Estructuras de datos en JAVA™

Compatible con JAVA 2

MARK ALLEN WEISS

Addison Wesley

Encuentra más libros en:

www.librosdejoe.com

MARK ALLEN WEISS

Estructuras de datos en JAVA™

Compatible con JAVA™2

Este libro presenta las estructuras de datos y algoritmos desde la perspectiva del pensamiento abstracto y la resolución de problemas. En este texto, el profesor Mark Allen Weiss adopta un enfoque actual, a través de la separación de la especificación y la implementación de las estructuras de datos. En concreto, en la Parte II del libro se presentan las interfaces de las estructuras de datos; a continuación, se ofrece a los lectores la oportunidad de usar dichas estructuras de datos en una gran variedad de ejemplos prácticos, antes de introducir en la Parte IV su implementación. Con este enfoque los lectores se acostumbrarán a razonar de forma más abstracta.

Características

- Contiene múltiples ejemplos de aplicación, de una cierta entidad, cuyo código original está disponible en Internet y ha sido comprobado y revisado por un programador profesional.
- Se proporciona una amplia introducción al lenguaje Java. También se trata en un apéndice las Interfaces Gráficas de Usuario (GUIs).
- Incluye abundante material pedagógico en forma de notas al margen y resúmenes adicionales al final de cada capítulo para aclarar los conceptos más complejos.
- Ofrece flexibilidad en el orden de estudio de los temas, minimizando las dependencias entre los distintos capítulos.

Acerca del Autor

Mark Allen Weiss es Catedrático de Informática en la Florida International University. Se doctoró en Ciencias de la Computación en la Universidad de Princeton donde estudió bajo la dirección del profesor Robert Sedgewick. Ha recibido diversos premios de investigación y reconocimientos por su espléndida labor docente. Tiene una amplia experiencia en el manejo de los lenguajes estructurados más modernos, que se ha materializado en diversos textos de amplia divulgación como *Algorithms, Data Structures and Problem Solving with C++* y los libros de la serie *Data Structures and Algorithm Analysis in Pascal, Ada, C y C++*, publicados por Addison-Wesley.

Pearson Educación

ISBN 84-7829-035-4

9 788478 290352

Signatura 4-64/722 N
Tipo préstamo O

Contenido

Prólogo XIX

Un enfoque novedoso XX
Prerrequisitos XX
Java XXI
Organización del libro XXII
Organización del libro por capítulos XXII
Dependencias entre capítulos XXIV
Partes independientes XXIV
Fundamentos matemáticos XXV
Organización del curso XXV
Ejercicios XXVI
Detalles pedadógicos XXVI
Disponibilidad del código XXVI
Guía del profesor XXVII
Agradecimientos XXVII

Prólogo a la edición en español XXIX

PRIMERA PARTE
Un recorrido por Java

1. Java básico 3

1.1. El entorno general 4
1.2. El primer programa 5
1.2.1. Comentarios 5
1.2.2. `main` 6
1.2.3. Salida por pantalla 6
1.3. Tipos primitivos 6
1.3.1. Los tipos primitivos 6
1.3.2. Las constantes 7
1.3.3. Declaración e inicialización de tipos primitivos 7
1.3.4. Entrada y salida por terminal 8

1.4. Operadores básicos 8
1.4.1. Operadores de asignación 8
1.4.2. Operadores aritméticos binarios 9
1.4.3. Operadores unarios 10
1.4.4. Conversiones de tipo 10
1.5. Instrucciones condicionales 11
1.5.1. Operadores relacionales y de igualdad 11
1.5.2. Operadores lógicos 11
1.5.3. La instrucción `if` 12
1.5.4. La instrucción `while` 13
1.5.5. La instrucción `for` 14
1.5.6. La instrucción `do` 15
1.5.7. `break` y `continue` 15
1.5.8. La instrucción `switch` 16
1.5.9. El operador condicional 17
1.6. Métodos 18
1.6.1. Sobrecarga de los nombres de los métodos 18
1.6.2. Clases de almacenamiento 19
Resumen 19
Elementos del juego 20
Errores comunes 21
En Internet 22
Ejercicios 22
Bibliografía 24

2. Referencias 25

2.1. ¿Qué es una referencia? 25
2.2. Nociones básicas sobre objetos y referencias 27
2.2.1. El operador punto (.) 28
2.2.2. Declaración de objetos 28
2.2.3. Recogida de basura 29
2.2.4. El significado de = ... 29
2.2.5. Paso de parámetros 30
2.2.6. El significado de == 31
2.2.7. Sobrecarga de operadores para objetos 31
2.3. Cadenas de caracteres 31
2.3.1. Conceptos básicos de la manipulación de `Strings` 32
2.3.2. Concatenación de cadenas 32
2.3.3. Comparando cadenas 33
2.3.4. Otros métodos del tipo `String` 33
2.3.5. Conversión entre cadenas y tipos primitivos 33
2.4. Vectores 34
2.4.1. Declaración, asignación y métodos 34
2.4.2. Expansión dinámica de vectores 36
2.4.3. Vectores multidimensionales 37
2.4.4. Argumentos de la línea de comandos 39
2.5. Manejo de excepciones 39
2.5.1. Procesamiento de excepciones 39

2.5.2. La cláusula `finally` 40
2.5.3. Excepciones más comunes 41
2.5.4. Las cláusulas `throw` y `throws` 42
2.6. Entrada y salida 42
2.6.1. Operaciones básicas de E/S 43
2.6.2. El objeto `StringTokenizer` 44
2.6.3. Ficheros de acceso secuencial 45
Resumen 46
Elementos del juego 47
Errores comunes 48
En Internet 48
Ejercicios 49
Referencias 49

3. Objetos y clases 51

3.1. ¿Qué es la programación orientada a objetos? 51
3.2. Un ejemplo sencillo 53
3.3. Javadoc 54
3.4. Métodos básicos 57
3.4.1. Constructores 57
3.4.2. Métodos modificadores y de acceso 57
3.4.3. Salida y `toString` 58
3.4.4. `equals` 58
3.4.5. Métodos `static` 59
3.4.6. `main` 60
3.5. Paquetes 60
3.5.1. La directiva `import` 60
3.5.2. La instrucción `package` 61
3.5.3. La variable de entorno `CLASSPATH` 62
3.5.4. Reglas de visibilidad amistosa dentro de un paquete 63
3.5.5. Compilación separada 64
3.6. Construcciones adicionales 64
3.6.1. La referencia `this` 64
3.6.2. La abreviatura `this` para constructores 65
3.6.3. El operador `instanceof` 65
3.6.4. Atributos estáticos 65
3.6.5. Inicializadores estáticos 66
Resumen 67
Elementos del juego 68
Errores comunes 69
En Internet 69
Ejercicios 70
Bibliografía 71

4. Herencia 73

4.1. ¿Qué es la herencia? 73
4.2. Sintaxis básica de Java 76
4.2.1. Reglas de visibilidad 76

4.2.2. El constructor y `super` ... 77
4.2.3. Métodos y clases finales ... 78
4.2.4. Sobreescribiendo un método ... 79
4.2.5. Métodos y clases abstractos ... 79
4.3. Ejemplo: extensión de la clase `Figura` ... 82
4.3.1. Disgresión: una introducción a la ordenación ... 84
4.4. Herencia múltiple ... 87
4.5. El interfaz ... 88
4.5.1. Especificación de una interfaz ... 88
4.5.2. Implementación de una interfaz ... 89
4.5.3. Varios inferfaces ... 91
4.6. Implementación de componentes genéricas ... 91
Resumen ... 94
Elementos del juego ... 95
Errores comunes ... 96
En Internet ... 96
Ejercicios ... 97
Bibliografía ... 99

SEGUNDA PARTE
Algoritmos y fundamentos de programación

5. Análisis de algoritmos ... 103

5.1. ¿Qué es el análisis de algoritmos? ... 104
5.2. Ejemplos de tiempo de ejecución de algoritmos ... 107
5.3. El problema de la subsecuencia de suma máxima ... 108
5.3.1. El algoritmo $O(N^3)$ obvio ... 109
5.3.2. Un algoritmo mejorado $O(N^2)$ ... 112
5.3.3. Un algoritmo lineal ... 113
5.4. Reglas generales para la notación O ... 114
5.5. Logaritmos ... 119
5.6. Problema de la búsqueda estática ... 121
5.6.1. Búsqueda secuencial ... 121
5.6.2. Búsqueda binaria ... 122
5.6.3. Búsqueda interpolada ... 124
5.7. Comprobar el análisis de un algoritmo ... 125
5.8. Limitaciones del análisis O ... 127
Resumen ... 127
Elementos del juego ... 128
Errores comunes ... 129
En Internet ... 129
Ejercicios ... 130
Bibliografía ... 134

6. Estructuras de datos ... 137

6.1. ¿Por qué necesitamos estructuras de datos? ... 137
6.2. Las pilas ... 139
6.2.1. Las pilas y los lenguajes de programación ... 141

6.3. Las colas ... 142
6.4. Listas enlazadas ... 144
6.5. Árboles generales ... 147
6.6. Árboles binarios de búsqueda ... 149
6.7. Tablas hash ... 153
6.8. Colas de prioridad ... 155
Resumen ... 158
Elementos del juego ... 159
Errores comunes ... 159
En Internet ... 160
Ejercicios ... 160
Bibliografía ... 163

7. Recursión ... 165

7.1. ¿Qué es la recursión? ... 165
7.2. Fundamentos: demostraciones por inducción matemática ... 166
7.3. Recursión básica ... 170
7.3.1. Impresión de números en cualquier base ... 170
7.3.2. ¿Por qué funciona? ... 172
7.3.3. Cómo funciona ... 174
7.3.4. Demasiada recursión puede ser peligrosa ... 175
7.3.5. Ejemplos adicionales ... 176
7.4. Aplicaciones numéricas ... 180
7.4.1. Aritmética modular ... 181
7.4.2. Exponenciación modular ... 181
7.4.3. Máximo común divisor e inversos multiplicativos ... 183
7.4.4. El sistema de criptografía RSA ... 185
7.5. Algoritmos divide y vencerás ... 188
7.5.1. El problema de la subsecuencia de suma máxima ... 188
7.5.2. Análisis de un algoritmo divide y vencerás sencillo ... 190
7.5.3. Una cota superior general para los tiempos de ejecución de los algoritmos divide y vencerás ... 195
7.6. Programación dinámica ... 197
7.7. Algoritmos de vuelta atrás ... 201
Resumen ... 205
Elementos del juego ... 205
Errores comunes ... 206
En Internet ... 207
Ejercicios ... 207
Bibliografía ... 211

8. Algoritmos de ordenación ... 213

8.1. ¿Por qué es importante la ordenación? ... 213
8.2. Preliminares ... 215
8.3. Análisis de la ordenación por inserción y otras ordenaciones simples ... 215
8.4. Shellsort ... 217
8.4.1. Rendimiento de Shellsort ... 219

8.5. Mergesort 221
8.5.1. Mezcla lineal de vectores ordenados 221
8.5.2. El algoritmo mergesort 224
8.6. Quicksort 225
8.6.1. El algoritmo quicksort 225
8.6.2. Análisis de quicksort 227
8.6.3. Seleccionando el pivote 230
8.6.4. Estrategia de partición 232
8.6.5. Elementos iguales al pivote 234
8.6.6. Partición con la mediana de tres 234
8.6.7. Vectores pequeños 235
8.6.8. Rutina de quicksort en Java 236
8.7. Selección rápida 238
8.8. Una cola inferior para la ordenación 240
Resumen 241
Elementos del juego 241
Errores comunes 242
En Internet 242
Ejercicios 242
Bibliografía 246

9. Números aleatorios 249

9.1. ¿Por qué son necesarios los números aleatorios? 249
9.2. Generadores de números aleatorios 250
9.3. Números aleatorios no uniformes 256
9.4. Generación de una permutación aleatoria 258
9.5. Algoritmos aleatorios 259
9.6. Test aleatorio de primalidad 261
Resumen 265
Elementos del juego 265
Errores comunes 266
En Internet 266
Ejercicios 267
Bibliografía 268

TERCERA PARTE
Aplicaciones

10. Juegos y diversión 273

10.1. Sopas de letras 273
10.1.1. Teoría 274
10.1.2. Implementación en Java 276
10.2. El juego de las tres en raya 281
10.2.1. Poda alfa-beta 281
10.2.2. Tablas de transposición 282
10.3. El ajedrez 285
Resumen 287
Elementos del juego 287

Errores comunes 288
En Internet 288
Ejercicios 288
Bibliografía 290

11. Las pilas y los compiladores 291

11.1. Analizador de símbolos equilibrados 291
11.1.1. El algoritmo básico 291
11.1.2. Implementación 293
11.2. Una calculadora sencilla 301
11.2.1. Máquinas postfijas 302
11.2.2. Conversión de notación infija a postfija 302
11.2.3. Implementación 305
11.2.4. Árboles sintácticos de expresiones 312
Resumen 313
Elementos del juego 313
Errores comunes 314
En Internet 314
Ejercicios 315
Bibliografía 316

12. Utilidades 317

12.1. Compresión de ficheros 317
12.1.1. Códigos sin prefijos 318
12.1.2. Algoritmo de Huffman 320
12.1.3. La fase de codificación 322
12.1.4. La fase de decodificación 323
12.1.5. Consideraciones prácticas 324
12.2. Generador de referencias cruzadas 324
12.2.1. Ideas básicas 324
12.2.2. Implementación en Java 325
Resumen 330
Elementos del juego 330
Errores comunes 330
En Internet 330
Ejercicios 331
Bibliografía 333

13. Simulación 335

13.1. El problema Josephus 335
13.1.1. La solución simple 336
13.1.2. Un algoritmo más eficiente 338
13.2. Simulación dirigida por eventos 340
13.2.1. Ideas básicas 340
13.2.2. Ejemplo: simulación de un banco de módems 341
Resumen 348
Elementos del juego 348

Errores comunes 349
En Internet 349
Ejercicios 349

14. Grafos y caminos 353

14.1. Definiciones 353
14.1.1. Representación 355
14.2. Problema del camino mínimo sin pesos 366
14.2.1. Teoría 366
14.2.2. Implementación en Java 370
14.3. Problema de los caminos mínimos con pesos positivos 371
14.3.1. Teoría: algoritmo de Dijkstra 371
14.3.2. Implementación en Java 375
14.4. Problema del camino mínimo con costes negativos 376
14.4.1. Teoría 376
14.4.2. Implementación en Java 377
14.5. Problemas de caminos en grafos acíclicos 379
14.5.1. Ordenación topológica 379
14.5.2. Teoría del algoritmo de caminos mínimos con un grafo acíclico 381
14.5.3. Implementación en Java 382
14.5.4. Una aplicación: análisis de caminos críticos 382
Resumen 386
Elementos del juego 386
Errores comunes 387
En Internet 388
Ejercicios 388
Bibliografía 391

CUARTA PARTE
Implementaciones

15. Pilas y colas 395

15.1. Implementación dinámica de vectores 395
15.1.1. Pilas 395
15.1.2. Colas 399
15.2. Implementaciones con listas enlazadas 404
15.2.1. Pilas 404
15.2.2. Colas 408
15.3. Comparación de los dos métodos 409
15.4. Colas dobles 411
Resumen 411
Elementos del juego 412
Errores comunes 412
En Internet 413
Ejercicios 413

16. Listas enlazadas 415

16.1. Ideas básicas 415
16.1.1. Nodos cabecera 417

16.1.2. Clases iteradoras 418
16.2. Implementación en Java 419
16.3. Listas doblemente enlazadas y listas enlazdas circulares 426
16.4. Listas enlazadas ordenadas 428
Resumen 430
Elementos del juego 431
Errores comunes 431
En Internet 431
Ejercicios 432

17. Árboles 435

17.1. Árboles generales 435
17.1.1. Definiciones 435
17.1.2. Implementación 437
17.1.3. Una aplicación: sistemas de ficheros 437
17.2. Árboles binarios 441
17.3. Árboles y recursión 448
17.4. Recorrido de árboles: clases iteradoras 450
17.4.1. Recorrido en postorden 453
17.4.2. Recorrido en orden simétrico 457
17.4.3. Recorrido en preorden 458
17.4.4. Recorrido por niveles 459
Resumen 461
Elementos del juego 461
Errores comunes 462
En Internet 462
Ejercicios 463

18. Árboles binarios de búsqueda 467

18.1. Ideas básicas 467
18.1.1. Las operaciones 468
18.1.2. Implementación en Java 470
18.2. Búsqueda por posición en el orden 477
18.2.1. Implementación en Java 477
18.3. Análisis de las operaciones de los árboles binarios de búsqueda 481
18.4. Árboles AVL 484
18.4.1. Propiedades 485
18.4.2. Rotación simple 487
18.4.3. Rotación doble 490
18.4.4. Resumen de la inserción en un árbol AVL 492
18.5. Árboles rojinegros 493
18.5.1. Inserción ascendente 495
18.5.2. Árboles rojinegros descendentes 497
18.5.3. Implementación en Java 503
18.5.4. Eliminación descendente 505
18.6. AA-Árboles 506
18.6.1. Inserción 509
18.6.2. Eliminación

18.6.3. Implementación en Java 509
18.7. B-Árboles 512
Resumen 519
Elementos del juego 520
Errores comunes 521
En Internet 521
Ejercicios 522
Bibliografía 524

19. Tablas hash 527

19.1. Ideas básicas 527
19.2. Función de localización 529
19.3. Exploración lineal 531
19.3.1. Análisis de la exploración lineal 532
19.3.2. Lo que sucede realmente: la agrupación primaria 533
19.3.3. Análisis de la operación `buscar` 534
19.4. Exploración cuadrática 536
19.4.1. Implementación en Java 541
19.4.2. Análisis de la exploración cuadrática 544
19.5. Hashing enlazado 545
Resumen 546
Elementos del juego 547
Errores comunes 548
En Internet 548
Ejercicios 549
Bibliografía 551

20. Una cola de prioridad: el montículo binario 553

20.1. Ideas básicas 553
20.1.1. Propiedad estructural 554
20.1.2. Propiedad de ordenación de los montículos 555
20.1.3. Operaciones permitidas 556
20.2. Implementación de las operaciones básicas 559
20.2.1. `insertar` 559
20.2.2. `eliminarMin` 562
20.3. `arreglarMonticulo`: construcción en tiempo lineal del montículo 564
20.4. Operaciones avanzadas: `reducirClave` y `mezclar` 568
20.5. Ordenación interna: método del montículo 568
20.6. Ordenación externa 571
20.6.1. Por qué necesitamos nuevos algoritmos 572
20.6.2. Modelo de ordenación externa 572
20.6.3. El algoritmo sencillo 572
20.6.4. Mezcla multiaria 574
20.6.5. Mezcla multifase 575
20.6.6. Selección del reemplazo 576
Resumen 578
Elementos del juego 578
Errores comunes 579

En Internet 579
Ejercicios 580
Bibliografía 583

QUINTA PARTE
Estructuras de datos avanzadas

21. Árboles de ensanchamiento 587

21.1. Auto-ajustamiento y análisis amortizado 587
21.1.1. Cotas de tiempo amortizadas 588
21.1.2. Una estrategia simple de auto-ajustamiento (que no funciona) 589
21.2. Árboles básicos de ensanchamiento ascendente 591
21.3. Operaciones básicas de los árboles de ensanchamiento 593
21.4. Análisis del ensanchamiento ascendente 594
21.4.1. Demostración de la cota de ensanchamiento 597
21.5. Árboles de ensanchamiento descendente 600
21.6. Implementación de los árboles con ensanchamiento descendente 602
21.7. Comparaciones de los árboles de ensanchamiento con otros árboles de búsqueda ... 608
Resumen 608
Elementos del juego 608
Errores comunes 609
En Internet 609
Ejercicios 609
Bibliografía 611

22. Colas de prioridad con mezcla 613

22.1. Los montículos sesgados 613
22.1.1. La mezcla es importante 613
22.1.2. Mezcla simple de árboles con ordenación de montículos 614
22.1.3. El montículo sesgado: una modificación sencilla 615
22.1.4. Análisis del montículo sesgado 616
22.2. Los montículos de emparejamientos 618
22.2.1. Operaciones del montículo de emparejamientos y teoría 618
22.2.2. Implementación del montículo de emparejamientos 620
22.2.3. Aplicación: el algoritmo de Dijkstra para la obtención de caminos mínimos. 626
Resumen 629
Elementos del juego 629
Errores comunes 630
En Internet 630
Ejercicios 630
Bibliografía 631

23. Estructura de partición 633

23.1. Relaciones de equivalencia 633
23.2. Equivalencia dinámicas y dos aplicaciones 634
23.2.1. Aplicación #1: árboles de recubrimiento mínimo 635
23.2.2. Aplicación #2: el problema del antecesor común más próximo 637

23.3. El algoritmo de búsqueda rápida ... 641
23.4. El algoritmo de unión rápida ... 642
23.4.1. Algoritmos de unión inteligentes ... 643
23.4.2. Compresión de caminos ... 645
23.5. Implementación en Java ... 647
23.6. Caso peor de la unión por rango y la compresión de caminos ... 648
23.6.1. Análisis del algoritmo unir/buscar ... 649
Resumen ... 654
Elementos del juego ... 655
Errores comunes ... 656
En Internet ... 656
Ejercicios ... 656
Bibliografía ... 658

APÉNDICES

A. Plataformas para Java ... 663

A.1. Estableciendo el entorno ... 663
A.1.1. Instrucciones para Unix ... 663
A.1.2. Instrucciones para Windows95/NT ... 664
A.2. JDK de Sun ... 665
A.3. Entornos de desarrollo visual ... 666
A.3.1. Symantec Café ... 667
A.3.2. Microsoft Visual J++ ... 671

B. Operadores ... 679

C. Algunas rutinas de librerías ... 681

C.1. Clases del paquete `java.lang` ... 681
C.1.1. `Character` ... 681
C.1.2. `Integer` ... 682
C.1.3. `Object` ... 683
C.1.4. `String` ... 684
C.1.5. `StringBuffer` ... 685
C.1.6. `System` ... 687
C.1.7. `Thread` ... 688
C.1.8. `Throwable` ... 689
C.2. Clases del paquete `java.io` ... 689
C.2.1. `BufferedReader` ... 689
C.2.2. `File` ... 690
C.2.3. `FileReader` ... 691
C.2.4. `InputStreamReader` ... 691
C.2.5. `PushbackReader` ... 691
C.3. Clases del paquete `java.util` ... 692
C.3.1. `Random` ... 692
C.3.2. `StringTokenizer` ... 693
C.3.3. `Vector` ... 694
En Internet ... 694

D. Interfaces gráficas de usuario 695

D.1. El Abstract Window Toolkit 695
D.2. Elementos básicos del AWT 696
D.2.1. Component 697
D.2.2. `Container` 697
D.2.3. Ventanas de alto nivel 698
D.2.4. Panel 700
D.2.5. Componentes importantes de la E/S 701
D.3. Principios básicos del AWT 705
D.3.1. Configuraciones 705
D.3.2. Gráficos 708
D.3.3. Eventos 710
D.3.4. Resumen: encajando las piezas 713
D.4. Animaciones y hebras 713
D.5. Applets 715
D.5.1. Lenguaje de hipertexto 716
D.5.2. Parámetros 719
D.5.3. Limitaciones de los applets 719
D.5.4. Conversión de una aplicación en un applet 720
D.5.5. Applets con animación 722
Resumen 724
Elementos del juego 724
Errores comunes 726
En Internet 727
Ejercicios 728
Bibliografía 730

Índice analítico 731

Prólogo

La versión original de este libro se diseñó para utilizarla en un curso de informática de dos semestres de duración, comenzando por lo que se conoce habitualmente como Estructuras de Datos.

El contenido de dicho curso ha ido evolucionando a lo largo del tiempo. Mientras que hay un cierto consenso general con respecto a los temas a cubrir, todavía hay mucho desacuerdo en relación con los detalles. Un tema uniformemente aceptado lo constituyen los principios del desarrollo de software, y más en concreto los conceptos de encapsulación y ocultamiento de la información. Todos los cursos de estructuras de datos tienden a incluir una introducción al análisis del tiempo de ejecución, la recursión, los algoritmos básicos de ordenación y las estructuras de datos elementales. En muchas universidades se ofrecen cursos avanzados que cubren a un nivel superior conceptos de estructuras de datos, algoritmos y análisis del tiempo de ejecución. El material de este libro se ha diseñado con la intención de ser usado en cursos de ambos niveles, eliminando así la necesidad de adquirir un segundo libro de texto.

Aunque los debates más enconados en este terreno se centran en torno a la elección del lenguaje de programación más apropiado, hay otras decisiones fundamentales a tomar, incluyendo las siguientes:

- Si se debe o no introducir pronto el diseño orientado a objetos y/o el diseño basado en objetos.
- El nivel de rigor matemático que se va a mantener.
- El equilibrio adecuado entre la implementación de las estructuras de datos y su utilización.
- Detalles de programación relacionados con el lenguaje elegido (por ejemplo, si las GUI deberían usarse pronto o no).

Mi objetivo a la hora de escribir este libro es proporcionar una introducción práctica a las estructuras de datos y los algoritmos desde el punto de vista del pensamiento abstracto y la resolución de problemas. Intento cubrir todos los detalles importantes concernientes a las estructuras de datos, su análisis y sus implementaciones en Java, aunque evitando aquellas estructuras que si bien son interesantes desde un punto de vista teórico, no se utilizan con asiduidad. Es imposible cubrir en un único curso todas las estructuras de datos presentadas en este libro, in-

cluyendo sus usos y análisis. Por ello, he diseñado el libro para permitir una cierta flexibilidad a los profesores a la hora de fijar su temario. Cada profesor tendrá que decidir el equilibrio adecuado entre teoría y práctica, y elegir en consecuencia aquellos temas que mejor encajen en su asignatura. Como indico más adelante en este prefacio, el texto está organizado procurando minimizar las dependencias entre los distintos capítulos.

Un enfoque novedoso

El libro adopta un enfoque novedoso separando la presentación de cada estructura de datos en su especificación (a través de una interfaz Java) e implementación. Este enfoque proporciona varios beneficios, entre ellos la promoción del pensamiento abstracto. Antes de conocer la implementación se presenta la interfaz de la clase, obligando así al estudiante a pensar desde el principio sobre la funcionalidad y eficiencia potencial de las distintas estructuras de datos. Por ejemplo, los estudiantes verán programas que usan las tablas hash cientos de páginas antes de implementarlas. Las interfaces de las estructuras de datos se describen en un único capítulo de la Parte II. En la Parte II se describen también las técnicas básicas de análisis, la recursión y el problema de la ordenación. La Parte III contiene una colección de aplicaciones que usan las distintas estructuras de datos. La implementación de las estructuras básicas no se estudia hasta la Parte IV, una vez se han usado dichas estructuras. Puesto que todo el código que aparece en el libro está disponible (véase *Disponibilidad del código* en la página XXVI), los estudiantes pueden diseñar proyectos de programación de cierta envergadura bastante pronto, usando el software existente. Las herramientas de desarrollo de software en todos los lenguajes vienen junto con grandes librerías, y la mayor parte de las estructuras de datos interesantes formarán parte de estas librerías más tarde o más temprano. Preveo que en los cursos de estructuras de datos se producirá un eventual desplazamiento del énfasis de la implementación al uso.

Muchos profesores preferirán un enfoque más tradicional en el que cada estructura de datos se define primero, luego se implementa y por último se usa. Puesto que no hay dependencias entre las partes III y IV, se puede también enseñar fácilmente siguiendo el método tradicional utilizando este libro.

Prerrequisitos

Los estudiantes que vayan a usar este libro deberían tener previamente conocimientos sobre algún lenguaje de programación modular u orientado a objetos. Se suponen conocimientos sobre las características básicas, incluyendo tipos primitivos, operadores, estructuras de control, funciones (métodos) y entrada/salida (pero no necesariamente de vectores y clases).

Los estudiantes que hayan asistido a un curso de Java pueden comenzar con el Capítulo 3 (o quizá más adelante). Los estudiantes que hayan usado otros lenguajes deberían empezar con el Capítulo 1. Ellos también deberían usar los apéndices, que combinados con la Parte I, proporcionan una gran cantidad de información sobre Java. Por si algún estudiante quisiera apoyarse en algún libro de referencia de Java, en el Capítulo 1 se dan algunas recomendaciones al respecto.

Los conocimientos sobre matemática discreta son útiles pero no absolutamente necesarios. Aunque hay varias demostraciones matemáticas, muchas de ellas están precedidas por una breve introducción matemática. Los capítulos 7, y del 18 al 23, requieren un cierto grado de sofisticación matemática. Eventualmente el profesor podría evitar los aspectos matemáticos de las demostraciones presentando solamente los resultados. Todas las demostraciones del libro están claramente marcadas y separadas del cuerpo central del texto.

Java

Este libro de texto presenta el material utilizando el lenguaje de programación Java. Éste es un lenguaje relativamente reciente que a menudo se compara con C++. Java ofrece muchas ventajas, y muchos programadores lo consideran un lenguaje mucho más seguro, portable y fácil de usar que C++.

El uso de Java requiere tomar algunas decisiones a la hora de escribir un libro de texto. Algunas de estas decisiones han sido las siguientes:

1. *Se usan exclusivamente construcciones de Java 1.1:* aunque cuando se escribió la versión original de este libro solamente había un compilador de Java 1.1 disponible, estoy seguro de que actualmente existen otros. Asegúrese de que está usando un compilador compatible con Java 1.1. La diferencia más notable en el texto principal es el uso de las clases para la E/S de Java 1.1 en lugar de las, venidas a menos, clases de Java 1.0.2. Entre ellas se encuentran `BufferedReader`, `FileReader` y `InputStreamReader`.
2. *No se hace mucho hincapié en las GUIs:* constituyen una característica fantástica de Java, y su uso se describe en el Apéndice D, pero son un detalle de implementación más que un tema del curso. Para satisfacer el objetivo del libro, el apéndice es el mejor lugar donde estudiar las GUIs. Además es la parte del lenguaje que más ha cambiado en Java 1.1. Es la parte más inestable del lenguaje.
3. *No se hace hincapié en los applets:* los applets usan GUIs. Además, lo importante del curso son las estructuras de datos, no las características del lenguaje. En el Apéndice D se estudian los applets. Los profesores pueden hacer que los estudiantes diseñen applets como parte del uso (o simulación) de las estructuras de datos.
4. *No se usan las características más novedosas de Java 1.1:* éstas incluyen clases internas, nuevas reglas para variables finales, etc. He intentado usar el corazón de Java para evitar excesivas características curiosas.
5. *Se estudia el concepto de puntero cuando se introducen las variables referencia:* Java no tiene un tipo puntero, sino un tipo referencia. Sin embargo, los punteros han sido un tema importante en los cursos de estructuras de datos que es necesario introducir. Ilustro el concepto de punteros en otros lenguajes al estudiar las variables referencia.
6. *Las hebras solamente se mencionan en el Apéndice D, y en relación con las animaciones:* algunos miembros de la comunidad informática defienden que la computación multihebra debería incluirse en un curso de este tipo. Si se desea, se puede utilizar para ello el Apéndice D.

Como todos los lenguajes de programación, Java también tiene desventajas. No soporta directamente programación genérica, sino que para hacerlo se requiere dar un rodeo que se estudia en el Capítulo 3. El soporte para la E/S en Java es mínimo. Por ello los ejemplos presentados hacen un mínimo uso de las facilidades de E/S de Java.

Organización del libro

En su Parte I el libro presenta el lenguaje Java y la programación orientada a objetos (particularmente la abstracción). Se estudian los tipos primitivos, los tipos referencia y algunas de las clases y excepciones predefinidas, continuando con el diseño de nuevas clases y la herencia.

La Parte II presenta la notación *O* y algunos paradigmas algorítmicos, como la recursión y los algoritmos aleatorios. Se dedica un capítulo completo al tema de la ordenación, y otro a la descripción de las estructuras de datos básicas. Se presentan las interfaces y los tiempos de ejecución de las operaciones *sin* dar una implementación. En este punto del texto, el profesor puede optar entre varios enfoques para presentar el material restante, incluyendo los dos siguientes:

1. Utilizar la correspondiente implementación de la Parte IV al describir cada tipo de datos. El profesor puede pedir a los alumnos que extiendan las clases de diversas maneras, como se sugiere en los ejercicios.
2. Mostrar cómo se usan las interfaces y cubrir la implementación más tarde en el curso. Los casos de estudio de la Parte III pueden utilizarse para seguir este enfoque. Ya que las implementaciones completas pueden obtenerse vía Internet, el profesor puede proporcionar una librería de clases para que se utilicen en las prácticas de programación.

La Parte V presenta estructuras de datos avanzadas, como los árboles de ensanchamiento, los montículos de emparejamientos y las particiones, que pueden ser cubiertas si el tiempo lo permite, o en un curso más avanzado.

Organización del libro por capítulos

La Parte I está formada por cuatro capítulos que describen los conceptos básicos de Java utilizados en el texto. El Capítulo 1 presenta los tipos primitivos e ilustra cómo escribir programas básicos en Java. El Capítulo 2 estudia los tipos referencia e ilustra el concepto básico de puntero (aunque Java no contiene punteros) de tal forma que los estudiantes aprendan este concepto básico de los cursos de Estructuras de Datos. Se ilustran varios tipos referencia básicos (cadenas, vectores y ficheros), y se discute el uso de excepciones. El Capítulo 3 continúa el estudio describiendo cómo se implementa una clase. El Capítulo 4 muestra el uso de la herencia al diseñar jerarquías (incluyendo clases de excepciones) y componentes genéricas.

La Parte II se centra en el estudio de los algoritmos y fundamentos de programación básicos. El Capítulo 5 proporciona una discusión completa de la complejidad en tiempo y de la notación *O*, también se estudia y analiza la búsqueda binaria. El Capítulo 6 es un capítulo crucial en el que se presentan las interfaces de las estructuras de datos, dando argumentos intuitivos sobre cual debería ser el tiempo de ejecución de cada operación soportada por cada estructura de datos. (La implementación de estas estructuras de datos se incluye en la Parte IV.) El Capítulo 7 describe la recursión introduciendo primero la noción de demostración por inducción. También estudia las técnicas de divide y vencerás, programación dinámica y vuelta atrás. Una sección estudia varios algoritmos numéricos que se utilizan en la implementación de un importante algoritmo de encriptación: el sistema de encriptación RSA. Para la mayoría de los estudiantes, la segunda mitad del Capítulo 7 correspondería más bien a un curso avanzado. El Capítulo 8 describe, implementa y analiza varios algoritmos básicos de ordenación, incluyendo la ordenación por inserción, Shellsort, mergesort y quicksort. También demuestra la cota inferior clásica de la ordenación y discute el problema relacionado de la selección. Por último, el Capítulo 9 es un breve capítulo sobre números aleatorios, que incluye su generación y la utilización de algoritmos aleatorios.

La Parte III proporciona varios casos de estudio, en la que cada capítulo gira en torno a un tema general. El Capítulo 10 ilustra varias técnicas importantes estudiando juegos. El Capítulo 11 estudia el uso de pilas en los lenguajes de programación, examinando un algoritmo que comprueba que los símbolos estén equilibrados y el algoritmo clásico de evaluación de expresiones con precedencia. Se dan implementaciones completas de ambos algoritmos. El Capítulo 12 estudia utilidades básicas de compresión de ficheros y generación de referencias cruzadas y proporciona una implementación completa de un generador de referencias cruzadas. El Capítulo 13 presenta una breve descripción del concepto de simulación utilizando un problema que puede interpretarse como de simulación, y del concepto de simulación dirigida por eventos. Por último, el Capítulo 14 ilustra cómo se utilizan ciertas estructuras de datos para implementar eficientemente varios algoritmos de búsqueda de caminos mínimos sobre grafos.

La parte IV presenta las implementaciones de las estructuras de datos correspondientes a las interfaces del Capítulo 6. En esta parte, especialmente en los Capítulos 18 y 20, se utilizan algunas nociones matemáticas, que podrían evitarse a juicio del profesor. El Capítulo 15 proporciona implementaciones de las pilas y las colas. Estas estructuras se implementan primero utilizando un vector extensible, y después utilizando listas enlazadas. Las listas enlazadas generales se estudian en el Capítulo 16. Se dejan como ejercicios extensiones como las listas doblemente enlazadas, las listas enlazadas circulares y la implementación de cursores. El Capítulo 17 estudia los árboles e ilustra los esquemas de recorrido estándar. El Capítulo 18 es un capítulo detallado que proporciona diversas implementaciones de los árboles binarios de búsqueda. Inicialmente se muestran los árboles binarios de búsqueda básicos y después árboles de búsqueda que soportan búsqueda por posición. Los árboles AVL se definen pero no se implementan; por el contrario, sí se implementan los árboles rojinegros y los AA-árboles, de mayor interés práctico. Por último, se estudian los B-árboles. El Capítulo 19 estudia las tablas hash e implementa un esquema de exploración cuadrática, después de examinar una alternativa más simple. El Capítulo 20 describe los montículos binarios y examina la ordenación mediante montículos y la ordenación externa.

La Parte V incluye un material más indicado para un curso más avanzado o como referencia general. Sin embargo, los algoritmos son accesibles incluso en un curso de primer nivel. Por completitud, se incluyen complicados análisis matemáticos. El Capítulo 21 describe los árboles de ensanchamiento, los cuales son árboles binarios de búsqueda que se comportan bien en la práctica y que compiten con

los montículos binarios en algunas aplicaciones que requieren colas de prioridad. El Capítulo 22 describe las colas de prioridad que soportan operaciones de mezcla y proporciona una implementación de los montículos de emparejamientos. Finalmente, el Capítulo 23 examina la estructura clásica de partición.

Los apéndices contienen material adicional sobre Java. El Apéndice A ilustra cómo compilar y ejecutar programas Java en diversas plataformas. El Apéndice B enumera los operadores y su precedencia. El Apéndice C resume las librerías Java utilizadas en el texto. El Apéndice D describe el AWT y los applets. También estudia las hebras de ejecución en el contexto de las animaciones.

Dependencias entre capítulos

Hablando de forma general, la mayoría de los capítulos son independientes entre sí. Enumeramos a continuación las dependencias más notables:

- *Parte I:* con la excepción del caso de estudio `Figura` del Capítulo 4, todo el material de la Parte I debería ser cubierta en orden, antes de continuar con el resto del libro. El concepto de interfaz debe ser introducido, pero los detalles sobre la herencia podrían cubrirse en sentido amplio sólo si el profesor así lo elige.
- *Capítulo 5 (Análisis de algoritmos):* debería cubrirse antes de los Capítulos 6 y 8. La recursión (Capítulo 7) podría cubrirse antes de este capítulo, pero el profesor debería dar detalles para evitar la recursión ineficiente.
- *Capítulo 6 (Estructuras de datos):* puede cubrirse antes o durante la explicación del material de las Partes III y IV.
- *Capítulo 7 (Recursión):* el material de las Secciones 7.1-7.3 debería cubrirse antes del estudio de los algoritmos de ordenación, los árboles, el ejemplo de las tres en raya y los algoritmos de búsqueda de caminos mínimos. Por su parte, material como el sistema de encriptación RSA, la programación dinámica y la vuelta atrás es opcional.
- *Capítulo 8 (Ordenación):* debería ir tras los Capítulos 5 y 7. Sin embargo, es posible cubrir Shellsort sin haber visto estos capítulos, ya que Shellsort no es recursivo y un análisis riguroso de su tiempo de ejecución es demasiado complejo y no se incluye en el libro.
- *Capítulos 15 y 16 (Pilas/Colas/Listas):* pueden cubrirse en cualquier orden. Sin embargo, es preferible ver antes el Capítulo 15, ya que incluye ejemplos más sencillos de listas enlazadas.
- *Capítulos 17 y 18 (Árboles/Árboles de búsqueda):* pueden cubrirse en cualquier orden o incluso simultáneamente.

Partes independientes

- *Capítulo 9 (Aleatoriedad):* el material sobre números aleatorios puede estudiarse en cualquier momento, según resulte más conveniente.
- *Parte III (Ejemplos):* puede ser estudiada en cualquier momento y en casi cualquier orden. Hay pocas referencias a capítulos anteriores que pueden seguirse fácilmente. Éstas incluyen: la Sección 10.2 (tres en raya), que referencia a la Sección 7.7, y la Sección 12.2 (generación de referencias cruzadas), que referencia a un código similar de análisis léxico de la Sección 11.1 (comprobación de símbolos equilibrados).
- *Capítulos 19 y 20 (Tablas Hash/Colas de prioridad):* pueden estudiarse en cualquier momento.
- *Parte V (Estructuras de datos avanzadas):* este material es autocontenido y normalmente se estudia en cursos superiores.
- *Apéndices (Más Java):* el material sobre GUI del Apéndice D puede estudiarse en cualquier momento tras el Capítulo 4, según resulte conveniente.

Fundamentos matemáticos

He intentado mostrar los resultados con el rigor matemático necesario para satisfacer las expectativas de aquellos cursos de Estructuras de Datos en los que se hace énfasis en la teoría, y para los cursos superiores, que incluyen más análisis. Sin embargo, este material se separa del texto principal presentándose en forma de teoremas separados y, en algunos casos, secciones separadas (o subsecciones). De este modo, puede prescindirse de él en los cursos en los que se decida no conceder tanta importancia a los resultados teóricos.

En cualquier caso, no es necesario entender la demostración de un teorema para comprender el resultado en sí. Esta es otra muestra de la distinción entre una interfaz (el enunciado de un teorema) y su implementación (su demostración). Así, parte del material inherentemente matemático, como la Sección 7.4 (Aplicaciones numéricas de la recursión), puede evitarse sin que ello afecte a la comprensión del resto de cada capítulo.

Organización del curso

Un aspecto fundamental a la hora de enseñar el contenido de este libro es decidir cómo emplear el material contenido en las Partes II, III y IV. El material de la Parte I ha de ser explicado con detenimiento, y el estudiante debería realizar varios programas que muestren cómo debe realizarse el diseño, la implementación, la prueba de clases (incluyendo las genéricas), y quizás el diseño orientado a objetos empleando herencia. El Capítulo 5 discute la notación O. Un ejercicio adecuado para comprobar si el estudiante la ha comprendido es hacerle escribir un pequeño programa, pidiéndole que compare su tiempo de ejecución con los resultados obtenidos en los análisis teóricos.

Dentro de la metodología de separación entre especificación e implementación, el concepto clave del Capítulo 6 es que diferentes estructuras de datos soportan distintos esquemas de acceso con eficiencias diferentes. Un posible ejercicio es pedir a los estudiantes que escriban una estructura de datos ineficiente, para probarla después con uno cualquiera de los casos de estudio (a excepción del algoritmo tres en raya, que emplea recursión). Así, los estudiantes podrían comparar su estructura de datos ineficiente con una rutina de librería eficiente (pueden conseguirse mediante ftp anónimo, tal y como se explicará más tarde). Siguiendo este esquema, pueden probarse todos los casos de estudio (salvo el juego de las tres en raya), estudiando el comportamiento de cada estructura de datos. De este modo, el estudiante podría ver las distintas interfaces y su utilización, pero no cómo se implementan de forma eficiente. Viendo las cosas de esta forma incrementamos la

capacidad de los estudiantes para pensar de forma abstracta. También se les podría pedir que extendiesen el caso de estudio, pero, de nuevo, para ello no es necesario que conozcan cada detalle de las correspondientes estructuras de datos.

La implementación de las estructuras de datos puede ser discutida posteriormente, y la recursión se introducirá siempre que el profesor lo considere adecuado, teniendo en cuenta que debe haberse hecho antes de estudiar los árboles binarios de búsqueda. Los detalles de los algoritmos de ordenación pueden explicarse en cualquier momento tras haber introducido la recursión. En este punto, el curso puede continuar examinando los mismos casos de estudio, pero modificando las implementaciones de las estructuras de datos. Por ejemplo, se pueden realizar diversas pruebas empleando los distintos tipos de árboles binarios de búsqueda equilibrados.

Aquellos profesores que opten por una aproximación más tradicional, pueden limitarse a discutir un caso de estudio de la Parte III, explicando después una implementación de una estructura de datos de las que se encuentran en la Parte IV. Insistimos en que los capítulos del libro están diseñados de forma tan independiente como ha sido posible.

Ejercicios

Los ejercicios presentados son de cuatro tipos diferentes. Las *cuestiones breves* se reducen a contestar pequeñas preguntas o simular sobre el papel algún algoritmo descrito en el texto. La sección de *problemas teóricos* incluye cuestiones que requieren algún análisis matemático, y preguntas acerca de soluciones a problemas, relevantes desde un punto de vista teórico. La parte de *problemas prácticos* contiene cuestiones sencillas sobre programación, incluyendo preguntas de sintaxis o sobre trucos empleados en la codificación. Por último, la sección de *prácticas de programación* contiene ideas para realizar ejercicios más complicados y sugerencias para desarrollar applets de Java.

Detalles pedagógicos

- Las notas al margen se emplean para destacar conceptos e ideas importantes.
- La sección *Elementos del juego* enumera los términos más importantes junto con sus definiciones.
- La sección de *Errores comunes*, que se encuentra al final de cada capítulo, muestra una lista de los errores más habituales.
- Al final de cada capítulo también se suministra una sección de *Bibliografía* para encontrar aclaraciones y ampliaciones de los temas estudiados.

Disponibilidad del código

El código incluido en el libro es completamente funcional y ha sido probado en su versión original sobre JDK 1.1 de Sun. Dicha versión (en inglés) puede obtenerse mediante ftp anónimo en `aw.com`, y se encuentra en la Word Wide Web en `http://www.aw.com/cseng/titles/0-201-54991-3/` (a partir de aquí deben seguirse los enlaces indicados en cada caso). La sección *En Internet*, que se encuentra al final de cada capítulo, enumera los nombres de los ficheros del código del capítulo correspondiente.

Guía del profesor

Se encuentra disponible una guía del profesor (correspondiente al texto original en inglés) que ilustra los distintos usos posibles del material. Incluye ejemplos de cuestiones de examen, prácticas a proponer a los alumnos y programas alternativos de asignaturas en las que se desarrolle parte del contenido del texto. También pueden encontrarse en ella soluciones de ejercicios seleccionados. Los profesores interesados en obtenerla deben contactar con su representante local de ventas de Addison-Wesley para una mayor información.

Agradecimientos

En la preparación de este libro ha participado una gran cantidad de gente. Mucha de ella ya fue reconocida en el trabajo anterior, *Algorithms, Data Structures, and Problem Solving with C++*, sobre el que se basa este libro. Otra gran cantidad de gente, demasiado numerosa para ser citada aquí, ha enviado mensajes electrónicos señalando errores e inconsistencias, que he intentado arreglar en esta versión (en inglés).

En esta ocasión, me gustaría mostrar mi agradecimiento a todos mis colaboradores de Addison-Wesley: mi editora, Susan Hartman, y editora asociada, Katherine Harutunian, que me han ayudado en las decisiones más delicadas sobre la organización del material Java y que han contribuido a que este libro se realizase. Los distintos revisores han sugerido la introducción de numerosos cambios que han contribuido a mejorar el texto. Ellos son: Laura K. Michaels, Phyllis Coyne y Sarah Hallet-Corey. Por otra parte, Syndi Hirsch realizó un diseño fantástico para la portada del libro y, como siempre, Tom Ziolkowski ha desarrollado una labor extraordinaria en el departamento de marketing. Agradezco especialmente a Pat Unubun, mi editora de producción, sus constantes esfuerzos en la coordinación completa del proyecto.

También doy las gracias al resto de mis revisores, quienes realizaron valiosos comentarios, muchos de los cuales he introducido en el texto: John Chenoweth, Rob Clark, John Franco, Susanne Hupfer, Steven L. Jenkins, Josephine DeGuzman Mendoza, Viera K. Proulx y Amr Sabry.

Parte del material empleado en el texto es una adaptación del que aparece en mi libro *Efficient C Programming: A Practical Approach* (Prentice-Hall, 1995) y ha sido empleado con el consentimiento de su editor. He intentado incluir las oportunas referencias a otros textos al final de cada capítulo.

Mi página en la World Wide Web, `http://www.cs.fiu.edu/~weiss`, contiene código fuente actualizado, una lista de erratas y un enlace para poder comunicarme otros fallos detectados (en el original en inglés).

Prólogo a la edición en español

El presente texto realiza un estudio exhaustivo de las estructuras de datos materializadas en el marco de la programación orientada a objetos. Para ello, se utiliza un lenguaje moderno como Java, que reúne unas características muy atractivas para usuarios, tanto de gustos más académicos como de un perfil más aplicado. No repetiremos aquí la descripción del texto que de forma esmerada y muy detallada realiza el autor en su prólogo; sin embargo, sí consideramos oportuno recalcar cuáles son los principales objetivos cubiertos por el mismo, tanto en el marco académico, como en el profesional. Comenzando por este último, el programador en Java, o aquél interesado en conocer las posibilidades de este lenguaje, en cuanto a estructuración de datos se refiere, encontrará en el texto una presentación detallada de la implementación y modo de aplicación de las estructuras de datos más utilizadas, así como de algunas otras más sofisticadas. Por su parte, los profesores y alumnos de disciplinas de Estructuras de Datos encontrarán en este texto una presentación de las mismas que hace hincapié en los principios de encapsulación y ocultamiento, hasta el extremo de discutir primero las aplicaciones de las estructuras estudiadas, partiendo de la descripción de las mismas, y dejar para más adelante el estudio de su implementación.

Como quiera que, en buena medida, el énfasis se pone en los detalles técnicos de las implementaciones y aplicaciones, el marco ideal de uso del texto sería el de un Laboratorio de desarrollo y uso de Estructuras de Datos; o bien el de una asignatura más general de estudio de dicha materia, soportada por la realización de prácticas sobre el computador. Es cierto que, en este último caso, si el profesor desea incluir un estudio teórico de los Tipos Abstractos de Datos, debería completar su bibliografía con un texto que cubra dicha materia.

Por último, remarcaremos el interés que tiene el soporte que el autor da a los lectores a través de la red, pues a través del mismo no sólo se pueden conseguir las versiones originales (en inglés) de todos los programas que aparecen en el texto, sino que se tiene acceso a eventuales versiones mejoradas y, en general, a diverso material de actualización y complemento del texto, sin tener que esperar a eventuales segundas ediciones revisadas. Al respecto, confiamos en que los lectores de esta versión traducida sepan aceptar como ineludible la consulta de dicho material

en su versión original, pues al enorme trabajo que supondría el mantenimiento actualizado de una versión traducida de dicha página electrónica, se une el hecho de que se trata de una página personal del autor. Por el contrario, al plantearnos la traducción del texto hemos considerado muy deseable, y por tanto así lo hemos hecho, traducir por completo el texto de los programas en el mismo. Con ello entendemos que se facilita enormemente la labor de comprensión de los mismos por parte de nuestros lectores, aun a pesar de la molestia que podría suponer el tener que manejar las versiones originales. Por otra parte, y sin que ello suponga asumir ninguna responsabilidad sobre posibles consecuencias de errores en las versiones traducidas, sí hemos hecho un enorme esfuerzo en este sentido, procurando que las versiones traducidas se comporten exactamente igual que las originales, siendo por tanto mutuamente reemplazables (lo que es de hecho un ejemplo de la doctrina de encapsulamiento y ocultación aplicada en el texto), haciendo por supuesto excepción de la necesidad de traducir los correspondientes identificadores que aparecen en las interfaces de las estructuras.

Por otra parte, y a pesar de que los programas originales estaban inicialmente concebidos para la versión 1.1 de Java, éstos han sido recopilados y ejecutados sin ningún problema en la nueva plataforma de Java 1.2, también llamada Java 2, gracias al esfuerzo del autor en usar lo máximo posible el corazón de Java. Por este último motivo es de esperar que los programas sigan siendo compatibles con versiones futuras de Java.

Los traductores y revisores técnicos
Madrid, marzo 2000

Parte I

Un recorrido por Java

CAPÍTULO 1

Java básico

El objetivo principal de este libro es estudiar técnicas de resolución de problemas que permitan la construcción de programas sofisticados y eficientes. Prácticamente todo el material utilizado se puede aplicar en cualquier lenguaje de programación. Se podría argumentar que para enseñar los conceptos basta con una descripción suficientemente amplia en pseudocódigo de estas técnicas. Sin embargo, creemos que es de vital importancia trabajar con código real.

No hay escasez de lenguajes de programación disponibles. En el momento en que se escribió este libro, C++ era el lenguaje más usado tanto académica como comercialmente. Sin embargo, en 1996, Java irrumpió en la escena como un contendiente viable.

La promoción publicitaria masiva sobre Java se debe en gran medida a su uso como lenguaje en el que se escriben applets. Los *applets* son programas Java ejecutados por un navegador de la World Wide Web, como *Netscape Navigator*. Actualmente, la mayoría de las páginas Web contienen animaciones, muchas de las cuales son applets de Java. Aunque los applets son fantásticos, ellos no son la razón por la que nosotros usamos Java. De hecho, en este libro solamente se estudian los applets en un breve apéndice.

Java proporciona también soporte para la programación concurrente, donde varios procesos se ejecutan en paralelo comunicándose entre sí de forma primitiva. Esta característica es importante en programación más avanzada, pero no se usa en este libro.

El principal atractivo de Java es que se trata de un lenguaje seguro y portable que soporta las construcciones de la programación moderna orientada a objetos. Muchas construcciones de C++ que confunden a los principiantes no existen en Java. Comparado con C++, Java detecta muchos de los errores comunes de programación, ya sea en tiempo de compilación o de ejecución. Java tiene un mecanismo de excepciones que obliga al programador a tratar de forma explícita los errores y un modelo relativamente simple que distingue entre tipos primitivos (como `int`) y tipos definidos por el usuario. Java carece de un tipo puntero explícito.

Java es portable: por ejemplo, un entero tiene el mismo rango de valores en todas las implementaciones de Java, independientemente de la arquitectura subyacente de la computadora. Java proporciona también una herramienta de interfaz gráfica de usuario (GUI) que permite que la entrada y la salida se lleven a cabo a

través de formularios. Aunque no estudiamos esta herramienta en el texto principal (*véase* Apéndice D), es bastante sencilla de utilizar. Lo más importante es que también es portable a cualquier implementación de Java. La filosofía de Java es «escribe una vez, ejecuta en todas partes».

En los primeros cuatro capítulos, estudiaremos las características de Java que se usan a lo largo del libro. Por el contrario, las características y aspectos técnicos que no se van a utilizar, no se cubren.

Quienes deseen una mayor información sobre Java, la encontrarán en los múltiples libros de Java disponibles.

Comenzamos discutiendo la parte del lenguaje semejante a la de un lenguaje de programación de los setenta como Pascal o C. Ésta incluye los tipos primitivos, operaciones básicas, construcciones condicionales y de repetición, y el equivalente en Java de las funciones.

En este capítulo, estudiaremos:

- Algunos conceptos básicos de Java, incluyendo los elementos léxicos simples.
- Los tipos primitivos de Java, incluyendo algunas de las operaciones que pueden llevarse a cabo con las variables de tipo primitivo.
- Cómo se implementan en Java las instrucciones condicionales y los bucles.
- El método estático; el equivalente en Java a la función y el procedimiento.

1.1 El entorno general

¿Cómo se escriben, se compilan y ejecutan los programas Java? La respuesta, por supuesto, depende de la plataforma particular sobre la que se encuentra instalado el compilador de Java. En el Apéndice A se describen algunos de los sistemas de desarrollo más populares.

El código fuente de Java se guarda en ficheros con nombres acabados en el sufijo `.java`. El compilador local, *javac*, compila el programa y genera ficheros `.class`, que contienen *código-j*. El código-j es un lenguaje portable intermedio que es interpretado por el intérprete de Java, *java*.

Para los programas de Java, la entrada puede provenir de muchas fuentes:

- El teclado, cuya entrada se denomina *entrada estándar*.
- Parámetros adicionales al invocar el programa ejecutable; *argumentos de la línea de comandos*.
- Una componente GUI.
- Un fichero.

En la Sección 2.4.4 se estudia los argumentos de la línea de comandos que son particularmente importantes para especificar opciones de programa. Java proporciona mecanismos para leer y escribir ficheros. Éstos se estudian en la Sección 2.6.3. Muchos sistemas operativos proporcionan una forma alternativa conocida como *redireccionamiento de fichero*, mediante la cual, el sistema operativo toma la entrada de (o envía la salida a) un fichero, de forma transparente al programa en ejecución. En Unix, por ejemplo, el comando

```
java Programa < ficheroentrada > ficherosalida
```

dispone automáticamente las cosas para que cualquier lectura del teclado sea redirigida para obtenerla de `ficheroentrada` y cualquier escritura en la pantalla sea redirigida sobre `ficherosalida`.

1.2 El primer programa

Comencemos examinando el sencillo programa Java de la Figura 1.1. Este programa imprime una frase corta en la pantalla. Observe que los números de línea mostrados a la izquierda del código *no son parte del programa*. Se proporcionan solamente para facilitar las referencias.

Guarde el programa en el fichero `PrimerPrograma.java` y después compílelo y ejecútelo. En el Apéndice A se describe cómo hacer esto en varias plataformas conocidas. Nótese que el nombre del fichero fuente debe coincidir con el nombre de la clase (mostrado en la línea 4).

1.2.1 Comentarios

Java admite tres formas de comentarios. La primera, heredada de C, comienza con el símbolo `/*` y termina con `*/`. He aquí un ejemplo:

```
/* Este es un comentario
   que ocupa dos líneas */
```

Los comentarios no pueden anidarse.

La segunda forma, heredada de C++, comienza con `//`. No hay símbolo de finalización, sino que el comentario se extiende hasta el final de la línea. Esta forma se ilustra en las líneas 1 y 2 de la Figura 1.1.

La tercera forma comienza con `/**` en lugar de `/*`. Esta forma se puede usar para proporcionar información a la utilidad *javadoc*, la cual generará documentación a partir de sus comentarios. Esta forma se estudia en la Sección 3.3.

Los comentarios aparecen para hacer el código más legible a las personas. Dichas personas pueden ser otros programadores que tengan que modificar o usar el código que se haya escrito. Un programa bien comentado es signo de un buen programador.

```
1  // Primer programa
2  // MW, 9/1/97
3
4  public class PrimerPrograma
5  {
6      public static void main( String [] args )
7      {
8          System.out.println( "¿Hay alguien ahí fuera?" );
9      }
10 }
```

Figura 1.1 Un primer programa simple.

1.2.2 main

Un programa Java está formado por una colección de clases que interactúan entre sí, las cuales contienen métodos. El equivalente en Java a una función o procedimiento es el *método estático*, que se describe en la Sección 1.6. Cuando se ejecuta un programa, se llama al método estático especial `main`. En la línea 6 de la Figura 1.1 se observa que se puede llamar al método estático `main` con argumentos en la línea de comandos. Es obligatorio que los tipos de los parámetros de `main` y el tipo del resultado (`void`) sean los mostrados en la figura.

1.2.3 Salida por pantalla

El programa de la Figura 1.1 consiste en una única instrucción, mostrada en la línea 8. `println` es el principal mecanismo de salida en Java. En dicha línea se coloca una cadena en la salida estándar `System.out` aplicando un método `println`. La entrada y la salida se estudian con más detalle en la Sección 2.6. Por ahora mencionamos solamente que se usa la misma sintaxis para producir la salida de cualquier entidad, ya sea un entero, un número en coma flotante, una cadena o un valor de cualquier otro tipo.

1.3 Tipos primitivos

Java tiene definidos ocho *tipos primitivos*. Además ofrece al programador una gran flexibilidad para definir nuevos tipos de objetos, llamados *clases*. Sin embargo, los tipos primitivos y los tipos definidos por el usuario poseen diferencias importantes en Java. En esta sección examinaremos los tipos primitivos y las operaciones básicas que se pueden realizar sobre ellos.

1.3.1 Los tipos primitivos

Java tiene ocho tipos primitivos, mostrados en la Figura 1.2. El más común es el entero, el cual se especifica mediante la palabra clave `int`. A diferencia de otros lenguajes, el rango de los enteros es independiente de la máquina. Es el mismo en cualquier implementación de Java, independientemente de la arquitectura subyacente. Java también permite entidades de tipo `byte`, `short` y `long`. Los números en coma flotante vienen representados por los tipos `float` y `double`. El tipo `double` tiene más dígitos significativos, por lo que se recomienda su uso en lugar de `float`. El tipo `char` se usa para representar caracteres. Un `char` utiliza 16 bits para representar el estándar Unicode. El estándar Unicode contiene más de 30.000 caracteres codificados, que cubren los principales lenguajes escritos. A bajo nivel, Unicode es idéntico a ASCII. El último tipo primitivo es `boolean`, cuyos valores son `true` y `false`.

Tipo primitivo	Lo que almacena	Rango
`byte`	entero de 8 bits	de −128 a 127
`short`	entero de 16 bits	de −32.768 a 32.767
`int`	entero de 32 bits	de −2.147.483.648 a 2.147.483.647
`long`	entero de 64 bits	de -2^{63} a $2^{63}-1$
`float`	número en coma flotante de 32 bits	6 dígitos significativos (10^{-46}, 10^{38})
`double`	número en coma flotante de 64 bits	15 dígitos significativos (10^{-324}, 10^{308})
`char`	Carácter Unicode	
`boolean`	Variable booleana	`false` y `true`

Figura 1.2 Los ocho tipos primitivos de Java.

1.3.2 Las constantes

Las *constantes enteras* se pueden representar en notación decimal, octal o hexadecimal. La notación octal se indica mediante un `0` delante del número; la notación hexadecimal se indica mediante `0x` o bien `0X` delante del número. Se muestran a continuación las distintas formas posibles de representar el entero 37: `37`, `045`, `0x25`. En este libro no usaremos la notación octal ni la hexadecimal. Sin embargo, debemos tenerlas en cuenta, de forma que usemos solamente un `0` delante de un número cuando pretendamos utilizarlas.

Una *constante de tipo carácter* se encierra entre dos comillas simples, como en `'a'`. Internamente este carácter se interpreta como un número pequeño. Las rutinas de salida interpretan posteriormente este número como el carácter correspondiente. *Una constante de tipo cadena* consiste en una secuencia de caracteres encerrados entre dobles comillas, como en `"Hola"`. Hay algunas secuencias especiales, conocidas como *secuencias de escape*, que se usan para representar ciertas constantes de tipo carácter (por ejemplo, ¿cómo se representa un salto de línea de interrogación?). En este libro usamos `'\n'`, `'\\'`, `'\''` y `'\"'`, que representan respectivamente, el carácter de nueva línea, el carácter de barra inclinada hacia atrás, la comilla simple y la comilla doble.

1.3.3 Declaración e inicialización de tipos primitivos

Cualquier variable, incluyendo las de tipo primitivo, se declara proporcionando su nombre, su tipo, y, opcionalmente, su valor inicial. El nombre debe ser un *identificador*. Un identificador está formado por cualquier combinación de letras, dígitos y el carácter de subrayado, si bien, no puede empezar con un dígito.

Las palabras reservadas, como `int`, no están permitidas. Ni tampoco se deben reutilizar nombres de identificadores que ya están visibles (por ejemplo, no usar `main` como nombre de una entidad).

Java es *sensible a las mayúsculas*, lo que quiere decir que `Edad` y `edad` son identificadores distintos. Este libro usa la siguiente convención para dar nombre a las variables: todos los identificadores comienzan con una letra minúscula y las palabras nuevas dentro de un identificador, con una letra mayúscula. Un ejemplo es el identificador `salarioMinimo`. A continuación se muestran algunos ejemplos de declaraciones:

```
int num3;                          // Inicialización por defecto
double salarioMinimo = 4.50;       // Inicialización estándar
int x = 0, num1 = 0;               // Declaración de dos variables
int num2 = num1;
```

Idealmente una variable debería estar declarada cerca de su primer uso. Como se mostrará más adelante, el lugar de una declaración determina su ámbito y significado.

1.3.4 Entrada y salida por terminal

La E/S básica por terminal con formato se lleva a cabo mediante `readLine` y `println`. La entrada estándar es `System.in`, y la salida estándar es `System.out`.

El mecanismo básico para E/S con formato utiliza el tipo `String`, que se estudia en la Sección 2.3. En la salida, `+` combina dos valores de tipo `String`. Si el segundo argumento no es un valor de tipo `String`, se crea un valor temporal de tipo `String` para él, siempre que sea de tipo primitivo. Estas conversiones al tipo `String` se pueden definir también para objetos (Sección 3.4.3). Para la entrada, debemos asociar un objeto `BufferedReader` a `System.in`. Entonces se lee un valor de tipo `String` que se puede analizar sintácticamente. En la Sección 2.6 puede encontrarse una discusión más detallada de la E/S, incluyendo el tratamiento de ficheros con formato.

1.4 Operadores básicos

Esta sección describe algunos de los operadores disponibles en Java. Estos operadores se usan para formar *expresiones*. Una constante o una variable es una expresión por sí misma, así como las combinaciones de constantes y variables mediante operadores. Una expresión seguida de un punto y coma es una instrucción simple. En la Sección 1.5, examinaremos otros tipos de instrucciones que introducen operadores adicionales.

1.4.1 Operadores de asignación

En la Figura 1.3 se muestra un programa Java sencillo que ilustra sobre el uso de algunos operadores. El operador básico de *asignación* es el signo de igualdad. Por ejemplo, en la línea 16 se asigna a la variable `a` el valor de la variable `c` (que en ese punto es 6). Cambios posteriores en la variable `c` no afectan a `a`. Los operadores de asignación se pueden encadenar, como en `z=y=x=0`.

```
1  public class TestOperador
2  {
3      // Programa de prueba de los operadores básicos.
4      // La salida es la siguiente:
5      // 12 8 6
6      // 6 8 6
7      // 6 8 14
8      // 22 8 14
9      // 24 10 33
10
11     public static void main( String [ ] args )
12     {
13         int a = 12, b = 8, c = 6;
14
15         System.out.println( a + " " + b + " " + c );
16         a = c;
17         System.out.println( a + " " + b + " " + c );
18         c += b;
19         System.out.println( a + " " + b + " " + c );
20         a = b + c;
21         System.out.println( a + " " + b + " " + c );
22         a++;
23         ++b;
24         c = a++ + ++b;
25         System.out.println( a + " " + b + " " + c );
26     }
27 }
```

Figura 1.3 Programa que ilustra el uso de los operadores.

Otro operador de asignación es `+=`, cuyo uso se muestra en la línea 18 de la figura. El operador `+=` añade el valor situado en el lado derecho (del operador de asignación `+=`) a la variable del lado izquierdo. Por tanto, en la figura, el valor de `c` pasa de ser 6, antes de la línea 18, a ser 14.

Java proporciona otros operadores de asignación como `-=`, `*=`, `/=`, que modifican el valor de la variable situada en el lado izquierdo del operador vía sustracción, multiplicación y división, respectivamente.

1.4.2 Operadores aritméticos binarios

En la línea 20 de la Figura 1.3 se muestra uno de los *operadores aritméticos binarios* típicos de todos los lenguajes de programación: el operador de suma (`+`). El operador `+` hace que los valores de `b` y `c` se sumen; `b` y `c` no sufren ningún cambio. El valor resultante se asigna a `a`. Otros operadores aritméticos típicos usados en Java son `-`, `*`, `/` y `%`, que se usan respectivamente para restar, multiplicar, dividir y obtener el resto de una división entera. La división entera devuelve solamente la parte entera y descarta el resto.

Como es habitual, la suma y la resta tienen la misma precedencia, y dicha precedencia es menor que la del grupo formado por los operadores de multiplicación, división y resto; en consecuencia `1+2*3` se evalúa a 7. Todos estos operadores asocian de izquierda a derecha (luego `3-2-2` se evalúa a –1). Todos los operadores tienen precedencia y asociatividad. La tabla completa de operadores se encuentra en el Apéndice B.

1.4.3 Operadores unarios

Además de los operadores aritméticos binarios, que requieren dos operandos, Java proporciona *operadores unarios*, que requieren un solo operando. El más familiar de estos operadores es el menos unario, que se utiliza para cambiar el signo del operando. Así, `-x` devuelve el valor de `x` cambiado de signo.

Java proporciona también el operador de autoincremento, para añadir 1 a una variable, denotado por `++`, y el operador de autodecremento, para restar 1 a una variable, denotado por `--`. El uso más benigno de esta característica se muestra en las líneas 22 y 23 de la Figura 1.3. En ambas líneas, el *operador de autoincremento* `++` añade 1 al valor de la variable. En Java, sin embargo, un operador aplicado a una expresión produce una expresión que tiene un valor. Aunque se garantiza que la variable será incrementada antes de la ejecución de la siguiente instrucción, surge una pregunta: ¿cuál es el valor de la expresión con autoincremento si se usa dentro de una expresión mayor?

En este caso, la posición de `++` tiene una importancia crucial. La semántica de `++x` es que el valor de la expresión es el nuevo valor de `x`. Esto recibe el nombre de *incremento prefijo*. Por el contrario, `x++` hace que el valor de la expresión sea el valor original de `x`. Esto recibe el nombre de *incremento postfijo*. Esta propiedad se muestra en la línea 24 de la Figura 1.3. Tanto `a` como `b` se incrementan en 1, mientras que `c` se obtiene sumando el valor original de `a` con el valor incrementado de `b`.

1.4.4 Conversiones de tipo

El *operador de conversión* se usa para generar una entidad temporal de un tipo nuevo. Considere, por ejemplo,

```
double cociente;
int x = 6;
int y = 10;
cociente = x / y;          // ¡Probablemente mal!
```

La primera operación es la división, y puesto que `x` e `y` son enteros, el resultado es su división entera, obteniéndose por tanto 0. El entero 0 se convierte entonces de forma implícita al tipo `double` para que pueda ser asignado a `cociente`. Pero nosotros pretendíamos asignar a `cociente` el valor 0,6. La solución es generar una variable temporal para `x` o para `y`, de forma que la división se lleve a cabo usando las reglas del tipo `double`. Esto se haría de la siguiente manera:

```
cociente = ( double ) x / y;
```

Observe que ni `x` ni `y` se modifican. Se crea una variable temporal sin nombre, cuyo valor se usa para efectuar la división. El operador de conversión de tipo tiene mayor precedencia que la división, luego la conversión de tipo se efectúa sobre el valor de `x` y después se lleva a cabo la división (en lugar de efectuarse la conversión tras la división de dos valores de tipo `int`).

1.5 Instrucciones condicionales

En esta sección se estudian instrucciones que afectan al flujo de control: instrucciones condicionales y bucles. Como consecuencia, se introducen nuevos operadores.

1.5.1 Operadores relacionales y de igualdad

El test básico que podemos realizar sobre los tipos primitivos es la comparación. Esta se lleva a cabo usando operadores de igualdad y desigualdad, así como los operadores relacionales (menor que, mayor que, etc.).

En Java, los operadores de igualdad son `==` y `!=`. Por ejemplo,

```
exprIzda == exprDcha
```

se evalúa a `true` si `exprIzda` y `exprDcha` son iguales; en caso contrario, se evalúa a `false`. Análogamente,

```
exprIzda != exprDcha
```

se evalúa a `true` si `exprIzda` y `exprDcha` son distintas; en caso contrario, se evalúa a `false`.

Los *operadores relacionales* son `<`, `<=`, `>` y `>=`. Tienen un significado natural para los tipos predefinidos. Los operadores relacionales tienen mayor precedencia que los operadores de igualdad. Ambos tienen menor precedencia que los operadores aritméticos pero mayor precedencia que los operadores de asignación, de modo que el uso de paréntesis no suele ser necesario. Todos estos operadores asocian de izquierda a derecha, pero este hecho no es de gran utilidad: en la expresión `a < b < 6`, por ejemplo, el primer `<` genera un valor de tipo `boolean` y el segundo es ilegal porque `<` no está definido para los valores de ese tipo. En la siguiente sección se describe la forma correcta de llevar a cabo esta comparación.

1.5.2 Operadores lógicos

Java proporciona *operadores lógicos* que se usan para simular los conceptos Y, O y NO del álgebra de Boole. Éstos se conocen con el nombre de *conjunción*, *disyunción* y *negación* respectivamente, y los operadores correspondientes son `&&`, `||` y `!`. El test de la sección anterior se implementa adecuadamente como `a < b && b < 6`. La precedencia de la conjunción y la disyunción es lo suficientemente baja como para que no sea necesario utilizar paréntesis. El operador `&&` tiene mayor precedencia que el operador `||`, mientras que `!` se agrupa con los demás operadores unarios. Los argumentos y resultados de estos operadores lógicos son valores de tipo `boolean`. En la Figura 1.4 se muestran los resultados de aplicar los operadores lógicos para todos los posibles valores de entrada.

X	Y	X && Y	X || Y	!X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Figura 1.4 Resultado de los operadores lógicos.

Una regla importante es que `&&` y `||` son operadores con evaluación de ciclo corto. La *evaluación de ciclo corto* consiste en que cuando el resultado de un operador lógico se puede determinar examinando la primera expresión, entonces no se evalúa la segunda. Por ejemplo, en

```
x != 0 && 1 / x != 3
```

si `x` es 0, entonces la primera mitad es `false`. Automáticamente el resultado del Y debe ser `false`, por lo que no se evalúa la segunda mitad. Esto es acertado pues la división por cero produciría un comportamiento erróneo. La evaluación de ciclo corto nos evita tener que preocuparnos por la división por cero[1].

1.5.3 La instrucción `if`

La instrucción `if` es la principal herramienta para tomar decisiones. Su forma básica es

```
if ( expresión )
    instrucción
siguiente instrucción
```

Si *`expresión`* toma el valor `true`, entonces se ejecuta *`instrucción`*; en caso contrario, no se ejecuta. Cuando se completa la ejecución de la instrucción `if` (sin ningún error sin tratar), el control pasa a la siguiente instrucción.

Opcionalmente, se puede usar una instrucción `if-else`, como la siguiente:

```
if( expresión )
    instrucción1
else
    instrucción2
siguiente instrucción
```

[1] ...damente) raros en los que es preferible no usar la evaluación de ciclo corto. En tales casos, los operadores `&` ...po `boolean`, garantizan que ambos argumentos se evalúan, incluso aunque el resultado de la operación pueda ... primer argumento.

En este caso, si *`expresión`* se evalúa a `true`, entonces se ejecuta *`instrucción1`*; en caso contrario se ejecuta *`instrucción2`*. En cualquier caso, el control pasa después a la siguiente instrucción, como en

```
System.out.print ( "1/x es " );
if( x != 0 )
    System.out.print( 1 / x );
else
    System.out.print( "Indefinido" );
System.out.println( );
```

Recuerde que se permite a lo sumo una instrucción en cada una de las cláusulas `if` y `else`, independientemente del sangrado. A continuación se puede observar dos errores:

```
if( x == 0 );     // ; es la instrucción vacía (y cuenta)
    System.out.print( "x es 0 " );
else
    System.out.print( "x es " );
    System.out.println( x ); // Dos instrucciones
```

El primer error es la inclusión del `;` al final del primer `if`. Este punto y coma cuenta como la *instrucción vacía*; como consecuencia, este fragmento no podrá compilarse (ya no hay ningún `if` con el que se asocie `else`). Una vez arreglado ese error, tenemos un error lógico: la última línea no es parte del `else`, incluso aunque el sangrado lo sugiera. Para arreglar este problema, tenemos que usar un *bloque*, en el cual encerramos una secuencia de instrucciones entre un par de llaves:

```
if( x == 0)
    System.out.println( "x es cero" );
else
{
    System.out.print( "x es " );
    System.out.println( x );
}
```

La instrucción `if` puede formar parte también de una cláusula `if` o `else`, al igual que otras instrucciones de control que estudiaremos más adelante en este capítulo. En el caso de instrucciones `if-else` anidadas, la cláusula `else` está asociada con la cláusula `if` más interna que esté sin emparejar. Será necesario añadir llaves en el caso de que no sea ése el significado pretendido.

1.5.4 La instrucción `while`

Java proporciona tres formas básicas de bucle: la instrucción `while`, la instrucción `for` y la instrucción `do`. La sintaxis de la *instrucción `while`* es la siguiente

```
while( expresión )
    instrucción
siguiente instrucción
```

Observe que al igual que en la instrucción `if`, no hay punto y coma en la sintaxis. Si está presente, se interpreta como la instrucción vacía.

Mientras que *expresión* tiene el valor `true`, se ejecuta *instrucción*; en consecuencia, *expresión* está continuamente reevaluándose. Si expresión vale inicialmente `false`, *instrucción* no se ejecutará nunca. En general, *instrucción* lleva a cabo alguna acción que potencialmente puede alterar el valor de *expresión*; en caso contrario, el bucle podría ser infinito. Cuando el bucle `while` termina (con normalidad), el control pasa a la siguiente instrucción.

1.5.5 La instrucción `for`

La instrucción `while` es suficiente para expresar todo tipo de repetición. Aun así, Java proporciona otras dos formas de bucle: la instrucción `for` y la instrucción `do`. La *instrucción* `for` se usa principalmente para la iteración simple. Su sintaxis es

```
for( inicialización; test; actualización )
    instrucción
siguiente instrucción
```

Aquí, *inicialización*, *test* y *actualización* son expresiones, siendo las tres opcionales. El valor por defecto de *test*, cuando no aparece, es `true`. No hay punto y coma tras el paréntesis cerrado.

Al ejecutar la instrucción `for`, lo primero que se lleva a cabo es la *inicialización*. Entonces, mientras *test* valga `true`, se llevan a cabo las siguientes acciones: se ejecuta *instrucción*, y se produce la *actualización*. Si se omiten *inicialización* y *actualización*, entonces la instrucción `for` se comporta exactamente igual a una instrucción `while`. La ventaja de la instrucción `for` es la claridad en el tratamiento de las variables contador (o iteradoras), ya que la instrucción `for` permite ver fácilmente cuál es el rango del contador. El siguiente fragmento produce los 100 primeros enteros positivos:

```
for( int i = 1; i <= 100; i++ )
    System.out.println( i );
```

Este fragmento ilustra la técnica habitual de declarar un contador en la parte de inicialización del bucle. El ámbito del contador es solamente el interior del bucle.

Tanto *inicialización* como *actualización* pueden usar una coma para permitir varias expresiones. El siguiente fragmento ilustra esta técnica habitualmente utilizada:

```
for( i = 0, suma = 0; i <= n; i++, suma += n )
    System.out.println( i + "\t" + suma );
```

Los bucles se anidan de la misma forma que las instrucciones `if`. Por ejemplo, podemos encontrar todos los números pequeños cuya suma es igual a su producto (como 2 y 2, cuya suma y producto es 4):

```
for( int i = 1; i <= 10; i++ )
    for( int j = 1; j <= 10; j++ )
        if( i + j == i * j )
            System.out.println( i + ", " + j );
```

Sin embargo, como veremos más adelante, cuando anidamos bucles podemos crear fácilmente programas cuyos tiempos de ejecución crecen rápidamente.

1.5.6 La instrucción `do`

La instrucción `while` lleva a cabo un test repetidamente. Si el test se evalúa a `true`, entonces se ejecuta el cuerpo del bucle. Sin embargo, si el test tiene inicialmente el valor `false`, el cuerpo del bucle no se ejecuta nunca. Ahora bien, en algunos casos nos gustaría garantizar que el cuerpo del bucle se ejecuta al menos una vez. Esto se puede conseguir usando la instrucción `do`. La *instrucción* `do` es idéntica a la instrucción `while`, excepto por el hecho de que el test se realiza tras la ejecución del cuerpo del bucle. La sintaxis es

```
do
    instrucción
while( expresión );
siguiente instrucción
```

Observe que la instrucción `do` incluye un punto y coma. En el siguiente fragmento en pseudocódigo se muestra un uso típico de la instrucción `do`:

```
do
{
    Pedir dato al usuario;
    Leer valor;
} while( el valor no es bueno );
```

La instrucción `do` es la menos usada de las tres formas de bucle. Sin embargo, cuando tenemos que hacer algo al menos una vez, y por alguna razón un bucle `for` no es lo más adecuado, entonces la instrucción `do` se convierte en nuestra elección.

1.5.7 `break` y `continue`

Las instrucciones `for` y `while` ofrecen una oportunidad de terminación antes de volver a ejecutar el cuerpo del bucle. La instrucción `do` ofrece una oportunidad de terminación tras la ejecución del cuerpo. Ocasionalmente, podríamos querer terminar la ejecución en mitad del cuerpo (instrucción compuesta) del bucle. La *instrucción* `break`, formada por la palabra reservada `break` seguida de un punto y coma, se puede usar para conseguirlo. Normalmente una instrucción `if` precede a la instrucción `break` como en

```
while( ... )
{
   ...
   if( algo )
      break;
   ...
}
```

La instrucción `break` provoca la salida del bucle más interno en el que está contenida (también se usa en conjunción con la instrucción `switch`, que se estudia en la siguiente sección). Si hay varios bucles de los que deseamos salir, la instrucción `break` no funcionará adecuadamente; además, lo más probable en tal caso es que su programa esté pobremente diseñado. Aun así, Java proporciona una instrucción *`break` etiquetada*. Para utilizar dicha instrucción, se comienza por poner una etiqueta a un bucle, de forma que la instrucción `break` se pueda aplicar a ese bucle, independientemente de cuantos bucles anidados haya. A continuación se muestra un ejemplo:

```
exterior:
  while( ... )
  {
    while( ... )
        if( desastre )
            break exterior; // Salta hasta después de exterior
  }
  // El control salta hasta aquí al salir del bucle exterior
```

A veces, queremos abandonar la ejecución del cuerpo del bucle en la iteración actual y saltar a la siguiente iteración. Esto se puede realizar con la *instrucción `continue`*. Al igual que la instrucción `break`, la instrucción `continue` incluye un punto y coma y se aplica solamente al bucle más interno. El siguiente fragmento produce los 100 primeros enteros, con la excepción de los divisibles por 10:

```
for( int i = 1; i <= 100; i++ )
{
   if( i % 10 == 0 )
      continue;
   System.out.println( i );
}
```

Por supuesto, en este ejemplo hay otras alternativas al uso de la instrucción `continue`. No obstante, `continue` se usa a menudo para evitar patrones `if-else` complicados dentro de los bucles.

1.5.8 La instrucción `switch`

La *instrucción `switch`* se usa para elegir entre varios valores enteros pequeños. Consiste en una expresión y un bloque. El bloque contiene una secuencia de instrucciones y una colección de *etiquetas*, que representan posibles valores de la expresión. Todas las etiquetas deben ser distintas. Una etiqueta opcional por defecto encaja con cualquier valor no representado por las demás. Si no hay ningún caso aplicable al valor de la expresión `switch`, la instrucción `switch` finaliza su ejecución; en caso contrario, el control pasa a la etiqueta apropiada y se ejecutan todas las instrucciones desde ese punto en adelante. Se puede usar una instrucción `break` para forzar la terminación de la instrucción `switch` utilizándose casi siempre para separar lógicamente los distintos casos. En la Figura 1.5 se muestra un ejemplo de esta estructura típica.

```
1  switch( algunCaracter )
2  {
3    case '(':
4    case '[':
5    case '{':
6      // Código para procesar los símbolos de apertura
7      break;
8
9    case ')':
10   case ']':
11   case '}':
12     // Código para procesar los símbolos de cierre
13     break;
14
15   case '\n':
16     // Código para tratar el carácter de fin de línea
17     break;
18
19   default:
20     // Código para tratar otros casos
21     break;
22 }
```

Figura 1.5 Ejemplo de una instrucción `switch`.

1.5.9 El operador condicional

El *operador condicional* `?:` se usa como abreviatura de una instrucción `if-else` simple. La forma general es

```
exprTest ? exprSi : exprNo
```

Primero se evalúa *`exprTest`*, seguida por *`exprSi`* o *`exprNo`*, produciendo así el resultado de la expresión completa. Si *`exprTest`* se evalúa a `true` entonces se evalúa *`exprSi`*; en caso contrario, se evalúa *`exprNo`*. La precedencia del operador condicional se encuentra justo por encima de la de los operadores de asignación. Esto nos evita tener que usar paréntesis cuando asignamos a una variable el resultado de un operador condicional. Como ejemplo, para asignar el mínimo entre `x` e `y` a la variable `valMin` haríamos lo siguiente:

```
valMin = x <= y ? x : y;
```

1.6 Métodos

Lo que en otros lenguajes se conoce como función o procedimiento, en Java se llama *método*. En el Capítulo 3 se proporciona un tratamiento más completo de los métodos. Esta sección presenta algunos conceptos básicos para escribir funciones del estilo de C, como `main`, de forma que podamos escribir algunos programas simples.

La *cabecera de un método* consiste en un nombre, una lista (posiblemente vacía) de parámetros y el tipo del resultado. El código que realmente implementa el método, a veces llamado el cuerpo del método, es formalmente un bloque. La *declaración de un método* está formada por una cabecera y un cuerpo. En la Figura 1.6 se muestra un ejemplo de un método y una rutina `main` que lo invoca.

Colocando delante de un método las palabras `public static`, podemos imitar una función global del estilo de C. Aunque en algunos ejemplos es una técnica útil, no debe sobreutilizarse.

El nombre del método es un identificador. La lista de parámetros consiste en cero o más *parámetros formales*, cada uno de ellos con un tipo. Cuando se llama a un método, los *argumentos reales* se pasan a los parámetros formales usando asignación normal. Esto significa que los tipos primitivos se pasan siempre usando *paso de parámetros por valor*. Los argumentos reales no pueden ser modificados por la función. Como en la mayoría de los lenguajes de programación modernos, las declaraciones de los métodos pueden aparecer en cualquier orden.

```
1  public class MinTest
2  {
3      public static void main( String [ ] args )
4      {
5          int a = 3;
6          int b = 7;
7
8          System.out.println( min( a, b ) );
9      }
10
11     // Definición de función
12     public static int min( int x, int y )
13     {
14         return x < y ? x : y;
15     }
16 }
```

Figura 1.6 Declaración y llamada de un método.

La *instrucción* `return` se usa para devolver un valor al punto de llamada. Si el tipo del resultado es `void`, entonces no se devuelve ningún valor, escribiéndose `return;` en tal caso.

1.6.1 Sobrecarga de los nombres de los métodos

Supongamos que necesitamos escribir una rutina que devuelva el máximo de tres valores de tipo `int`. Una cabecera razonable para el método sería

```
int max( int a, int b, int c )
```

En algunos lenguajes, esto puede ser inaceptable si `max` se ha declarado previamente. Por ejemplo, podríamos tener también

```
int max( int a, int b )
```

Java permite la *sobrecarga* de nombres de los métodos. Esto significa que varios métodos pueden tener el mismo nombre y declararse en el mismo ámbito de clase mientras sus *signaturas* (es decir, los tipos de la lista de parámetros) sean diferentes. Cuando se produce una llamada a `max`, el compilador puede deducir cuál es el método que se pretende aplicar en base a la lista de argumentos reales. Dos signaturas pueden tener el mismo número de parámetros, siempre que el tipo de alguno de ellos sea distinto.

Observe que el tipo del resultado no está incluido en la signatura. Esto significa que no está permitido tener dos métodos dentro del mismo ámbito de clase cuya única diferencia sea el tipo del resultado. Métodos que se encuentren en ámbitos de clase distintos pueden tener el mismo nombre, signatura e incluso tipo del resultado; esto se estudia en el Capítulo 3.

1.6.2 Clases de almacenamiento

Las entidades declaradas dentro del cuerpo de un método son variables locales y solamente se puede acceder a ellas a través de su nombre, dentro del cuerpo del método. Dichas entidades se crean cuando se ejecuta el cuerpo del método y desaparecen cuando el cuerpo del método termina.

Una variable declarada fuera del cuerpo de un método es global a la clase. Es similar a las variables globales en otros lenguajes si se usa la palabra `static` (lo cual es bastante probable si se desea permitir el acceso a la entidad por parte de métodos estáticos). Si se usan las palabras `static` y `final`, se trata de constantes simbólicas globales. Por ejemplo,

```
static final double PI = 3.1415926535897932;
```

Observe la convención habitual de usar nombres en mayúsculas para los nombres de constantes simbólicas. Si el nombre está formado por varias palabras, se separan mediante el carácter de subrayado, como en `MAX_INT_VALUE`.

Si se omite la palabra `static`, entonces la variable (o constante) tiene un significado diferente, que se discute en la Sección 3.4.5.

Resumen

En este capítulo se han estudiado las características básicas de Java, tales como los tipos primitivos, los operadores, las instrucciones condicionales, los bucles y los métodos, las cuales se encuentran prácticamente en cualquier lenguaje.

Cualquier programa no trivial requerirá el uso de tipos no primitivos, llamados *tipos referencia*, de los que se habla en el siguiente capítulo.

Elementos del juego

bloque Secuencia de instrucciones entre llaves.
cabecera de un método Consiste en el nombre, tipo del resultado y lista de parámetros.
código-j Código intermedio portable generado por el compilador de Java.
comentarios Hacen el código más legible pero no tienen significado semántico. Java admite tres formas de comentarios.
constante de tipo cadena Constante consistente en una secuencia de caracteres encerrados entre dobles comillas.
constantes enteras en octal y hexadecimal Las constantes enteras se pueden representar en notación decimal, octal o hexadecimal. La notación octal se indica añadiendo un `0` delante del número; la hexadecimal se indica añadiendo `0x` o `0X` delante del número.
declaración de un método Consiste en una cabecera y un cuerpo.
entidad `static final` Constante global.
entrada estándar Se trata del teclado, a menos que se redirija. También hay canales para la salida estándar y el error estándar.
evaluación de ciclo corto Proceso por el cual, si el resultado de un operador lógico se puede determinar a partir del primer operando, entonces no se evalúa el segundo operando.
identificador Usado para nombrar una variable o un método.
instrucción `break` Instrucción que provoca la salida del bucle más interno o de la instrucción `switch` más interna.
instrucción `break` etiquetada Instrucción `break` usada para salir de bucles anidados.
instrucción `continue` Instrucción que salta a la siguiente iteración del bucle más interno.
instrucción `do` Construcción de bucle que garantiza que el cuerpo del bucle se ejecuta al menos una vez.
instrucción `for` Construcción de bucle usada principalmente para la iteración simple.
instrucción `if` La construcción principal para tomar decisiones.
instrucción `return` Instrucción usada para devolver información al punto de llamada.
instrucción `switch` Instrucción usada para elegir entre varios enteros pequeños.
instrucción vacía Instrucción que consiste en un punto y coma.
instrucción `while` La forma más básica de bucle.
java Intérprete de Java.
javac Compilador de Java; genera código-j.
`main` Función especial que se invoca al ejecutarse el programa.
método El equivalente en Java a una función.
método estático Método equivalente a una función global.
operador condicional (`?:`) Operador usado como abreviatura de instrucciones `if-else` simples.
operador de conversión de tipo Operador usado para generar una variable temporal sin nombre de un tipo nuevo.
operadores aritméticos binarios Se usan para realizar operaciones aritméticas básicas. Java proporciona varios, entre ellos `+`, `-`, `*`, `/` y `%`.
operadores de asignación En Java se usan para modificar el valor de una variable. Estos operadores incluyen `=`, `+=`, `-=`, `*=` y `/=`.
operadores de autoincremento (`++`) y autodecremento (`--`) Operadores que suman y restan 1, respectivamente. Hay dos formas de incrementar y decrementar: prefija y postfija.
operadores de igualdad En Java, `==` y `!=` se usan para comparar dos valores; devuelven `true` o `false` (según corresponda).
operadores lógicos Son `&&`, `||` y `!`, y se usan para simular los conceptos Y, O y NO del álgebra de Boole.
operadores relacionales En Java son `<`, `<=`, `>` y `>=`, y se usan para decidir cuál es el mayor o el menor de dos valores; devuelven `true` o `false`.
operadores unarios Requieren un operando. Se definen diversos operadores unarios, entre los que se encuentran el menos unario (`-`) y los operadores de autoincremento y autodecremento (`++` y `--`).
paso de parámetros por valor Mecanismo de paso de parámetros de Java en el que los argumentos reales se copian en los parámetros formales.
secuencia de escape Se usa para representar algunas constantes de tipo carácter.
signatura Combinación del nombre del método y los tipos de la lista de parámetros. El tipo del resultado no forma parte de la signatura.
sobrecarga del nombre de un método Consiste en permitir a varios métodos tener el mismo nombre mientras los tipos de la lista de parámetros sean distintos.
tipos enteros `byte`, `char`, `short`, `int` y `long`.
tipos primitivos En Java son los enteros, números en coma flotante, booleanos y caracteres.
Unicode Conjunto internacional de caracteres que contiene más de 30.000 caracteres distintos, los cuales cubren los principales lenguajes escritos.

Errores comunes

1. Añadir un punto y coma innecesario produce errores lógicos, pues el punto y coma representa la instrucción vacía. Esto significa que un punto y coma a continuación de una instrucción `for`, `while` o `if` puede pasar inadvertido y estropear el programa.
2. En tiempo de compilación, Java es capaz de detectar algunos casos en los que un método que se supone que devuelve un valor, falla al hacerlo. Pero en último lugar, es su responsabilidad el recordar devolver un valor.
3. Un `0` delante de un número convierte una constante entera en octal; luego `037` es equivalente al decimal `31`.
4. Usar `&&` y `||` para las operaciones lógicas; `&` y `|` no evalúan en ciclo corto.
5. La cláusula `else` se asocia a la cláusula `if` más interna sin emparejar. Es habitual olvidar incluir las llaves necesarias para asociar un `else` con un `if` alejado.
6. Cuando se usa una instrucción `switch`, es habitual olvidar la instrucción `break` entre los casos lógicos. Cuando esto sucede, el control pasa al siguiente caso; lo que, en general, no es el comportamiento deseado.
7. Las secuencias de escape comienzan con la barra inclinada hacia atrás `\`, no con la inclinada hacia delante `/`.

8. Llaves sin emparejar pueden dar lugar a respuestas erróneas o engañosas. Usar `comprobarEquilibrados`, descrito en la Sección 11.1, para comprobar si ésta es la causa de un mensaje de error del compilador.
9. El nombre del fichero que contiene el código fuente del programa en Java debe ser igual que el nombre de la clase que se está compilando.

En Internet

Indicamos a continuación los ficheros disponibles correspondientes a este capítulo. Todos ellos son autocontenidos, y no serán posteriormente utilizados en el texto. Los programas se encuentran en el directorio **Chapter01**.

FirstProgram.java	Nuestro primer programa, que aparece traducido en la Figura 1.1.
MinTest.java	Para ilustrar el uso de los métodos, que aparece traducido en la Figura 1.6.
OperatorTest.java	Demostración del uso de varios operadores. Aparece traducido en la Figura 1.3.

Ejercicios

Cuestiones breves

1.1. ¿Qué extensiones se usan para los ficheros fuente y compilados en Java?
1.2. Describa los tres tipos de comentarios usados en Java.
1.3. ¿Cuáles son los ocho tipos primitivos en Java?
1.4. ¿Cuál es la diferencia entre los operadores `*` y `*=`?
1.5. Explique las diferencias entre los operadores de incremento prefijo y postfijo.
1.6. Describa los tres tipos de bucle en Java.
1.7. Describa todos los usos de una instrucción `break`.
1.8. ¿Qué hace la instrucción `do`?
1.9. ¿Qué es la sobrecarga de métodos?
1.10. Describa el paso de parámetros por valor.

Problemas teóricos

1.11. Supongamos que `b` tiene el valor 5 y `c` el valor 8. ¿Cuál es el valor de `a`, `b` y `c` después de la ejecución de cada una de las líneas del siguiente fragmento de programa?

```
a = b++ + c++;
a = b++ + ++c;
a = ++b + c++;
a = ++b + ++c;
```

1.12. ¿Cuál es el resultado de `true && false || true`?

1.13. Busque un ejemplo en el que el bucle `for` de la izquierda no sea equivalente al bucle `while` de la derecha:

```
for( inic; test; actual)
{
    instrucciones
}
```

```
inic;
while( test )
{
    instrucciones
    actual;
}
```

1.14. ¿Cuáles son las posibles salidas de este programa?

```
public class QueEsX
{
   public static void f(int x)
      { /* cuerpo desconocido */ }

   public static void main( String [ ] args )
   {
         int x = 0;
         f( x );
         System.out.println( x );
   }
}
```

Problemas prácticos

1.15. Escriba una instrucción `while` que sea equivalente al siguiente fragmento `for`. ¿Para qué podría ser esto útil?

```
for( ; ; )
    instrucción
```

1.16. Escriba un programa para generar las tablas de suma y multiplicación de números de un sólo dígito (las tablas que los estudiantes de la escuela elemental están acostumbrados a memorizar).
1.17. Presente dos métodos estáticos, el primero de los cuales debe devolver el máximo de tres enteros, mientras que el segundo devolverá el máximo de cuatro enteros.
1.18. Escriba un método estático que tome un año como parámetro y devuelva `true` cuando el año sea bisiesto.

Prácticas de programación

1.19. Escriba un programa que produzca todos los pares de enteros positivos, (a, b), tales que $a < b < 1000$ y $(a^2 + b^2 + 1)/(ab)$ sea entero.

1.20. Escriba un método que dado un número entero, muestre su representación en números romanos. En particular, si el dato es `1998`, la salida sería `MCMLXLVIII`.

1.21. Supongamos que queremos mostrar números entre corchetes, con el siguiente formato: `[1] [2] [3]`, etc. Escriba un método que tome dos parámetros: `cuantos` y `longLinea`. El método debe mostrar los números de línea desde 1 hasta `cuantos` en el formato antes mencionado, pero no debe mostrar más de `longLinea` caracteres en cada línea. Además, no debe empezar con el correspondiente a un nuevo número a menos que pueda escribir en la misma línea el correspondiente.

1.22. En el siguiente puzzle aritmético, se asigna un dígito a cada una de las diferentes letras. Escriba un programa que encuentre todas las soluciones posibles, a continuación mostramos una de ellas.

```
  MARK    A=1 W=2 N=3 R=4 E=5        9147
+ ALLEN   L=6 K=7 I=8 M=9 S=0      + 16653
 -----                             -----
 WEISS                             25800
```

Bibliografía

Una parte del material en el estilo de C usado en este capítulo se ha tomado de [5]. La especificación completa del lenguaje Java se puede encontrar en [4]. En [3] se listan algunos paquetes de librerías y se proporcionan ejemplos completos. Algunos libros introductorios de Java son [1] y [2][2].

1. G. Cornell y C. S. Horstmann, *Core Java*, 3.ª ed., Prentice-Hall, Englewood Cliffs, NJ (1998).
2. J. Lewis y W. Loftus, *Java Software Solutions: Foundations of Program Design,* Addison-Wesley, Reading, Mass. (1997).
3. D. Flanagan, *Java in a Nutshell*, 2.ª ed., O'Reilly and Associates, Sebastopol, Calif. (1997).
4. J. Gosling, B. Joy, y G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, Mass. (1996).
5. M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1995).

describe Java 1.0.2. En el momento en que se escribió la versión original de este libro, Java 1.1 ya se había incorpora algunos cambios significativos. En una posible segunda edición de este texto se utilizarían aquellas s del lenguaje que puedan ser de utilidad para mejorar la presentación en algunos puntos. Las referencias [1],

CAPÍTULO

2 Referencias

En el Capítulo 1 hemos estudiado los tipos primitivos de Java. Todos los tipos que no sean ninguno de los ocho tipos primitivos de Java son tipos referencia, incluyéndose en estos últimos entidades tan importantes como las cadenas de caracteres, los vectores y los canales de E/S.

En este capítulo veremos:

- Qué es un tipo referencia y qué es un valor.
- En qué se distinguen los tipos referencia de los tipos primitivos.
- Ejemplos de tipos referencia, incluyendo las cadenas de caracteres, los vectores y los canales de E/S.
- Cómo se emplean las excepciones para detectar comportamientos erróneos.

2.1 ¿Qué es una referencia?

El Capítulo 1 describe los ocho tipos primitivos, junto con algunas de las operaciones que se pueden realizar con sus elementos. Todos los demás tipos de Java son tipos referencia, incluyendo las cadenas de caracteres, los vectores y los ficheros. Pero, ¿qué es una referencia? Una *variable referencia* (o simplemente *referencia*) en Java es una variable que guarda la dirección de memoria en la que se almacena un objeto.

Por ejemplo, en la Figura 2.1 hay dos objetos de tipo `Punto`. Supongamos que estos objetos están almacenados en las posiciones de memoria 1000 y 1024, respectivamente. Para ambos existen tres referencias: `punto1`, `punto2` y `punto3`. `punto1` y `punto3` referencian al objeto situado en la posición 1000 y `punto2` referencia al objeto situado en la posición 1024. Observe que las direcciones de los objetos, como 1000 y 1024, son asignadas por el compilador de forma arbitraria (siempre que encuentre memoria disponible), por lo que estos valores no son útiles externamente como números. Sin embargo, el hecho de que `punto1` y `punto3` almacenen valores idénticos es útil: significa que están referenciando al mismo objeto.

Una referencia siempre contiene la dirección de memoria en la que un objeto está guardado, a menos que no referencie a ningún objeto. En este caso, almacena la *referencia nula*, `null`. Java no permite referencias a variables primitivas.

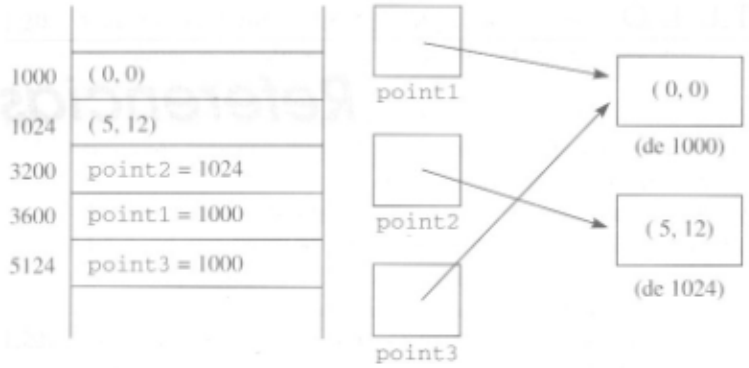


Figura 2.1 Representación gráfica de una referencia: el objeto de tipo `Punto` almacenado en la posición de memoria 1000 es referenciado por `punto1` y `punto3`. El objeto de tipo `Punto` almacenado en la posición de memoria 1024 es referenciado por `punto2`. Las posiciones de memoria donde se guardan las variables son arbitrarias.

Existen dos amplias categorías de operaciones que pueden aplicarse a variables referencia. Una de ellas nos permite examinar o manipular el valor referencia. Por ejemplo, si modificamos el valor almacenado en `punto1` (1000) pasaríamos a estar referenciando a otro objeto. También podemos comparar `punto1` y `punto3`, y determinar si están referenciando al mismo objeto. La otra categoría de operaciones se aplica al objeto referenciado; es posible examinar o cambiar el estado interno de uno de los objetos de tipo `Punto`. Por ejemplo, podemos averiguar las coordenadas *x* e *y* de algunos `Puntos`.

Antes de describir qué puede hacerse con referencias, vamos a ver lo que no está permitido. Considérese la expresión `punto1*punto2`. Como los valores almacenados por `punto1` y `punto2` son 1000 y 1024, respectivamente, su producto debería ser 1024000. Sin embargo, éste es un cálculo sin sentido que no tendría ninguna utilidad. Las variables referencia almacenan direcciones de memoria, y no puede asociarse ningún significado lógico al producto de dos direcciones de memoria.

Análogamente, `punto1++` no tiene sentido en Java; esta expresión sugiere que `punto1` —1000— debería incrementarse a 1001, pero en ese caso podría no referenciar a un objeto válido de tipo `Punto`. Muchos lenguajes de programación definen el *puntero*, que se comporta como una variable referencia. Por el contrario, los punteros de C++ son mucho más peligrosos, pues se permiten operaciones aritméticas sobre las direcciones de memoria almacenadas. Así, en C++, `punto1++` tiene sentido. Además, C++ permite punteros a tipos primitivos, por lo que se debe ser cuidadoso al distinguir entre cálculos con direcciones de memoria y cálculos con los objetos referenciados. Esto se hace explícitamente *desreferenciando* el puntero. En la práctica, los inseguros punteros de C++ causan numerosos errores de programación.

En Java los únicos operadores permitidos para manipular los tipos referencia (con una única excepción en el caso de los `Strings`) son las asignaciones vía =, y las comparaciones de igualdad, a través de == y !=.

La Figura 2.2 ilustra el operador de asignación para variables referencia. Asignando a `punto3` el valor almacenado en `punto2` se consigue que `punto3` referencie al mismo objeto que `punto2` estaba referenciando. Ahora `punto2==punto3` es `true` ya que `punto2` y `punto3` almacenan el valor 1024, y por lo tanto, referencian al mismo objeto. `punto1!=punto2` es también cierto ya que `punto1` y `punto2` referencian objetos distintos.

La otra categoría de operaciones maneja el objeto que está siendo referenciado. Sobre él sólo pueden realizarse tres acciones básicas:

1. Realizar una conversión de tipos (Sección 1.4.4).
2. Acceder a un campo interno o invocar a un método, empleando el operador punto (.) (Sección 2.2.1).
3. Emplear el operador `instanceof` para comprobar que el objeto almacenado es de un tipo determinado (Sección 3.6.3).

La siguiente sección muestra con más detalle las operaciones con referencias más comunes.

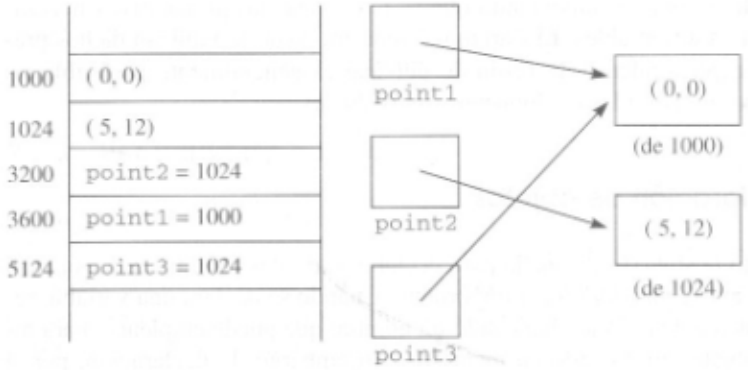


Figura 2.2 El resultado de `punto3=punto2`: ahora `punto3` referencia el mismo objeto que `punto2`.

2.2 Nociones básicas sobre objetos y referencias

En Java, un *objeto* es un elemento de cualquiera de los tipos no primitivos. Los objetos son tratados de forma diferente a los valores de los tipos primitivos. Los tipos primitivos, como hemos visto, se manipulan por *valor*, es decir, los valores supuestos para las variables primitivas se almacenan en dichas variables y son copiados de unas a otras en las asignaciones. Como se ha explicado en la Sección 2.1, las variables referencia guardan referencias a objetos. El objeto actual se almacena en algún lugar de la memoria, y la variable referencia guarda la posición de memoria del objeto. Así, una variable referencia representa simplemente un nombre de una zona de la memoria. Esto significa que las variables primitivas y

las variables referencia se comportan de forma diferente. Esta sección estudia con más detalle dichas diferencias e ilustra las operaciones permitidas con variables referencia.

2.2.1 El operador punto (.)

El operador punto (.) se emplea para seleccionar un método y aplicarlo a un objeto. Por ejemplo, supongamos que se tiene un objeto de tipo `Circulo`, en el que se define el método `area`. Si `elCirculo` referencia a un `Circulo`, entonces podemos calcular el área del `Circulo` referenciado (y guardarla en una variable de tipo `double`) haciendo lo siguiente:

```
double elArea = elCirculo.area( );
```

Es posible que `elCirculo` contenga la referencia `null`. En este caso, el empleo del operador punto generará una `NullPointerException` cuando el programa se ejecute. Generalmente, esto causará la terminación anormal del programa.

El operador punto también puede emplearse para acceder a las componentes individuales de un objeto, suponiendo que se han hecho los preparativos necesarios para hacerlas observables. El Capítulo 3 muestra cómo se realizan dichos preparativos. Se explica además la razón de por qué es generalmente preferible no permitir acceso directo a las componentes individuales.

2.2.2 Declaración de objetos

Ya hemos visto la sintaxis adecuada para declarar variables primitivas. En el caso de los objetos existe una diferencia importante. Cuando se declara una variable referencia, simplemente estamos indicando un nombre que puede emplearse para referenciar un objeto almacenado en memoria. Sin embargo, la declaración, por sí misma, no genera ese objeto. Por ejemplo, supongamos que existe un objeto de tipo `Button` que deseamos añadir en un `Panel p` dado, empleando para ello el método `add` (todo ello definido en las librerías de Java). Para hacerlo utilizaríamos las instrucciones

```
Button b;              // b puede referenciar a un objeto de tipo Button
b.setLabel( "No" ); // La etiqueta del botón b es "No"
p.add( b );            // b se añade al panel p
```

Todo parece estar bien hasta que recordamos que `b` es el nombre de un objeto de tipo `Button`, pero que aún no se ha creado ninguno de ellos. Como resultado, después de la declaración de `b` el valor almacenado por la variable referencia `b` es `null`, indicando que `b` todavía no apunta a ningún objeto válido de tipo `Button`. Como consecuencia, la segunda línea es incorrecta ya que está intentando alterar un objeto inexistente. En este contexto, el compilador detectará el error, indicando que `b` no está inicializada. En otros casos, el compilador no detectará el fallo y se producirá un error durante la ejecución, obteniéndose el críptico mensaje `NullPointerException`.

Una forma de almacenar un objeto (la única común a todos los tipos referencia) es emplear la palabra reservada `new`. `new` se utiliza para construir un nuevo objeto. Un modo de hacerlo es el siguiente:

```
Button b;              // b puede referenciar a un objeto de tipo Button
b = new Button( );     // Ahora b referencia a un objeto creado
b.setLabel( "No" ); // La etiqueta del botón b es "No"
p.add( b );            // b se añade al panel p
```

Obsérvese que se necesitan los paréntesis tras el tipo del objeto.

También es posible combinar la declaración y la construcción de un objeto, como en

```
Button b = new Button( );
b.setLabel( "No" );  // La etiqueta de botón b es "No"
p.add( b );           // b se añade al panel p
```

Muchos objetos pueden construirse dándoles un valor inicial. Por ejemplo, el objeto de tipo `Button` puede construirse con un `String` que sea su etiqueta:

```
Button b = new Button( "No" );
p.add( b );            // b se añade al panel p
```

2.2.3 Recogida de basura

Como todos los objetos deben construirse, podríamos suponer que cuando uno ya no se vaya a emplear más, debemos destruirlo explícitamente. En Java, cuando un objeto creado no es referenciado por ninguna variable referencia, la memoria que consume es reclamada automáticamente y pasa a estar disponible. Esta técnica se conoce como *recogida de basura*.

2.2.4 El significado de =

Supongamos que se tienen dos variables primitivas `lizq` y `lder`, donde empleamos `lizq` y `lder` para denotar los *lados izquierdo* y *derecho*, respectivamente. Entonces, la instrucción de asignación

```
lizq = lder;
```

tiene un sencillo significado: el valor almacenado en `lder` es copiado en la variable primitiva `lizq`. Los cambios posteriores en alguna de las variables no afectan a la otra.

En los objetos, el significado de = es el mismo: los valores almacenados se copian. Si `lizq` y `lder` son referencias (de tipos compatibles), después de la instrucción de asignación, `lizq` señalará el mismo objeto al que `lder` apunta. Aquí lo que se copian son las direcciones. El objeto al que `lizq` referenciaba dejará de ser referenciado por esta variable. Si `lizq` era la única referencia a ese objeto,

entonces dicho objeto se encuentra ahora desreferenciado por lo que será incluido en la próxima recogida de basura. Observe que los objetos no se copian.

Veamos algunos ejemplos. En primer lugar, supongamos que se quiere tener dos objetos de tipo `Button`. Intentamos lograrlo comenzando por crear `noBoton`. Después creamos `siBoton` modificando `noBoton` del modo siguiente:

```
Button noBoton = new Button( "No" );
Button siBoton = noBoton;
siBoton.setLabel( "Si" );
p.add( noBoton );
p.add( siBoton );
```

Esto no funciona ya que sólo ha sido creado un objeto de tipo `Button`. De este modo, la segunda instrucción simplemente indica que `siBoton` es otro nombre para el botón creado en la primera línea. Dicho botón tiene ahora dos nombres. En la tercera línea, el objeto de tipo `Button` construido ha cambiado su etiqueta por `Si`, pero esto significa que el único objeto `Button`, identificado por dos nombres, se etiqueta ahora con `Si`. Las dos últimas líneas añaden al `Panel p` el objeto `Button` dos veces.

El hecho de que `siBoton` nunca consulte su propio objeto carece de importancia en este ejemplo. El problema es la asignación. Considere

```
Button noBoton = new Button( "No" );
Button siBoton = new Button( );
siBoton = noBoton;
siBoton.setLabel( "Si" );
p.add( noBoton );
p.add( siBoton );
```

Las consecuencias son exactamente las mismas. Ahora sí han sido generados dos objetos `Button`, pero al final de la secuencia el primer objeto está referenciado por `noBoton` y `siBoton`, mientras que el segundo objeto está sin referenciar.

A primera vista, el hecho de que los objetos no puedan copiarse parece una limitación importante. En realidad no lo es, aunque cuesta un poco acostumbrarse. (Para ser exactos algunos objetos sí necesitan ser copiados. Para ellos, si está disponible un método `clone`, debe usarse. `clone` invoca a `new` para obtener una réplica del objeto. Sin embargo, `clone` no se utiliza en este libro.)

2.2.5 Paso de parámetros

A causa del paso de parámetros por valor, los parámetros actuales se pasan en lugar de los parámetros formales empleando la asignación usual. Si el parámetro es un tipo referencia, entonces ya sabemos que la asignación significa que el parámetro formal ahora referencia al mismo objeto que el parámetro actual. Cualquier método aplicado al parámetro formal se aplica también al parámetro actual. En otros lenguajes esto se conoce como *paso de parámetros por referencia*. Emplear esta terminología en Java podría provocar equívocos, pues haría pensar que el paso de parámetros es diferente. En realidad el paso de parámetros no ha cambiado; en su lugar, son los parámetros los que han variado, pasando de corresponder a tipos no referenciales a corresponder a tipos referencia.

2.2.6 El significado de ==

En los tipos primitivos, `==` es verdadero cuando los valores almacenados son idénticos. En los tipos referencia su significado es diferente, aunque totalmente consistente con la discusión previa.

Dos variables referencia son iguales vía `==` si ambas apuntan al mismo objeto (o bien son las dos `null`). Consideremos, por ejemplo, la siguiente secuencia de asignaciones:

```
Button a = new Button( "Si" );
Button b = new Button( "Si" );
Button c = b;
```

Aquí se han generado dos objetos. El primero es accesible mediante el nombre `a`, mientras que el segundo se accede con cualquiera de los dos nombres, `b` y `c`. En consecuencia `b==c` es `true`. Sin embargo, aunque `a` y `b` estén referenciando objetos que parecen tener el mismo valor, `a==b` es `false`, ya que apuntan a objetos distintos. Para `!=` se aplican reglas similares.

En algunas ocasiones es importante conocer si los objetos referenciados son idénticos. Todos los objetos pueden compararse mediante `equals`, pero para muchos (incluyendo los de tipo `Button`) `equals` devuelve `false` a menos que ambas referencias estén apuntando al mismo objeto (en otras palabras, para algunos objetos `equals` no es otra cosa que el test `==`). En la Sección 2.3, donde se estudia el tipo `String`, veremos un ejemplo en el que `equals` resulta muy útil.

2.2.7 Sobrecarga de operadores para objetos

Salvo para la excepción descrita en la sección siguiente, los operadores no pueden definirse para ser aplicados sobre objetos. Así, no existe ningún operador `<` disponible sobre ningún tipo de objetos. En su lugar, deberíamos definir para realizar esta función un método con identificador, como por ejemplo, `menorQue`.

2.3 Cadenas de caracteres

Las cadenas de caracteres en Java se manipulan mediante el tipo `String`. El lenguaje hace aparecer a `String` como un tipo primitivo ya que define la `+` y el operador `+=` para la concatenación de cadenas. Pero éste es el único tipo referencia para el que la sobrecarga de operadores está permitida. Por lo demás, el tipo `String` se comporta como cualquier otro tipo no primitivo.

2.3.1 Conceptos básicos de la manipulación de `Strings`

Existen dos reglas fundamentales sobre el tipo `String`. En primer lugar, se comporta como cualquier otro tipo no primitivo, a excepción de lo que hace referencia a los operadores de concatenación. En segundo lugar, el tipo `String` es *inmutable*. Esto significa que, una vez que un objeto `String` se ha creado, su contenido no puede modificarse.

Como un `String` es inmutable, siempre es seguro emplear el operador `==` con él. Así, un `String` puede declararse de la forma siguiente:

```
String vacia      = "";
String mensaje    = "Hola";
String repeticion = mensaje;
```

Después de estas declaraciones, se ha creado dos objetos `String`. El primero de ellos es la cadena vacía, referenciada por `vacia`. El segundo es el `String "Hola"`, referenciado por `mensaje` y `repeticion`. En otros casos la doble referencia mediante `mensaje` y `repeticion` podría generar algún problema. Sin embargo, como los `Strings` son inmutables, la compartición de objetos de tipo `String` es segura a la vez que eficiente. El único modo de modificar el valor que `repeticion` está referenciando es crear un nuevo `String` y hacer que `repeticion` lo referencie. Esto no tiene efecto alguno sobre el `String` que referencia `mensaje`.

2.3.2 Concatenación de cadenas

Java no permite la sobrecarga de operadores para tipos referencia. Sin embargo, tenemos una excepción en la concatenación de cadenas.

El operador `+` realiza la concatenación cuando, al menos, uno de los operandos es un `String`. El resultado es una referencia al objeto de tipo `String` recién construido. Por ejemplo,

```
"este" + "aquel"    // Genera "esteaquel"
"abc" + 5           // Genera "abc5"
5 + "abc"           // Genera "5abc"
"a" + "b" + "c"     // Genera "abc"
```

Las cadenas de un solo carácter no deben reemplazarse por caracteres individuales; el Ejercicio 2.5 pide explicar el porqué. Observe que el operador `+` es asociativo por la izquierda, de modo que

```
"a" + 1 + 2       // Genera "a12"
1 + 2 + "a"       // Genera "3a"
1 + ( 2 + "a" )   // Genera "12a"
```

También el operador `+=` está definido sobre el tipo `String`. El efecto de `str+=exp` es el mismo que el de `str=str+exp`. Más concretamente, esto significa que `str` referenciará al mismo objeto de tipo `String` generado por `str+exp`.

2.3.3 Comparando cadenas

Como el operador básico de asignación funciona para los `Strings`, es lógico suponer que los operadores relacionales y de igualdad también funcionarán. Sin embargo, ello no es cierto.

De acuerdo con la prohibición de la sobrecarga de operadores, los operadores relacionales (`<`, `>`, `<=`, `>=`) no están definidos para el tipo `String`. Mas aún, `==` y `!=` tienen el significado usual para las variables referencia. Así, para dos objetos de tipo `String`, `lizq` y `lder`, `lizq==lder` es `true` sólo cuando ambos referencian al mismo objeto `String`. De modo que, si referencian dos objetos distintos con el mismo contenido, `lizq==lder` es `false`. Para `!=` valen razonamientos similares.

Para comparar la igualdad de dos objetos `String` empleamos `equals`. `lizq.equals(lder)` es `true` si `lizq` y `lder` referencian `Strings` que almacenan valores idénticos.

Una comprobación más general puede realizarse con el método `compareTo`. `lizq.compareTo(lder)` compara dos objetos `String`, `lizq` y `lder`. Se devuelve un número negativo, cero o un número positivo, según `lizq` sea menor, igual o mayor que `lder`, respectivamente.

2.3.4 Otros métodos del tipo `String`

La longitud de un objeto `String` (la cadena vacía tiene longitud cero) puede obtenerse con el método `length`. Como `length` es un método, es necesario emplear paréntesis.

Se tienen definidos dos métodos para acceder a los caracteres individuales de un `String`. El método `charAt` obtiene un solo carácter, especificando su posición (la primera posición es la posición 0). El método `substring` devuelve una referencia a un nuevo `String`. La llamada se realiza especificando la posición de inicio y la primera posición no incluida.

Veamos a continuación un ejemplo de estos tres métodos:

```
String saludo = "hola";
int lon = saludo.length( );              // lon es 4
char ch = saludo.charAt( 1 );            // ch es 'o'
String sub = saludo.substring( 2, 4 );   // sub es "la"
```

2.3.5 Conversión entre cadenas y tipos primitivos

El método `toString` puede emplearse para convertir cualquier valor de tipo primitivo en un objeto de tipo `String`. Por ejemplo, `toString(45)` devuelve una referencia al recién creado `String "45"`. Muchos tipos tienen implementado el método `toString`. De hecho, cuando el operador `+` tiene un solo argumento `String`, el argumento que no lo es se convierte automáticamente en una cadena aplicando `toString`. Para los tipos de enteros, una forma alternativa de `toString` permite la especificación de la base de la representación. Así

```
System.out.println( Integer.toString( 55,2 ) );
```

imprime la representación binaria de 55.

El valor de tipo `int` representado por un `String` puede obtenerse llamando al método `Integer.parseInt`. Este método genera una excepción si el `String` no representa un valor de tipo `int`. Las excepciones se estudian en la Sección 2.5. Para obtener un valor `double` a partir de un `String` es necesario más trabajo. Aquí se muestran algunos ejemplos:

```
int    x = Integer.parseInt( "75" );
double y = Double.valueOf( "3.14" ).doubleValue( );
```

2.4 Vectores

Un *agregado* es una colección de entidades almacenadas en una unidad. El *vector* es el mecanismo básico para almacenar una colección de entidades del mismo tipo. En Java el vector no es un tipo primitivo, pero se comporta de forma muy similar al resto de los objetos. Así, muchas de las reglas de los objetos pueden aplicarse también a los vectores.

Cada entidad en el vector puede ser accedida mediante el *operador de indexación de vectores* `[ ]`. Decimos que el operador `[ ]` *indexa* el vector en el sentido de que especifica qué objeto debe ser accedido. A diferencia de C y C++, el chequeo de los límites se realiza automáticamente.

En Java, los vectores se indexan comenzando siempre en cero. Así, un vector `a` de tres elementos almacena `a[0]`, `a[1]` y `a[2]`. El número de elementos que puede ser almacenado en un vector `a` se obtiene con `a.length`. Observe que en esta ocasión no hay paréntesis. Un bucle típico para recorrer un vector se basaría en

```
for ( int i = 0; i < a.length; i++ )
```

2.4.1 Declaración, asignación y métodos

Un vector es un objeto, así que dada una declaración de vector

```
int [ ] vector1;
```

aún no se ha asignado memoria para guardar el vector. `vector1` es simplemente una referencia de un vector, por lo que en este momento es `null`. Para generar 100 valores de tipo `int`, por ejemplo, aplicaríamos la instrucción `new` como sigue

```
vector1 = new int [ 100 ];
```

Ahora `vector1` referencia un vector de 100 `int`s.

Existen otras formas de declarar vectores. Por ejemplo, en algunos contextos

```
int [ ] vector2 = new int [ 100 ];
```

es aceptable. También puede emplearse listas de inicialización, como en C o C++, para especificar valores iniciales. En el siguiente ejemplo, un vector de cuatro `int`s es almacenado y referenciado por `vector3`.

```
int [ ] vector3 = { 3, 4, 10, 6 };
```

Los corchetes pueden colocarse antes o después del nombre del vector. Situándolos antes es más sencillo ver que el nombre corresponde a un objeto de tipo vector, por lo que éste es el estilo que emplearemos en este texto. La declaración de un vector de objetos (en lugar de tipos primitivos) requiere la misma sintaxis. Obsérvese, sin embargo, que cuando se guarda un vector de objetos, cada objeto almacena inicialmente una referencia `null`. Además cada una de ellas debe redefinirse para que pase a referenciar a un objeto creado. Por ejemplo, un vector de cinco botones se construye como sigue

```
Button [ ] vectorDeBotones;
vectorDeBotones = new Button [ 5 ];
for ( int i = 0; i < vectorDeBotones.length; i++ )
    vectorDeBotones[ i ] = new Button( );
```

La Figura 2.3 muestra el empleo de los vectores en Java, mediante su aplicación a la modelización del siguiente problema: en la lotería primitiva, cada semana se seleccionan seis números distintos entre 1 y 49 (ambos inclusive). El programa de la figura elige números repetidamente para 1.000 sorteos. La salida es la cantidad de veces que cada número ha sido escogido. La línea 15 declara un vector de enteros que lleva la cuenta de las apariciones de cada número. Como los vectores se indexan comenzando en el cero, el `+1` es crucial. Sin él, tendríamos un vector cuyo rango indexable se extendería desde el 0 hasta el 48, y como consecuencia el acceso al índice 49 estaría fuera de rango. El bucle de las líneas 16 y 17 inicializa las entradas del vector a cero. El resto del programa es relativamente sencillo. Se emplea el tipo `Random` definido en el Capítulo 9. Las llamadas al método `randomInt` producen repetidamente un número aleatorio dentro del rango especificado. La salida de los resultados se genera en las líneas 26 y 27.

Como el tipo vector es un tipo referencia, = no copia vectores. En cambio, si `lizq` y `lder` son vectores, el efecto de

```
int [ ] lizq = new int [ 100 ];
int [ ] lder = new int [ 100 ];
   ...
lizq = lder
```

es que el objeto vector referenciado por `lder` está ahora también referenciado por `lizq`. Como consecuencia, el cambio de `lizq[0]` también modifica `lder[0]`. (Para hacer que `lizq` sea una copia independiente de `lder` se debería emplear el método `clone`.)

Por último, un vector puede emplearse como parámetro de un método. El comportamiento que se produce se corresponde con el hecho de que un nombre de vector es una referencia. Consideramos un método `metodoLlamada` que tiene un vector de `int` como parámetro. Las vistas del invocador/invocado son

```
metodoLlamada( vectorActual );        // llamada al método
metodoLlamada( int [ ] vectorFormal ) // declaración del método
```

De acuerdo con los convenios sobre el paso de parámetros para los tipos referencia de Java, `vectorFormal` referencia al mismo objeto vector que `vectorActual`. Como consecuencia, `vectorFormal[i]` accede a `vectorActual[i]`.

```
1  import Soporte.Random;
2
3  public class Lotería
4  {
5     // Genera los números de la lotería (1-49)
6     // Imprime la cantidad de ocurrencias de cada número
7
8     public static final int NUMEROS_DIF    =   49;
9     public static final int NUMEROS_JUEGO  =    6;
10    public static final int JUEGOS         = 1000;
11
12    public static void main( String [ ] args )
13    {
14       // Generación de números
15       int [ ] numeros = new int [ NUMEROS_DIF + 1 ];
16       for( int i = 0; i < numeros.length; i++ )
17          numeros[ i ] = 0;
18
19       Random r = new Random( );
20
21       for( int i = 0; i < JUEGOS; i++ )
22          for( int j = 0; j < NUMEROS_JUEGO; j++ )
23             numeros[ r.randomInt( 1, NUMEROS_DIF ) ]++;
24
25       // Salida de resultados
26       for( int k = 1; k <= NUMEROS_DIF; k++ )
27          System.out.println( k + ": " + numeros[ k ] );
28    }
29 }
```

Figura 2.3 Una simple muestra de vectores.

Esto significa que las variables representadas por el vector indexado son modificables. Éste será siempre el caso. Nótese también que una instrucción como

```
vectorFormal = new int [ 20 ];
```

no tiene ningún efecto sobre `vectorActual`. Finalmente, como los nombres de los vectores no son otra cosa que referencias, pueden devolverse como resultados.

2.4.2 Expansión dinámica de vectores

Supongamos que queremos leer una secuencia de números y almacenarlos en un vector para su procesamiento. La propiedad básica de un vector es que debemos especificar su tamaño para que el compilador pueda reservarle la cantidad de memoria adecuada. Además debemos realizar su declaración antes del primer acceso al vector. Si no tenemos idea a priori de cuántos elementos deberemos manejar, es difícil realizar una elección razonable de dicho tamaño. Esta sección muestra cómo expandir vectores si su tamaño inicial resulta ser al final demasiado pequeño. Esta técnica recibe el nombre de *expansión dinámica de vectores* y nos permite almacenar en memoria vectores de cualquier tamaño, agrandándolos o empequeñeciéndolos durante la ejecución del programa.

El método de almacenar vectores en memoria que hemos visto anteriormente es

```
int [ ] a = new int [ 10 ];
```

Suponga que después de las declaraciones, observamos que necesitaremos 12 valores de tipo `int` en lugar de 10. En tal caso podemos emplear la siguiente estrategia:

```
int [ ] original = a;      // 1. Se guarda una referencia a a
a = new int [ 12 ];        // 2. Se crea una referencia con más memoria
for ( int i = 0; i < 10; i++ )  // 3. Se copian los datos antiguos
    a[ i ] = original[ i ];
```

Examinando detenidamente el código anterior llegamos a la conclusión de que ésta es una operación costosa. Esto es debido a que se copian todos los elementos de `original` en `a`. Si, por ejemplo, esta expansión se produce como respuesta a una entrada de datos, sería ineficiente aumentar el vector cada vez que se lean unos pocos datos. Por esta razón, cuando se implementa la expansión de vectores, lo hacemos de modo que el tamaño aumente según una constante multiplicativa. Por ejemplo, en cada paso podemos expandirlo duplicando su longitud. De esta forma, cuando alargamos un vector de N elementos a $2N$ elementos, el coste de N copias puede distribuirse sobre los siguientes N elementos, que pueden ser insertados en el vector sin necesidad de expandirlo.

Para mayor detalle, la Figura 2.4 muestra un programa que lee una cantidad ilimitada de enteros de la entrada estándar y almacena el resultado de la lectura en un vector que se expande dinámicamente. La rutina `reajustar` realiza la expansión del vector (o su contracción), devolviendo una referencia al nuevo vector. De forma análoga, el método `leerEnteros` devuelve una referencia al vector donde se almacenan los datos.

Al principio de `leerEnteros`, `elemLeidos` se inicializa a 0 y se comienza la lectura con un vector de cinco elementos. La lectura reiterada de los datos se realiza en las líneas 28 y 30. Si el vector está lleno, hecho que se comprueba en la línea 31, el vector se expande llamando a `reajustar`. En las líneas 44 a 50 el vector se agranda aplicando la estrategia comentada. En la línea 33, el dato que acaba de ser leído se inserta en el vector y el contador de datos introducidos se incrementa. Si se produce un error en la lectura, se detiene la ejecución del programa. Por último, en la línea 38 se reduce el tamaño del vector para ajustarlo al número de datos leídos.

2.4.3 Vectores multidimensionales

En algunas ocasiones es necesario acceder a un vector a través de varios índices. Un ejemplo típico son las matrices. Un *vector multidimensional* es un vector en el que el acceso a un elemento se realiza empleando más de un índice. La cantidad de memoria que se le reserva se determina al especificar el tamaño de sus índices. Accedemos a cada elemento colocando cada índice entre su propio par de corchetes. Por ejemplo, la declaración

```
int [ ] [ ] x = new int [ 2 ] [ 3 ];
```

```
1  import java.io.*;
2  public class LecturaEnteros
3  {
4     public static void main( String [ ] args )
5     {
6        int [ ] vector = leerEnteros( );
7        for( int i = 0; i < vector.length; i++ )
8           System.out.println( vector[ i ] );
9     }
10
11    // Lee una cantidad ilimitada de ints sin recuperación
12    // de errores; devuelve un int [ ]
13    public static int [ ] leerEnteros( )
14    {
15    //BufferedReader se estudia en la Sección 2.6
16       BufferedReader in = new BufferedReader( new
17                                 InputStreamReader( System.in ) );
18       int entradaVal = 0;
19       int [ ] vector = new int[ 5 ];
20       int elemLeidos = 0;
21       String unaLinea;
22
23       System.out.println( "Introduce una cantidad cualquiera" +
24                           "de enteros, uno por línea: " );
25       try
26       {
27          while( ( unaLinea = in.readLine( ) ) != null )
28          {
29             entradaVal = Integer.parseInt( unaLinea );
30             if( elemLeidos == vector.length )
31                vector = reajustar( vector, vector.length * 2 );
32             vector[ elemLeidos++ ] = entradaVal;
33          }
34       }
35       catch( Exception e ) { } // No se procesan los errores
36       System.out.println( "Lectura finalizada" );
37       return reajustar( vector, elemLeidos );
38    }
39
40    // Reajustar el int[ ] vector; devuelve el nuevo vector
41    public static int [ ] reajustar( int [ ] vector, int nuevoTamanyo )
42    {
43       int [ ] original = vector;
44       int numCopia = Math.min( original.length, nuevoTamanyo );
45
46       vector = new int[ nuevoTamanyo ];
47       for( int i = 0; i < numCopia; i++ )
48          vector[ i ] = original[ i ];
49       return vector;
50    }
51 }
```

Figura 2.4 Código para leer una cantidad ilimitada de `ints` y devolverlos como salida.

define el vector bidimensional `x`, con el primer índice tomando valores entre 0 y 1 y el segundo moviéndose entre 0 y 2 (para guardar un total de seis objetos). El compilador asigna seis posiciones de memoria a dichos objetos.

2.4.4 Argumentos de la línea de comandos

Los argumentos de la línea de comandos aparecen cuando examinamos el parámetro de `main`. El vector de cadenas de caracteres representa los argumentos adicionales de la línea de comandos. Por ejemplo, cuando se ejecuta el programa

```
java Eco este aquel
```

`args[0]` referencia al `String "este"` y `args[1]` referencia al `String "aquel"`. El programa de la Figura 2.5 implementa el programa `Eco`.

```
1  public class Eco
2  {
3     // Muestra los argumentos de la línea de comandos
4     public static void main( String [ ] args )
5     {
6        for( int i = 0; i < args.length - 1; i++ )
7           System.out.print( args[ i ] + " " );
8        if( args.length != 0 )
9           System.out.println( args[ args.length - 1 ] );
10       else
11          System.out.println( "No hay argumentos que mostrar" );
12    }
13 }
```

Figura 2.5 El comando `eco`.

2.5 Manejo de excepciones

Las *excepciones* son elementos que almacenan información y la transmiten fuera de la secuencia normal de retorno de datos. Las excepciones se propagan hacia atrás a través de la secuencia habitual de llamada, hasta que sean *recogidas* por alguna rutina. Se emplean para detectar hechos excepcionales, como por ejemplo, errores.

2.5.1 Procesamiento de excepciones

El código de la Figura 2.6 muestra el empleo de excepciones. El código que eventualmente puede generar una excepción se encierra en el bloque `try`. El bloque `try` se extiende desde la línea 15 hasta la línea 19. Inmediatamente después del bloque `try` se encuentran los manejadores de excepciones. Esta parte del código se ejecuta sólo si se produce una excepción. En el momento en el que ésta se lanza, el bloque `try` del que procede se considera finalizado. Cada bloque `catch` se prueba en orden hasta encontrar un manejador de excepciones adecuado. Como

```
1  import java.io.*;
2
3  public class DividePorDos
4  {
5      public static void main( String [ ] args )
6      {
7          // BufferedReader se discute en la Sección 2.6
8          BufferedReader in = new BufferedReader( new
9                                  InputStreamReader( System.in ) );
10         int x;
11         String unaLinea;
12
13         System.out.println( "Introduce un entero: " );
14         try
15         {
16             unaLinea = in.readLine( );
17             x = Integer.parseInt( unaLinea );
18             System.out.println( "La mitad de x es " + ( x / 2 ) );
19         }
20         catch( Exception e )
21           { System.out.println( e ); }
22         }
23 }
```

Figura 2.6 Un programa sencillo para ilustrar las excepciones.

`Exception` incluye los tipos de todas las excepciones de nuestro interés, recoge cualquier excepción generada por el bloque `try`. Más concretamente, estas excepciones son del tipo `IOException`, generadas por `readLine` si se produce algún error inesperado de lectura, y del tipo `NumberFormatException`, generadas por `parseInt`, si `unaLinea` no es convertible a entero.

Una vez capturada la excepción correspondiente se ejecuta el código del bloque `catch` —en nuestro caso la línea 21—. Entonces este bloque y el que contiene la combinación `try/catch`[1] se consideran terminados. En el ejemplo se imprime un mensaje significativo para el objeto `e` de tipo `Exception`. Como alternativa, podría realizarse un procesamiento adicional o bien darse mensajes de error más detallados.

2.5.2 La cláusula `finally`

Algunos objetos construidos en el bloque `try` deberían ser eliminados antes de concluir la ejecución del mismo. Por ejemplo, los ficheros que se abren en el bloque `try` necesitan cerrarse antes de abandonar dicho bloque. Uno de los problemas que aparece es que si se lanza una excepción dentro de un bloque `try`, la eliminación podría no producirse al causar la excepción la salida inmediata del bloque `try`. Aunque la eliminación de los objetos puede realizarse justo después del último bloque `catch`, esto sólo es efectivo si la excepción es recogida por alguna cláusula `catch`, y esto es difícil de garantizar con seguridad.

y `catch` requieren un bloque de instrucciones en lugar de una instrucción única, por lo que las llaves no son piladores anteriores de Java aceptan, de forma errónea, código que no incluye las llaves necesarias.

La cláusula `finally` que sigue al último bloque `catch` (o al bloque `try` si no hay ningún bloque `catch`) se emplea en estas situaciones. La cláusula `finally` consiste en la palabra clave `finally` seguida del bloque `finally`. Existen tres situaciones básicas:

1. Si el bloque `try` se ejecuta sin generarse excepciones, el control pasa al bloque `finally`. Esto es así aún cuando el bloque `try` termine antes de alcanzar su última instrucción, mediante un `return`, `break` o `continue`.
2. Si se produce una excepción dentro del bloque `try` que no es recogida, el control pasa al bloque `finally`. Tras ejecutar éste la excepción se propaga.
3. Si se produce una excepción dentro del bloque `try` que es recogida en un `catch` posterior, el control pasa al bloque `catch` correspondiente. Tras ejecutarse, se procesa el bloque `finally`.

2.5.3 Excepciones más comunes

En Java existen muchos tipos de excepciones estándar. Las excepciones *run-time* (o en tiempo de ejecución) incluyen eventos como la división-por-cero de enteros y los accesos ilegales a vectores. Como estos eventos pueden producirse casi en cualquier punto, sería agotador incorporar constantemente manejadores de excepciones para ellos. En cualquier caso, si se especifica para ellos un bloque `catch`, estas excepciones se comportan como cualquier otra. Pero si no se especifica ninguno, se lanza una excepción estándar que se propaga de la forma usual, pudiendo sobrepasar `main`. En este caso el programa termina de forma anómala, presentando un mensaje de error. En la Figura 2.7 se muestran algunas de las excepciones *run-time* más usuales.

La mayoría de las excepciones son del tipo *standard checked*. Si un método puede lanzar una de estas excepciones directa o indirectamente, el programador debe definir el bloque `catch` adecuado para ella o indicar de forma explícita que

Excepciones Run-time	Significado
`ArithmeticException`	Desbordamiento o división entera por cero.
`NumberFormatException`	Conversión ilegal de un `String` a un tipo numérico.
`IndexOutOfBoundsException`	Acceso a un elemento inexistente de un vector o de un `String`.
`NegativeArraySizeException`	Intento de creación de un vector de longitud negativa.
`NullPointerException`	Intento de uso de una referencia nula.
`SecurityException`	Violación de la seguridad en tiempo de ejecución.

Figura 2.7 Excepciones *run-time* más comunes.

la excepción va a propagarse, incluyendo para esto último la cláusula `throws` en la declaración del método. Observe que antes o después deben definirse manejadores para ellas, ya que el método `main` no debería tener cláusula `throws`. En el Capítulo 4 se muestra cómo definir nuevos tipos de excepciones. En la Figura 2.8 se recogen algunas de las excepciones más usuales de este tipo.

Los *errores* son excepciones, pero no se ajustan al tipo `Exception`. Típicamente son irrecuperables. El error más común es `OutOfMemoryError`. Para capturar cualquier excepción posible debe recogerse un objeto `Throwable`.

Excepciones Checked	Significado
`java.io.EOFException`	Final de fichero antes de lo esperado.
`java.io.FileNotFoundException`	Fichero inexistente.
`java.io.IOException`	Incluye muchas excepciones de E/S.
`InterruptedException`	Lanzada por el método `Thread.sleep`.

Figura 2.8 Excepciones *standard checked* más comunes.

2.5.4 Las cláusulas `throw` y `throws`

El programador puede lanzar una excepción empleando la cláusula `throw`. Por ejemplo, podemos lanzar una excepción de tipo `Exception` mediante

```
throw new Exception( "Malas Noticias" );
```

Normalmente, no se lanzan excepciones del tipo `Exception`, sino que se lanzan excepciones definidas por el usuario. Los detalles de esto se explican en el Capítulo 4.

Como se ha mencionado anteriormente, las excepciones del tipo *standard checked* deben ser recogidas o propagadas explícitamente hacia la rutina que ha realizado la llamada, pero en algunas ocasiones, y como último recurso, deben procesarse en `main`. Para hacer esto último, el método que no desea capturar la excepción debe indicar, a través de la cláusula `throws`, qué excepciones se van a propagar. La cláusula `throws` se escribe al final de la cabecera del método. La Figura 2.9 muestra un método que propaga las excepciones del tipo `IOException` que encuentra, éstas deben ser recogidas en `main` (ya que no incluiremos una cláusula `throws` en `main`).

2.6 Entrada y salida

La *entrada y salida* (*E/S*) se realiza en Java empleando el paquete `java.io`. La instrucción

```
import java.io.*;
```

```
1  import java.io.*;
2
3  public class DemoThrow
4  {
5        public static void procesaFichero( String Fichero )
6                                                    throws IOException
7        {
8        // Esta implementación omitida propaga cualquier
9        // IOException hacia la rutina invocadora
10       }
11
12       public static void main( String [ ] args )
13       {
14             for( int i = 0; i < args.length; i++ )
15             {
16                   try
17                     { procesaFichero( args[ i ] ; }
18                   catch( IOException e )
19                     { System.out.println( e ); }
20             }
21       }
22 }
```

Figura 2.9 Muestra de la cláusula `throws`.

debe aparecer en cualquier programa que emplee las rutinas de E/S de un modo no trivial. La librería de Java es bastante compleja y tiene multitud de opciones. Examinaremos sólo las más básicas, concentrándonos por completo en la E/S con formato.

2.6.1 Operaciones básicas de E/S

Contamos con tres canales predefinidos para realizar la E/S mediante el terminal. `System.in`, que es la entrada estándar, `System.out`, que es la salida usual, y `System.err`, para la salida de errores.

Como ya hemos mencionado, los métodos `print` y `println` se emplean para la salida con formato. Gracias al método `toString`, un valor de cualquier tipo puede convertirse en un `String` que puede imprimirse; en muchos casos esta transformación es automática. A diferencia de C y C++, que tienen multitud de opciones de formato, la salida de Java se realiza, casi exclusivamente, mediante la concatenación de `String`s sin formato alguno.

Un modo sencillo de realizar la lectura de la entrada es leer una sola línea, almacenándola en un objeto de tipo `String`, para lo que se emplea `readLine`. Este método lee hasta que encuentra un final de línea o el final del fichero. Los caracteres leídos, salvo el final de línea (en el caso de que se encuentre), se devuelven como un nuevo `String`. Para emplear `readLine`, se debe construir primero un objeto `BufferedReader` sobre un objeto `InputStreamReader`, que a su vez se crea partiendo de `System.in`. Esto se muestra en las líneas 8 y 9 de la Figura 2.6. Observe que las `IOException` son del tipo *standard checked*, por lo que, en algunos casos, deben ser recogidas. En muchas situaciones, se permite que la `IOException` se propague hasta un bloque `catch` del método `main`; esta técnica se ilustra en la Figura 2.9.

2.6.2 El objeto `StringTokenizer`

Recuerde que para leer un solo carácter de tipo primitivo, como un `int`, se debe emplear `readLine` para leer la línea y aplicar después el método adecuado para generar el tipo primitivo a partir del `String`. Para el tipo `int` podemos emplear `parseInt`.

En algunos casos nos encontramos con más de un elemento en la misma línea. Por ejemplo, supongamos que cada línea contiene dos valores de tipo `int`. Java tiene definido el objeto `StringTokenizer` que permite separar en `tokens` un `String`. Para usarlo debemos emplear el mandato `import`

```
import java.util.*;
```

El empleo de `StringTokenizer` se ilustra en la Figura 2.10. En primer lugar, en la línea 19 construimos un objeto de este tipo, a partir del `String` que representa la línea de entrada. El método `countTokens`, en la línea 20, devuelve el número de tokens del `String`. En este ejemplo deberían ser dos, de no ser errónea la entrada. El método `nextToken` devuelve el siguiente token como un `String`. Este método lanza una `NoSuchElementException` en el caso de no encontrar ningún token, que representa un error y por tanto no debe ser recogido. En las líneas 22 y 23 usamos `nextToken`, seguido de `parseInt`, para obtener un valor de

```
1  import java.io.*;
2  import java.util.*;
3
4  public class MaxTest
5  {
6     public static void main( String args[ ] )
7     {
8        BufferedReader in = new BufferedReader( new
9                                 InputStreamReader( System.in ) );
10       String unaLinea;
11       StringTokenizer str;
12       int x;
13       int y;
14
15       System.out.println( "Introduzca 2 ints en una misma
            línea: " );
16       try
17       {
18          unaLinea = in.readLine( );
19          str = new StringTokenizer( unaLinea );
20          if( str.countTokens( ) != 2 )
21              throw new NumberFormatException( );
22          x = Integer.parseInt( str.nextToken( ) );
23          y = Integer.parseInt( str.nextToken( ) );
24          System.out.println( "Max: " + Math.max( x, y ) );
25       }
26       catch( Exception e )
27         { System.err.println( "Error: se necesitan dos
             enteros" ); }
28    }
29 }
```

Figura 2.10 Programa que muestra el troceo de cadenas de caracteres.

tipo `int`. Todos los errores, incluyendo el producido cuando no se obtienen exactamente dos tokens, son tratados en el bloque `catch`. Como es habitual, con un mayor esfuerzo se podrían generar mejores mensajes de error.

Por defecto, los tokens se separan mediante espacios en blanco. El `StringTokenizer` puede construirse de modo que reconozca otros caracteres como delimitadores, o para considerar dichos delimitadores como tokens.

2.6.3 Ficheros de acceso secuencial

Una de las reglas básicas de Java es que lo que funciona en la E/S con el terminal también funciona en la E/S con ficheros. Para trabajar con un fichero no construimos un objeto `BufferedReader` a partir de un `InputStreamReader`, sino a partir de un `FileReader`, que es creado a su vez especificando el nombre de un fichero.

En la Figura 2.11 se muestra un ejemplo que ilustra estas ideas básicas. Se trata de un programa que imprime el contenido de los ficheros de texto que se

```
1  import java.io.*;
2
3  public class ContenidoFicheros
4  {
5     public static void main( String [ ] args )
6     {
7        if( args.length == 0 )
8           System.out.println( "No se especifican ficheros" );
9        for( int i = 0; i < args.length; i++ )
10          contenidoFichero( args[ i ] );
11    }
12
13    public static void contenidoFichero( String nombreFichero )
14    {
15       FileReader elFichero;
16       BufferedReader lector = null;
17       String unaLinea;
18
19       System.out.println( "Fichero: " + nombreFichero );
20       try
21       {
22          elFichero = new FileReader( nombreFichero );
23          lector = new BufferedReader( elFichero );
24          while( ( unaLinea = lector.readLine( ) ) != null )
25             System.out.println( unaLinea );
26       }
27       catch( Exception e )
28         { System.out.println( e ); }
29
30       // Cierre del fichero
31       try
32       {
33          if(lector != null )
34             lector.close( );
35       }
36       catch( IOException e ) ( )
37    }
38 }
```

Figura 2.11 Programa que imprime el contenido de un fichero.

indiquen en la línea de comandos. La rutina `main` simplemente recoge los argumentos de la línea de comandos, pasando cada uno a `contenidoFichero`. En `contenidoFichero` se construye el objeto `FileReader` en la línea 22, que se emplea para crear el objeto `BufferedReader` – `lector` – en la línea 23. A partir de este punto la lectura es igual a la ya explicada.

Después de terminar con un fichero, debemos cerrarlo, ya que de otro modo podríamos quedarnos sin canales. Observe que esto no puede hacerse al final del bloque `try`, pues una excepción causaría la salida del bloque prematuramente. En consecuencia, cerramos el fichero tras la secuencia `try/catch`. Como todos los errores recuperables están incluidos en `Exception`, no es necesaria una cláusula `finally`. Estamos seguros de alcanzar la instrucción de cierre de fichero, por si hay más trabajo que hacer.

La salida con formato es similar a la entrada con formato. Como no la usaremos en este texto, remitimos al lector a cualquier referencia de Java si desea conocer los detalles de la misma.

Resumen

En este capítulo hemos estudiado los tipos referencia. Una referencia es una variable que almacena, o bien una dirección de memoria donde se almacena un objeto, o bien una referencia especial `null`. Sólo pueden referenciarse los objetos, aunque un objeto puede ser referenciado por más de una variable. Cuando dos referencias son comparadas vía `==`, el resultado es `true` si ambas referencias apuntan al mismo objeto. De forma similar, `=` hace que una variable referencia apunte a otro objeto. Las referencias pueden manipularse con muy pocas operaciones. La más importante es la representada por el operador punto, que permite la selección de un método de una clase, o bien el acceso a sus atributos internos.

Debido a que en Java sólo existen ocho tipos primitivos, casi cualquier cosa es un objeto accesible a través de una referencia. Esto incluye las cadenas de caracteres, los vectores, los objetos de la clase `Exception`, los canales de entrada y salida de datos y el `StringTokenizer`.

El `String` es un tipo referencia especial pues sobre el mismo están definidos los operadores de concatenación `+` y `+=`. Por lo demás, un `String` se comporta como cualquier otra referencia; así, se necesita `equals` para comprobar si los estados de dos `String`s son idénticos. Un vector es una colección de elementos del mismo tipo. El vector se indexa partiendo de 0 y se garantiza el chequeo del rango. Los vectores se pueden expandir dinámicamente, usando `new` para asignar mayor cantidad de memoria y copiando después los elementos uno a uno.

Las excepciones se emplean para detectar eventos excepcionales. Una excepción se lanza mediante la cláusula `throw`, y se propaga hasta que es recogida por algún bloque `catch` asociado a un bloque `try`. Excepto para las excepciones en tiempo de ejecución (*run-time exceptions*) y los `Error`es, cada método debe indicar las excepciones que puede propagar empleando la cláusula `throws`.

El `StringTokenizer` se emplea para partir un `String`, obteniendo nuevos `String`s. Normalmente se emplea junto a otras rutinas de procesamiento de la entrada. La entrada se manipula a través de los objetos `BufferedReader`, `InputStreamReader` y `FileReader`.

El siguiente capítulo muestra cómo diseñar nuevos tipos definiendo una *clase*

Elementos del juego

agregado Colección de objetos almacenados en la misma unidad.
argumento de la línea de comandos Empleado como parámetro del método `main`.
atributo `length` Usado para conocer la longitud de un vector.
bloque `catch` Se emplea para el procesamiento de excepciones.
bloque `try` Comprende el código que puede generar una excepción.
`BufferedReader` Se emplea para el procesamiento línea a línea de la entrada.
concatenación de cadenas Realizada a través de los operadores `+` y `+=`.
construcción Para los objetos se efectúa mediante la palabra reservada `new`.
entrada y salida (E/S) Implementada en el paquete `java.io`.
`equals` Se emplea para comprobar si los valores de dos objetos almacenados son iguales.
`Error` Excepción irrecuperable.
excepción del tipo *standard checked* Debe ser recogida o declarada explícitamente en la cláusula `throws` del método correspondiente.
excepción *run-time* No necesita ser declarada. Ejemplos notables son `ArithmeticException` y `NullPointerException`.
excepción Sirve para detectar situaciones excepcionales tales como errores.
expansión dinámica de vectores Permite aumentar el tamaño de los vectores cuando ello sea preciso.
`FileReader` Empleado cuando la entrada se efectúa por fichero.
`finally` Se ejecuta siempre antes de salir de un bloque `try/catch`.
inmutable Se dice de un objeto que no puede cambiar de estado. Por ejemplo, los `String`s son inmutables.
`java.io` Paquete que debe ser importado para producir cualquier salida no trivial.
`java.util` Paquete que debe ser importado para poder emplear el objeto `StringTokenizer`.
`lizq` y `lder` Variables empleadas para indicar los lados izquierdo y derecho, respectivamente.
método `length` Usado para conocer la longitud de una cadena de caracteres.
`new` Empleado para crear un nuevo objeto.
`null` Valor de una variable referencia que no apunta a ningún objeto.
`NullPointerException` Generada cuando se intenta acceder a una referencia `null`.
objeto Entidad no primitiva.
operador de indexación de vectores `[ ]` Permite el acceso a cada elemento de un vector.
operador punto (`.`) Permite el acceso a cada componente de una estructura.
paso por referencia En muchos lenguajes de programación significa que el parámetro formal es una referencia al parámetro actual. Éste es el efecto producido en Java cuando el paso por valor se utiliza sobre un tipo referencia.
recogida de basura Recuperación automática de la memoria que ya no es referenciada.
`String` Objeto especial empleado para almacenar una secuencia de caracteres.
`StringTokenizer` Empleado para obtener `String`s delimitados a partir de una sola cadena de caracteres.

`System.in`, `System.out` y `System.err` Canales de E/S predefinidos.
`throw` Empleado para lanzar una excepción.
`throws` Indica que un método puede propagar una excepción.
tipo referencia Cualquier tipo no primitivo.
`toString` Convierte un tipo primitivo, o un objeto, en un `String`.
vector Almacena una colección de elementos del mismo tipo.
vector multidimensional Vector cuya manipulación requiere varios índices.

Errores comunes

1. Para tipos referencia y vectores, = no hace una copia de los objetos sino de las referencias.
2. Para tipos referencia y cadenas de caracteres, para comprobar si dos objetos tienen exactamente el mismo estado debe emplearse `equals` en lugar de `==`.
3. Los errores por salida de rango son habituales en todos los lenguajes de programación.
4. Los tipos referencia se inicializan a `null` por defecto; es decir, no se construye ningún objeto nuevo sin haber llamado previamente a `new`. Una «variable referencia sin inicializar» o la `NullPointerException` indica que el programador ha olvidado crear un objeto.
5. En Java los vectores se indexan desde 0 hasta `N-1`, donde `N` es el tamaño del vector. El acceso a componentes fuera de ese rango se detecta de forma automática en tiempo de ejecución.
6. Los vectores bidimensionales se indexan como `A[i][j]` y no como `A[i,j]`.
7. Las excepciones *standard checked* deben ser recogidas, o declaradas en la cláusula `throws` del método correspondiente para permitir su propagación.
8. Debe emplearse `" "` en lugar de `' '`, para mostrar un espacio en blanco.

En Internet

A continuación mostramos los ficheros disponibles para este capítulo. Todos ellos son autocontenidos y no vuelven a utilizarse en el resto del texto. Se encuentran en el directorio **Chapter02**.

DivideBytwo.java	Código correspondiente a la Figura 2.6. Es la versión inglesa de la clase `DividePorDos`.
Echo.java	Código correspondiente a la Figura 2.5. Es la versión inglesa de la clase `Eco`.
ListFiles.java	Código correspondiente a la Figura 2.11. Es la versión inglesa de la clase `ContenidoFicheros`.
Lottery.java	Código correspondiente a la Figura 2.3. Es la versión inglesa de la clase `Loteria`.
MaxTest.java	Código correspondiente a la Figura 2.10. Es la versión inglesa de la clase `MaxTest`.
ReadInts.java	Código correspondiente a la Figura 2.4. Es la versión inglesa de la clase `LecturaEnteros`.

Ejercicios

Cuestiones breves

2.1. Enumere las principales diferencias entre tipos referencia y tipos primitivos.
2.2. Enumere cinco operaciones que pueden ser aplicadas a los tipos referencia.
2.3. Describa cómo funcionan las excepciones en Java.
2.4. Enumere las principales operaciones que pueden realizarse con cadenas de caracteres.

Problemas teóricos

2.5. Si `x` e `y` tienen los valores 5 y 7, respectivamente, determine la salida de la secuencia siguiente:

```
System.out.println( x + ' ' + y );
System.out.println( x + " " + y );
```

Problemas prácticos

2.6. Un *testsum* es el entero 32-bit calculado sumando todos los caracteres Unicode de un fichero (se permite `DesbordamientoInferior` silencioso, aunque éste es improbable cuando todos los caracteres son ASCII). Dos ficheros iguales tienen el mismo testsum. Escriba un programa que calcule el testsum de un fichero cuyo nombre se introducirá en la línea de comandos.
2.7. Modifique el programa de la Figura 2.11 de modo que no se introduzca ningún parámetro por la línea de comandos, sino que se utilice la entrada estándar.
2.8. Escriba un método que devuelva `true` si `String str1` es un prefijo de `String str2`. No debe emplear ninguna rutina de búsqueda de cadenas de caracteres, excepto `charAt`.

Prácticas de programación

2.9. Escriba un programa que muestre la cantidad de caracteres, palabras y líneas de los ficheros que se indiquen en la línea de comandos.
2.10. Implemente un programa de copia de ficheros. Incluya un test que se asegure de que el fichero de origen y el de destino son diferentes. Necesitará leer alguna referencia sobre Java para aprender a programar la salida de datos en fichero.

Referencias

Se puede encontrar más información en las referencias listadas al final del Capítulo 1.

CAPÍTULO

3 Objetos y clases

En este capítulo se comienza a estudiar la *programación orientada a objetos*. Una componente fundamental de la programación orientada a objetos es la especificación, implementación y uso de objetos. En el Capítulo 2 vimos algunos ejemplos de objetos, incluyendo las cadenas y los ficheros, que son parte de la librería obligatoria de Java. También vimos que estos objetos tienen un estado interno que se puede manipular aplicando el operador punto para seleccionar un método. En Java, el estado y la funcionalidad de un objeto vienen dados por la definición de una *clase*. Un objeto es una instancia de una clase.

En este capítulo veremos:

- Cómo usa Java las clases para conseguir *encapsulación* y *ocultamiento de información*.
- Cómo se implementan y se documentan de forma automática las clases.
- Cómo se agrupan las clases en paquetes.

3.1 ¿Qué es la programación orientada a objetos?

La programación orientada a objetos parece estar surgiendo como el paradigma dominante de los noventa. En esta sección se estudian algunos elementos de la orientación a objetos que soporta Java y se mencionan algunos de los principios de la programación orientada a objetos.

El corazón de la programación orientada a objetos es el *objeto*. Un objeto es un tipo de datos con estructura y estado. Cada objeto tiene definidas operaciones que pueden acceder a ese estado o manipularlo. Como ya hemos visto, en Java los objetos se distinguen de los valores de los tipos primitivos, pero ésta es más bien una característica de Java que de la programación orientada a objetos. Además de ejecutar acciones generales, podemos hacer lo siguiente:

- Crear nuevos objetos, posiblemente con un valor inicial.
- Copiar o comprobar la igualdad.
- Realizar E/S sobre esos objetos.

También vimos el objeto como una *unidad atómica* que el usuario no debería pretender diseccionar. La mayoría de nosotros ni siquiera pensaría en jugar con los bits que representan a un número en coma flotante, y consideraríamos completamente ridículo el intentar incrementar nosotros mismos un número en coma flotante alterando su representación interna.

El principio de atomicidad se conoce como *ocultamiento de la información*. El usuario no tiene acceso directo a las partes del objeto o a su implementación; solamente se puede acceder a ellas de forma indirecta a través de métodos proporcionados junto con el objeto. Podemos pensar que cada objeto viene con el aviso «No abrir —partes no utilizables por el usuario». En la vida real, la mayoría de la gente que intenta arreglar las cosas a pesar de tales avisos, termina produciendo más perjuicio que beneficio. En este sentido, la programación imita al mundo real. El agrupamiento de datos y las operaciones que se aplican sobre ellos, forman un agregado, mientras que el ocultamiento de los detalles del agregado, se conoce como *encapsulación*.

Un objetivo importante de la programación orientada a objetos es soportar la reutilización de código. Al igual que los ingenieros usan las misma componentes repetidamente en sus diseños, los programadores deberían ser capaces de reutilizar objetos en lugar de implementarlos repetidamente. Cuando tenemos una implementación del objeto exacto que necesitamos usar, la reutilización es simple. El problema está en utilizar un objeto disponible cuando no es exactamente el que se necesita, sino solamente uno muy parecido.

Los lenguajes orientados a objetos proporcionan varios mecanismos para satisfacer este objetivo. Uno es el uso de código *genérico*. Si la implementación es idéntica independientemente del tipo de los objetos sobre los que se aplica, no hay necesidad de rescribir completamente el código: en lugar de ello, se escribe el código genéricamente, de forma que funcione para cualquier tipo. Por ejemplo, la lógica usada para ordenar un vector de objetos es independiente del tipo de los objetos que se están ordenando, de modo que se podría usar un algoritmo genérico.

El mecanismo de *herencia* nos permite extender la funcionalidad de un objeto. En otras palabras, podemos crear tipos nuevos con las propiedades del tipo original restringidas (o extendidas). La herencia nos permite avanzar un largo trecho hacia el objetivo deseado de reutilización de código.

Otro principio importante de la programación orientada a objetos es el *polimorfismo*. Un tipo referencia polimórfico puede referenciar objetos de varios tipos diferentes. Cuando se aplican métodos al tipo polimórfico, se selecciona automáticamente la operación apropiada para el objeto referenciado en ese momento. En Java, esto se implementa como parte de la herencia. El polimorfismo nos permite implementar clases que comparten una lógica común. Como se estudia en el Capítulo 4, esto se aplica reiteradamente en las librerías de Java. El uso de la herencia para crear estas jerarquías distingue la programación orientada a objetos de la más simple *programación basada en objetos*.

En Java, los algoritmos genéricos se implementan como parte de la herencia. En el Capítulo 4 se estudia la herencia y el polimorfismo. En este capítulo, describimos cómo Java usa las clases para conseguir encapsulación y ocultamiento de información.

Un *objeto* en Java es una instancia de una clase. Una *clase* es similar a una estructura en C o un registro en Pascal o Ada, excepto por el hecho de que hay dos mejoras importantes. En primer lugar, las componentes pueden ser funciones o datos, las cuales se conocen respectivamente por *métodos* y *atributos*. En segundo lugar, se puede restringir la visibilidad de dichas componentes. Puesto que los métodos que manipulan el estado del objeto son componentes de la clase, se accede a ellos mediante el operador punto, al igual que a los atributos. En la terminología de la programación orientada a objetos, cuando hacemos una llamada a un método, estamos pasando un mensaje al objeto.

3.2 Un ejemplo sencillo

Recuerde que cuando está diseñando una clase, es importante ser capaz de ocultar los detalles internos al usuario de la clase. Esto se consigue de dos formas. En primer lugar, la clase puede definir su funcionalidad mediante componentes de la clase, llamadas métodos. Algunos de estos métodos describen cómo se crea e inicializa una instancia de la estructura, otros cómo se realiza las comprobaciones de igualdad o cómo se lleva a cabo la salida. Otras funciones serían específicas a la estructura particular. La idea es que los atributos internos que representan el estado del objeto no deberían ser manipulados directamente por el usuario de la clase, sino solamente a través de los métodos. Esta idea se puede reforzar ocultando componentes al usuario. Para hacer esto podemos especificar que se almacenen en una sección *privada*. El compilador hará cumplir la regla que convierte a las componentes de la sección privada en inaccesibles para el usuario del objeto. En general, todas las componentes de datos deberían ser privadas.

En la Figura 3.1 se muestra una declaración de clase para un objeto `CeldaEntero`[1].La declaración consta de dos partes: una pública y otra privada. La sección *pública* representa la porción visible al usuario del objeto. Puesto que esperamos ocultar los datos, en general solamente debería aparecer métodos y constantes en la sección pública. En nuestro ejemplo, tenemos métodos que leen y escriben en el objeto `CeldaEntero`. La sección *privada* contiene los datos: ésta es invisible para el usuario del objeto. Se debe acceder al atributo `valorAlmacenado` a través de las rutinas públicas `leer` y `escribir`; `main` no puede acceder directamente a ella. En la Figura 3.2 encontramos una representación gráfica de la situación.

```
1  // Clase CeldaEntero
2  // int leer( )              -->  Devuelve el valor almacenado
3  // void escribir( int x )   -->  Se almacena x
4
5  public class CeldaEntero
6  {
7          // Métodos públicos
8      public int leer( ) {return valorAlmacenado;}
9      public void escribir( int x ) {valorAlmacenado = x;}
10
11         // Representación interna privada del dato
12     private int valorAlmacenado;
13 }
```

Figura 3.1 Una declaración completa de una clase `CeldaEntero`.

[1] Recuerde que las clases deben guardarse en ficheros del mismo nombre que la clase. Luego `CeldaEntero` fichero *CeldaEntero.java*.

Figura 3.2 Componentes de `CeldaEntero`: `leer` y `escribir` son accesibles pero `valorAlmacenado` está oculto.

La Figura 3.3 muestra cómo se usan los objetos de la clase `CeldaEntero`. Puesto que `leer` y `escribir` son componentes de la clase `CeldaEntero`, se puede acceder a ellas usando el operador punto. También se podría acceder al atributo `valorAlmacenado` usando el operador punto, pero debido a que es privada, el acceso en la línea 14 sería ilegal, si ésta no estuviera comentada.

A continuación resumimos la terminología. La clase define *componentes*, que pueden ser *atributos* (datos) o *métodos* (funciones). Los métodos pueden actuar sobre los atributos y pueden llamar a otros métodos. El modificador de visibilidad `public` significa que la componente es accesible para cualquiera a través del operador punto. El modificador de visibilidad `private` significa que la componente solamente es accesible a través de otros métodos de esa clase. Si no hay modificador de visibilidad, tenemos acceso amistoso, el cual se estudia en la Sección 3.5.4. Hay un cuarto modificador conocido como `protected`, que se estudia en el Capítulo 4.

```
1  //Ejercicio con la clase CeldaEntero
2
3  public class TestCeldaEntero
4  {
5      public static void main( String [ ] args )
6      {
7          CeldaEntero m = new CeldaEntero( );
8
9          m.escribir( 5 );
10         System.out.println( "Contenidos de la celda: " + m.leer( ) );
11
12         // La siguiente línea sería ilegal si no estuviera comentada
13         // porque valorAlmacenado es una componente privada
14     // m.valorAlmacenado = 0;
15     }
16 }
```

Figura 3.3 Una rutina simple de comprobación para mostrar cómo se accede a los objetos de la clase `CeldaEntero`.

3.3 Javadoc

Cuando se diseña una clase, la *especificación de la clase* representa el diseño de la clase y nos dice qué se puede hacer sobre un objeto. La *implementación* representa los detalles internos de cómo se consigue esto. En lo que respecta al usuario, estos detalles no son importantes. En muchos casos, la implementación representa información del propietario que el diseñador de la clase no desea compartir. Sin embargo, la especificación debe estar compartida; en caso contrario, no se podría usar la clase.

En muchos lenguajes, los objetivos de compartir la especificación pero con ocultamiento de la implementación se consigue colocando especificación e implementación en ficheros fuente separados. Por ejemplo, en C++ la interfaz de clase se coloca en un archivo con extensión `.h`, y la implementación de la clase en un archivo `.cpp`. En el fichero `.h`, la interfaz de clase muestra los métodos (proporcionando las cabeceras de los métodos) implementados por la clase.

Java tiene un enfoque distinto. Es fácil ver que se puede documentar automáticamente una lista de métodos en una clase, con signaturas y tipos del resultado, a partir de la implementación. Java usa esta idea: el programa *javadoc*[2], proporcionado en todos los sistemas Java, se puede ejecutar para generar automáticamente documentación para las clases. La salida de *javadoc* es un conjunto de ficheros HTML que se puede ver o imprimir con un navegador.

El fichero de implementación también puede añadir comentarios *javadoc* que comiencen con el token `/**`. Dichos comentarios se añaden automáticamente, de una forma uniforme y consistente, a la documentación producida por *javadoc*.

Hay también varias marcas especiales que se pueden incluir en los comentarios *javadoc*. Algunas de ellas son `@author`, `@param`, `@return` y `@exception`. En la Figura 3.4 se muestra el uso de los comentarios *javadoc* en la definición de clases. En la línea 3, se usa la marca `@author`. Esta marca debe preceder a la definición de la clase. La línea 10 ilustra el uso de la marca `@return`, y la 19 el de la marca `@param` . Estas marcas deben aparecer antes de la declaración de un método. El primer token que sigue a la marca `@param` es el nombre del parámetro. La marca `@exception` no se ilustra, pero tiene la misma sintaxis que `@param`.

```
1  /**
2   * Clase para simular una celda de memoria para un entero
3   * @author Mark A. Weiss
4   */
5
6  public class CeldaEntero
7  {
8      /**
9       * Obtiene el valor almacenado.
10      * @return el valor almacenado.
11      */
12     public int leer( )
13     {
14         return valorAlmacenado;
15     }
16
17     /**
18      * Almacena un valor.
19      * @param x el número a almacenar.
20      */
21     public void escribir( int x )
22     {
23         valorAlmacenado = x;
24     }
25
26     private int valorAlmacenado;
27 }
```

Figura 3.4 Declaración de `CeldaEntero` con comentarios *javadoc*.

[2] *N. del T.*: La Figura 3.5 corresponde a la versión inglesa de la Figura 3.4.

En la Figura 3.5 se muestra la salida que resulta de la ejecución de *javadoc*. Ejecute *javadoc* proporcionando el nombre (incluida la extensión `.java`) del fichero fuente. Para obtener dibujos para los constructores, métodos, etc., debe tener un subdirectorio `images` en el directorio donde se generan los ficheros HTML

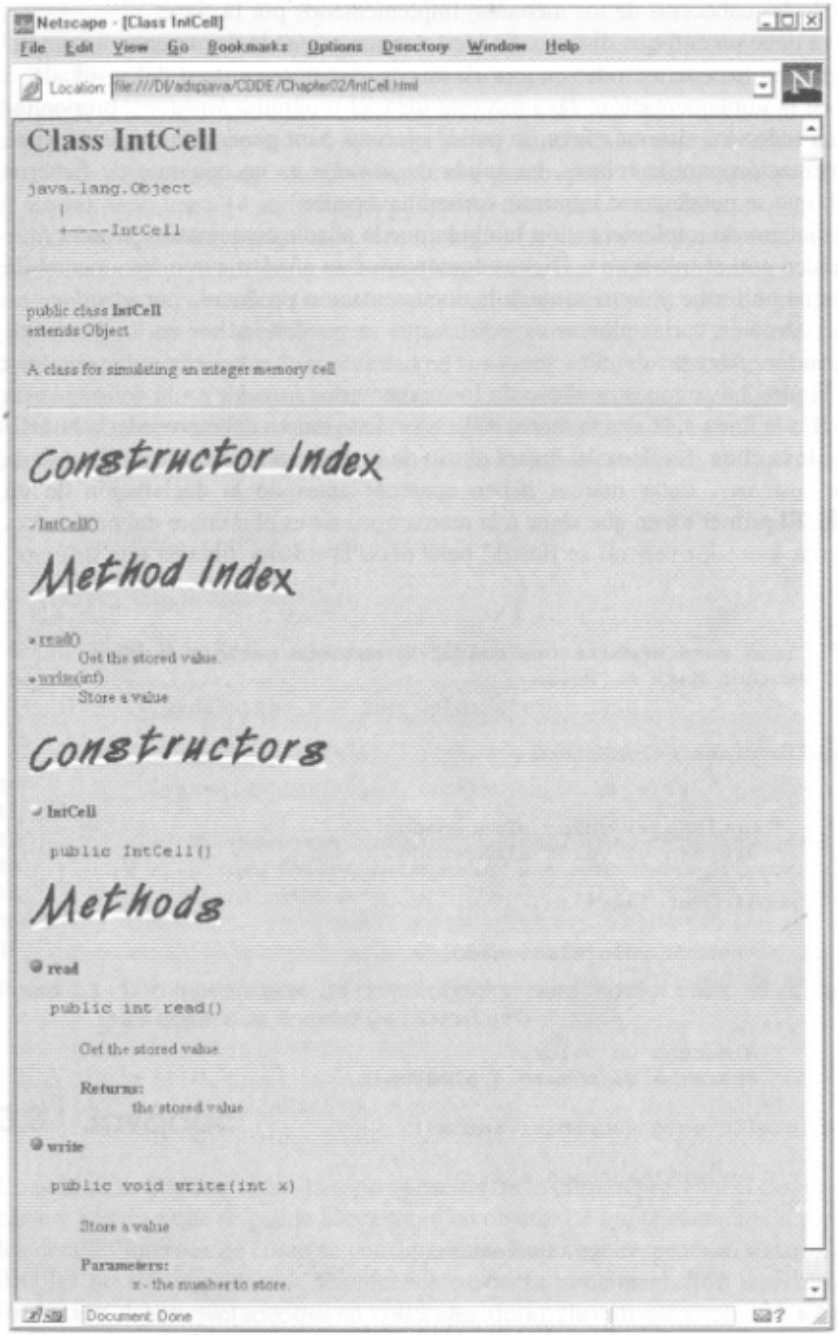


Figura 3.5 Salida de *javadoc* para la Figura 3.4.

(este subdirectorio es parte de la distribución JDK y está replicado en el código que se suministra en la red).

La salida de *javadoc* está formada exclusivamente por comentarios, excepto en el caso de las cabeceras de los métodos. El compilador no comprueba que dichos comentarios estén implementados. A pesar de ello, nunca nos quedaremos cortos al valorar la importancia de disponer de documentación apropiada. *javadoc* hace más fácil la tarea de generar documentación con un buen formato.

3.4 Métodos básicos

Algunos métodos son comunes a todas las clases. En esta sección se estudia los métodos *modificadores* y *de acceso*, y tres métodos especiales: los constructores, `toString` y `equals`. También se habla de los métodos estáticos, uno de los cuales es `main`.

3.4.1 Constructores

Como se ha mencionado anteriormente, una propiedad básica de los objetos es que se pueden definir, posiblemente con un valor inicial. En Java, los métodos que controlan cómo se crea e inicializa un objeto reciben el nombre de *constructores*. Gracias a la sobrecarga, en una clase se puede definir diversos constructores.

Si no se proporciona ningún constructor, como en el caso de la clase `CeldaEntero` de la Figura 3.1, se genera un constructor por defecto que inicializa cada atributo usando los correspondientes valores por defecto. Esto significa que los atributos de tipo primitivo se inicializan a cero y los tipos referencia se inicializan a la referencia `null`. Por tanto, en el caso de `CeldaEntero`, la componente `valorAlmacenado` es 0.

Para escribir un constructor, proporcionamos un método con el mismo nombre que la clase y sin resultado. En la Figura 3.6 hay dos constructores: uno comienza en la línea 7 y el otro en la línea 15. Usando estos constructores, podemos construir objetos `Fecha` de las dos formas siguientes:

```
Fecha f1 = new Fecha( );
Fecha f2 = new Fecha( 15, 4, 1998 );
```

Observe que una vez que se escribe un constructor, ya no se utiliza un constructor por defecto sin parámetros. Si desea volver a contar con él, tiene que escribirlo explícitamente. Así, el constructor de la línea 7 es obligatorio para poder construir el objeto que referencia `f1`.

3.4.2 Métodos modificadores y de acceso

Los atributos de una clase se declaran normalmente como privados. En consecuencia, las rutinas que no pertenecen a la clase no pueden acceder directamente a

ellos. Sin embargo, algunas veces nos gustaría examinar el valor de un atributo, e incluso podríamos querer cambiarlo directamente.

Una alternativa para hacer esto es declarar los atributos como públicos. Ésta es, sin embargo, una elección pobre, pues viola el principio de ocultamiento de información. En su lugar podemos proporcionar métodos para examinar y cambiar cada atributo. Un método que examina pero no cambia el estado de un objeto es un *método de acceso*. Un método que cambia el estado es un *modificador* (porque modifica el estado del objeto).

Casos particulares de métodos de acceso y modificadores trabajan solamente sobre un atributo. Estos métodos de acceso usualmente tendrán nombres que empiezan por `obtener`[3], como `obtenerMes`, mientras que los modificadores tendrán nombres que comienzan por `cambiar`[4], como `cambiarMes`.

La ventaja de usar un modificador es que de esta forma podemos asegurar que los cambios realizados en el estado del objeto son consistentes. Así, un modificador que cambie el atributo `dia` en un objeto `Fecha` nos garantizará que solamente se generan fechas legales.

3.4.3 Salida y `toString`

Normalmente queremos mostrar el estado de un objeto usando `print`. Esto se consigue definiendo un método `toString`[5] en la clase. Dicho método devuelve un valor de tipo `String` apto para producir una salida. Como ejemplo, en la Figura 3.6 se muestra una implementación básica del método `toString` para la clase `Fecha`.

3.4.4 `equals`

El método `equals` se usa para comprobar si dos referencias describen el mismo valor. La signatura es siempre la siguiente

```
public boolean equals( Object lder )
```

Observe que el parámetro es del tipo `Object`, en lugar del tipo de la clase. El método `equals` sobre una clase `NombreClase` se implementará de modo que devuelva `true` sólo cuando `lder` es una instancia de `NombreClase` y, si tras la conversión a `NombreClase`, todos los atributos primitivos son iguales (usando `==`) y todos los atributos referencia son iguales (usando una aplicación componente a componente de `equals`).

En la Figura 3.6 se proporciona un ejemplo de cómo se implementa `equals`. El operador `instanceof` se estudia en la Sección 3.6.3.

nglés.
nglés.
ndos `toString`, `equals` y `compareTo` no se han traducido ya que dichos métodos están predefinidos para el ones dadas en este libro sobreescriben a los predefinidos.

```
1  // Clase Fecha básica para ilustrar algunas características de
2  // Java sin comprobaciones de error ni comentarios javadoc
3
4  public class Fecha
5  {
6      // Constructor de cero parámetros
7      public Fecha( )
8      {
9          dia = 1;
10         mes = 1;
11         anyo = 1998;
12     }
13
14     // Constructor de tres parámetros
15     public Fecha( int elDia, int elMes, int elAnyo )
16     {
17         dia = elDia;
18         mes = elMes;
19         anyo = elAnyo;
20     }
21
22     // Devuelve true si dos valores son iguales
23     public boolean equals( Object lder )
24     {
25         if( ! ( lder instanceof Fecha ) )
26             return false;
27         Fecha lderFecha = ( Fecha ) lder;
28         return lderFecha.dia == dia && lderFecha.mes == mes &&
29                lderFecha.anyo == anyo;
30     }
31
32     // Conversión a String
33     public String toString( )
34     {
35         return dia + "/" + mes + "/" + anyo;
36     }
37
38     // Atributos
39     private int dia;
40     private int mes;
41     private int anyo;
42 }
```

Figura 3.6 Una clase `Fecha` básica que ilustra el uso de constructores y los métodos `equals` y `toString`.

3.4.5 Métodos `static`

Un *método estático* es un método que no necesita un objeto que lo controle. El método estático más común es `main`. En las clases `String`, `Integer` y `Math` se puede encontrar otros métodos estáticos. Algunos ejemplos son `String.valueOf`, `Integer.parseInt`, `Math.sin` y `Math.max`. El acceso a los métodos estáticos utiliza las mismas reglas de visibilidad que los atributos estáticos.

Recuerde del Capítulo 1 que algunos atributos de la clase usan el modificador `static`. Usándolo, en conjunción con la palabra `final`, obtenemos las constantes. Sin la palabra `final`, tenemos *atributos estáticos*, que tienen otro significado distinto, estudiado en la Sección 3.6.4.

3.4.6 main

Cuando se usa el comando *java* para ejecutar el intérprete, se llama al método `main` de la clase del fichero referenciado por el comando. Por tanto, cada clase puede tener su propia función `main` sin ningún problema. Sin embargo, aunque así se puede probar la funcionalidad, colocar `main` en una clase proporciona a esta función más visibilidad que la que estaría permitida en general. Por tanto, llamadas de `main` a métodos no públicos tendrán éxito en la prueba, aunque serán ilegales en un entorno más general.

3.5 Paquetes

Los paquetes se usan para organizar una serie de clases relacionadas. Cada paquete consta de un conjunto de clases. Dos clases en un mismo paquete tienen restricciones ligeramente menores de visibilidad entre ellas que si estuvieran en paquetes distintos.

Java proporciona varios paquetes predefinidos, entre ellos `java.applet`, `java.awt`, `java.io`, `java.lang` y `java.util`. El paquete `java.lang` incluye, entre otras, las clases `Integer`, `Math`, `String` y `System`. Algunas clases del paquete `java.util` son `Date`, `Random` y `StringTokenizer`. El paquete `java.io` se usa para la E/S e incluye las clases de canales vistas en la Sección 2.6. El paquete `java.applet` incluye clases usadas para diseñar applets. El paquete `java.awt` incluye clases que forman el Abstract Window Toolkit[6], que se usa para generar interfaces gráficas de usuario.

Una clase `C` en un paquete `P` se especifica como `P.C`. Por ejemplo, podemos tener un objeto `Date` construido con la fecha y hora actuales como estado inicial usando

```
Java.util.Date hoy = new java.util.Date( );
```

Observe que incluyendo un nombre de paquete evitamos conflictos con clases nombradas de forma idéntica en otros paquetes.

3.5.1 La directiva import

Usar un nombre completo con el paquete y la clase puede resultar pesado. Para evitar esto, usamos la directiva `import`.

```
import NombrePaquete.NombreClase;
import NombrePaquete.*;
```

Con la primera forma, se puede usar `NombreClase` como abreviatura de un nombre de clase totalmente cualificado. Con la segunda, todas las clases del paquete se pueden abreviar con el nombre de la clase correspondiente.

uñol: herramienta de ventanas abstractas. En lo sucesivo se usará sin traducir, puesto que sus siglas, awt, se usan exto, así como en el nombre del paquete.

Por ejemplo, con estas directivas `import`,

```
import java.util.Date;
import java.io.*;
```

podemos usar

```
Date hoy = new Date( );
FileReader elFichero = new FileReader( nombre );
```

El uso de las directivas `import` ahorra escritura. Y puesto que se ahorra más con la segunda forma, la verá usada muy a menudo. Sin embargo, las directivas `import` tienen dos desventajas. En primer lugar, la abreviatura hace difícil saber, a partir de la lectura del código, qué clase se está usando cuando hay varias directivas `import`. Además, la segunda forma puede permitir abreviaturas para clases que no se pretendía abreviar, e introducir conflictos de nombres que deberán ser resueltos escribiendo nombres de clase completamente cualificados.

Suponga que usamos

```
import java.util.* // Paquete de librería
import util.*;     // Paquete definido por el usuario
```

con la intención de importar la clase `java.util.Random` y un paquete que hemos escrito nosotros mismos. Entonces, si tuviéramos nuestra propia clase `Date`, la directiva `import` generaría un conflicto con `java.util.Date`, de forma que para referirnos a estas clases deberíamos utilizar los nombres totalmente cualificados. Y lo que es más, si estamos usando una clase de uno de estos paquetes, no sabremos si procede del paquete de librería o de nuestro propio paquete. Habríamos evitado ese problema si hubiéramos usado la primera forma:

```
import java.util.Random;
```

Las directivas `import` deben aparecer antes del comienzo de la declaración de una clase. Vimos un ejemplo en la Figura 2.11. Además, el paquete `java.lang` se importa de forma automática. Por ello podemos usar abreviaturas como `Math.max`, `Integer.parseInt`, `System.out`, etc.

3.5.2 La instrucción package

Para indicar que una clase es parte de un paquete, debemos hacer dos cosas. En primer lugar, debemos incluir la *instrucción* `package` como primera línea, antes de la definición de clase. En segundo lugar, debemos colocar el código en un subdirectorio apropiado.

En este libro, usamos los tres paquetes mostrados en la Figura 3.7. Otros programas, incluyendo los programas de prueba y las aplicaciones de la Parte III del libro, son clases aisladas que no forman parte de un paquete.

Paquete	Uso
`EstructurasDatos`	Clases que implementan estructuras de datos y ordenación.
`Excepciones`	Clases de excepciones.
`Soporte`	Varias clases de soporte e interfaces.

Figura 3.7 Paquetes definidos en este libro.

En la Figura 3.8 se muestra un ejemplo de cómo se usa la instrucción `package`. Tenemos un método estático `pausaLarga` que simplemente se duerme durante mil millones de segundos (aproximadamente dos semanas). Este método es útil porque cuando algunos entornos integrados ejecutan aplicaciones de consola desde dentro de sus entornos, cierran la salida de la consola tan pronto como termina el programa. Esto puede hacer difícil ver la salida. El método `pausaLarga` mantiene la consola sin cerrar en tales casos.

```
1  package Soporte;
2
3  public class Saliendo
4  {
5      // Suspende la ejecución del programa actual durante un tiempo
6      public static void pausaLarga( )
7      {
8          try
9              { Thread.sleep( 1000000000 ); }
10         catch( InterruptedException e ) { }
11     }
12 }
```

Figura 3.8 Una clase `Saliendo` con un único método estático, que forma parte del paquete `Soporte`.

3.5.3 La variable de entorno `CLASSPATH`

Los paquetes se buscan en los lugares indicados por la variable `CLASSPATH`. ¿Qué significa esto? A continuación se muestran dos posibles asignaciones de valores a la variable `CLASSPATH`, la primera para un sistema Windows 95, y la segunda para un sistema Unix:

```
SET CLASSPATH=.;C:\cafe\java\lib\
setenv CLASSPATH .:$HOME/java:/usr/java/lib
```

En ambos casos, la variable `CLASSPATH` lista los directorios (o ficheros comprimidos) que contienen los ficheros de las clases de los paquetes. Por ejemplo, si su variable `CLASSPATH` tiene un valor incorrecto, no podrá compilar ni siquiera el programa más trivial porque no se encontrará `java.lang`.

Una clase de un paquete `P` debe estar en un directorio `P`, el cual se encontrará buscando en la lista de `CLASSPATH`. El directorio actual (directorio `.`) siempre está en la variable `CLASSPATH`, luego si va a trabajar en un único directorio principal, simplemente puede crear subdirectorios en él. Lo más probable, sin embargo, es que quiera crear un directorio java separado y después crear subdirectorios en él. Entonces tendría que aumentar la variable `CLASSPATH` para incluir el directorio java. En la definición Unix previa se hace esto al añadir `$HOME/java` a `CLASSPATH`. Dentro del directorio java, cree subdirectorios llamados `Soporte`, `EstructurasDatos` y `Excepciones`. Coloque el código de la clase `Saliendo` en el subdirectorio `Soporte`.

Una aplicación puede usar entonces el método `pausaLarga` escribiendo

```
Soporte.pausaLarga( );
```

o simplemente usando `pausaLarga`, si se proporciona una directiva `import` apropiada.

3.5.4 Reglas de visibilidad amistosa dentro de un paquete

Los paquetes tienen varias reglas de visibilidad importantes. En primer lugar, si no se especifica ningún modificador de visibilidad para un atributo, entonces el atributo es *amistoso*. Esto significa que solamente es visible para las demás clases del mismo paquete. Este tipo de visibilidad es mayor que la de la parte privada (que es invisible incluso para otras clases dentro del mismo paquete) pero menor que la de la parte pública (que es visible también para clases que no pertenecen a ese paquete).

En segundo lugar, solamente las clases públicas de un paquete se pueden usar fuera del paquete. Ésta es la razón por la que siempre hemos usado el cualificador `public` delante de las clases. Las clases no se pueden declarar con la palabra `private`. El acceso amistoso se extiende a las clases también. Si una clase no se declara con la palabra `public`, entonces solamente las demás clases del paquete pueden acceder a ella; se trata de una *clase amistosa*. En la Parte IV veremos que las clases amistosas se pueden usar sin violar por ello los principios de ocultamiento de información. De hecho hay situaciones en las que las clases amistosas pueden sernos de mucha utilidad.

Todas la clases que no forman parte de un paquete, pero se pueden alcanzar a través de la variable `CLASSPATH`, se consideran parte de un mismo paquete por defecto. Como resultado el acceso amistoso se aplica entre todas ellas. Ésta es la razón por la que la visibilidad no queda afectada si se omite el modificador `public` en las clases que no pertenecen a ningún paquete. Sin embargo, este uso del acceso amistoso es bastante pobre. Evite el acceso amistoso excepto en casos como los que aparecen en algunas de las implementaciones de estructuras de datos de la Parte IV.

3.5.5 Compilación separada

Cuando un programa consta de varios ficheros `.java`, cada fichero se debe compilar por separado. Normalmente colocamos cada clase en su fichero `.java` correspondiente. Como resultado, obtenemos una colección de ficheros `.class`. Como se ha mencionado antes, cuando se escribe el comando *java* para ejecutar el intérprete, se llama al método `main` de la clase situada en el fichero al que se hace referencia en el comando. Los ficheros fuente en Java se pueden compilar en cualquier orden, siempre que los ficheros `.class` estén disponibles cuando se ejecute el intérprete.

3.6 Construcciones adicionales

Tres palabras clave adicionales son `this`, `instanceof` y `static`. La palabra `this` tiene varios usos en Java; dos de ellos se estudia en esta sección. La palabra `instanceof` también tiene varios usos; aquí la usaremos para asegurar que una conversión de tipo puede tener éxito. La palabra `static` tiene también varios usos. Ya hemos estudiado los *métodos estáticos*. Esta sección cubre los *atributos estáticos* y el *inicializador estático*.

3.6.1 La referencia `this`

El primer uso de `this` es como referencia al objeto actual. Piense en la referencia `this` como en un dispositivo de orientación que, en cualquier instante de tiempo, le dice dónde está. Un uso importante de la referencia `this` es el manejo del caso especial de autoasignación.

Un ejemplo de ello es un programa que copia un fichero en otro. Un algoritmo normal comienza borrando el fichero destino. Si no se realiza ninguna comprobación para asegurar que los ficheros fuente y destino son de hecho distintos, entonces podría borrarse el fichero fuente —difícilmente una característica deseable. Cuando se manejan dos objetos, de forma que sobre uno de ellos se escribe un valor y del otro se lee algún valor, deberíamos comprobar primero si nos encontramos en este caso especial, que se conoce como *aliasing*.

Como segundo ejemplo, supongamos que tenemos una clase `Cuenta` con un método `transferenciaFinal`. Este método mueve todo el dinero de una cuenta a otra. En principio, es una rutina sencilla de escribir:

```
// Transferir todo el dinero de lder a la cuenta actual
public void transferenciaFinal( Cuenta lder )
{
    dolares += lder.dolares;
    lder.dolares = 0;
}
```

Sin embargo, considere lo que sucede si hacemos:

```
Cuenta cuenta1;
Cuenta cuenta2;
  ...
cuenta2 = cuenta1;
cuenta1.transferenciaFinal( cuenta2 );
```

Puesto que estamos transfiriendo de una cuenta a ella misma, no debería haber cambio en dicha cuenta. Sin embargo, la última instrucción en `transferenciaFinal` provoca que la cuenta se vacíe. Una forma de evitar esto es usar un test de alias:

```
// Transferir todo el dinero de lder a la cuenta actual
public void transferenciaFinal( Cuenta lder )
{
    if( this == lder )    // Test de alias
        return;
    dolares += lder.dolares;
    lder.dolares = 0;
}
```

3.6.2 La abreviatura `this` para constructores

Muchas clases tienen varios constructores que se comportan de una forma similar. Podemos usar `this` dentro de un constructor para llamar a otro de los constructores de clase. Una alternativa al constructor de cero parámetros `Fecha` de la Figura 3.6 sería

```
public Fecha( )
{
    this( 1, 1, 1998); // Llama al constructor de tres parámetros
}
```

Son posibles usos más complicados, pero siempre la llamada a `this` debe ser la primera instrucción en el constructor; después puede aparecer otras instrucciones.

3.6.3 El operador `instanceof`

El operador `instanceof` lleva a cabo una comprobación en tiempo de ejecución. El resultado de

```
exp instanceof NombreClase
```

es `true` si `exp` es una instancia de `NombreClase`, y `false` en caso contrario. Si `exp` es `null`, el resultado es siempre `false`. El operador `instanceof` se usa normalmente para realizar una conversión de tipo y es `true` cuando ésta puede llevarse a cabo con éxito.

3.6.4 Atributos estáticos

Los atributos estáticos se usan cuando necesitamos que una variable esté compartida por todas las componentes de una clase. Normalmente se trata de una constante simbólica, pero no necesariamente. Cuando una variable se declara `static` en una clase, se crea una única instancia de la variable. No forma parte de ninguna instancia de la clase. En su lugar, se comporta como una variable global pero con el ámbito de la clase. En otras palabras, en la declaración

```
public class Ejemplo
{
    private int x;
    private static int y;
}
```

cada objeto `Ejemplo` almacena su propia `x`, pero hay solamente una `y` compartida.

Un uso común de un atributo estático es para representar una constante. Por ejemplo, la clase `Integer` define el valor `MAX_VALUE` como

```
public final static int MAX_VALUE = 2147483647;
```

Si una constante no fuera un atributo estático, entonces cada instancia de `Integer` tendría un atributo llamado `MAX_VALUE`, lo cual supondría un absurdo gasto de espacio. En lugar de ello, hay una única variable llamada `MAX_VALUE`. Cualquiera de los métodos de `Integer` puede acceder a ella usando el identificador `MAX_VALUE`. También se puede acceder a ella desde un objeto `Integer` usando `Obj.MAX_VALUE`, como con cualquier otro atributo. Observe que esto está permitido solamente porque `MAX_VALUE` es pública. Finalmente, se puede acceder a `MAX_VALUE` usando el nombre de la clase, como en `Integer.MAX_VALUE` (de nuevo permitido por ser pública). Esto no estaría permitido si se tratara de un atributo no estático.

También sin el cualificador `final`, los atributos estáticos son útiles. Supongamos que queremos llevar la cuenta del número de objetos `CeldaEntero` que hemos construido. Lo que necesitamos es una variable estática. Declaramos en la clase `CeldaEntero`

```
private static int instanciasActivas = 0;
```

Entonces podríamos incrementar `instanciasActivas` en el constructor. Sin el modificador `static`, tendríamos un comportamiento incorrecto, puesto que cada objeto `CeldaEntero` tendría su propio atributo, por lo que `instanciasActivas` se incrementaría de 0 a 1 en el constructor.

Observe que dado que un atributo estático no tiene objeto que lo controle, sino que es compartido por todas las instancias de la clase, cualquier método estático de la clase puede acceder a él y modificarlo, si la visibilidad lo permite. Un método estático solamente puede acceder a un atributo no estático, que es parte de cada instancia de la clase, si se proporciona un objeto controlador.

3.6.5 Inicializadores estáticos

Los atributos estáticos se inicializan cuando se carga la clase. A veces la inicialización es compleja. Por ejemplo, supongamos que necesitamos un vector estático que almacena las raíces cuadradas de los primeros 100 enteros. Sería deseable que esos valores se calcularan automáticamente. Una posibilidad es proporcionar un método estático y obligar al programador a llamarlo antes de usar el vector.

Una alternativa es usar un *inicializador estático*. En la Figura 3.9 se muestra un ejemplo, donde el inicializador se extiende de la línea 5 a la 9. La forma más sencilla de inicializador estático consiste en colocar el código de la inicialización de los atributos estáticos en un bloque precedido por la palabra clave `static`. El inicializador estático debe aparecer a continuación de la declaración del atributo estático.

```
1  public class Raices
2  {
3      private static double raicesCuadradas[ ] = new double[ 100 ];
4
5      static
6      {
7          for( int i = 0; i < raicesCuadradas.length; i++ )
8              raicesCuadradas[ i ] = Math.sqrt( ( double ) i );
9      }
10
11     // Resto de la clase
12 }
```

Figura 3.9 Ejemplo de un inicializador estático.

Resumen

En este capítulo se han descrito las clases y los paquetes de Java. La clase es el mecanismo de Java utilizado para crear nuevos tipos referencia; los paquetes se usan para agrupar clases similares. Para cada clase podemos

- definir la construcción de objetos,
- proporcionar ocultamiento de información y atomicidad, y
- definir métodos para manipular objetos.

Las clases constan de dos partes: la especificación y la implementación. La especificación le dice al usuario de la clase lo que hace la clase; la implementación lo hace. La implementación frecuentemente contiene código de propietario y en algunos casos se distribuye solamente como un fichero `.class`. Sin embargo, la especificación es de conocimiento público. En Java, usando *javadoc* se puede generar una especificación que lista los métodos de la clase a partir de la implementación.

El ocultamiento de información se puede lograr usando la directiva `private`. La inicialización de los objetos está controlada por las funciones constructoras, y las componentes del objeto se pueden examinar y modificar mediante funciones de acceso y funciones modificadoras, respectivamente. Otros métodos auxiliares, como `toString` y `equals`, también pueden ser escritos.

Las características estudiadas en este capítulo implementan los aspectos fundamentales de la programación orientada a objetos. En el siguiente capítulo se estudiará la herencia, la cual es otro de los pilares de la programación orientada a objetos.

Elementos del juego

acceso amistoso Las componentes que no tengan modificadores de visibilidad solamente podrán ser accedidas desde los métodos de las clases del mismo paquete.
aliasing Caso especial en el que el mismo objeto aparece con más de un rol.
atributo Componente de los objetos de una clase que almacena datos.
atributo estático Esencialmente es una variable global con ámbito de clase.
clase Consta de atributos y métodos que se aplican a instancias de la clase.
clase amistosa Clase no pública a la que solamente pueden acceder otras clases del mismo paquete.
compilación separada Cada clase se compila por separado, en cualquier orden.
constructor Nos dice cómo se declara e inicializa un objeto. El constructor por defecto es una inicialización por defecto componente a componente, tal que los atributos de tipo primitivo se inicializan a cero y los atributos referencia a `null`.
directiva `import` Se usa para proporcionar abreviaturas de nombres de clase totalmente cualificados.
encapsulación Agrupación de datos y operaciones que se aplican sobre ellos para formar un agregado, a la vez que se oculta la implementación del agregado.
especificación de una clase Describe su funcionalidad, pero no la implementación.
implementación Representa los detalles internos de cómo se satisfacen las especificaciones. En lo que respecta al usuario, la implementación no es importante.
inicializador estático Bloque de código que se usa para inicializar los atributos estáticos.
instrucción `package` Indica que una clase es miembro de un paquete. Debe preceder a la definición de la clase.
javadoc Genera automáticamente documentación para las clases.
llamada al constructor `this` Usado para hacer una llamada a otro constructor de la misma clase.
marca *javadoc* Entre ellas tenemos `@author`, `@param`, `@return` y `@exception`. Se usan dentro de comentarios *javadoc*.
método Función proporcionada como componente adicional que, si no es estática, opera sobre instancias de la clase.
método de acceso Método que examina un objeto pero no cambia su estado.
método `equals` Se puede implementar para comprobar si dos referencias describen el mismo valor. El parámetro formal es siempre de tipo `Object`.
método `toString` Devuelve un valor del tipo `String` basado en el estado del objeto.
modificador Método que cambia el estado de un objeto.
objeto Entidad con estructura y estado, para el que están definidas operaciones que pueden acceder o manipular el estado.
ocultamiento de información Convierte en inaccesibles los detalles de implementación, incluyendo las componentes de un objeto.
operador `instanceof` Comprueba si una expresión es una instancia de una clase.
paquete Usado para organizar una colección de clases.
privado Componente no visible para los métodos que no pertenecen a la clase.
programación basada en objetos Usa las características de la encapsulación y el ocultamiento de información de los objetos pero no usa la herencia.
programación orientada a objetos Se distingue de la programación basada en objetos en que hace uso de la herencia para formar jerarquías de clases.
público Componente que es visible para los métodos que no pertenecen a la clase.
referencia `this` Referencia al objeto actual. Se puede usar para enviar el objeto actual, como un todo, a otro método.
unidad atómica En relación a un objeto, sus partes no pueden ser diseccionadas por los usuarios generales del objeto.
variable `CLASSPATH` Especifica los directorios y ficheros donde se debe buscar para encontrar las clases.

Errores comunes

1. No se puede acceder a las componentes privadas desde fuera de la clase. Recuerde que, por defecto, las componentes de la clase son amistosas, por lo que son visibles solamente dentro del paquete.
2. Use `public class` en lugar de `class` a menos que esté escribiendo una clase de ayuda que luego vaya a deshechar.
3. El parámetro formal de `equals` debe ser del tipo `Object`. En caso contrario, aunque el programa se compilaría correctamente, hay casos en los que se usará un `equals` por defecto (que siempre devuelve `false`).
4. Los métodos estáticos no pueden acceder a componentes no estáticas que no tengan un objeto controlador.
5. Un error común es olvidar el `.*` en la segunda forma de escribir una directiva `import`.
6. Las clases que forman parte de un paquete deben colocarse en un directorio que se llame igual que el paquete y que debe poderse alcanzar a través de la variable `CLASSPATH`.
7. `this` es una referencia final y no se puede modificar.

En Internet

Los siguientes ficheros están disponibles:

Date.java	Traducido por `Fecha.java`, contiene la clase `Date`, traducida como `Fecha` en la Figura 3.6. Se encuentra en el directorio **Chapter03**.
Exiting.java	Traducido por `Saliendo.java`, contiene el método `longPause`, traducido por `pausaLarga` en la Figura 3.8. Se encuentra en el directorio **Supporting**, traducido por `Soporte`.
IntCell.java	Traducido por `CeldaEntero.java`, contiene la clase `IntCell`, traducida por `CeldaEntero` en la Figura 3.4. Se encuentra en el directorio **Chapter03**. También se encuentra allí la salida producida por *javadoc*.
Squares.java	Traducida por `Raices.java`, contiene el inicializador estático de la Figura 3.9. Se encuentra en el directorio **Chapter03**.

TestIntCell.java Traducido por `TestCeldaEntero.java`, contiene un `main` que prueba `IntCell` (`CeldaEntero`) que se muestra en la Figura 3.3. Se encuentra en el directorio **Chapter03**.

Ejercicios

Cuestiones breves

3.1. ¿Qué es el ocultamiento de información? ¿Qué es la encapsulación? ¿Cómo soporta Java estos conceptos?

3.2. Explique las secciones pública y privada de una clase.

3.3. Describa la función del constructor.

3.4. ¿Qué sucede si una clase no proporciona constructores?

3.5. ¿Qué es el acceso amistoso en un paquete?

3.6. ¿Cómo se lleva a cabo la salida en una clase `NombreClase`?

3.7. Describa las dos formas de escribir una directiva `import` que permitan que la clase `pausaLarga` sea usada sin proporcionar el nombre del paquete `Soporte`.

Problemas teóricos

3.8. ¿Por qué no se puede declarar como privadas las clases?

3.9. Supongamos que el método `main` de la Figura 3.3 formaba parte de la clase `CeldaEntero`.

a) ¿Todavía funcionaría el programa?

b) ¿Se podría quitar el comentario a la línea comentada sin generar un error?

Problemas prácticos

3.10. Un cerrojo con combinación tiene las siguientes propiedades básicas: la combinación (una secuencia de tres números) está oculta; el cerrojo se puede abrir proporcionando la combinación; y la combinación se puede cambiar, pero solamente por alguien que conoce la combinación actual. Diseñe una clase con métodos públicos `abrir` y `cambiarComb`, y atributos privados para almacenar la combinación. La combinación debería asignarse en el constructor. Deshabilite la copia de cerrojos con combinación.

Prácticas de programación

3.11. Escriba una clase que implemente los números racionales. Sus atributos serán dos variables de tipo `long` que almacenarán el numerador y denominador. Almacene los números racionales en forma simplificada, con el denominador siempre no negativo. Proporcione un conjunto razonable de constructores, los métodos `sumar`, `restar`, `multiplicar` y `dividir`, así como `toString`, `equals` y `compareTo` (que se comporta como el de la clase `String`). Asegúrese de que `toString` trata correctamente el caso en que el denominador es cero.

3.12. Implemente una clase `Fecha` simple. Debería poder representar cualquier fecha desde el 1 de enero de 1.800 al 31 de diciembre del 2.500; restar dos fechas; incrementar una fecha en un número de días; y comparar dos fechas usando `equals` y `compareTo`. Una `Fecha` se representa internamente como el número de días a partir de una cierta fecha de inicio, que aquí es el comienzo de 1.800. Esto convierte en triviales todos los métodos excepto `toString`.

La regla para los años bisiestos es que un año es bisiesto si es divisible por cuatro y no es divisible por 100 a menos que también lo sea por 400. Por tanto 1.800, 1.900 y 2.100 no son años bisiestos, pero el 2.000 sí lo es. El constructor debe comprobar la validez de la fecha, al igual que debe hacerlo `toString`. La `Fecha` podría no ser válida si un incremento o decremento provocan que se salga fuera del rango.

Una vez ha decidido la especificación, puede hacer una implementación. La parte difícil es la conversión entre las representaciones interna y externa de la fecha. A continuación se explica un posible algoritmo.

Cree dos atributos estáticos de tipo vector. El primer vector, `diasHastaPrimeroDeMes`, contendrá el número de días hasta el primero de cada mes en un año no bisiesto. Luego contiene 0, 31, 59, 90 y así sucesivamente. El segundo vector, `diasHastalEnero`, contendrá el número de días hasta el primero de cada año, comenzando con `primerAnyo`. Luego contiene 0, 365, 730, 1.095, 1.460, 1.826 y así sucesivamente porque 1.800 no es un año bisiesto, pero 1.804 sí lo es. Su programa debería inicializar este vector usando un inicializador estático. Entonces puede usar el vector para convertir la representación interna en la externa.

3.13. Implemente una clase `Complejos` de números complejos. Recuerde que un número complejo consta de una parte real y una imaginaria. Soporte las mismas operaciones que la clase de los racionales, cuando tengan sentido (por ejemplo `compareTo` no tiene sentido). Añada métodos de acceso para extraer las partes real e imaginaria.

3.14. Implemente una clase `TipoEntero` que soporte un conjunto razonable de constructores, `sumar`, `restar`, `multiplicar`, `dividir`, `equals`, `compareTo` y `toString`. Mantenga `TipoEntero` como un vector lo suficientemente largo. Para esta clase, la operación difícil es la división, seguida de cerca por la multiplicación.

Bibliografía

Se puede encontrar más información en las referencias, al final del Capítulo 1.

CAPÍTULO

4 Herencia

Tal y como se ha comentado en el Capítulo 3, un objetivo importante de la programación orientada a objetos es la reutilización de código. Igual que los ingenieros emplean repetidamente en sus diseños las mismas componentes, los programadores deberían ser capaces de reutilizar objetos, en lugar de implementarlos repetidamente. En un lenguaje de programación orientado a objetos, el mecanismo básico de reutilización de código es la *herencia*. La herencia permite extender la funcionalidad de un objeto. En otras palabras, podemos definir nuevos tipos restringiendo (o aumentando) propiedades del tipo original, generándose así una jerarquía de clases. La herencia es también el mecanismo que Java emplea para implementar métodos y clases genéricos.

En este capítulo veremos:

- Los principios generales de la herencia, incluyendo el *polimorfismo*.
- Cómo se implementa la herencia en Java.
- Cómo puede derivarse una colección de clases a partir de una única clase abstracta.
- La *interfaz*, que es un tipo especial de clase.
- Cómo implementa Java la programación genérica usando herencia.

4.1 ¿Qué es la herencia?

La *herencia* es el principio básico de la programación orientada a objetos empleado para reutilizar código entre clases relacionadas. La herencia modela las *relaciones ES-UN(A)*. En una relación ES-UN(A), decimos que una clase derivada *es una* variación de la clase base. Por ejemplo, un Círculo ES-UN(A) Figura y un Coche ES-UN(A) Vehículo. Sin embargo, no se tiene que una Elipse ES-UN(A) Círculo. Las relaciones de herencia forman *jerarquías*. Por ejemplo, podemos extender Coche formando otras clases, ya que CocheExtranjero ES-UN(A) Coche (y paga impuestos) y CocheNacional ES-UN(A) Coche (y no paga impuestos).

Otro tipo de relaciones son las relaciones *CONTIENE*. Esta clase de relaciones no posee las propiedades asociadas a la jerarquía de herencia. Un ejemplo de relación CONTIENE es que un Coche CONTIENE Volante. Las relaciones CONTIENE no deben ser modeladas a través de la herencia. Por el contrario, deben implementarse mediante la técnica de la *composición*, en la que las componentes son sencillamente atributos de datos privados.

El lenguaje Java emplea extensamente la herencia al implementar sus librerías de clases. Dos ejemplos son las *excepciones* y las *componentes* en el *Abstract Window Toolkit*:

- *Excepciones*. Java define la clase `Exception`. Como ya hemos visto, existe una gran variedad de excepciones incluyendo `NullPointerException` y `ArrayOutOfBoundsException`. Cada una es una clase distinta, pero sobre todas ellas pueden aplicarse los métodos `toString` y `printStackTrace` para ayudarnos en la depuración.
- *Componentes*. El *Abstract Window Toolkit* (descrito con más detalle en el Apéndice D) define un objeto conocido como `Component`. Existen muchos tipos de `Component`es, incluyendo `Button`, `Canvas`, `Checkbox`, `List` y `TextField`. Cualquiera de ellas puede añadirse a un `Frame`, `Panel` o `Window` (que a su vez son `Component`es).

En ambos casos, la herencia modela una relación ES-UN(A). Un `Button` ES-UN(A) `Component`. Una `NullPointerException` ES-UN(A) `Exception`. Debido a las características de la relación ES-UN(A), la propiedad fundamental de la herencia garantiza que cualquier método ejecutable por `Exception` puede ser ejecutado también por `NullPointerException`, y más aún, un objeto `NullPointerException` puede estar referenciado por una variable `Exception`. (Observe que lo contrario no es cierto.) Entonces, como `printStackTrace` es un método disponible en la clase `Exception`, podemos escribir

```
catch( Exception e ) { e.printStackTrace( ); }
```

Si `e` referencia a un objeto `NullPointerException`, `e.printStackTrace` tiene sentido. Dependiendo de las circunstancias de la jerarquía de clases que se tenga, el método `printStackTrace` puede permanecer igual o *especializarse* en cada clase. Cuando un método permanece invariante a lo largo de la jerarquía, en el sentido de tener la misma funcionalidad para todas las clases en la jerarquía, evitamos tener que reescribir una implementación del método para cada clase.

La llamada a `printStackTrace` muestra también un importante principio de la programación orientada a objetos conocido como *polimorfismo*. Una variable referencia polimórfica puede referenciar objetos de distintos tipos. Cuando una operación ha de aplicarse a una referencia tal, se selecciona automáticamente la más adecuada al objeto referenciado en ese momento. En Java todos los tipos referencia son polimórficos. En el caso de una referencia de tipo `Exception` se toma una decisión en tiempo de ejecución: el método `printStackTrace` del objeto que `e` referencia en tiempo de ejecución es el empleado. Esto es conocido como *tipado dinámico*.

En la *herencia*, se tiene una *clase base* de la que derivamos el resto de clases. La clase base es la clase sobre la que se basa la herencia. Una *clase derivada* hereda todas las propiedades de la clase base, de modo que todos los métodos públicos disponibles en la clase base se convierten en métodos públicos, con implementaciones idénticas en la clase derivada. Además pueden añadirse atributos y métodos adicionales y cambiar el significado de los métodos heredados. Cada clase derivada es una clase completamente nueva. Por otra parte, los cambios realizados en las clases derivadas no afectan en absoluto a la clase base. De este modo, el diseño de clases derivadas no afecta a la clase base, lo que simplifica notablemente el mantenimiento del software.

Una clase derivada es de tipo compatible con la clase base, de modo que una variable referencia de la clase base puede referenciar a un objeto de la clase derivada, pero no al revés. Las clases hermanas (es decir, las clases derivadas de una clase común) no son de tipos compatibles.

Como hemos mencionado anteriormente, el uso de la herencia genera una jerarquía de clases. La Figura 4.1 muestra una pequeña parte de la jerarquía `Exception`. Nótese que `NullPointerException` se deriva indirectamente de `Exception`. Este hecho es transparente al usuario de las clases ya que las relaciones ES-UN(A) son transitivas. En otras palabras, si *X* ES-UN(A) *Y* e *Y* ES-UN(A) *Z*, entonces *X* ES-UN(A) *Z*. La jerarquía `Exception` muestra las decisiones de diseño usuales en las que los rasgos comunes se reúnen en las clases base y la especialización se hace en las clases derivadas. En esta jerarquía se dice que la clase derivada es una *subclase* de la clase base y que la clase base es una *superclase* de la clase derivada. Estas relaciones son transitivas. Por otra parte, el operador `instanceof` puede emplearse con las subclases. Así, si `obj` es de tipo `X` (y no `null`) entonces `obj instanceof Z` es `true`.

En las siguientes secciones responderemos a las siguientes cuestiones:

- ¿Cuál es la sintaxis adecuada para derivar una clase de otra clase base ya existente?
- ¿Cómo afecta esto a lo público y lo privado?
- ¿Cómo se especifica que un método no se redefine a lo largo de la jerarquía de clases?
- ¿Cómo se especializa un método?
- ¿Cómo podemos eliminar las diferencias generando una clase abstracta para crear después una jerarquía?
- ¿Se puede derivar una nueva clase a partir de varias clases (*herencia múltiple*)?
- ¿Cómo se emplea la herencia para implementar la genericidad?

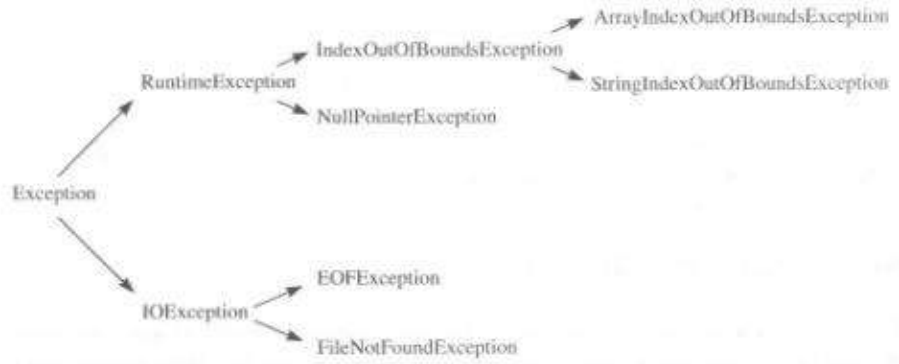


Figura 4.1 Parte de la jerarquía `Exception`.

Algunas de estas cuestiones se ilustran en la implementación de la clase `Figura` (en las Secciones 4.2.5 y 4.3) de la que se derivan `Circulo`, `Cuadrado` y `Rectangulo`. En este ejemplo puede observarse cómo Java implementa el polimorfismo en tiempo de ejecución, y comprobar cómo puede emplearse la herencia para implementar métodos genéricos.

4.2 Sintaxis básica de Java

Recuerde que una clase derivada hereda todas las propiedades de cada una de sus clases base. Después podemos añadir nuevos atributos, sobrescribir métodos y añadir nuevos métodos. Cada clase derivada es una clase completamente nueva. En la Figura 4.2 se muestra un esqueleto típico de herencia, en el que se usa la *cláusula* `extends`. Una cláusula `extends` indica que una clase se deriva de otra. Una clase derivada *extiende* una clase base. Presentamos a continuación una breve descripción genérica de una clase derivada:

- Generalmente todos los datos son privados, así que al añadir nuevos atributos a la clase derivada debemos especificarlos en la sección privada.
- Cualquier método de una clase base que no se especifique en la clase derivada es heredado sin modificación alguna, a excepción del constructor. Este caso especial se detalla en la Sección 4.2.2.
- Cualquier método de una clase base que se defina de nuevo en la sección pública de la clase derivada se sobrescribe. La nueva definición se aplicará a los objetos de la clase derivada.
- Los métodos públicos de la clase base no deben redefinirse en la sección privada de la clase derivada.
- En la clase derivada puede añadirse métodos adicionales.

```
1  public class Derivada extends Base
2  {
3      // Cualquier componente no listada se hereda sin cambios
4      // excepto el constructor
5
6          // componentes públicos
7      // Constructor(es) si el constructor por defecto no es válido
8      // Métodos de la clase base redefinidos
9      // Métodos públicos adicionales
10
11         // componentes privados
12     // Atributos adicionales (generalmente privados)
13     // Métodos privados adicionales
14 }
```

Figura 4.2 Esquema general de la herencia.

4.2.1 Reglas de visibilidad

Ya sabemos que cualquier componente privada es accesible sólo para los métodos de su clase. De este modo, las componentes privadas de la clase base no son accesibles desde la clase derivada.

Es posible que en ocasiones necesitemos que la clase derivada tenga acceso a algunas componentes de su clase base. Existen dos formas de lograrlo. La primera consiste en emplear el acceso público o el acceso amistoso. Sin embargo, el acceso público permite acceder a otras clases además de a la clase derivada, y el acceso amistoso sólo funciona si ambas clases están en el mismo paquete.

Si deseamos permitir el acceso sólo a las clases derivadas, podemos declarar componentes protegidas. Una *componente protegida* de una clase es privada para cualquier clase salvo para las clases derivadas (y las clases que se encuentren en el mismo paquete). El declarar componentes como públicas o protegidas viola el espíritu de la encapsulación y el ocultamiento de información, por lo que sólo se debería admitir como licencia. Una mejor alternativa es escribir métodos de modificación o de acceso para aprovechar el acceso amistoso. Sin embargo, si una declaración protegida permite evitar un código complicado, es razonable emplearla. En este texto, utilizaremos en ocasiones atributos protegidos por esta razón. El empleo de métodos protegidos es también una práctica habitual en el texto. Esto permite que una clase derivada herede un método interno sin convertirlo en accesible fuera de la jerarquía de paquetes.

4.2.2 El constructor y `super`

Cada clase derivada debe definir sus propios constructores. Si no se implementa ninguno entonces se genera un constructor sin parámetros por defecto. Este constructor invocará al constructor sin parámetros de la clase base para inicializar la parte heredada, y se empleará la inicialización por defecto sobre los nuevos atributos (o sea, utilizará 0 para los tipos primitivos y `null` para los tipos referencia).

En la práctica es habitual construir una clase derivada construyendo primero la parte heredada. De hecho, esto se hace así siempre por defecto, incluso cuando se tiene un constructor para la clase derivada. Esto es lógico si se tiene en cuenta que la encapsulación considera la porción heredada como una sola entidad, y el constructor de la clase base indica como inicializar dicha entidad.

Los constructores de la clase base pueden ser llamados explícitamente empleando el método `super`. Así, el constructor por defecto de una clase derivada es en realidad

```
public Derivada( )
{
    super( );
}
```

La llamada al método `super` puede realizarse con los parámetros que se ajusten al constructor de la clase base. Por ejemplo, la Figura 4.3 muestra la clase `DesbordamientoInferior` empleada en la implementación de las estructuras de datos. `DesbordamientoInferior` se lanza cuando se intenta obtener información de una estructura de datos vacía. Los objetos de tipo `DesbordamientoInferior` se crean especificando un `String`. `DesbordamientoInferior` no incorpora nuevos atributos, por lo que la construcción de objetos de dicha subclase se reduce a construir el fragmento heredado, empleando el constructor de `Exception`.

```
1 package Excepciones;
2
3 public class DesbordamientoInferior extends Exception
4 {
5     public DesbordamientoInferior( String lanzador )
6     {
7         super( lanzador );
8     }
9 }
```

Figura 4.3 Constructor para una nueva excepción `DesbordamientoInferior` que emplea `super`.

El método `super` sólo puede emplearse como primera línea de un constructor. Si éste no aparece se genera de forma automática una llamada sin parámetros a `super`.

4.2.3 Métodos y clases finales

Como se ha descrito anteriormente, la clase derivada o bien sobrescribe o bien deja intactos los métodos de la clase base. En muchos casos, está claro que un determinado método de la clase base debe permanecer invariante a lo largo de la jerarquía de clases, esto es, que ninguna clase derivada debe redefinirlo. En este caso podemos declarar que el método es *final*, de modo que no podrá sobrescribirse.

La declaración de un método invariante como `final` no es sólo una buena costumbre de programación, sino que puede conducirnos a código más eficiente. Es una buena costumbre de programación, porque además de declarar nuestras intenciones al lector del programa y de la documentación, se evita la redefinición accidental de un método que no debe ser rescrito.

Para comprobar por qué el uso de `final` puede generar código más eficiente, suponga que una clase base `Base` declara un método final `f` y que `Derivada` extiende a la clase `Base`.

Considere la rutina

```
void hazlo( Base obj )
{
    obj.f( );
}
```

Como `f` es un método final, no importa si `obj` referencia a un objeto de `Base` o `Derivada`; la definición de `f` es invariante, de modo que sabremos lo que hará `f`. Como consecuencia, la llamada al método se resuelve en tiempo de compilación en lugar de en tiempo de ejecución. Esto se conoce por *tipado estático*. El programa se ejecutará más rápido, ya que los tipos se determinan durante la compilación y no durante la ejecución. En qué grado esto es importante depende de cuántas veces evitemos tomar decisiones durante la ejecución del programa.

Un corolario de esta observación es que si `f` es un método trivial, tal como un mero acceso a un atributo, y se declara como `final`, el compilador puede reemplazar la llamada a `f` por su definición. Así, la llamada al método será sustituida por una sola línea que realiza el acceso al atributo, ganando tiempo. Si `f` no se declara como `final` esto es imposible, ya que `obj` podría referenciar a un objeto de la clase derivada, para el que la definición de `f` sería diferente. Los métodos estáticos no necesitan de objeto controlador y sus llamadas se resuelven en tiempo de compilación mediante el tipado estático.

Las *clases finales* son similares a los métodos finales. Una clase final no puede ser extendida. Como consecuencia, todos sus métodos son automáticamente métodos finales. Por ejemplo, la clase `Integer` es una clase final. Nótese que el hecho de que una clase tenga métodos finales no implica que la clase sea también final. Las clases finales se conocen también por *clases hoja* ya que en la jerarquía de la herencia, que se asemeja a un árbol, las clases finales están en los bordes, como las hojas.

4.2.4 Sobrescribiendo un método

Los métodos de la clase base se sobrescriben en la clase derivada con sólo definir en la clase derivada un método con la misma signatura. El método de la clase derivada debe tener el mismo tipo del retorno y no puede añadir excepciones en la lista `throws`.

En algunas ocasiones los métodos de la clase derivada necesitan invocar a los métodos de la clase base. Esto se conoce con el nombre de *sobrescritura parcial*. Esto es, deseamos hacer lo mismo que la clase base y un poco más, en lugar de hacerlo todo de manera completamente diferente. Las llamadas a un método de la clase base se hacen a través de `super`. Aquí se muestra un ejemplo:

```
public class TrabajadorAdicto extends Trabajador
{
    public trabaja( )
    {
        super.trabaja( ); // Trabaja como un Trabajador
        bebeCafe( );      // Se toma un descanso
        super.trabaja( ); // Trabaja como un Trabajador un poco más
    }
}
```

4.2.5 Métodos y clases abstractos

Ya hemos visto que algunos métodos permanecen invariantes a lo largo de la jerarquía de clases (éstos son los métodos finales), y que otros pueden ver modificado su significado en la jerarquía. Una tercera posibilidad es que el método tenga sentido para las clases derivadas y que deba implementarse en ellas, mientras que su implementación no tiene sentido en la clase base. En este caso podemos declarar el método en la clase base como *abstracto* mediante la palabra reservada *abstract*.

Un *método abstracto* es un método que declara una funcionalidad que debe ser implementada en todas las clases derivadas de la clase en que se vaya a utilizar. En otras palabras, a través de un método tal indicamos lo que deberían poder hacer los objetos de dichas subclases. No se implementa en la clase base, sino que cada subclase debe tener su propia implementación.

Una clase que tiene al menos un método abstracto es una *clase abstracta*. En Java se exige que todas las clases abstractas sean declaradas como tales. Cuando en una clase derivada no se (re)define un método abstracto mediante una implementación, el método continúa siéndolo en la clase derivada. Como resultado, si en una clase no definida como abstracta no se implementa un método abstracto, el compilador detectará esa inconsistencia y devolverá un error.

Un ejemplo viene dado por la clase abstracta `Figura`, que se empleará en un ejemplo mayor de este capítulo, un poco más tarde. Las figuras concretas, como `Circulo` o `Rectangulo`, se derivan de `Figura`. `Cuadrado` puede derivarse como un `Rectangulo` especial. La Figura 4.4 ilustra la jerarquía resultante.

La clase `Figura` puede contener atributos comunes a todas las clases. En un ejemplo más concreto podrían incluirse como tales las coordenadas de los extremos de los objetos. La citada clase da la definición de algunos métodos, como `posicionDe`, que son independientes del tipo del objeto referenciado en cada momento; `posicionDe` será entonces un método final. También se declaran métodos que se aplican a cada tipo concreto de objetos. Algunos de estos métodos podrían ser implementados genéricamente en la clase abstracta `Figura`. Por lo que, por ejemplo, es imposible indicar cómo calcular el área de un objeto abstracto; el método `area` debería ser declarado abstracto mediante `abstract`.

Como ya hemos mencionado anteriormente, la existencia de un método abstracto convierte a la clase en abstracta, no permitiéndose la creación de objetos de dicha clase. De este modo, no podemos construir un objeto de la clase `Figura`; sólo podemos crear objetos de las clases derivadas. Sin embargo, como de costumbre, una `Figura` puede referenciar cualquier objeto concreto de las clases derivadas, como `Circulo` o `Rectangulo`. Así,

```
Figura a, b;
a = new Circulo( 3.0 );         // Correcto
b = new Figura ( "Circulo" );   // Incorrecto
```

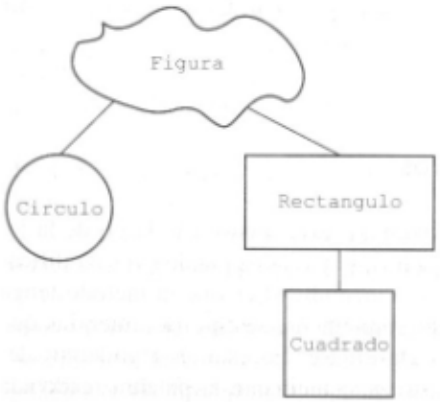


Figura 4.4 La jerarquía de las figuras.

```
1  // Clase base abstracta para las figuras
2  //
3  // CONSTRUCCIÓN: no se permite. Figura es un constructor abstracto
4  //     con un solo parámetro invocado por las clases derivadas
5  //
6  // ******************OPERACIONES PÚBLICAS*******************
7  // double area( )           --> Devuelve el área (abstracto)
8  // boolean menorQue( lder ) --> Compara el área de dos objetos
9  // String toString( )       --> Método toString usual
10
11 public abstract class Figura
12 {
13     abstract public double area( );
14
15     public Figura( String nombreFigura )
16     { nombre = nombreFigura; }
17
18     final public boolean menorQue( Figura lder )
19     { return area( ) < lder.area( ); }
20
21     final public String toString( )
22     { return nombre + " con área " + area( ); }
23
24     private String nombre;
25 }
```

Figura 4.5 La clase abstracta `Figura`.

La Figura 4.5 muestra la clase abstracta `Figura`. En la línea 24 se declara un `String` que almacena el tipo de la figura. Esto se emplea únicamente en las clases derivadas. Este atributo es privado, así que las clases derivadas no tienen acceso directo a él. El resto de la clase especifica una colección de métodos.

El constructor nunca se invoca directamente, ya que `Figura` es una clase abstracta. Sin embargo, necesitamos un constructor, al que la clase derivada invocará cuando necesite inicializar los componentes privados. El constructor de `Figura` inicializa el atributo interno `nombre`.

La línea 13 declara el método abstracto `area`. En tiempo de ejecución se decidirá el método `area` adecuado de la correspondiente clase derivada. `area` es un método abstracto, ya que no tiene ningún significado por defecto que pueda ser especificado para aplicarse sobre una clase heredada que opte por no implementarlo.

El método de comparación de figuras implementado en las líneas 18 y 19, no es abstracto ya que puede ser aplicado en todas las clases derivadas. De hecho, su definición es invariante a lo largo de la jerarquía de figuras, así que lo hemos convertido en un método final.

El método `toString`, mostrado en las líneas 21 y 22, imprime el nombre de una figura y su área. Al igual que el método de comparación, permanece invariante a lo largo de la jerarquía completa, y se declara como final.

Antes de continuar, vamos a resumir los distintos tipos de métodos existentes:

1. *Métodos finales*. La sobrecarga se resuelve en tiempo de compilación. Empleamos un método final sólo cuando dicho método es invariante a lo largo de la jerarquía de herencia (esto es, cuando el método no se redefine nunca).

2. *Métodos abstractos*. La sobrecarga se resuelve durante la ejecución del programa. La clase base no implementa estos métodos y es abstracta a su vez. Es necesario que las clases derivadas implementen dichos métodos o sean también abstractas.
3. *Métodos estáticos*. La sobrecarga se resuelve durante la compilación ya que no existe ningún objeto controlador.
4. *Otros métodos*. La sobrecarga se resuelve en tiempo de ejecución. La clase base tiene una implementación por defecto que puede ser, o bien sobrescrita en las clases derivadas, o permanecer sin cambios en ellas.

4.3 Ejemplo: extensión de la clase `Figura`

En esta sección implementamos las clases derivadas de `Figura` y mostramos cómo se emplean de manera polimórfica. Para ello consideramos el siguiente problema:

ORDENANDO FIGURAS
Léanse los datos de N figuras (círculos, cuadrados o rectángulos) y muéstrense ordenadas por área.

La implementación de las clases derivadas, mostrada en la Figura 4.6, es muy sencilla y no incluye nada que no haya sido comentado anteriormente. El único punto novedoso es que `Cuadrado` se define a partir de `Rectangulo`, que se deriva a su vez de `Figura`. Esta derivación se realiza exactamente igual que cualquier otra. En la implementación de estas clases debemos hacer lo siguiente:

1. Definir un nuevo constructor.
2. Estudiar cada método que no sea ni abstracto ni final para decidir si se acepta su definición por defecto. De no ser así, debemos escribir para él una nueva definición.
3. Escribir una definición para cada método abstracto.
4. Si es necesario, definir métodos adicionales.

Para cada clase hemos definido un sencillo constructor en el que se inicializan las dimensiones básicas (el radio para los círculos, y las longitudes de los lados para cuadrados y rectángulos). Primero se inicializa la parte heredada empleando `super`. En cada clase debe definirse un método `area` ya que en `Figura` aparece como método abstracto. Si el método `area` no es implementado en alguna de las clases derivadas se producirá un error durante la compilación. Esto es debido a que si falta la implementación de `area`, la clase derivada también debería declararse como abstracta. Nótese que `Cuadrado` hereda el método `area` de `Rectangulo`, por lo que no es necesario implementarlo de nuevo.

Ahora que hemos escrito las clases, ya podemos resolver el problema propuesto. Para ello empleamos un vector de `Figuras`. Es importante observar que dicho vector no almacenará directamente `Figuras` (lo que no es posible) sino que guardará referencias de tipo `Figura`. Estas referencias pueden apuntar a `Circulos`, `Cuadrados` y `Rectangulos`.

```
1  // Clases Circulo, Cuadrado y Rectangulo
2  //     basadas todas en Figura
3  //
4  // CONSTRUCCIÓN: con un radio (para el círculo), longitud del lado
5  //     (para el cuadrado), y largo y ancho (para el rectángulo).
6  // ******************OPERACIONES PÚBLICAS*******************
7  // double area( ) --> Implementa el método area abstracto en Figura
8
9  public class Circulo extends Figura
10 {
11     public Circulo( double r )
12     {
13         super( "Círculo" );
14         radio = r;
15     }
16
17     public double area( )
18     {
19         return PI * radio * radio;
20     }
21
22     private static final double PI = 3.14159265358979323;
23     private double radio;
24 }
25
26 public class Rectangulo extends Figura
27 {
28     public Rectangulo( double largo, double ancho )
29     {
30         super( "Rectangulo" );
31         base = largo;
32         alt = ancho;
33     }
34
35     public double area( )
36     {
37         return base * alt;
38     }
39
40     private double base;
41     private double alt;
42 }
43
44 public class Cuadrado extends Rectangulo
45 {
46     public Cuadrado( double lado )
47     {
48     super( lado, lado );
49     }
50 }
```

Figura 4.6 Clases `Circulo`, `Cuadrado` y `Rectangulo`, situadas en ficheros distintos.

La Figura 4.7 muestra este enfoque del problema. En primer lugar, se realiza la lectura de los objetos. En la línea 23, la llamada a `leerFigura` consiste en leer un carácter y las dimensiones de una figura, para crear después un objeto de tipo

```
1  import java.io.*;
2
3  class TestFigura
4  {
5      private static BufferedReader in;
6
7      private static Figura leer Figura( )
8        { /* Implementación en la Figura 4.8 */
9
10     public static void main( String [ ] args )
11     {
12         try
13         {
14               // Lectura del número de figuras
15             System.out.println( "Introduzca # de figuras: ");
16             in = new BufferedReader( new
17                                 InputStreamReader( System.in ) );
18             int numFiguras = Integer.parseInt( in.readLine( ) );
19
20               // Lectura de las figuras
21             Figura [ ] vector = new Figura [ numFiguras ];
22             for( int i = 0; i < numFiguras; i++ )
23                vector[ i ] = leerFigura( );
24
25               // Ordenación y Salida
26             ordenacionPorInsercion( vector );
27             System.out.println( "Ordenados por area" );
28             for ( int i = 0; i < numFiguras; i++)
29                 System.out.println( vector[ i ] );
30         }
31         catch( Exception e )
32         { System.out.println( e ); }
33     }
34 }
```

Figura 4.7 Rutina `main` para leer figuras y mostrarlas en orden creciente de áreas.

`Figura`. La Figura 4.8 muestra una implementación directa de ello. Nótese que si se produce una entrada de datos incorrectos, se crea un círculo de radio cero y se devuelve una referencia apuntando hacia él. Una solución más elegante en este caso sería definir y lanzar una excepción. Haga esto como Ejercicio 4.14.

A continuación, cada objeto creado en `leerFigura` se referencia a través de una posición del vector. Después la llamada a `ordenacionPorInsercion` ordena las figuras. Finalmente, se muestra por pantalla el vector de `Figuras`, llamando implícitamente al método `toString`.

4.3.1 Disgresión: una introducción a la ordenación

La ordenación en el ejemplo anterior se ha implementado usando un algoritmo denominado *ordenación por inserción*. Esta técnica se considera una buena solución cuando son pocos los elementos a ordenar, ya que el algoritmo en cuestión es muy sencillo. Sin embargo, consume demasiado tiempo, y si queremos ordenar una gran cantidad de elementos, la ordenación por inserción es una pobre elección. En tal caso debemos emplear algoritmos mejores, tal y como se estudia en el Capítulo 8. La ordenación por inserción está implementada en la Figura 4.9.

```
1  // Crea la figura adecuada basándose en la entrada.
2  // Es parte de la clase TestFigura de la Figura 4.7.
3  // El usuario debe teclear 'c', 's' o 'r' para indicar la figura
4  // y después introducir las dimensiones.
5  // Ante cualquier error se devuelve un circulo de radio cero.
6
7      private static Figura leerFigura( )
8      {
9          double rad;
10         double largo;
11         double ancho;
12         String unaLinea;
13
14         try
15         {
16             System.out.println( "Introduzca el tipo de figura:" );
17             do
18             {
19                 unaLinea = in.readLine( );
20             } while( unaLinea.length( ) == 0 );
21
22             switch( unaLinea.charAt( 0 ) )
23             {
24               case 'c':
25                 System.out.println( "Introduzca el radio: " );
26                 rad = Double.valueOf( in.readLine( ).doubleValue( );
27                 return new Circulo( rad );
28
29               case 's':
30                 System.out.println( "Introduzca el lado: " );
31                 largo=Double.valueOf(in.readLine()).doubleValue();
32                 return new Cuadrado( largo );
33
34               case 'r':
35                 System.out.println( "Introduzca el largo y el ancho "
36                                     + "en líneas separadas: " );
37                 largo=Double.valueOf(in.readLine()).doubleValue();
38                 ancho=Double.valueOf(in.readLine()).doubleValue();
39                 return new Rectangulo( largo, ancho );
40
41               default:
42                 System.err.println( "Me faltan c, r o s" );
43                 return new Circulo( 0 );
44             }
45         }
46         catch( IOException e )
47         {
48             System.err.println( e );
49             return new Circulo( 0 );
50         }
51 }
```

Figura 4.8 Rutina para leer los datos de entrada y devolver una `Figura`.

La ordenación por inserción trabaja del siguiente modo. En el estado inicial se parte de que el primer elemento considerado aisladamente está ordenado. En el estado final que queremos obtener se tienen ordenados todos los elementos del vector. La Figura 4.10 muestra que la acción básica de la ordenación consiste en

```
1  // ordenacionPorInsercion: ordena el vector a.
2      private static void ordenacionPorInsercion( Figura [ ] a )
3      {
4          for( int p = 1; p < a.length; p++ )
5          {
6              Figura tmp = a[ p ];
7              int j = p;
8
9              for( ; j > 0 && tmp.menorQue( a[ j - 1 ] ); j-- )
10                 a[ j ] = a[ j - 1 ];
11             a[ j ] = tmp;
12         }
13     }
```

Figura 4.9 Ordenación por inserción.

Posición del vector	0	1	2	3	4	5
Estado inicial:	8	5	9	2	6	3
Tras ordenar a(0..1):	5	8	9	2	6	3
Tras ordenar a(0..2):	5	8	9	2	6	3
Tras ordenar a(0..3):	2	5	8	9	6	3
Tras ordenar a(0..4):	2	5	6	8	9	3
Tras ordenar a(0..5):	2	3	5	6	8	9

Figura 4.10 Acción básica de la ordenación por inserción (la parte sombreada está ya ordenada).

conseguir que los elementos entre las posiciones 0 y `p` (donde `p` toma valores entre 1 y $N-1$ y N es el número de elementos a ordenar) queden correctamente ordenados. En cada paso, `p` aumenta en una unidad, lo que es controlado en el bucle de la línea 4 en la Figura 4.9.

Cuando se ejecuta el cuerpo del bucle `for` en la línea 6, tenemos garantizado que los elementos del vector entre las posiciones 0 y `p` − 1 están ya ordenados. La Figura 4.11 muestra más detalladamente lo que vamos haciendo, resaltando en cada caso la parte relevante del vector. En cada paso necesitamos añadir el elemento en negrita en la parte del vector previamente ordenada. Esto se logra fácilmente almacenando dicho elemento en una variable temporal y desplazando todos los elementos mayores que él una posición a la derecha. Después de esto, podemos copiar el contenido de la variable temporal en la posición anterior al elemento recolocado que esté más a la izquierda (indicado con un sombreado más claro en la línea siguiente). Se mantiene un contador `j`, que es la posición en la que debe colocarse la variable temporal. `j` decrece en una unidad cada vez que se desplaza un elemento del vector. Las líneas entre la 6 y la 11 implementan lo indicado.

Posición del vector	0	1	2	3	4	5
Estado inicial:	8	**5**				
Tras ordenar a(0..1):	5	8	**9**			
Tras ordenar a(0..2):	5	8	9	**2**		
Tras ordenar a(0..3):	2	5	8	9	**6**	
Tras ordenar a(0..4):	2	5	6	8	9	**3**
Tras ordenar a(0..5):	2	3	5	6	8	9

Figura 4.11 Detalle de la ordenación por inserción (el sombreado oscuro indica la parte ordenada; el sombreado claro indica dónde debe colocarse el nuevo elemento).

Es importante comprobar que la ordenación por inserción se comporta correctamente en los dos casos extremos. En primer lugar, si en la Figura 4.11 el elemento en negrita es el mayor de los que se encuentran en el grupo, entonces se copia en la variable temporal e inmediatamente se vuelve a insertar en el vector, por lo que el algoritmo es correcto. Si en la misma figura, el elemento en negrita es el menor de todos los del grupo, entonces el grupo entero se desplaza y la variable temporal se coloca en la posición cero. Sólo debemos tener cuidado de no salirnos fuera de los límites del vector. Así, podemos asegurar que, cuando el bucle `for` más externo termina, el vector está ordenado.

4.4 Herencia múltiple

Todos los ejemplos de herencia vistos hasta ahora derivan una clase a partir de una única clase base. En la *herencia múltiple* una clase puede derivarse de varias clases base. Por ejemplo, supóngase que se tienen las clases `Estudiante` y `Trabajador`, un `EstudianteTrabajador` podría derivarse entonces de ambas clases.

Aunque la herencia múltiple es atractiva y en algunos lenguajes (incluyendo C++) se puede implementar, está rodeada de sutiles detalles que complican su diseño. Por ejemplo, es posible que las dos clases base contengan dos métodos con la misma signatura pero distintas implementaciones. Del mismo modo, ambas clases pueden tener atributos con nombres idénticos. Entonces, ¿cuál deberíamos usar?

Supongamos que en el ejemplo del `EstudianteTrabajador`, `Persona` es una clase con el atributo `nombre` y el método `toString`. Supóngase además que `Estudiante` extiende a `Persona` y sobrescribe el método `toString` para incluir el año de graduación del estudiante. Supongamos, también, que `Empleado` extiende a `Persona` pero no redefine `toString`, sino que lo declara como `final`.

1. Como `EstudianteTrabajador` hereda los atributos de `Estudiante` y `Trabajador`, ¿se tienen dos copias de `nombre`?
2. Si `EstudianteTrabajador` no redefine `toString`, ¿qué método `toString` debemos emplear?

Cuando en la herencia intervienen más clases, los problemas son aún mayores. Estas dificultades se traducen en implementaciones o identificadores de atributos conflictivos. Como consecuencia, Java no permite la herencia múltiple; sin embargo, presenta una alternativa llamada *interfaz*.

4.5 La interfaz

En Java la `interface` es la clase más abstracta posible. Consiste sólo en métodos públicos abstractos y atributos públicos y finales.

Se dice que una clase *implementa* una interfaz si define todos los métodos abstractos de la interfaz. Una clase que implementa una interfaz se comporta como si hubiese extendido una clase abstracta especificada por la interfaz.

En principio, la diferencia fundamental entre una interfaz y una clase abstracta es que, aunque ambas especifican cómo deben comportarse las subclases, la interfaz no puede dar ningún detalle acerca de la implementación de los métodos ni de los atributos. La consecuencia de estas restricciones es que la existencia de varias interfaces no presenta los mismos problemas potenciales que la herencia múltiple. De este modo, mientras que una clase puede extender sólo a otra clase base, sí que puede implementar a varias interfaces.

4.5.1 Especificación de una interfaz

Sintácticamente, nada es más sencillo que especificar una interfaz. La interfaz tiene la apariencia de una declaración de clase, salvo que emplea la palabra clave `interface`. Consiste en una lista de métodos que debe ser posteriormente implementados. Un ejemplo es la interfaz `Comparable`, mostrada en la Figura 4.12 y empleada varias veces a lo largo del texto.

La interfaz `Comparable` especifica dos métodos que cualquier subclase suya debe implementar: `compara` y `menorQue`. `compara` se comporta como el método `String compareTo`. Observe que no es necesario especificar que estos métodos son públicos (mediante `public`) y abstractos (mediante `abstract`). Como esto está forzado en los métodos de una interfaz, estas palabras claves pueden, y deben, ser omitidas.

```
1  package Soporte;
2
3  public interface Comparable
4  {
5      int compara( Comparable lder );      // Similar a compareTo
6      boolean menorQue( Comparable lder );
7  }
```

Figura 4.12 Interfaz `Comparable`.

4.5.2 Implementación de una interfaz

Una clase implementa una interfaz si cumple los siguientes requisitos:

1. declara que implementa la interfaz y
2. define implementaciones para todos los métodos de la interfaz.

En la Figura 4.13 se muestra un ejemplo. En él se define la clase `MiEntero`, empleada como clase estándar a lo largo del texto. La clase `MiEntero` se comporta de forma idéntica a la clase predefinida de Java `Integer`, pero además podrá usarse siempre que se requiera la manipulación de objetos `Comparables`.

En la línea 3, en la que la clase declara que implementa la interfaz, se usa la palabra clave *`implements`* en lugar de `extends`. La clase puede contener todos los métodos que desee, pero entre ellos deben figurar los listados en la interfaz. La interfaz se implementa entre las líneas 23 y 30. Nótese que es obligatorio implementar *exactamente cada método* indicado en la interfaz. Por ello, estos métodos tienen como párametro un objeto `Comparable` en lugar de un objeto `MiEntero`.

Una clase que implementa una interfaz puede ser extendida a su vez, siempre que no sea final. Así, como `MiEntero` no es final, podría ser extendida posteriormente.

```
1  package Soporte;
2
3  final public class MiEntero implements Comparable
4  {
5          // Constructor
6      public MiEntero( int x )
7        { valor = x; }
8
9          // Algunos métodos
10     public String toString( )
11       { return Integer.toString( valor ); }
12
13     public int intValor( )
14       { return valor; }
15
16     public boolean equals( Object lder )
17     {
18         return lder instanceof Integer &&
19                valor == ((MiEntero)lder).valor;
20     }
21
22         // Implementando la interfaz
23     public int compara( Comparable lder )
24     {
25         return valor < ((MiEntero)lder).valor? -1 :
26                valor == ((MiEntero)lder).valor? 0 : 1;
27     }
28
29     public boolean menorQue( Comparable lder )
30       { return valor < ((MiEntero)lder).valor; }
31
32         // Atributos
33     private int valor;
34 }
```

Figura 4.13 La clase `MiEntero` (versión preliminar), que implementa la interfaz `Comparable`.

Además, cualquier clase que implemente una interfaz puede extender también a otra clase. Por ejemplo, podríamos haber escrito

```
public class MiEntero extends Integer implements Comparable
```

Esto sería incorrecto, sólo porque `Integer` es una clase final que no puede extenderse.

```
1  package Soporte;
2
3  public interface Hashable
4  {
5      int hash( int tamanyoTabla );
6  }
```

Figura 4.14 La interfaz `Hashable`.

```
1  package Soporte;
2
3  /**
4   * Clase empleada en estructuras de datos genéricas.
5   * Se comporta igual que Integer.
6   */
7  final public class MiEntero implements Comparable, Hashable
8  {
9      public MiEntero( int x )
10       { /* Como antes */ }
11     public String toString( )
12       { /* Como antes */ }
13     public int intValor( )
14       { /* Como antes */ }
15     public boolean equals( Object lder )
16       { /* Como antes */ }
17
18     public int compara( Comparable lder )
19     {
20         return valor < ((MiEntero)lder).valor? -1 :
21                valor == ((MiEntero)lder).valor? 0 : 1;
22     }
23
24     public boolean menorQue( Comparable lder )
25     {
26         return valor < ((MiEntero)lder).valor;
27     }
28
29     public int hash( int tamanyoTabla )
30     {
31         if( valor < 0 )
32             return -valor % tamanyoTabla;
33         else
34             return valor % tamanyoTabla;
35     }
36
37     private int valor;   /* Como antes */
38 }
```

Figura 4.15 La clase `MiEntero`, que implementa dos interfaces.

4.5.3 Varias interfaces

Como ya hemos mencionado anteriormente, una clase puede implementar varias interfaces y la sintaxis para hacerlo es muy sencilla. Una clase implementa varias interfaces si cumple los siguientes requisitos:

1. enumera las interfaces que implementa y
2. define implementaciones para todos los métodos de dichas interfaces.

La Figura 4.14 muestra la interfaz `Hashable`, estudiada con detalle en la Sección 6.7 y el Capítulo 19. La clase `MiEntero` implementa las interfaces `Comparable` y `Hashable`. El resultado (en el que para mayor brevedad hemos eliminado los métodos que no corresponden a ninguna interfaz) se muestra en la Figura 4.15.

La interfaz es la clase más abstracta posible y una elegante solución al problema de la herencia múltiple.

4.6 Implementación de componentes genéricas

Recordemos que un objetivo central de la programación orientada a objetos es la reutilización de código. Un mecanismo destacado que hace posible alcanzarlo es la programación *genérica*: cuando una implementación no depende del tipo de los objetos que manipula, puede emplearse una *implementación genérica* para describir la funcionalidad básica. Por ejemplo, si se pretende ordenar un vector de elementos, la lógica empleada para hacerlo es independiente del tipo de los objetos manipulados, luego puede emplearse un método genérico.

A diferencia de muchos de los lenguajes más recientes (como C++, que emplea plantillas para implementar la programación genérica), Java no soporta directamente las implementaciones genéricas. Esto es debido a que la programación genérica puede implementarse a partir de la herencia. Esta sección describe cómo las clases y métodos genéricos pueden implementarse en Java empleando los conceptos básicos de la herencia[1].

En Java la idea básica es que se puede implementar una clase genérica utilizando una superclase adecuada, como `Object`. En Java, si una clase no extiende a ninguna otra se entiende que extiende a la clase `Object` (definida en `java.lang`). Como resultado, cualquier clase es una subclase de `Object`.

Considere la clase `CeldaEntera` de la Figura 3.1 y recuerde también que `CeldaEntero` soporta los métodos `leer` y `escribir`. Podemos convertir esta clase en una clase genérica llamada `CeldaMemoria`. El resultado se muestra en la Figura 4.16.

Cuando se emplea esta estrategia debe tenerse en cuenta dos detalles. Ambos se ilustran en la Figura 4.17, en la que se implementa un método `main` que escribe un 5 en un objeto `CeldaMemoria` y después lee el contenido de dicho objeto. En primer lugar, debe recordarse que un tipo primitivo no es un objeto. Como resultado, `m.escritura(0)` sería incorrecto. Sin embargo, esto no representa un grave problema, ya que Java define *clases envoltorio* para los ocho tipos primitivos. Así, el objeto `CeldaMemoria` guardará un objeto de clase `Integer` (y no un valor de tipo `int`).

[1] El soporte directo de métodos y clases genéricos se encuentra en la actualidad bajo una fuerte consider ampliar para ello el lenguaje con la palabra clave `generic`. Actualmente la técnica descrita en esta sección es la

```
1  // Clase CeldaMemoria
2  // Object leer( )          --> Devuelve el valor guardado
3  // void escribir( Object x --> x es almacenado
4
5  public class CeldaMemoria
6  {
7          // Métodos públicos
8      public Object leer( )          { return valorGuardado; }
9      public void escribir( Object x ) { valorAlmacenado = x; }
10
11         // Representación interna y privada de datos
12     private Object valorAlmacenado;
13 }
```

Figura 4.16 Clase genérica `CeldaMemoria`.

```
1  public class TestCeldaMemoria
2  {
3      public static void main( String [ ] args )
4      {
5          CeldaMemoria m = new CeldaMemoria( );
6
7          m.escribir( new Integer( 5 ) );
8          System.out.println( "El contenido es: " +
9                              ( (Integer) m.leer( ) ).intValue( ) );
10     }
11 }
```

Figura 4.17 Empleo de la clase genérica `CeldaMemoria`.

El segundo detalle es que el resultado de `m.leer( )` es un `Object`. Para generar el valor almacenado en cada momento, debemos realizar previamente una conversión de tipos, convirtiendo el `Object` en un `Integer`. Después, aplicaremos el método `intValue` para obtener un `int`[2].

Debido a que las clases envoltorio son clases finales, el constructor y el método de acceso `intValue` pueden ser expandidos por el compilador, obteniéndose código tan eficiente como si un `int` hubiera sido manipulado directamente.

Un segundo ejemplo es el problema de la ordenación. Ya hemos escrito anteriormente un método `ordenacionPorInsercion` que trabaja con un vector de `Figuras`. Ahora sería oportuno escribirlo para un vector genérico.

La Figura 4.18 muestra una rutina genérica de inserción que es casi idéntica al método de la Figura 4.9. Este método está incluido en la clase `Ordenacion` del paquete `EstructurasDatos`; posteriormente se irá añadiendo nuevas rutinas de ordenación a dicha clase, que será ampliamente estudiadas en el Capítulo 8 del libro. Es importante observar que en la rutina de ordenación no se emplea la clase `Object`. Por el contrario, ordenamos un vector de elementos `Comparables`, puesto que es necesario emplear el método `menorQue`. Como consecuencia, sólo las clases que implementan la interfaz `Comparable` pueden ser ordenadas con el nuevo `ordenacionPorInsercion`. Así, este método, tal y como está escrito,

ecesita aplicar `intValue`, ya que el método `toString` está directamente definido en la clase `Integer`. Aquí ar cómo el valor de tipo `int` podría obtenerse de una forma más general.

```
1  // ordenacionPorInsercion: ordenación del vector a.
2  // Forma parte de la clase EstructurasDatos.Ordenacion
3  // El paquete importa Soporte.*
4
5      private static void ordenacionPorInsercion( Comparable [ ] a )
6      {
7          for( int p = 1; p < a.length; p++ )
8          {
9              int j = p;
10             Comparable tmp = a[ p ];
11
12             for( ; j > 0 && tmp.menorQue( a[ j - 1 ] ); j-- )
13                 a[ j ] = a[ j - 1 ];
14             a[ j ] = tmp;
15         }
16     }
```

Figura 4.18 Rutina genérica de inserción.

no puede emplearse para la ordenación de un vector de `Figuras`, ya que la clase `Figura` no implementa la interfaz `Comparable`. El Ejercicio 4.15 pide al lector que realice las modificaciones necesarias a la clase `Figura` para poder hacerlo.

Para comprobar cómo se emplea el método genérico `Ordenacion` vamos a escribir un programa que lea una cantidad ilimitada de enteros, los ordene y muestre el resultado. El mismo se muestra en la Figura 4.19. El método `leerEnteros`, escrito en la Figura 2.4, se emplea para la lectura del vector de `ints` en la línea 11.

```
1  import Soporte.*;
2
3  // Programa de prueba para leer enteros del terminal (uno por
4  // línea), ordenarlos y finalmente mostrarlos.
5
6  public class OrdenaEnteros
7  {
8      public static void main( String [ ] args )
9      {
10             // Lectura del vector de enteros
11         int [ ] arr = LecturaEnteros.leerEnteros( ); // Figura 2.4
12
13             // Conversión a un vector de MiEnteros
14         MiEntero [ ] elVector = new MiEntero[ arr.length ];
15         for( int i = 0; i < elVector.length; i++ )
16             elVector[ i ] = new MiEntero( arr[ i ] );
17
18             // Aplicación del algoritmo de ordenación
19         EstructurasDatos.Ordenacion.ordenacionPorInsercion( elVector );
20
21             // Muestra el resultado ordenado
22         System.out.println( "Resultado ordenado: " );
23         for( int i = 0; i < elVector.length; i++ )
24             System.out.println( elVector[ i ] );
25     }
26 }
```

Figura 4.19 Lectura de un conjunto de `ints`, ordenación de los mismos y muestra del resultado.

A continuación, se genera un vector de elementos que implementen nuestra interfaz `Comparable`. La clase predefinida `Integer` no implementa `Comparable`, pero sí lo hace la clase `MiEntero`, presentada en este capítulo. En las líneas 14 a 16 se realiza la construcción apropiada. Para ello se crea por cada `int` el correspondiente `MiEntero`, que luego se guarda en el vector. La ordenación se efectúa en la línea 19 (no empleamos la cláusula `import`, por lo que el método debe ser invocado con su nombre completo, incluyendo los paquetes en los que está integrado). Por último, el código para mostrar el resultado se encuentra en las líneas 22 a 24. Observe que el método `toString` se invoca implícitamente a través de `MiEntero`.

Resumen

La herencia es una potente herramienta, esencial en la programación orientada a objetos y en Java. Nos permite abstraer funcionalidad dando lugar a las clases abstractas base e implementar clases derivadas expandiendo dicha funcionalidad. En la clase base pueden especificarse varios tipos de métodos, tal y como resume la Figura 4.20.

Método	Sobrecarga	Comentarios
final	En tiempo de compilación	Invariante a lo largo de la jerarquía de herencia (el método nunca se redefine).
abstract	En ejecución	La clase base no facilita ninguna implementación y es abstracta. Las clases derivadas deben facilitar la implementación.
static	En tiempo de compilación	No hay objeto controlador.
Otros	En ejecución	La clase base facilita una implementación que puede sobrescribirse o no en las clases derivadas.

Figura 4.20 Cuatro tipos de métodos.

Las clases más abstractas, en las que no se puede incorporar ninguna implementación, son las `interfaces`. Una interfaz enumera todos los métodos que deben ser implementados por cada clase derivada. Las clases derivadas deben implementar todos estos métodos, o ser, a su vez, abstractas, además de indicar, mediante la palabra clave *implements*, que implementan la interfaz. Una clase puede implementar varias interfaces, representando esto una alternativa sencilla a la herencia múltiple.

Finalmente, la herencia permite escribir de forma sencilla métodos y clases genéricas mediante las cuales representamos una amplia gama de tipos genéricos. Para lograrlo tendremos habitualmente que utilizar la conversión de tipos.

Con este capítulo concluye la primera parte del texto, en la que hemos visto las características básicas de Java y de la programación orientada a objetos. Pasaremos ahora a estudiar los algoritmos y las estructuras de datos empleados en la resolución de problemas de programación.

Elementos del juego

clase abstracta Clase de la que no pueden construirse objetos pero que sirve para especificar la funcionalidad de sus clases derivadas.
clase base Clase sobre la que se basa la herencia.
clase derivada Clase completamente nueva que presenta cierta compatibilidad con la clase de la que se deriva.
clase envoltorio Clase que facilita objetos en los que se almacenan elementos de un tipo primitivo. Un ejemplo de clase envoltorio es la clase `Integer`.
clase final Clase que no puede ser extendida.
clase hoja Clase final.
composición Mecanismo más adecuado para implementar la herencia cuando la relación subyacente no es una relación ES-UN(A). En tal caso se dice que un objeto de clase *B* contiene un objeto de clase *A*.
`extends` Cláusula empleada para indicar que una clase es una subclase de otra.
herencia Proceso gracias al cual podemos derivar una clase partiendo de otra, llamada base, sin modificar la implementación de la clase base. También permite el diseño de jerarquías de clases, como `Exception` y `Component`.
herencia múltiple Proceso en el que se deriva una clase a partir de varias clases base. Java no soporta la herencia múltiple; sin embargo sí se tiene la alternativa de implementar simultáneamente varias interfaces.
`implements` Cláusula necesaria para declarar que una clase implementa los métodos de una interfaz.
interfaz Tipo especial de clase abstracta que no contiene ningún detalle de implementación.
ligado (tipado) dinámico Decisión tomada en tiempo de ejecución en la que se determina qué metodo debe aplicarse al objeto referenciado en ese momento.
ligado (tipado) estático La decisión de qué método se aplica se toma en tiempo de compilación. Se emplea únicamente para métodos estáticos o finales.
llamada al constructor `super` Llamada al constructor de la clase base.
método abstracto Método especificado pero no definido en la clases abstractas, y que, por ello, tendrá que ser implementado en las clases derivadas.
método final Método que no puede sobrescribirse y, por tanto, permanece invariante a lo largo de la jerarquía de herencia. Para estos métodos se emplea el ligado (tipado) estático.
miembro de clase protegido Accesible para las clases derivadas pero privado para cualquier otra.
ordenación por inserción Algoritmo de ordenación sencillo, adecuado para entradas de pequeño tamaño.
polimorfismo Capacidad que poseen las variables referencia para apuntar a objetos de diferentes tipos. Cuando se realizan operaciones con una variable tal, se

selecciona automáticamente la adecuada al objeto referenciado en el momento actual.

programación genérica Se emplea para implementar la lógica independiente del tipado.

referencia `super` Referencia empleada en la sobrescritura (sobrecarga) parcial mediante la cual se invoca un método de la correspondiente clase base.

relación CONTIENE Relación en la que la clase derivada contiene una instancia de la correspondiente clase base.

relación ES-UN(A) Relación en la que la clase derivada es una variación de la clase base.

sobrescritura (sobrecarga) parcial Hecho de extender el cuerpo de un método de una clase base, para realizar tareas adicionales, aunque no completamente distintas.

Errores comunes

1. Las componentes privadas de una clase base no son accesibles desde sus clases derivadas.
2. Los objetos de una clase abstracta no puede instanciarse.
3. Si la clase derivada no implementa un método heredado de la clase abstracta, la primera se convierte también en abstracta. Si esto no es indicado explícitamente, el compilador generará un error.
4. Los métodos finales no deben ser sobrescritos. Del mismo modo, las clases finales no pueden extenderse.
5. Los métodos estáticos emplean el ligado estático, incluso si son redefinidos en una clase derivada.
6. En una clase derivada, las componentes heredadas sólo deben inicializarse a través del método `super`. Si estas componentes son públicas o protegidas, puede ser leídas o asignadas posteriormente.
7. En una clase derivada, la lista `throws` de un método sobrescrito no puede incluir ninguna excepción que no sea lanzada en el mismo método de la clase base. Además los tipos del resultado deben coindir en ambos métodos.
8. En la mayoría de los casos en los que un método genérico devuelve una referencia genérica, debe realizarse una conversión de tipos para obtener el objeto devuelto.

En Internet

El código para el caso de estudio `Figura` se encuentra en el directorio **Chapter04**. `Sort.java` es parte del paquete `DataStructures`, contenido en el directorio del mismo nombre. El código que ilustra el ejemplo sobre genericidad `CeldaMemoria` también se encuentra disponible en **Chapter04**. Por último, los interfaces de las Figuras 4.12, 4.14 y 4.15 pueden encontrarse en el directorio **Supporting**.

Circle.java	La clase `Circulo`.
Comparable.java	La interfaz `Comparable` de la Figura 4.12, formando parte del paquete `Soporte`.
Hashable.java	La interfaz `Hashable` de la Figura 4.14, formando parte del paquete `Soporte`.
MemoryCell.java	La clase `CeldaMemoria` de la Figura 4.16.
MyInteger.java	La interfaz `MiEntero` de la Figura 4.15, formando parte del paquete `Soporte`.
Rectangle.java	La clase `Rectangulo`.
Shape.java	La clase abstracta `Figura`.
Sort.java	Colección de rutinas genéricas de ordenación, se encuentran en el paquete `EstructurasDatos`.
SortInts.java	Contiene el código de la Figura 4.19 (clase `OrdenaEnteros`).
Square.java	La clase `Cuadrado`.
TestMemoryCell.java	Programa de prueba para la clase de celdas de memoria, mostrado en la figura 1.17 (clase `CeldaMemoria`).
TestShape.java	Un programa de prueba para el ejemplo del `Cuadrado`.
Underflow.java	La clase excepción `DesbordamientoInferior` de la Figura 4.3, formando parte del paquete `Soporte`.

Ejercicios

Cuestiones breves

4.1. ¿Qué componentes de una clase heredada pueden emplearse en la clase derivada? ¿Qué componentes se convierten en públicos para los usuarios de la clase derivada?

4.2. ¿Qué es la composición?

4.3. Explique el concepto de polimorfismo.

4.4. Explique el ligado (tipado) dinámico. ¿Cuándo no se emplea este tipo de ligado?

4.5. ¿Qué es un método final?

4.6. Considere el programa de la Figura 4.21, para estudiar la visibilidad.

a) ¿Qué accesos son incorrectos?

b) Si `main` fuese un método de `Base`, ¿qué accesos serían incorrectos?

c) Si `main` fuese un método de `Derivada`, ¿qué accesos serían incorrectos?

d) ¿Cómo cambiarían las respuestas anteriores si se eliminase de la línea 4 la cláusula `protected`?

e) Escriba un constructor de `Base` con tres parámetros. Escriba, también, un constructor de `Derivada` con cinco parámetros.

f) La clase `Derivada` tiene cinco atributos enteros. ¿Cuáles son accesibles para la clase `Test`?

g) Un método en la clase `Derivada` se aplica sobre un objeto de `Base`. ¿A qué componentes de `Base` puede acceder `Derivada`?

```
1  public class Base
2  {
3      public    int bPublico;
4      protected int bProtegido;
5      private   int bPrivado;
6      // Se omiten los métodos públicos
7  }
8
9  public class Derivada extends Base
10 {
11     public    int dPublico;
12     private   int dPrivado;
13     // Se omiten los métodos públicos
14 }
15
16 public class Test
17 {
18     public static void main( String [ ] args )
19     {
20         Base b    = new Base( );
21         Derivada d = new Derivada( );
22
23         System.out.println( b.bPublico + " " + b.bProtegido + " "
24                             + b.bPrivado + " " + d.dPublico + " "
25                             + d.dPrivado );
26     }
27 }
```

Figura 4.21 Programa para comprobar la visibilidad.

4.7. ¿Cuál es la diferencia entre las clases finales y el resto de clases? ¿Para qué se emplean las clases finales?

4.8. ¿Qué es un método abstracto?

4.9. ¿Qué es una clase abstracta?

4.10. ¿Qué es una interfaz? ¿En qué se diferencia una interfaz de una clase abstracta? ¿Cómo deben ser los componentes de una interfaz?

4.11. ¿Cómo se implementan en Java los algoritmos genéricos?

Problemas prácticos

4.12. Escriba dos métodos genéricos `min` y `max`. Cada uno de ellos debe tener dos parámetros de tipo `Comparable`. Emplee ambos métodos sobre la clase `MiEntero`.

4.13. Escriba dos métodos genéricos `min` y `max` que tengan como parámetro de entrada un vector de `Comparables`. Emplee ambos métodos sobre la clase `MiEntero`.

4.14. Modifique en el ejemplo de la clase `Figura` los métodos `leerFigura` y `main`, de modo que lancen y recogan una excepción (en lugar de crear un círculo de radio cero), cuando se detecte un error en la entrada.

4.15. Modifique la clase `Figura` para que el algoritmo genérico de ordenación pueda emplearse para manipular objetos de ese tipo.

4.16. Un `BufferSimple` soporta las operaciones `lectura` y `escritura`: dicho `BufferSimple` almacena un solo valor, de modo que un atributo suyo nos indica si está lleno o vacío. Una operación de `escritura` sólo podrá efectuarse cuando el `BufferSimple` esté vacío, mientras que las operaciones de `lectura` se ejecutan cuando haya un elemento almacenado. Una `lectura` borra el contenido del `BufferSimple` y devuelve el valor de lo que almacenaba justo antes. Escriba una clase genérica `BufferSimple`, definiendo las excepciones necesarias para la detección de errores.

Prácticas de programación

4.17. Rescriba la jerarquía de `Figura` de modo que se calcule el área de los objetos en el constructor de la misma y se almacene en un atributo de la clase. Los constructores de las clases derivadas deben pasar el resultado de ese cálculo al método `super`. El método `area` debe ser final y devolver únicamente el valor de ese atributo.

4.18. Añada a la jerarquía de `Figura` el concepto de posición, incluyendo como atributos las coordenadas necesarias. Añada, después, un método `distancia`.

4.19. Escriba una clase abstracta para manipular `Fechas` y su clase derivada `FechaGregoriana`.

4.20. Implemente una jerarquía de contribuyentes que consista en una interfaz `Contribuyente` y en las clases `ContribuyenteSoltero` y `ContribuyenteCasado` que implementan dicha interfaz.

Bibliografía

Los siguientes libros describen los principios generales del desarrollo del software orientado a objetos:

1. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ (1988).
2. B. Booch, *Object-Oriented Design and Analysis with Applications*, 2.ª ed., Benjamin/Cummings, Redwood City, Calif. (1994).
3. I. Jacobson, M. Christerson, P. Jonsson, y G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach* (revisado), Addison-Wesley, Reading, Mass. (1992).
4. D. de Champeaux, D. Lea, y P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, Mass. (1993).

Parte II

Algoritmos y fundamentos de programación

CAPÍTULO

5 Análisis de algoritmos

En la Parte I hemos visto cómo la programación orientada a objetos puede ayudar en el diseño e implementación de grandes sistemas. Ésta es, sin embargo, la mitad de la historia.

Generalmente utilizamos un ordenador porque necesitamos procesar gran cantidad de datos. Cuando ejecutamos un programa sobre una gran cantidad de datos debemos estar seguros de que el programa termina dentro de un plazo razonable. Esto es independiente, en la mayoría de los casos, del lenguaje de programación utilizado, e incluso de la metodología empleada (tales como procedimental u orientada a objetos).

Un *algoritmo* es un conjunto de instrucciones claramente especificadas que el ordenador debe seguir para resolver un problema. Una vez que se ha dado un algoritmo para resolver un problema y se ha probado que es correcto, el siguiente paso es determinar la cantidad de recursos, tales como tiempo y espacio, que el algoritmo requerirá para su aplicación. Este paso se llama *análisis de algoritmos*. Un algoritmo que necesita varios gigabytes de memoria principal no es útil en la mayoría de las máquinas actuales, aunque sea completamente correcto.

En este capítulo veremos:

- Cómo estimar el tiempo necesario para ejecutar un algoritmo.
- Diversas técnicas que reducen radicalmente el tiempo de ejecución de un algoritmo.
- Un marco matemático que describe de forma rigurosa el tiempo de ejecución de un algoritmo.
- Cómo escribir una rutina simple de *búsqueda binaria*.

5.1 ¿Qué es el análisis de algoritmos?

El tiempo necesario para ejecutar un algoritmo depende casi siempre de la cantidad de datos que el mismo debe procesar. Es de esperar, por ejemplo, que ordenar diez mil elementos requiera más tiempo que ordenar diez. El tiempo de ejecución de un algoritmo es, por tanto, función del tamaño de la entrada. El valor exacto de esta función depende de muchos factores, tales como la velocidad de la máquina, la calidad del compilador, y en algunos casos, de la calidad del programa. Para un programa fijo ejecutándose sobre un ordenador dado podemos dibujar la gráfica que representa la función del tiempo de ejecución. La Figura 5.1 ilustra una de estas gráficas sobre cuatro programas. Las curvas representan cuatro funciones típicas en el análisis de algoritmos: lineal, $O(N \log N)$, cuadrática y cúbica. El tamaño de la entrada N varía de 1 a 100 elementos, y los tiempos de ejecución

asociados varían de 0 a 10 milisegundos. Un simple vistazo a las Figuras 5.1 y 5.2 sugiere que las curvas lineal, $O(N \log N)$, cuadrática y cúbica representan tiempos de ejecución en orden decreciente de preferencia.

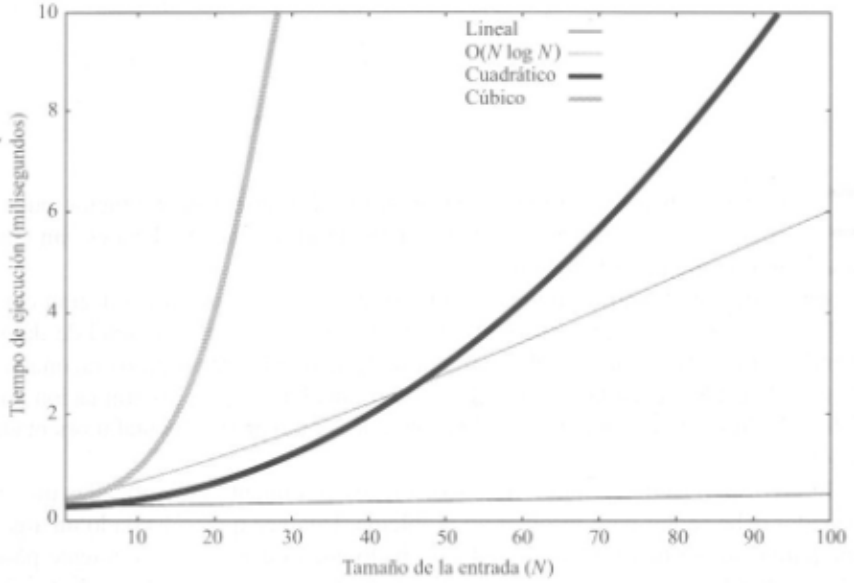


Figura 5.1 Tiempos de ejecución para tamaños de entrada pequeños.

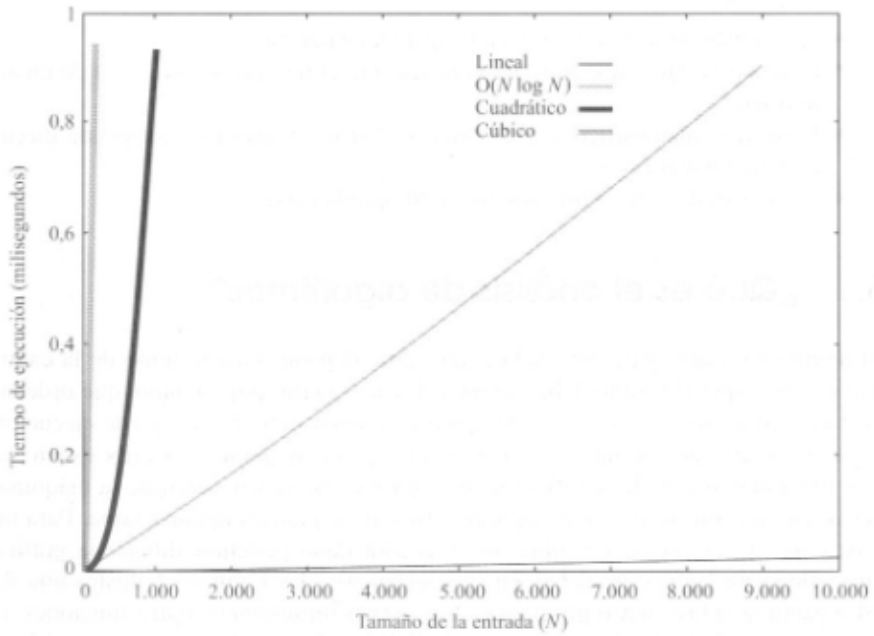


Figura 5.2 Tiempos de ejecución para tamaños de entrada mayores.

El problema de descargar un fichero de Internet nos sirve como ejemplo. Supongamos que hay un retraso inicial de 2 segundos (para establecer la conexión), después del cual la descarga se lleva a cabo a 1,6 K/seg. Entonces, si el tamaño del fichero es de N kilobytes, el tiempo de descarga viene descrito por la fórmula $T(N) = N/1{,}6 + 2$. Ésta es una *función lineal*. Podemos ver que la descarga de un fichero de 80K tardaría aproximadamente 52 segundos, mientras que descargar un fichero el doble de grande (160K) consumiría alrededor de 102 segundos, es decir casi el doble. Esta propiedad, por la cual el tiempo de ejecución es esencialmente proporcional al tamaño de la entrada, caracteriza a un *algoritmo lineal*, y es el escenario más favorable.

Como puede verse en las gráficas, algunas de las curvas no lineales conducen a tiempos de ejecución mayores. Esto indica, en particular, que un algoritmo lineal es mucho mejor que uno cúbico.

Este capítulo trata varias cuestiones importantes:

- ¿Es importante estar siempre sobre la curva más favorable?
- ¿Cuánto mejor es una curva que otra?
- ¿Cómo decidimos sobre qué curva se encuentra un algoritmo particular?
- ¿Cómo diseñamos algoritmos que eviten estar sobre las curvas malas?

Una *función cúbica* es una función cuyo término dominante es N^3 multiplicado por alguna constante. Por ejemplo, $10N^3 + N^2 + 40N + 80$ es una función cúbica. De igual manera, una función cuadrática tiene como término dominante N^2 por alguna constante, y una función lineal tiene como término dominante N por una constante. $O(N \log N)$ representa una función cuyo término dominante es N veces el logaritmo de N. El logaritmo es una función que crece lentamente; por ejemplo, el logaritmo de 1.000.000 (en base 2) es sólo 20. La función logaritmo crece más lentamente que la raíz cuadrada, la raíz cúbica, o cualquier raíz. El logaritmo se estudia en mayor profundidad en la Sección 5.5.

Dadas dos funciones, una cualquiera puede ser menor que la otra en un punto dado, por lo que en general no tiene sentido afirmar, por ejemplo, que $F(N) < G(N)$. No obstante, nosotros medimos el índice de crecimiento de las funciones. Esto viene justificado por tres razones. Primero, para funciones cúbicas, como la mostrada en la Figura 5.2, cuando N es 1.000 el valor de la función está determinado casi completamente por el término cúbico. En la función $10N^3 + N^2 + 40N + 80$, para $N = 1.000$, el valor es 10.001.040.080, del cual 10.000.000.000 es debido al término $10N^3$. Si utilizáramos solo el término cúbico para estimar el valor total de la función, obtendríamos un error de, aproximadamente, el 0,01 por ciento. Para un N suficientemente grande, el valor de la función está completamente determinado por su término dominante (el significado del término *suficientemente grande* varía para cada función).

La segunda razón por la cual medimos la tasa de crecimiento de las funciones es que el valor exacto del coeficiente del término dominante no se conserva al cambiar de máquina (aunque los valores relativos de este coeficiente para funciones de crecimiento idéntico pueden serlo). Por ejemplo, la calidad del compilador podría tener gran influencia sobre esta constante. Y tercero, los valores pequeños de N generalmente no son importantes. Para $N = 20$, la Figura 5.1 muestra que todos los algoritmos terminan antes de 5 ms. La diferencia entre el mejor y el peor algoritmo es menor que un parpadeo de ojos.

Utilizaremos la notación O para representar el índice de crecimiento. Por ejemplo, el tiempo de ejecución de un algoritmo cuadrático se describe como $O(N^2)$ (pronunciado «del orden de N al cuadrado»). La notación O nos permite establecer un orden relativo entre funciones, comparando los términos dominantes. La notación O se discute más formalmente en la Sección 5.4.

Para valores pequeños de N (por ejemplo, aquellos menores que 30), la Figura 5.1 muestra que hay puntos para los cuales una curva es inicialmente mejor que otra, aunque a la postre esto deja de ser cierto. Por ejemplo, la curva cuadrática es mejor inicialmente que la curva $O(N \log N)$, pero cuando N se hace suficientemente grande el algoritmo cuadrático pierde esta ventaja. Para valores pequeños del tamaño de la entrada, es difícil comparar las funciones porque las constantes multiplicativas llegan a ser muy significativas. La función $N+2.500$ es mayor que N^2 cuando N es menor que 50. A partir de un punto, la función lineal será siempre menor que la función cuadrática. Lo que es más importante, para tamaños pequeños de la entrada los tiempos de ejecución son insignificantes, por lo que no tenemos que preocuparnos de ellos. Por ejemplo, la Figura 5.1 muestra que cuando N es menor que 25, todos los algoritmos descritos para este problema se ejecutan en menos de 10 ms. Por lo tanto, cuando los tamaños de la entrada son muy pequeños una regla práctica es utilizar el algoritmo más sencillo.

La Figura 5.2 muestra claramente las diferencias entre las diferentes curvas cuando el tamaño de la entrada es grande. Un algoritmo lineal resuelve un problema de tamaño 10.000 en una pequeña fracción de segundo. El algoritmo $O(N \log N)$ utiliza aproximadamente diez veces este tiempo. Nótese que las diferencias de tiempo dependen de las constantes involucradas y, por tanto, podrían ser mayores o menores. Al depender de estas constantes, un algoritmo $O(N \log N)$ podría ser más rápido que uno lineal para tamaños de la entrada bastante grandes. Sin embargo, en la práctica, para algoritmos de complejidad lógica similar, los algoritmos lineales suelen ser mejores que los algoritmos $O(N \log N)$.

Esto no es cierto, sin embargo, para los algoritmos cuadráticos y cúbicos. Los algoritmos cuadráticos son casi siempre impracticables cuando el tamaño de la entrada es mayor que unos pocos miles, y los algoritmos cúbicos son impracticables para tamaños de la entrada tan pequeños como unos pocos cientos. Para ver esto, intente ejecutar el algoritmo `ordenacionPorInsercion` de la Sección 4.6 con 100.000 elementos. Esté preparado para esperar un largo rato porque la ordenación por inserción es un algoritmo cuadrático. Los algoritmos de ordenación que se discuten en el Capítulo 8 se ejecutan en un tiempo *subcuadrático* (es decir, mejor que $O(N^2)$), lo que hace que la ordenación de grandes vectores sea practicable.

La característica más notoria de estas curvas es que los algoritmos cuadráticos y cúbicos no pueden competir con los restantes para tamaños de la entrada razonablemente grandes. Podemos implementar un algoritmo cuadrático en un lenguaje máquina altamente eficiente e implementar de forma simple un algoritmo lineal, y el algoritmo cuadrático seguirá perdiendo. Incluso los trucos de programación más inteligentes no pueden hacer que sea rápido un algoritmo ineficiente. Por tanto, antes de perder tiempo intentando optimizar un código, debemos tratar de optimizar el algoritmo. La Figura 5.3 ordena en orden creciente de tasa de crecimiento distintas funciones que describen comúnmente el tiempo de ejecución de los algoritmos.

Función	Nombre
c	Constante
$\log N$	Logarítmica
$\log^2 N$	Logarítmica al cuadrado
N	Lineal
$N \log N$	$N \log N$
N^2	Cuadrática
N^3	Cúbica
2^N	Exponencial

Figura 5.3 Funciones en orden creciente de índice de crecimiento.

5.2 Ejemplos de tiempo de ejecución de algoritmos

Esta sección examina tres problemas. También esboza posibles soluciones y determina qué clase de tiempo de ejecución tendrán los correspondientes algoritmos, sin dar programas detallados. El objetivo de la sección es proporcionar al lector cierta intuición sobre el análisis de algoritmos. La próxima sección proporciona más detalles de este proceso, mientras que la Sección 5.4 da un enfoque formal al problema del análisis de algoritmos.

Los problemas que examinamos en esta sección son:

ELEMENTO MÍNIMO DE UN VECTOR
Dado un vector de N elementos, encontrar el elemento más pequeño.

PUNTOS MÁS CERCANOS EN EL PLANO
Dados N puntos en un plano (es decir, un conjunto de pares de coordenadas x-y), encontrar el par de puntos que se encuentran más cercanos.

PUNTOS COLINEALES EN EL PLANO
Dados N puntos en un plano (es decir, en un sistema de coordenadas x-y), determinar si existen tres que se encuentren en línea recta.

El problema del elemento mínimo es fundamental en computación. Puede ser resuelto de la siguiente manera:

1. Mantener una variable `min` que almacene el elemento mínimo.
2. Inicializar `min` al primer elemento.
3. Hacer un recorrido secuencial del vector, actualizando `min` de forma adecuada.

El tiempo de ejecución de este algoritmo será lineal, es decir $O(N)$, pues repetimos una cantidad fija de trabajo por cada elemento del vector. Un algoritmo lineal es todo lo bueno que podemos esperar. Esto es debido a que tenemos que

examinar cada elemento del vector, un proceso que inexorablemente requiere un tiempo lineal.

El problema del par de puntos más cercano es un problema fundamental en las artes gráficas que podemos resolver de la siguiente manera:

1. Calcular la distancia entre cada par de puntos.
2. Conservar la mínima distancia.

Sin embargo éste es un cálculo costoso, ya que hay $N(N-1)/2$ pares de puntos[1]. Por tanto, hay aproximadamente N^2 pares de puntos. Examinar todos estos pares y calcular la distancia mínima entre ellos supondrá un tiempo cuadrático. Existe un algoritmo mejorado que se ejecuta en tiempo $O(N \log N)$ que trabaja evitando el cálculo de todas las distancias. Existe además un algoritmo del que se supone que tarda un tiempo $O(N)$. Estos dos últimos algoritmos utilizan sutiles observaciones para proporcionar resultados de forma más rápida, y están más allá del alcance de este texto.

El problema de los puntos colineales es importante en muchos algoritmos gráficos. Esto se debe a que la existencia de puntos colineales introduce un caso degenerado que requiere un tratamiento especializado. Puede ser resuelto directamente enumerando todos los grupos de tres puntos. Esta solución es aún más costosa computacionalmente que la del problema del par de puntos más cercano, ya que el número de grupos diferentes de tres puntos es $N(N-1)(N-2)/6$ (utilizando un razonamiento similar al usado para el problema del par de puntos más cercano). Esto nos indica que el enfoque directo nos llevaría a un algoritmo cúbico. Existe también una estrategia más inteligente (también más allá del alcance de este texto) que resuelve el problema en un tiempo cuadrático (y mejoras adicionales del mismo son un área de continua investigación).

La siguiente sección trata un problema que ilustra la diferencia entre algoritmos lineales, cuadráticos y cúbicos. Muestra también cómo el rendimiento de estos algoritmos se compara con una predicción matemática. Por último, después de discutir las ideas básicas, se estudia la notación O de manera más formal.

5.3 El problema de la subsecuencia de suma máxima

En esta sección consideramos el siguiente problema:

> ***PROBLEMA DE LA SUBSECUENCIA DE SUMA MÁXIMA***
> *Dada la secuencia de enteros (posiblemente negativos) $A_1, A_2, \ldots, A_N$, encontrar (e identificar la subsecuencia correspondiente) el valor máximo de $\sum_{k=i}^{j} A_k$. Cuando todos los enteros son negativos entendemos que la subsecuencia de suma máxima es la vacía, siendo su suma cero.*

Por ejemplo, para la entrada $\{-2, \mathbf{11}, \mathbf{-4}, \mathbf{13}, -5, 2\}$, la respuesta es 20. La misma corresponde a la subsecuencia que abarca los elementos del segundo al cuarto (mostrados en negrita). Como un segundo ejemplo, para la entrada $\{1, -3, \mathbf{4}, \mathbf{-2}, \mathbf{-1}, \mathbf{6}\}$, la respuesta es 7, y en este caso la subsecuencia abarca los últimos cuatro elementos.

En Java, los vectores empiezan en el índice cero, por lo que un programa en Java representaría la entrada como una secuencia desde A_0 hasta A_{N-1}. Éste es un detalle de programación por lo que no forma parte del diseño del algoritmo.

Antes de discutir los distintos algoritmos para este problema, merece la pena comentar el caso degenerado en el cual todos los enteros son negativos. El enunciado del problema establece una suma máxima de cero para este caso. Uno podría preguntarse por qué lo hacemos así, en vez de devolver simplemente el mayor (es decir, el menor en magnitud) entero negativo. La razón es que la subsecuencia vacía, formada por ningún entero, también es una subsecuencia, y su suma es claramente cero. Esto es análogo al hecho de que el conjunto vacío es subconjunto de cualquier conjunto. Es importante ser consciente de que el caso vacío es siempre una posibilidad y que en muchas situaciones no es un caso especial.

El problema de la subsecuencia de suma máxima es interesante, principalmente, porque hay muchos algoritmos para resolverlo; y el rendimiento de estos algoritmos varía radicalmente. Esta sección discute tres de estos algoritmos. El primero es un algoritmo inmediato de búsqueda exhaustiva. Lamentablemente, también es muy ineficiente. El segundo es una mejora sobre el primero realizada a partir de la observación de un hecho simple. El tercero es un algoritmo muy eficiente, pero nada obvio. Probaremos que su tiempo de ejecución es lineal.

El Capítulo 7 presenta un cuarto algoritmo, con un tiempo de ejecución $O(N \log N)$. Dicho algoritmo no es tan eficiente como el algoritmo lineal, pero sí bastante más eficiente que los otros dos, siendo un ejemplo típico de la clase de algoritmos con tiempo de ejecución $O(N \log N)$. Las gráficas mostradas en las Figuras 5.1 y 5.2 son representativas de estos cuatro algoritmos.

5.3.1 El algoritmo $O(N^3)$ obvio

El algoritmo más simple consiste en una búsqueda exhaustiva, o un algoritmo de fuerza bruta como se muestra en la Figura 5.4. Las líneas 9 y 10 controlan dos bucles que iteran sobre todas las posibles subsecuencias. Para cada posible subsecuencia, el valor de su suma se calcula en las líneas de la 12 a la 15. Si esa suma es mejor que la calculada hasta el momento, entonces se actualiza el valor de `sumaMax`, que finalmente se devuelve en la línea 25. Dos valores de tipo `int` —`secIni` y `secFin` (atributos estáticos de la clase)— se actualizan también cuando se encuentra una subsecuencia mejor.

El algoritmo de búsqueda exhaustiva tiene el mérito de ser extremadamente simple; cuanto menos complejo es un algoritmo, más probable es que se programe con corrección. Sin embargo, por lo general, los algoritmos de búsqueda exhaustiva no son tan eficientes como sería posible. El resto de esta sección muestra que el tiempo de ejecución del algoritmo es cúbico. Contaremos cuántas veces (como función del tamaño de la entrada) se evalúan las expresiones de la Figura 5.4. Sólo necesitamos un resultado en notación O, así que una vez encontrado el término dominante, podremos ignorar otros términos de menor grado y las constantes multiplicativas.

[1] se que cada uno de los N puntos puede ser emparejado con $N-1$ puntos, lo que nos lleva a un total de argo, así se cuentan dos veces los pares A, B y B, A, por lo que hay que dividir por dos.

```
1  /**
2   * Algoritmo cúbico para la subsecuencia de suma máxima.
3   * secIni y secFin representan la secuencia mejor actual.
4   */
5  public static int subsecuenciaSumaMaxima( int [ ] a )
6  {
7      int sumaMax = 0;
8
9      for( int i = 0; i < a.length; i++ )
10         for( int j = i; j < a.length; j++ )
11         {
12             int sumaActual = 0;
13
14             for( int k = i; k <= j; k++ )
15                 sumaActual += a[ k ];
16
17             if( sumaActual > sumaMax )
18             {
19                 sumMax = sumaActual;
20                 secIni = i;
21                 secFin = j;
22             }
23         }
24
25     return sumaMax;
26 }
```

Figura 5.4 Algoritmo cúbico para el problema de la subsecuencia de suma máxima.

El tiempo de ejecución del algoritmo está dominado completamente por el bucle `for` más interno de las líneas 14 y 15. Cuatro expresiones se ejecutan de forma repetida:

1. La inicialización `k = i`
2. La comprobación `k <= j`
3. La modificación `sumaActual += a[ k ]`
4. El incremento `k++`

El número de veces que se ejecuta la instrucción 3 la hace el término dominante de entre las cuatro expresiones. Para ver que esto es cierto, observe primero que cada inicialización está acompañada de al menos una comprobación. Nosotros ignoramos las constantes, por lo que podemos descartar el coste de las inicializaciones; éstas no pueden ser el coste dominante del algoritmo. Ya que la comprobación de la expresión 2 es falsa exactamente una vez por bucle, el número de comprobaciones sin éxito realizadas por la expresión 2 es exactamente el mismo que el de inicializaciones. El número de comprobaciones con éxito, el número de modificaciones realizadas por la expresión 3 y el número de incrementos realizados por la expresión 4 son todos idénticos. Por tanto, el número de modificaciones realizadas por la expresión 3 (es decir, el número de veces que se ejecuta la línea 15) es una medida dominante del trabajo realizado por el bucle más interno.

El número de veces que se ejecuta la línea 15 es exactamente igual al número de ternas ordenadas (i, j, k) que satisfacen $1 \leqslant i \leqslant k \leqslant j \leqslant N$.[2] Esto es así porque

[2] s varían de 0 a $N-1$. Hemos utilizado el equivalente algorítmico de 1 a N, para simplificar el análisis.

el índice i recorre el vector entero, j varía desde i al final del vector, y k varía desde i hasta j. Una estimación rápida y burda nos lleva a que el número de ternas es algo menor que $N \times N \times N$, o N^3, pues i, j y k pueden cada uno tomar uno de los N valores. La restricción adicional $i \leqslant k \leqslant j$ sirve para reducir este valor. Un cálculo preciso es algo difícil de obtener y se lleva a cabo en el Teorema 5.1.

La parte más importante del Teorema 5.1 no es su demostración, sino más bien el resultado. Hay dos formas de calcular el número de ternas. Una consiste en evaluar la suma $\Sigma_{i=1}^{N} \Sigma_{j=1}^{N} \Sigma_{k=i}^{j} 1$. Podríamos calcular esta suma de dentro a fuera (*véase* Ejercicio 5.8). Utilizaremos, en cambio, una forma alternativa.

> *El número de ternas ordenadas de enteros (i, j, k) que satisfacen $1 \leqslant i \leqslant k \leqslant j \leqslant N$ es $N(N+1)(N+2)/6$.*
>
> *Coloque las siguientes $N+2$ bolas en una caja: N bolas numeradas desde 1 hasta N, una bola roja sin numerar, y una bola azul sin numerar. Retire tres bolas de la caja. Si ha extraído una bola roja, numérela con el menor número de las bolas extraídas. Si ha extraído una bola azul, numérela con el número más alto de las bolas extraídas. Observe que si sacamos la bola roja y la bola azul, el efecto será que tendremos tres bolas numeradas de la misma manera. Ordene las tres bolas. Cada posible ordenación corresponde a una terna solución de la ecuación en el Teorema 5.1. El número de ordenaciones posibles es el número de formas diferentes de extraer tres bolas sin reemplazamiento de una colección de $N+2$ bolas. Esto es similar al problema de seleccionar tres puntos de un conjunto de N, que calculamos en la Sección 5.2, por lo que inmediatamente obtenemos el resultado deseado.*

La consecuencia del Teorema 5.1 es que el bucle `for` más interno justifica un tiempo de ejecución cúbico. El resto de trabajo del algoritmo no tiene consecuencias ya que se repite, como mucho, una vez por iteración del bucle más interno. Dicho de otra forma, el coste de las líneas 17 a 22 no tiene consecuencias porque se ejecuta solamente tan a menudo como la inicialización del bucle `for` más interno, en vez de tan a menudo como el cuerpo de este bucle. En consecuencia, el algoritmo es $O(N^3)$.

El argumento combinatorio previo nos permite obtener un cálculo preciso del número de iteraciones del bucle más interno. Para un cálculo en un orden de crecimiento O, esto no es realmente necesario; sólo necesitamos saber que el término principal es un número constante de veces N^3. Mirando el algoritmo, vemos un bucle de tamaño potencial N dentro de un bucle de tamaño potencial N dentro de otro bucle de tamaño potencial N. Esto nos dice que el triple bucle genera potencialmente $N \times N \times N$ iteraciones. Este potencial es sólo alrededor de seis veces mayor que lo que nuestro cálculo preciso mostraba que realmente ocurría. Ya que de cualquier manera las constantes se ignoran, podemos adoptar la regla general por la cual cuando tenemos bucles anidados podemos multiplicar el coste de la instrucción más interna por el tamaño de cada bucle anidado para obtener una cota superior. En la mayoría de los casos, esta cota superior no supondrá una gran

sobreestimación[3]. Por tanto, un programa con tres bucles anidados, cada uno de los cuales recorre una gran porción de un vector, es probable que tenga un comportamiento $O(N^3)$. Obsérvese que tres bucles consecutivos (no anidados) tendrán un comportamiento lineal; es el anidamiento lo que lleva a una explosión combinatoria. Por consiguiente, para mejorar el algoritmo, necesitamos eliminar un bucle. Esto se logra en la próxima sección.

5.3.2 Un algoritmo mejorado $O(N^2)$

Si podemos eliminar un bucle del algoritmo, generalmente podremos reducir el tiempo de ejecución. ¿Cómo podemos eliminar un bucle? Obviamente, no siempre podemos. Sin embargo, el algoritmo anterior contiene muchos cálculos innecesarios. La ineficiencia que el algoritmo mejorado corrige puede verse observando que como quiera que se tiene $\Sigma_{k=i}^{j} A_k = A_j + \Sigma_{k=i}^{j-1} A_k$, el cálculo en el bucle `for` más interno de la Figura 5.4 es exageradamente costoso. Más concretamente, una vez que acabamos de calcular la suma de la subsecuencia que se extiende desde i hasta $j-1$, calcular la suma de la subsecuencia que se extiende desde i hasta j no debería llevarnos mucho tiempo, pues sólo necesitamos una suma adicional. Sin embargo, el algoritmo cúbico no utiliza esta información. Si utilizamos esta observación, obtenemos el algoritmo mejorado mostrado en la Figura 5.5. Tenemos dos bucles anidados en vez de tres, de modo que el tiempo de ejecución es $O(N^2)$.

```
1  /**
2   * Algoritmo cuadrático para la subsecuencia de suma máxima.
3   * secIni y secFin representan la subsecuencia mejor actual.
4   */
5  public static int subsecuenciaSumaMaxima( int [ ] a )
6  {
7      int sumaMax = 0;
8
9      for( int i = 0; i < a.length; i++ )
10     {
11         int sumaActual = 0;
12
13         for( int j = i; j < a.length; j++ )
14         {
15             sumaActual += a[ j ];
16
17             if( sumaActual > sumaMax)
18             {
19                 sumaMax = sumaActual;
20                 secIni = i;
21                 secFin = j;
22             }
23         }
24     }
25
26     return sumaMax;
27 }
```

Figura 5.5 Algoritmo cuadrático para la obtención de la subsecuencia de suma máxima.

lustra un ejemplo en el cual la multiplicación del tamaño de los bucles lleva a sobreestimar el resultado O.

5.3.3 Un algoritmo lineal

Para pasar de un algoritmo cuadrático a otro lineal, necesitaríamos eliminar un bucle anidado más. Sin embargo, a diferencia de la reducción ilustrada en las Figuras 5.4 y 5.5, donde la eliminación de un bucle fue sencilla, no es fácil librarse de un bucle más. El problema es que el algoritmo cuadrático representa aún una búsqueda exhaustiva; es decir, estamos examinando todas las posibles subsecuencias. La única diferencia entre los algoritmos cúbico y cuadrático es que el coste de comprobar cada subsecuencia sucesiva es constante $O(1)$ en vez de lineal $O(N)$. Ya que el número de subsecuencias posibles es cuadrático, la única forma de conseguir una cota subcuadrática es encontrar un razonamiento astuto que nos permita dejar de considerar un gran número de subsecuencias, sin calcular sus sumas ni comprobar si esa suma es un nuevo máximo. Esta sección muestra como hacerlo.

Eliminamos primero un gran número de subsecuencias posibles. Denotaremos por $A_{i,j}$ la subsecuencia que abarca los elementos del i al j, siendo $S_{i,j}$ su suma.

> *Sea $A_{i,j}$ una secuencia cualquiera con $S_{i,j} < 0$. Tenemos que si $q > j$, entonces $A_{i,q}$ no es la subsecuencia de suma máxima.*
>
> *La suma de los elementos de A desde i hasta q es la suma de los elementos de A desde i hasta j más la suma de los elementos desde $j+1$ hasta q. Tenemos, por tanto, $S_{i,q} = S_{i,j} + S_{j+1,q}$. Ya que $S_{i,j} < 0$, sabemos que $S_{i,q} < S_{j+1,q}$. Podemos entonces deducir que $A_{i,q}$ no es la subsecuencia de suma máxima.*

La Figura 5.6 muestra una representación gráfica de las sumas generadas por i, j y q. El Teorema 5.2 demuestra que es posible evitar examinar diversas subsecuencias, incorporando una comprobación adicional: siempre que `sumaActual` sea menor que cero, podemos salir del bucle más interno de la Figura 5.5. Intuitivamente, si vemos una subsecuencia cuya suma es negativa, la misma no puede formar parte de la subsecuencia máxima, pues podemos obtener una subsecuencia mayor suprimiéndola. Esta observación por sí misma no es suficiente para reducir el tiempo de ejecución por debajo del cuadrático. Una observación similar también es cierta: todas las subsecuencias que bordean la subsecuencia máxima deben tener suma negativa o cero (de otra forma, las añadiríamos). Esto tampoco reduce el tiempo de ejecución por debajo del cuadrático. Sin embargo, una tercera observación, ilustrada en la Figura 5.7, sí que lo hace. La formalizamos en el Teorema 5.3.

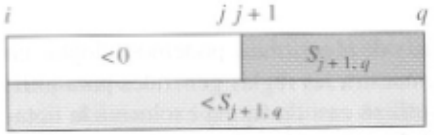


Figura 5.6 Las subsecuencias utilizadas en el Teorema 5.2.

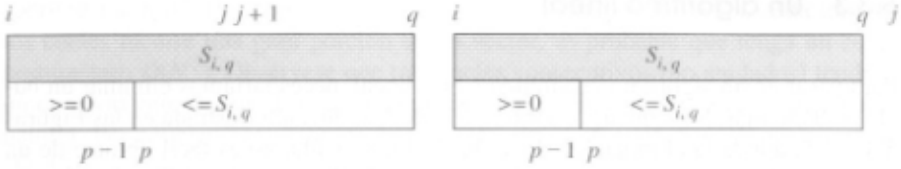


Figura 5.7 Las subsecuencias utilizadas en el Teorema 5.3. La secuencia desde p hasta q tiene una suma que es, como mucho, igual que la subsecuencia de i a q. En la parte de la izquierda la secuencia de i a q no es la máxima (por el Teorema 5.2). En la parte derecha, la secuencia de i a q ya ha sido comprobada.

Para cada i, sea $A_{i,j}$ la primera secuencia que satisfaga $S_{i,j} < 0$. Entonces, para cualquier $i \leqslant p \leqslant j$ y $p \leqslant q$, $A_{p,q}$ o bien no es una subsecuencia máxima, o es igual a una subsecuencia máxima ya considerada.

Si $p = i$, entonces es aplicable el Teorema 5.2. En otro caso, como en el Teorema 5.2, tenemos $S_{i,j} = S_{i,p-1}$. Puesto que j es el menor índice para el cual $S_{i,j} < 0$, se sigue que $S_{i,p-1} \geqslant 0$. Por tanto, $S_{p,q} \leqslant S_{i,q}$. Si $q > j$ (ilustrado en la parte de la izquierda de la Figura 5.7), entonces por el Teorema 5.2 $A_{i,q}$ no es una subsecuencia máxima, ni lo es $A_{p,q}$. En otro caso, como se muestra en la parte derecha de la Figura 5.7, la subsecuencia $A_{p,q}$ tiene una suma como mucho igual a la de la subsecuencia $A_{i,q}$ que ya ha sido considerada.

El Teorema 5.3 nos dice que cuando se detecta una subsecuencia negativa, no sólo podemos salir del bucle más interno, sino que podemos avanzar `i` hasta `j+1`. La Figura 5.8 muestra cómo podemos rescribir el algoritmo utilizando un solo bucle. Claramente, el tiempo de ejecución de este algoritmo es lineal: en cada paso del bucle, avanzamos `j`, así que el bucle se itera como mucho N veces. La corrección de este algoritmo es menos obvia que en los casos anteriores. Esto es típico: los algoritmos que utilizan la estructura del problema para evitar una búsqueda exhaustiva requieren habitualmente algún tipo de demostración de corrección. Hemos probado que el algoritmo (aunque no el programa Java correspondiente) es correcto utilizando un argumento formal. Nuestro propósito no es realizar una completa discusión matemática, sino dar una idea de las técnicas que podrían ser necesarias en un trabajo más avanzado.

5.4 Reglas generales para la notación *O*

Una vez tenemos las ideas básicas del análisis de algoritmos, podemos adoptar un enfoque un poco más formal. Esta sección muestra las reglas generales para utilizar la notación O. Aunque en este texto se utiliza casi de forma exclusiva la notación O, se definen otros tres tipos de notación que están relacionados con ella y que se utilizarán de forma ocasional más adelante en el texto.

```
1  /**
2   * Algoritmo lineal para la subsecuencia de suma máxima.
3   * secIni y secFin representan la secuencia mejor actual.
4   */
5  public static int subsecuenciaSumaMaxima( int [ ] a )
6  {
7      int sumaMax = 0;
8      int sumaActual = 0;
9
10     for( int i = 0, j = 0; j < a.length; j++ )
11     {
12         sumaActual += a[ j ];
13
14         if( sumaActual > sumaMax )
15         {
16             sumaMax = sumaActual;
17             secIni = i;
18             secFin = j;
19         }
20         else if( sumaActual < 0 )
21         {
22             i = j + 1;
23             sumaActual = 0;
24         }
25     }
26
27     return sumaMax;
28 }
```

Figura 5.8 Algoritmo lineal para la obtención de la subsecuencia de suma máxima.

DEFINICIÓN: (O) $T(N)$ es $O(F(N))$ si existen constantes positivas c y N_0 tales que para $N \geqslant N_0$ se verifica $T(N) \leqslant cF(N)$.

DEFINICIÓN: (Ω) $T(N)$ es $\Omega(F(N))$ si existen constantes positivas c y N_0 tales que para $N \geqslant N_0$ se verifica $T(N) \geqslant cF(N)$.

DEFINICIÓN: (Θ) $T(N)$ es $\Theta(F(N))$ si y sólo si $T(N)$ es $O(F(N))$ y $T(N)$ es $\Omega(F(N))$.

DEFINICIÓN: (o) $T(N)$ es $o(F(N))$ si y sólo si $T(N)$ es $O(F(N))$ y $T(N)$ no es $\Theta(F(N))$.

La primera definición, correspondiente a la notación O, afirma que existe un punto N_0 tal que para todos los valores de N después de este punto, $T(N)$ está acotada por algún múltiplo de $F(N)$. Éste es el N suficientemente grande que mencionábamos anteriormente. Por tanto, si el tiempo de ejecución $T(N)$ de un algoritmo es $O(N^2)$, entonces, ignorando las constantes, podemos garantizar que a partir de un punto se puede acotar el tiempo de ejecución mediante una función cuadrática. Obsérvese que si el tiempo de ejecución real es lineal, la afirmación de que el tiempo de ejecución es $O(N^2)$ es técnicamente correcta, porque la desigualdad es válida. Sin embargo, decir que dicho tiempo es $O(N)$ sería la afirmación más precisa posible.

Si utilizamos los operadores de desigualdad tradicionales para comparar tasas de crecimiento, entonces la primera definición nos dice que la tasa de crecimiento de $T(N)$ es menor o igual que la de $F(N)$.

La segunda definición, $T(N) = \Omega(F(N))$, correspondiente a la llamada *notación Omega*, nos dice que la tasa de crecimiento de $T(N)$ es mayor o igual que la de $F(N)$. Por ejemplo, podríamos decir que cualquier algoritmo que examine todas las posibles subsecuencias en el problema de la subsecuencia de suma máxima debe tener un tiempo de ejecución $\Omega(N^2)$, porque son posibles un número cuadrático de subsecuencias. Éste es un argumento sobre una cota inferior, que se utiliza en análisis más avanzados. Más tarde en este texto, veremos un ejemplo de ello, donde se demuestra que cualquier algoritmo de ordenación de propósito general requiere un tiempo $\Omega(N \log N)$.

La tercera definición, $T(N) = \Theta(F(N))$, correspondiente a la denominada *notación Theta*, nos dice que la obtención de la tasa de crecimiento de $T(N)$ es igual que la de $F(N)$. Por ejemplo, el algoritmo para la subsecuencia de suma máxima de la Figura 5.5 se ejecuta en un tiempo $\Theta(N^2)$. Esto significa que el tiempo de ejecución está acotado por una función cuadrática y que esta cota no puede mejorarse, pues el algoritmo también está acotado inferiormente por otra función cuadrática. Cuando utilizamos la notación Θ, estamos proporcionando no solamente una cota superior del tiempo de ejecución del algoritmo, sino que garantizamos que el análisis que nos lleva a esta cota superior es lo más ajustado posible. A pesar de la precisión adicional facilitada por la notación Θ, la notación O se utiliza más extensamente, excepto por investigadores en el campo del análisis de algoritmos.

La última definición, $T(N) = o(F(N))$, correspondiente a la denominada notación o, nos dice que la tasa de crecimiento de $T(N)$ es estrictamente menor que la de $F(N)$. Esto difiere de la notación O, pues O admite la posibilidad de que las tasas de crecimiento sean iguales. Por ejemplo, si el tiempo de ejecución de un algoritmo es $o(N^2)$, se garantiza que crece con una tasa menor que cuadrática (es decir, es *subcuadrático*). Por tanto, tener como cota $o(N^2)$ es mejor que tener como cota $\Theta(N^2)$. La Figura 5.9 ilustra los significados de estas cuatro definiciones.

Resulta oportuno aquí comentar un par de notas de estilo. No es de buen estilo incluir constantes o términos de orden menor que el dominante dentro de la notación O. No escriba $T(N) = O(2N^2)$ ni $T(N) = O(N^2 + N)$. En ambos casos, la forma correcta sería $T(N) = O(N^2)$. Recuerde que en cualquier análisis que requiera una respuesta O, son posibles todo tipo de abreviaturas. Los términos de orden inferior, las constantes y los símbolos relacionales pueden eliminarse todos.

Expresión matemática	Índices de crecimiento relativos
$T(N) = O(F(N))$	El crecimiento de $T(N)$ es $\leq$ que el crecimiento de $F(N)$.
$T(N) = \Omega(F(N))$	El crecimiento de $T(N)$ es $\geq$ que el crecimiento de $F(N)$.
$T(N) = \Theta(F(N))$	El crecimiento de $T(N)$ es $=$ que el crecimiento de $F(N)$.
$T(N) = o(F(N))$	El crecimiento de $T(N)$ es $<$ que el crecimiento de $F(N)$.

Figura 5.9 Significado de las diferentes funciones de crecimiento.

Una vez formalizados estos conceptos matemáticos, los pondremos en relación con el análisis de algoritmos. La regla más importante es *que el tiempo de ejecución de un bucle es como mucho el tiempo de ejecución de las instrucciones dentro del bucle* (*incluyendo los tests*) *multiplicado por el número de iteraciones*. Como vimos antes, la inicialización y la comprobación de la condición del bucle es habitualmente menos dominante que las instrucciones que forman el cuerpo del bucle.

El tiempo de ejecución de las instrucciones dentro de un grupo de bucles anidados es el tiempo de ejecución de las instrucciones (incluyendo la comprobación de la condición del bucle más interno) multiplicado por el tamaño de todos los bucles. El tiempo de ejecución de una secuencia de bucles consecutivos es igual al tiempo de ejecución del bucle dominante. La diferencia en tiempo entre dos bucles anidados cuyos índices varían de 1 a N y dos bucles consecutivos no anidados pero que se ejecutan sobre los mismos índices es la misma que la diferencia de espacio entre una matriz bidimensional y dos vectores unidimensionales. El segundo caso es lineal, pues $N+N$ es $2N$, que es $O(N)$. En ocasiones, esta simple regla puede sobrestimar el tiempo de ejecución, pero en la mayoría de los casos no lo hace. En cualquier caso no hay problemas, pues la notación O no garantiza una respuesta asintótica exacta, sino tan sólo una cota superior.

Los análisis que hemos realizado hasta ahora son *cotas en el caso peor*, que garantizan el tiempo sobre todas las entradas de cierto tamaño. Otro tipo de análisis es el *análisis en el caso medio*. En este caso, el tiempo de ejecución se obtiene como la media sobre todas las posibles entradas de tamaño N. La media podría diferir del caso peor si, por ejemplo, una instrucción condicional que depende de la entrada particular hace que el bucle termine rápidamente. Las cotas en el caso medio se discuten más formalmente en la Sección 5.8. Por ahora, tenga en cuenta que el hecho de que un algoritmo tenga mejor cota en el caso peor que otro, no implica nada sobre sus cotas en el caso medio. Sin embargo, en muchos casos las cotas en el caso peor y en el caso medio están fuertemente relacionadas. Cuando no lo estén, se discutirán por separado.

El último asunto referente a la notación O que discutimos es cómo crece el tiempo de ejecución para cada tipo de curva. Ya vimos esto en las gráficas de las Figuras 5.1 y 5.2. Pero queremos una repuesta más cuantitativa a la siguiente pregunta: si un algoritmo tarda un tiempo $T(N)$ en resolver un problema de tamaño N, ¿cuánto tiempo tardará en resolver un problema de tamaño mayor? Por ejemplo, ¿cuánto tardará en resolver el problema cuando el tamaño de la entrada se multiplica por diez? Las respuestas exactas experimentales se muestran en la Figura 5.10. Sin embargo, nos gustaría contestar la pregunta sin tener que ejecutar el programa, confiando en que nuestra respuesta analítica corresponda con el comportamiento observado.

Empecemos examinado el algoritmo cúbico. Por hipótesis, asumimos que el tiempo de ejecución está aproximado de manera razonable por $T(N) = cN^3$. En consecuencia tendremos $T(10N) = c(10N)^3$. Ahora unas simples manipulaciones matemáticas nos llevan a

$$T(10N) = 1.000cN^3 = 1.000T(N).$$

Por tanto, el tiempo de ejecución de un programa cúbico se incrementa por 1.000 (asumiendo un tamaño N suficientemente grande) cuando la entrada se incrementa

	Figura 5.4	Figura 5.5	Figura 7.18	Figura 5.8
N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0,00103	0,00045	0,00066	0,00034
100	0,47015	0,01112	0,00486	0,00063
1.000	448,77	1,1233	0,05843	0,00333
10.000	NA	111,13	0,68631	0,03042
100.000	NA	NA	8,01130	0,29832

Figura 5.10 Tiempos de ejecución observados (en segundos) de los diferentes algoritmos para el cálculo de la subsecuencia de suma máxima.

por un factor de 10. Esto se confirma de forma aproximada examinando el crecimiento en los tiempos de ejecución desde $N = 100$ hasta 1.000 en la Figura 5.10. Recuerde que no esperamos una respuesta exacta, sino una aproximación razonable. También esperaríamos que para $N = 10.000$, hubiera otro crecimiento por mil en el tiempo de ejecución.

El resultado sería que el uso de un algoritmo cúbico requeriría, más o menos, dos semanas de tiempo de cómputo. En general, si el tamaño de la entrada se incrementa por un factor f, entonces el tiempo de ejecución de un algoritmo cúbico se incrementa aproximadamente por un factor f^3.

Podemos realizar cálculos similares para los algoritmos cuadráticos y lineales. Para el algoritmo cuadrático, asumimos que $T(N) = cN^2$. Se sigue que $T(10N) = c(10N)^2$. Y al expandir, obtenemos

$$T(10N) = 100cN^2 = 100T(N).$$

Por tanto, cuando el tamaño de la entrada se incrementa por un factor 10, el tiempo de ejecución de un algoritmo cuadrático se incrementa por un factor aproximado de 100. Esto también puede confirmarse en la Figura 5.10. En general, un incremento por un factor f del tamaño de la entrada produce un incremento por un factor f^2 del tiempo de ejecución de un algoritmo cuadrático.

Finalmente, para un algoritmo lineal, cálculos similares mostrarían que un incremento por diez del tamaño de la entrada tiene como resultado un incremento por diez del tiempo de ejecución. Una vez más, esto puede ser confirmado experimentalmente. Observe, sin embargo, que para el programa lineal el término *suficientemente grande* significa una entrada algo mayor que en los otros casos. Esto es debido a la sobrecarga de 0,0003 segundos que aparece en todos los casos. Para un programa lineal, este término es todavía significativo para tamaños de entrada moderados.

El análisis utilizado en estos casos no sirve cuando hay términos logarítmicos. Cuando el tamaño de la entrada de un algoritmo $O(N \log N)$ se multiplica por diez, el tiempo de ejecución se incrementa en un factor que es algo mayor que diez. De forma específica, tenemos que $T(10N) = c(10N) \log (10N)$. Cuando expandimos obtenemos

$$T(10N) = 10cN \log (10N) = 10cN \log N + 10cN \log 10 = 10T(N) + c'N.$$

donde $c' = 10c \log 10$. Cuando N se hace muy grande, el cociente $T(10N)/T(N)$ se va aproximando a 10, pues $c'N/T(N) \approx (10 \log 10)/\log N$ se va haciendo cada vez más pequeño al incrementar N. Por consiguiente, si el algoritmo es competitivo comparado con uno lineal para un N muy grande, seguirá siendo competitivo para un N un poco mayor.

¿Significa todo esto que un algoritmo cuadrático o cúbico no es útil? La respuesta es no. En algunos casos, el algoritmo conocido más eficiente es cuadrático o cúbico. En otros, el algoritmo más eficiente es incluso peor (exponencial). Además, cuando el tamaño de la entrada es pequeño, cualquier algoritmo es útil, y frecuentemente los algoritmos que no son asintóticamente eficientes son fáciles de programar. Finalmente, hay que señalar que una buena forma de evaluar un algoritmo lineal complejo es comparar su comportamiento con un algoritmo de búsqueda exhaustiva. La Sección 5.8 discute otras limitaciones del modelo O.

5.5 Logaritmos

La lista de funciones de tasas de crecimiento típicas incluye varias funciones que contienen *logaritmos*. Esta sección describe con más detalle el fundamento matemático en el que se basan los logaritmos. La siguiente sección ilustra como aparecen en un simple algoritmo.

Empezaremos con la definición formal para seguir con una serie de consecuencias.

DEFINICIÓN: Para cualquier B, $N > 0$, $\log_B N = K$ si $B^K = N$.

En esta definición, B es la base del logaritmo. En computación, cuando la base se omite, por defecto se toma el valor 2. Esto es natural por varias razones, como se verá más adelante en este mismo capítulo. Demostraremos un resultado matemático, el Teorema 5.4, en el que mostramos que, en lo que se refiere a la notación O, la base no es importante. Veremos también cómo pueden ser derivadas diversas relaciones que involucran logaritmos.

> *La base no es importante. Para cualquier constante $B > 1$, $\log_B N = O(\log N)$.*
>
> *Sea $\log_B N = K$. Entonces $B^K = N$. Sea $C = \log B$. Entonces $2^C = B$. De este modo, $B^K = (2^C)^K = N$. Por tanto, tenemos que $2^{CK} = N$, lo cual implica que $\log N = CK = C \log_B N$. Por tanto, $\log_B N = (\log N)/(\log B)$, lo que completa la demostración.*

En el resto del texto, utilizaremos exclusivamente logaritmos en base 2. Un hecho importante sobre los logaritmos es que crecen lentamente. Como quiera que

$2^{10} = 1.024$, $\log 1.024 = 10$. Cálculos adicionales muestran que el logaritmo de un millón es aproximadamente 20, y el logaritmo de un billón aproximadamente 40. Por consiguiente, el rendimiento de un algoritmo $O(N \log N)$ está mucho más cerca de un algoritmo lineal $O(N)$ que de uno cuadrático $O(N^2)$, incluso para tamaños de entrada moderadamente grandes. Antes de ver un algoritmo realista cuyo tiempo de ejecución incluye un logaritmo, veamos algunos ejemplos de cómo los logaritmos entran en juego.

BITS EN UN NÚMERO BINARIO

¿Cuántos bits son necesarios para representar N enteros consecutivos?

Un entero `short` de 16 bits representa los 65.536 enteros en el rango de -32.768 a 32.767. En general, B bits son suficientes para representar 2^B enteros diferentes. Por tanto, el número de bits B necesario para representar N enteros consecutivos satisface la ecuación $2^B \geq N$. Obtenemos entonces $B \geq \log N$, de donde se sigue que el número de bits mínimo es $\lceil \log N \rceil$. (Aquí $\lceil X \rceil$ es la función techo y representa el entero más pequeño mayor o igual que X. La función suelo correspondiente, $\lfloor X \rfloor$, representa el mayor entero menor o igual que X.)

DUPLICACIONES REPETIDAS

Empezando con $X = 1$, ¿cuántas veces debe ser X duplicado antes de que sea mayor que N?

Supongamos que empezamos con 1 euro y lo duplicamos cada año. ¿Cuánto tardaríamos en ahorrar un millón de euros? En este caso, después de un año tendríamos 2 euros; después de 2 años, 4 euros; después de 3 años, 8 euros, etc. En general, después de K años tendríamos 2^K euros, por lo que queremos encontrar el menor K que satisfaga $2^K \geq N$. Ésta es la misma ecuación de antes, por lo que $K = \lceil \log N \rceil$. Después de 20 años, tendríamos alrededor de un millón de euros. El *principio de las duplicaciones repetidas* afirma que empezando en 1, podemos duplicar sólo $\lceil \log N \rceil$ veces antes de alcanzar N.

SUCESIVAS DIVISIONES POR LA MITAD

Empezando con $X = N$, si N se divide de forma repetida por la mitad, ¿cuántas veces hay que dividir para hacer N menor o igual que 1?

Si la división redondea al entero más cercano (o es división real, no entera) tenemos el mismo problema que con las duplicaciones repetidas, excepto que ahora nos movemos en el sentido opuesto. Una vez más la respuesta es $\lceil \log N \rceil$ iteraciones. Si la división redondea hacia abajo, la respuesta sería $\lfloor \log N \rfloor$. La diferencia puede verse empezando con $X = 3$. Se necesitan dos divisiones, a menos que la división redondee hacia abajo, en cuyo caso sólo se necesita una.

Muchos de los algoritmos examinados en este texto tendrán logaritmos en sus tiempos de ejecución debido al *principio de sucesivas divisiones por la mitad*. Dicho principio nos dice que un algoritmo es $O(\log N)$ si tarda un tiempo constante ($O(1)$) en dividir el tamaño del problema por una fracción constante (que generalmente es 1/2). Esto se deduce directamente del hecho de que habrá $O(\log N)$ iteraciones en el bucle. Cualquier fracción constante sirve, ya que la fracción está reflejada en la base del logaritmo, y el Teorema 5.4 nos dice que la base no es importante.

Todas las apariciones de logaritmos que faltan se introducen (directa o indirectamente) aplicando el Teorema 5.5. La demostración utiliza nociones de cálculo, pero comprender la demostración no es necesario para utilizar el teorema.

Siendo $H_N = \sum_{i=1}^{N} 1/i$ se tiene que $H_N = \Theta(\log N)$. Una estimación más precisa vendría dada por $\ln N + 0{,}577$, donde $\ln$ denota el logaritmo neperiano.

La intuición de la demostración es que una suma discreta se puede aproximar con una integral (continua). La demostración utiliza una construcción para mostrar que la suma H_N puede ser acotada superior e inferiormente por $\int \frac{dx}{x}$, con límites apropiados. Los detalles se ven en el Ejercicio 5.17. H_N es conocido como el número armónico N-ésimo.

La próxima sección muestra cómo el principio de sucesivas divisiones por la mitad conduce a un algoritmo de búsqueda eficiente.

5.6 Problema de la búsqueda estática

Un uso importante de las computadoras es la búsqueda de datos. Si los datos no pueden cambiar (por ejemplo, si están almacenados es un CD-ROM), decimos que son estáticos. Una *búsqueda estática* accede a datos estáticos. El problema de la búsqueda estática se formula como sigue:

PROBLEMA DE LA BÚSQUEDA ESTÁTICA

Dado un entero X y un vector A, devolver la posición de X en A o una indicación de que no está presente. Si X aparece más de una vez, devolver cualquier aparición. El vector A nunca será modificado.

Un ejemplo de búsqueda estática es buscar a una persona en el listín telefónico. La eficiencia de un algoritmo de búsqueda estática depende de si el vector donde se busca está o no ordenado. En el caso del listín telefónico, buscar por nombre es fácil, pero buscar por número de teléfono es algo desesperante (al menos para los humanos). En esta sección, examinamos algunas soluciones al problema de búsqueda estática.

5.6.1 Búsqueda secuencial

Cuando el vector no está ordenado, tenemos pocas alternativas fuera de una *búsqueda secuencial* lineal, recorriendo todo el vector secuencialmente hasta que se encuentre el elemento buscado. La complejidad del algoritmo se analiza de tres formas. Primero, daremos el coste de una búsqueda sin éxito. Después veremos el coste en el caso peor de una búsqueda con éxito. Finalmente, veremos el coste en

el caso medio de una búsqueda con éxito. El analizar de forma separada los casos en los que la búsqueda tiene éxito es típico. También es típico el que las búsquedas sin éxito consuman más tiempo que las búsquedas con éxito (simplemente piense en la última vez que perdió algo en su casa). Para la búsqueda secuencial, el análisis es muy fácil.

Una búsqueda sin éxito requiere examinar cada uno de los elementos del vector, así que el tiempo será $O(N)$. En el peor caso, una búsqueda con éxito también requiere examinar cada elemento del vector porque podríamos no encontrar el elemento hasta la última posición. Luego el tiempo de ejecución en el caso peor en también lineal. En promedio, sin embargo, sólo buscamos en la mitad del vector. Es decir, para cada búsqueda con éxito en la posición i, hay una búsqueda con éxito correspondiente en la posición $N - i$ (suponiendo que empezamos a numerar desde 1). Sin embargo, $N/2$ sigue siendo $O(N)$. Como mencionamos anteriormente en este mismo capítulo, todos estos términos en notación O deberían ser, más correctamente, términos Θ. Sin embargo, el uso de la notación O es más popular.

5.6.2 Búsqueda binaria

Si el vector de entrada está ordenado, entonces existe una alternativa a la búsqueda secuencial, la *búsqueda binaria*. La búsqueda binaria se realiza desde la mitad del vector, en vez de desde el final.

En todo momento, mantendremos dos variables `inicio` y `fin`, que delimitan la parte del vector en la que el elemento, si aparece, debe encontrarse. Inicialmente, el rango es desde 0 hasta $N - 1$. Si `inicio` es mayor que `fin`, sabemos que el elemento no está presente, así que lanzamos una excepción. En otro caso, igualamos la variable `medio` al punto medio en el rango actual (redondeando hacia abajo si el rango tiene un número par de elementos) y comparamos el elemento que estamos buscando con el elemento en la posición `medio`. Si hay éxito, hemos acabado. Si el elemento que estamos buscando es menor que el elemento en la posición `medio` entonces debería estar en el rango desde `inicio` hasta `medio -1`. Si es mayor, entonces debe encontrarse en el rango desde `medio +1` hasta `fin`. En la Figura 5.11, las líneas de la 18 a la 21 modifican el rango posible, esencialmente dividiéndolo por la mitad. Por el principio de divisiones sucesivas por la mitad, sabemos que el número de iteraciones será $O(\log N)$.

En una búsqueda sin éxito, el número de iteraciones del bucle es $\lfloor \log N \rfloor + 1$. Esto es debido a que el rango de búsqueda se divide por la mitad (redondeando hacia abajo si el rango tiene un número par de elementos) en cada iteración. Añadimos 1 porque el último rango abarca cero elementos. En una búsqueda con éxito, el caso peor consiste en $\lfloor \log N \rfloor$ iteraciones, porque en el caso peor reducimos la parte de vector hasta un único elemento. En el caso medio precisaremos sólo una iteración menos. Esto se debe a que la búsqueda de la mitad de los elementos nos conduce al caso peor, un cuarto de los elementos ahorran una búsqueda, y sólo uno entre 2^i elementos ahorrará i iteraciones con respecto al caso peor. Hemos pues de calcular una media ponderada, calculando la suma de una serie finita. El resultado es que el tiempo medio de ejecución para cada búsqueda es $O(\log N)$. En el Ejercicio 5.19 se pide completar este cálculo.

Para valores de N razonablemente grandes, la búsqueda binaria se ejecuta de forma más eficiente que la búsqueda secuencial. Por ejemplo, si N es 1.000, la

```
1  /**
2   * Realiza la búsqueda binaria estándar
3   * utilizando dos comparaciones por nivel.
4   * @exception ElementoNoEncontrado si es apropiado.
5   * @return índice donde se encuentra el elemento.
6   */
7  public static int busquedaBinaria( Comparable [ ] a,
8                                     Comparable x ) throws ElementoNoEncontrado
9  {
10     int inicio = 0;
11     int fin = a.length - 1;
12     int medio;
13
14     while( inicio <= fin )
15     {
16         medio = ( inicio + fin ) / 2;
17
18         if( a[ medio ].compara( x ) < 0 )
19             inicio = medio + 1;
20         else if( a[ medio ].compara( x ) > 0 )
21             fin = medio - 1;
22         else
23             return medio;
24     }
25
26     throw new ElementoNoEncontrado ( "La búsqueda binaria falla" );
27 }
```

Figura 5.11 Búsqueda binaria básica que distingue tres casos.

búsqueda secuencial requiere en promedio 500 comparaciones aproximadamente. En media, la búsqueda binaria requiere 8 iteraciones en una búsqueda con éxito (utilizando la fórmula anterior). Ya que cada iteración utiliza 1,5 comparaciones en promedio (unas veces 1 y otras 2), el total es de 12 comparaciones para una búsqueda con éxito. La búsqueda binaria mejora aún más en el caso peor o cuando las búsquedas no tienen éxito.

Si queremos hacer la búsqueda binaria aún más rápida, tenemos que hacer más ajustado el bucle más interno. Una estrategia posible es eliminar la comprobación (implícita) en el caso de una búsqueda con éxito del bucle más interno y reducir el rango a un solo elemento en todos los casos. Entonces podemos utilizar una comprobación más fuera del bucle para determinar si el elemento buscado está en el vector o no, como se muestra en la Figura 5.12. Si el elemento que estamos buscando en la Figura 5.12 no es mayor que el elemento en la posición `medio`, entonces está en el rango que incluye la posición `medio`. Cuando salimos del bucle, el rango es 1, y podemos comprobar si está ahí el elemento buscado.

En el algoritmo mejorado, el número de iteraciones es siempre $\lfloor \log N \rfloor$, pues siempre se reduce el rango a la mitad, posiblemente redondeando. El número de comparaciones necesarias es siempre $\lfloor \log N \rfloor + 1$.

La búsqueda binaria es sorprendentemente delicada de implementar. El Ejercicio 5.5 ilustra algunos errores comunes.

Observe que para valores N pequeños, como valores menores que 6, puede no merecer la pena la búsqueda binaria. Utiliza más o menos el mismo número de comparaciones para una búsqueda con éxito típica, pero tiene la sobrecarga de la línea 19 en cada iteración. En realidad, las últimas iteraciones de la búsqueda binaria pro-

```
1  /**
2   * Realiza la búsqueda binaria estándar
3   * utilizando una comparación por nivel.
4   * @exception ElementoNoEncontrado si es apropiado.
5   * @return índice donde se encuentra el elemento.
6   */
7  public static int busquedaBinaria( Comparable [ ] a, Comparable x )
8                                      throws ElementoNoEncontrado
9  {
10     if (a.length == 0) throw
11         new ElementoNoEncontrado ( "La búsqueda ha fallado" );
12
13     int inicio = 0;
14     int fin = a.length - 1;
15     int medio;
16
17     while( inicio < fin )
18     {
19         medio = ( inicio + fin ) / 2;
20
21         if( a[ medio ].compara( x ) < 0 )
22             inicio = medio + 1;
23         else
24             fin = medio;
25     }
26
27     if (a[ inicio ].compara( x ) == 0 )
28         return inicio;
29
30     throw new ElementoNoEncontrado ( "La búsqueda ha fallado" );
31 }
```

Figura 5.12 Búsqueda binaria básica que distingue dos casos.

gresan lentamente. Uno puede adoptar una estrategia híbrida en la cual el bucle de búsqueda binaria termina cuando el rango es pequeño, aplicándose entonces una búsqueda secuencial para terminar. De forma similar, la gente busca en un listín telefónico de forma no secuencial. Una vez que han reducido el rango de la búsqueda a una columna, realizan una búsqueda secuencial. La búsqueda en un listín telefónico no es secuencial, pero tampoco es binaria. Se discute en la próxima sección.

5.6.3 Búsqueda interpolada

La búsqueda binaria es muy rápida sobre un vector estático ordenado. De hecho, es tan rápida que difícilmente podríamos usar algo mejor. Sin embargo, un método de búsqueda estática que es en ocasiones más rápido es la *búsqueda interpolada*. Para que la búsqueda interpolada sea práctica, deben satisfacerse dos hipótesis:

1. Cada acceso debe ser muy costoso comparado con una instrucción típica. Por ejemplo, el vector podría estar en un disco en vez de en memoria principal, y cada comparación requiere un acceso a disco.
2. Los datos además de estar ordenados deben estar también uniformemente distribuidos. Por ejemplo, un listín telefónico está uniformemente distribuido. Por el contrario, si los elementos de la entrada son {1, 2, 4, 8, 16, ...}, entonces la distribución no sería uniforme.

Estas hipótesis son bastante restrictivas, de modo que es posible que en la práctica nunca nos convenga utilizar una búsqueda interpolada. Pero es interesante ver que hay más de una forma de resolver un problema, y que ningún algoritmo, incluso la clásica búsqueda binaria, es el mejor en todas las situaciones.

La idea básica de la búsqueda interpolada consiste en que estamos dispuestos a invertir más tiempo si con ello conseguimos una suposición más certera con respecto a donde podría encontrarse el elemento. La búsqueda binaria utiliza siempre el punto medio. Sin embargo, sería absurdo buscar a *Pablo Álvarez* en la mitad del listín telefónico; algún sitio cercano al principio sería claramente más apropiado. Por tanto, en vez de `medio`, utilizamos `proximo` para indicar el próximo elemento al que accederemos.

Veamos ahora un ejemplo de hasta qué punto la búsqueda interpolada podría ser buena. Supongamos que el rango de búsqueda contiene 1.000 elementos, el elemento menor en el rango es 1.000, el mayor es 1.000.000, y estamos buscando el valor 12.000. Si los elementos están uniformemente distribuidos, entonces podemos esperar encontrarlo en algún lugar cercano al duodécimo elemento. La fórmula aplicable es

$$próximo = inicio + \left\lceil \frac{x - a[inicio]}{a[fin] - a[inicio]} \times (fin - inicio - 1) \right\rceil$$

El sustraer 1 es un ajuste técnico que se ha mostrado positivo en la práctica. Claramente, este cálculo es más costoso que el cálculo de la búsqueda binaria. Involucra una división extra (la división por 2 en la búsqueda binaria es simplemente un desplazamiento de un bit, igual que dividir por 10 es sencillo para las personas), una multiplicación y cuatro restas. Estos cálculos deben ser realizados utilizando operaciones de punto flotante. Una iteración podría incluso ser más lenta que la búsqueda binaria completa. Sin embargo, si el coste de estas operaciones es insignificante comparado con el coste de acceder a un elemento, entonces no es importante; sólo debemos preocuparnos del número de iteraciones.

En el caso peor, cuando los datos no están uniformemente distribuidos, el tiempo de ejecución podría ser lineal y todos los elementos podrían tener que ser examinados. El Ejercicio 5.18 pide construir uno de estos casos. Sin embargo, bajo la suposición de que los elementos están razonablemente distribuidos, como en un listín telefónico, el número medio de comparaciones es $O(\log \log N)$. Esto significa que aplicamos el logaritmo dos veces seguidas. Para $N = 4$ billones, $\log N$ es aproximadamente 42, y $\log \log N$ es más o menos 5. Por supuesto, hay algunas constantes multiplicativas ocultas en la notación O, pero el logaritmo extra puede reducir considerablemente el número de iteraciones, mientras no se presente un caso malo. Sin embargo, probar el resultado de forma rigurosa es bastante complicado.

5.7 Comprobar el análisis de un algoritmo

Una vez que hemos efectuado el análisis de un algoritmo, queremos ver si es correcto y tan bueno como sea posible. Una forma de hacerlo es implementar el programa y ver si el tiempo de ejecución observado empíricamente se ajusta con el tiempo de ejecución predicho por el análisis.

Cuando N se incrementa por un factor de diez, el tiempo de ejecución se eleva en un factor de 10 en los programas lineales, de 100 en los programas cuadráticos y de 1.000 en los cúbicos. Los programas que se ejecutan en un tiempo $O(N \log N)$ tardarán algo más de diez veces más en las mismas circunstancias. Estos aumentos pueden ser difíciles de observar si los términos de orden inferior tienen coeficientes relativamente grandes y N no es suficientemente grande. Un ejemplo, es el salto de $N = 10$ a $N = 100$ en el tiempo de ejecución de las diferentes implementaciones para el problema del cálculo de la subsecuencia de suma máxima. También puede ser difícil diferenciar un programa lineal de uno $O(N \log N)$ basándose tan sólo en la evidencia empírica.

Otro truco utilizado a menudo para verificar si un programa es $O(F(N))$ es calcular los valores de $T(N)/F(N)$ para una serie de valores de N (generalmente espaciados por múltiplos de dos), donde $T(N)$ es el tiempo de ejecución observado empíricamente. Si $F(N)$ es una respuesta ajustada al tiempo de ejecución, entonces los valores calculados convergen a una constante positiva. Si $F(N)$ es una sobreestimación, los valores convergen a cero. Si $F(N)$ es una subestimación, y por tanto errónea, los valores divergen.

Por ejemplo, supongamos que escribimos un programa para realizar N búsquedas aleatorias utilizando el algoritmo de búsqueda binaria. Ya que cada búsqueda es logarítmica, esperamos que el tiempo de ejecución total del programa sea $O(N \log N)$. La Figura 5.13 muestra los valores observados para esta rutina, para varios tamaños de la entrada, sobre una computadora real. La tabla muestra que la última columna es la columna convergente, confirmando, por tanto, nuestra predicción, mientras que los números crecientes para T/N sugieren que $O(N)$ es una subestimación, y los valores rápidamente decrecientes de T/N^2 sugieren que $O(N^2)$ es una sobreestimación.

Observe en particular que no tenemos una convergencia absolutamente diáfana. Un problema es que el reloj que utilizamos para medir el tiempo del programa

N	Tiempo de CPU T (milisegundos)	T/N	T/N^2	$T/(N \log N)$
10.000	100	0,01000000	0,00000100	0,00075257
20.000	200	0,01000000	0,00000050	0,00069990
40.000	440	0,01100000	0,00000027	0,00071953
80.000	930	0,01162500	0,00000015	0,00071373
160.000	1.960	0,01225000	0,00000008	0,00070860
320.000	4.170	0,01303125	0,00000004	0,00071257
640.000	8.770	0,01370313	0,00000002	0,00071046

Figura 5.13 Tiempo de ejecución empírico para N búsquedas binarias en un vector de N elementos.

hace tictac sólo cada 10 ms. Observe también que no hay una gran diferencia entre $O(N)$ y $O(N \log N)$. Por supuesto, un algoritmo $O(N \log N)$ está más cerca de ser lineal que de ser cuadrático. Finalmente, tenga en cuenta que la máquina utilizada en el ejemplo tiene suficiente memoria para almacenar 640.000 objetos (en el caso de este experimento, enteros). Si esto no es cierto en su máquina, entonces no podrá reproducir similares resultados.

La próxima sección discute algunas de las limitaciones del análisis basado en la notación O.

5.8 Limitaciones del análisis O

El análisis O es una herramienta muy efectiva, pero es importante conocer sus limitaciones. Como ya se ha mencionado, no es apropiado para pequeñas cantidades de datos. Con pequeñas entradas, utilice el algoritmo más simple. También, para un algoritmo particular, la constante implícita en la notación O puede resultar demasiado grande en la práctica. Por ejemplo, si el tiempo de ejecución de un algoritmo está gobernado por la fórmula $2N \log N$ y otro tiene un tiempo de ejecución de $1.000N$, entonces el primer algoritmo será probablemente mejor, aunque su tasa de crecimiento sea mayor. Las constantes grandes pueden entrar en juego cuando un algoritmo es excesivamente complejo. También aparecen porque nuestro análisis no tiene en cuenta las constantes y por tanto no puede diferenciar cosas como accesos a memoria (que no son costosos) de accesos a disco (que típicamente son muchos miles de veces más costosos). Nuestro análisis supone memoria infinita, pero en las aplicaciones que involucran grandes conjuntos de datos, la falta de memoria suficiente puede ser un serio problema.

A veces, aun cuando las constantes y los términos no dominantes son tenidos en cuenta, se prueba empíricamente que el análisis era una sobreestimación. En tal caso, el análisis debe ser ajustado (generalmente mediante una observación más inteligente). Otras veces la cota de tiempo de ejecución en promedio puede ser significativamente menor que la cota en el caso peor, y por tanto ninguna mejora en la cota es posible. Hay muchos algoritmos complicados para los cuales la cota en el caso peor se alcanza para una mala entrada, pero en la práctica es generalmente una sobreestimación. Dos ejemplos son los métodos de ordenación Shellsort y quicksort (ambos se discuten en el Capítulo 8).

Sin embargo, las cotas en el caso peor son normalmente más fáciles de obtener que sus contrapartidas en el caso medio. Por ejemplo, hasta la fecha no se ha podido obtener un análisis matemático del tiempo de ejecución en el caso medio de Shellsort. A veces, incluso definir con exactitud lo que significa «caso medio» es difícil. Utilizamos un análisis en el caso peor porque es conveniente, y también porque, en la mayoría de los casos, el análisis en el caso peor es bastante significativo. Al realizar el análisis, frecuentemente podremos decir si también sería aplicable al caso promedio.

Resumen

En este capítulo se ha introducido el amplio tema del análisis de algoritmos y se ha mostrado que decisiones algorítmicas generalmente influyen más en el tiempo

de ejecución de un programa que los trucos de programación. También se ha mostrado la enorme diferencia entre los algoritmos lineales y cuadráticos y se ha visto cómo los algoritmos cúbicos son, en la mayor parte, insatisfactorios. Se ha examinado un algoritmo que podría tomarse como base para nuestra primera estructura de datos. La búsqueda binaria soporta de forma eficiente operaciones estáticas (es decir, búsqueda pero no actualización), proporcionando por tanto una búsqueda logarítmica en el caso peor. Capítulos posteriores del texto examinan estructuras de datos dinámicas que soportan eficientemente las actualizaciones (tanto inserción como eliminación).

En el próximo capítulo definimos las estructuras de datos y sus operaciones permitidas. También se detiene en algunas aplicaciones de las estructuras de datos y discute su eficiencia.

Elementos del juego

algoritmo lineal Un algoritmo cuyo tiempo de ejecución crece como $O(N)$. Si el tamaño de la entrada aumenta por un factor f, entonces el tiempo de ejecución también aumenta por el mismo factor. Es el tiempo de ejecución más favorable para un algoritmo.

búsqueda binaria El método de búsqueda utilizado cuando el vector de entrada está ordenado. Las búsquedas se realizan desde la mitad en vez desde el final. La búsqueda binaria es logarítmica porque el rango de búsqueda se reduce a la mitad en cada iteración.

búsqueda estática Encuentra un elemento en un conjunto de datos que nunca es modificado.

búsqueda interpolada Algoritmo de búsqueda estática que tiene mejor rendimiento O en el caso medio que la búsqueda binaria, pero que sólo se puede utilizar en casos limitados y tiene una complejidad mala en el caso peor.

búsqueda secuencial Método lineal de búsqueda que recorre el vector hasta que encuentra el elemento.

cota en el caso medio La cota en la cual el tiempo de ejecución se obtiene mediante la media sobre todas las posibles entradas de tamaño N.

cota en el caso peor Una garantía del coste sobre todas las entradas de cierto tamaño.

logaritmo El logaritmo de N (en base 2) es el valor X tal que 2 elevado a X es igual a N.

notación Ω Notación similar al mayor o igual cuando consideramos tasas de crecimiento.

notación Θ Notación similar al igual cuando consideramos tasas de crecimiento.

notación o Notación similar al menor estricto cuando se consideran tasas de crecimiento.

notación O Notación utilizada para capturar el término más dominante de una función. La notación O es similar al menor o igual cuando consideramos tasas de crecimiento.

números armónicos El número armónico N-ésimo es la suma de los inversos de los N primeros enteros positivos. La tasa de crecimiento de los números armónicos es logarítmica.

principio de duplicaciones repetidas Principio por el cual si empezamos con 1, podemos multiplicar por 2 sólo un número logarítmico de veces, hasta alcanzar N.

principio de sucesivas divisiones por la mitad Principio por el cual si empezamos con N, podemos dividir de forma repetida por la mitad solamente un número de veces logarítmico, hasta alcanzar 1. Se utiliza para obtener rutinas de búsqueda logarítmicas.

subcuadrático Algoritmo cuyo tiempo de ejecución es estrictamente menor que cuadrático. El tiempo de ejecución puede ser escrito como $o(N^2)$.

Errores comunes

1. En el caso de bucles anidados, el tiempo total es función del producto del tamaño de los bucles. En el caso de bucles consecutivos, no.
2. No cuente simplemente el número de bucles. Dos bucles anidados, cada uno de los cuales se ejecuta desde 1 hasta N^2, representan un tiempo $O(N^4)$.
3. No escriba expresiones como $O(2N^2)$ o $O(N^2+N)$. Sólo es necesario el término dominante, sin la constante multiplicativa.
4. Utilice igualdades con la notación O, Ω, etc. No escriba que el tiempo de ejecución es $>O(N^2)$; esto no tiene sentido porque la notación O es una cota superior. No escriba que el tiempo de ejecución es $<O(N^2)$; si el propósito es decir que el tiempo de ejecución es estrictamente menor que cuadrático, utilice la notación o.
5. Utilice la notación Ω, y no la notación O, para una cota inferior.
6. Utilice el logaritmo para describir el tiempo de ejecución de un problema resuelto dividiendo por la mitad su tamaño en tiempo constante. No obstante si tarda más de un tiempo constante en dividir por la mitad el tamaño, el logaritmo no se aplica.
7. La base del logaritmo no es importante cuando se utiliza la notación O. Es por tanto innecesario precisarla, aunque por ello mismo podemos elegir en cada caso la base más favorable para nuestros razonamientos.

En Internet

Los tres algoritmos para la obtención de la subsecuencia de suma máxima, así como un cuarto tomado de la Sección 7.5, están disponibles, junto con una función `main` que realiza los tests de tiempo de ejecución. También se proporciona un algoritmo de búsqueda binaria en el paquete `DataStructures`. He aquí los nombres de ficheros y los directorios:

BinarySearch.java	Traducido por `BusquedaBinaria.java`, contiene el algoritmo de la búsqueda binaria de la Figura 5.12, aunque ligeramente modificado. Se encuentra en el directorio **DataStructures**. La Figura 5.11 se encuentra en el directorio **Chapter05**.
MaxSumTest.java	Traducido por `TestSumaMax.java`, contiene cuatro algoritmos para el problema de la subsecuencia de suma máxima. Se encuentra en el directorio **Chapter05**.

Ejercicios

Cuestiones breves

5.1. Se extraen tres bolas de una caja como se especifica en el Teorema 5.1 obteniéndose las combinaciones desde *a*) hasta *d*). ¿Cuáles son los correspondientes valores de i, j y k?

a) Roja, 5, 6.
b) Azul, 5, 6.
c) Azul, 3, Roja.
d) 6, 5, Roja.

5.2. ¿Por qué no es suficiente una implementación basada exclusivamente en el Teorema 5.2 para obtener un tiempo de ejecución subcuadrático para el problema de la obtención de la subsecuencia de suma máxima?

5.3. Suponga que $T_1(N) = O(F(N))$ y $T_2(N) = O(F(N))$. ¿Cuáles entonces de las siguientes afirmaciones son ciertas?

a) $T_1(N) + T_2(N) = O(F(N))$.
b) $T_1(N) - T_2(N) = O(F(N))$.
c) $T_1(N)/T_2(N) = O(1)$.
d) $T_1(N) = O(T_2(N))$.

5.4. Se analizan dos programas A y B y se ve que en el caso peor tienen tiempos de ejecución no mayores que $150N \log N$ y N^2, respectivamente. Responda a las siguientes cuestiones, cuando ello sea posible:

a) ¿Qué programa da la mejor garantía en tiempo de ejecución para valores grandes de N ($N > 10.000$)?
b) ¿Qué programa da la mejor garantía en tiempo de ejecución para valores pequeños de N ($N < 100$)?
c) ¿Qué programa se ejecutará más rápido *en promedio* para $N = 1.000$?
d) ¿Es posible que el programa B se ejecute más rápidamente que el programa A en *todas* las posibles entradas?

5.5. Para la rutina de búsqueda binaria de la Figura 5.11, muestre las consecuencias de las siguientes modificaciones en el código:

a) Línea 14: utilizando el test `inicio < fin`.
b) Línea 16: haciendo la asignación `medio = inicio + fin / 2`.
c) Línea 19: asignando `inicio = medio`.
d) Línea 21: asignando `fin = medio`.

Problemas teóricos.

5.6. Determine, para los algoritmos típicos que utiliza para realizar cálculos a mano, el tiempo de ejecución en hacer los siguiente:

a) Sumar dos enteros de N dígitos.
b) Multiplicar dos enteros de N dígitos.
c) Dividir dos enteros de N dígitos.

5.7. En términos de N, ¿cuál es el tiempo de ejecución del siguiente algoritmo para calcular X^N?

```
public static double potencia( double x, int n )
{
    double resultado = 1.0;

    for( int i = 0; i < n; i++ )
        resultado *= x;
    return resultado;
}
```

5.8. Evalúe directamente el triple sumatorio que precede al Teorema 5.1. Verifique que la respuesta es idéntica a la allí obtenida.

5.9. Para el algoritmo cuadrático, determine con precisión cuántas veces se ejecuta el bucle más interno.

5.10. Un algoritmo tarda 0,5 milisegundos para una entrada de tamaño 100. ¿Cuánto tardará con una entrada de tamaño 500 si el tiempo de ejecución es el siguiente?

a) Lineal.
b) $O(N \log N)$.
c) Cuadrático.
d) Cúbico.

5.11. Un algoritmo tarda 0,5 milisegundos para una entrada de tamaño 100. ¿Qué tamaño de entrada puede procesar en un minuto, si el tiempo de ejecución es el siguiente?

a) Lineal.
b) $O(N \log N)$.
c) Cuadrático.
d) Cúbico.

5.12. Complete la tabla de la Figura 5.10 con estimaciones para tiempos de ejecución que son demasiado grandes para ser simulados. Interpole los tiempos de ejecución para los cuatro algoritmos y estime el tiempo necesario para calcular la subsecuencia de suma máxima de un millón de números. ¿Qué hipótesis ha asumido?

5.13. Ordene las siguientes funciones por tasa de crecimiento: N, $\sqrt{N}$, $N^{1.5}$, N^2, $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log (N^2)$, $2/N$, 2^N, $2^{N/2}$, 37, N^3 y $N^2 \log N$. Indique qué funciones crecen a la misma velocidad.

5.14. Para cada uno de los siguientes fragmentos de programas, haga lo siguiente:

a) Dé un análisis en notación O del tiempo de ejecución.
b) Implemente el código y ejecútelo para diferentes valores de N.
c) Compare su análisis con los tiempos de ejecución reales.

```
// Fragmento #1
for( int i = 0; i < n; i++ )
    suma++;
```

```
// Fragmento #2
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++)
        suma++;

// Fragmento #3
for( int i = 0; i < n; i++ )
    suma++;
for( int j = 0; j < n; j++)
    suma++;

// Fragmento #4
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++)
        suma++;

// Fragmento #5
for( int i = 0; i < n; i++ )
    for( int j = 0; j < i; j++)
        suma++;

// Fragmento #6
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++)
        for( int k = 0; k < j; k++)
            suma++;
```

5.15. En ocasiones, multiplicar el tamaño de los bucles anidados puede hacer que el tiempo de ejecución en notación O sea una sobreestimacion. Repita el Ejercicio 5.14 para el siguiente fragmento de programa:

```
for( int i = 1; i < n; i++ )
    for( int j = 0; j < i * i; j++ )
        if( j % i == 0 )
            for( int k = 0; k < j; k++ )
                suma++;
```

5.16. En un juicio reciente, un juez condenó a una ciudad por desacato y ordenó una multa de 2 euros para el primer día. Cada día posterior, hasta que la ciudad acatara la orden del juez, la multa sería elevada al cuadrado (es decir, la multa aumentaría de la siguiente manera: 2 euros, 4 euros, 16 euros, 256 euros, 65536 euros, ...).

a) ¿Cuál sería la multa el día N?
b) ¿Cuántos días harían falta para que la multa alcanza E euros? (Una respuesta en notación O será suficiente.)

5.17. Demuestre el Teorema 5.5. *Pista*: Muestre que $\sum_2^N \frac{1}{i} < \int_1^N \frac{dx}{x}$. Obtenga luego de forma similar una cota inferior.

5.18. Construya un ejemplo en el cual una búsqueda interpolada examine cada elemento del vector.

5.19. Analice el coste en promedio de una búsqueda con éxito para el algoritmo de búsqueda binaria.

Problemas prácticos

5.20. Escriba un algoritmo eficiente para determinar si existe un entero i tal que $A_i = i$ en un vector de enteros ordenados de forma creciente. ¿Cuál es su tiempo de ejecución?

5.21. Un número primo no tiene divisores aparte de 1 y de él mismo. Se pide lo siguiente:

a) Escriba un programa para determinar si un entero positivo N es primo. En términos de N, ¿cuál es el tiempo de ejecución en el caso peor de su programa?
b) Sea B el número de bits en la representación binaria de N. ¿Cuál es el valor de B?
c) En términos de B, ¿cuál es el tiempo de ejecución en el caso peor de su programa?
d) Compare los tiempos de ejecución necesarios para determinar si un número de 20 bits y otro de 40 bits son primos.

5.22. Un problema importante en análisis numérico consiste en encontrar una solución a la ecuación $F(X) = 0$ para un F arbitrario. Si la función es continua y tenemos dos puntos *bajo* y *alto* para los cuales $F(bajo)$ y $F(alto)$ tienen signo opuesto, entonces debe existir una raíz entre *bajo* y *alto*, que puede ser encontrada bien mediante una búsqueda binaria o con una búsqueda interpolada. Escriba una función que tome como parámetros F, *bajo* y *alto* y encuentre una raíz de la ecuación. ¿Qué debe hacer para asegurar la terminación?

5.23. Un elemento mayoritario en un vector A de tamaño N es un elemento que aparece más de $N/2$ veces (en consecuencia habrá a lo sumo uno). Por ejemplo, el vector

3, 3, 4, 2, 4, 4, 2, 4, 4

contiene un elemento mayoritario (4), mientras que el vector

3, 3, 4, 2, 4, 4, 2, 4

no tiene elemento mayoritario. Escriba un algoritmo para encontrar el elemento mayoritario cuando exista, o que diga que no exista. ¿Cuál es el tiempo de ejecución de su algoritmo? (Hay una solución $O(N)$.)

Prácticas de programación

5.24. La criba de Eratóstenes es un método utilizado para calcular todos los primos menores que N. Empiece haciendo una tabla de enteros desde 2 hasta N. Encuentre el menor entero i que no esté tachado. Entonces imprima i y tache i, $2i$, $3i$, etc. Cuando $i > \sqrt{N}$, el algoritmo termina. Se ha probado que el tiempo de ejecución es $O(N \log \log N)$. Escriba un programa para implementar la criba y verifique que el tiempo de ejecución es en efecto el señalado. ¿Qué dificultad ha encontrado para distinguir el tiempo de ejecución entre $O(N)$ y $O(N \log N)$?

5.25. La ecuación $A^5 + B^5 + C^5 + D^5 + E^5 = F^5$ tiene exactamente una solución entera que satisfaga $0 < A \leqslant B \leqslant C \leqslant D \leqslant E \leqslant F \leqslant 75$. Escriba un programa para encontrar dicha solución. *Pista*: primero, calcule todos los valores de X^5 y almacénelos en un vector. Entonces, para cada tupla (A, B, C, D, E), sólo necesita comprobar si existe algún F en el vector. Hay varias formas de buscar F. Un método consiste en aplicar búsqueda binaria. Sin embargo podríamos probar que otros métodos pueden ser más eficientes.

5.26. Implemente los algoritmos para el problema de la obtención de la subsecuencia de suma máxima, para obtener datos equivalentes a los de la Figura 5.10. Compile los programas con las opciones que conduzcan a una mayor eficiencia.

Bibliografía

El problema de la obtención de la subsecuencia de suma máxima se ha tomado de [5]. Los libros [4], [5] y [6] muestran cómo optimizar la velocidad de los programas. La búsqueda interpolada fue sugerida por primera vez en [14] y analizada en [13]. Los libros [1], [8] y [17] proporcionan un tratamiento más riguroso del análisis de algoritmos. La serie de tres partes [10], [11] y [12], actualizada recientemente, sigue siendo la principal referencia en el tema.

Los conocimientos matemáticos necesarios para el análisis de algoritmos más avanzado se proporciona en [2], [3], [7], [15] y [16]. Un libro especialmente bueno de análisis avanzado es [9].

1. A. V. Aho, J. E. Hopcroft, y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).
2. M. O. Albertson y J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, New York, NY (1988).
3. Z. Bavel, *Math Companion for Computer Science*, Reston Publishing Company, Reston, Va (1982).
4. J. L. Bentley, *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, NJ (1982).
5. J. L. Bentley, *Programming Pearls*, Addison-Wesley. Reading, Mass. (1986).
6. J. L. Bentley, *More Programming Pearls*, Addison-Wesley, Reading, Mass. (1988).
7. R. A. Brualdi, *Introductory Combinatorics*, North-Holland, New York, NY (1977).
8. T. H. Cormen, C. E. Leiserson, y R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass. (1990).
9. R. L. Graham, D. E. Knuth, y O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, Mass. (1989).
10. D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, 3.ª ed., Addison-Wesley, Reading, Mass. (1997).
11. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, 3.ª ed., Addison-Wesley, Reading. Mass. (1997).
12. D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching*, 2.ª ed., Addison-Wesley, Reading, Mass. (1997).
13. Y. Pearl, A. Itai, y H. Avni, «Interpolation Search—A log log N Search», *Communications of the ACM* **21** (1978), 550-554.
14. W. W. Peterson, «Addressing for Random Storage», *IBM Journal of Research and Development* **1** (1957), 131-132.
15. F. S. Roberts, *Applied Combinatorics*, Prentice-Hall, Englewood Cliffs, NJ (1984).
16. A. Tucker, *Applied Combinatorics*, 2.ª ed., John Wiley & Sons, New York, NY (1984).
17. M. A. Weiss, *Data Structures and Algorithm Analysis in C*, 2.ª ed., Benjamin/Cummings, Redwood City, Calif. (1997).

CAPÍTULO

6 Estructuras de datos

Muchos algoritmos requieren una representación apropiada de los datos para lograr ser eficientes. Esta representación junto con las operaciones permitidas se llama *estructura de datos*. Típicamente todas las estructuras de datos permiten inserciones arbitrarias. Las estructuras de datos varían en cómo permiten el acceso a miembros del grupo. Algunas permiten tanto accesos como operaciones de borrado arbitrarios. Otras imponen restricciones, tales como permitir el acceso sólo al elemento más recientemente insertado, o al menos recientemente insertado del grupo.

Este capítulo discute siete de las más comunes estructuras de datos: pilas, colas, listas enlazadas, árboles, árboles binarios de búsqueda, tablas hash y colas de prioridad. El objetivo es definir cada estructura de datos, así como dar una idea intuitiva de la complejidad en tiempo de las operaciones de inserción, borrado y acceso. La presentación de una implementación eficiente se retrasa hasta las Partes IV y V.

En este capítulo veremos:

- Descripciones de las estructuras de datos más comunes, sus operaciones y sus tiempos de ejecución.
- Para cada estructura de datos, una interfaz Java que contenga el protocolo que debe ser implementado.
- Algunas aplicaciones de las estructuras de datos.

Estas estructuras de datos son utilizadas en los casos de estudio de la Parte III del libro y son implementadas en la Parte IV. El objetivo es mostrar que la especificación, que describe la funcionalidad, es independiente de la implementación. No necesitamos conocer *cómo* se implementa algo con tal de que sepamos que *está* implementado.

6.1 ¿Por qué necesitamos estructuras de datos?

Las estructuras de datos nos permiten lograr un importante objetivo de la programación orientada a objetos: la reutilización de componentes. Como se muestra más tarde en el capítulo, las estructuras de datos descritas en esta sección (e implementadas más adelante) tienen usos repetidos. Una vez que la estructura de da-

tos ha sido implementada, puede ser utilizada una y otra vez en diversas aplicaciones. En este capítulo sólo se proporciona las correspondientes interfaces. Las consideraciones sobre implementaciones eficientes se retrasan hasta la Parte IV.

Este enfoque —la separación de la interfaz y la implementación— es parte del paradigma de la programación orientada a objetos. El usuario de la estructura de datos no necesita ver la implementación, sólo las operaciones disponibles. Ésta es la parte de ocultamiento y encapsulación, de la programación orientada a objetos. Sin embargo, otra parte importante de la programación orientada a objetos es la *abstracción*. Debemos pensar cuidadosamente el diseño de las estructuras de datos porque debemos escribir programas que utilicen esas estructuras de datos sin tener en cuenta su implementación. Esto hace a la interfaz más limpia, más flexible (más reutilizable) y generalmente más fácil de implementar.

Todas las estructuras son fáciles de implementar si no nos preocupamos del rendimiento. Esto nos permite colocar componentes «baratas» dentro de nuestro programa en la fase de depuración del mismo. Los ejercicios al final de este capítulo le piden escribir implementaciones ineficientes, útiles para procesar pequeñas cantidades de datos. Más tarde, reemplazaremos las implementaciones de estructuras de datos «baratas» por implementaciones que tienen un mejor rendimiento en tiempo y/o espacio, y que son útiles para manejar grandes cantidades de datos. Ya que las interfaces están fijadas, estos cambios no requerirán virtualmente ningún cambio en los programas que utilizan las estructuras de datos.

Este capítulo describe las estructuras de datos utilizando interfaces. En las Partes IV y V implementamos cada interfaz, derivando entonces una nueva clase. Por ejemplo, las pilas se especifican con la interfaz `Pila`. Cuando la implementemos en el Capítulo 15, la clase resultante será denominada `PilaVec` (implementación basada en vectores). En todos los casos, la nueva clase implementará las especificaciones en la interfaz y podrá incluir alguna funcionalidad adicional.

Por ejemplo, la Figura 6.1 presenta una interfaz para la celda de memoria descrita en la Sección 4.6. La interfaz describe las funciones disponibles; la clase derivada concreta debe proporcionar su definición. La implementación se muestra en la Figura 6.2 y es idéntica a la de la Figura 4.16, excepto por lo que se refiere a la cláusula `implements`. La rutina `main` de la Figura 4.17 puede utilizarse sin cambios.

Para ahorrar espacio, las interfaces mostradas en este texto contienen comentarios concisos al comienzo de cada clase, en vez de los comentarios más extensos para Javadoc. El código en Internet tiene ambos tipos de comentarios. Es importante observar que las estructuras de datos implementadas aquí almacenan referencias a los elementos insertados y no realizan copias internas. Por tanto, es una buena

```
1  // Interfaz CeldaMem: simula una celda RAM genérica
2  //
3  // ******************OPERACIONES PÚBLICAS**********************
4  // Object lectura( )          --> Devuelve el valor almacenado
5  // void escritura( Object x ) --> Almacena x
6
7  public interface CeldaMem
8  {
9      Object leer( );
10     void escribir( Object x );
11 }
```

Figura 6.1 Interfaz para la clase de celdas abstractas de memoria.

```
1  // Clase CeldaMemoria
2  //
3  // ******************OPERACIONES PÚBLICAS**********************
4  //
5  // Object leer( )             --> Devuelve el valor almacenado
6  // void escribir( Object x )  --> Almacena x
7
8  public class CeldaMemoria implements CeldaMem
9  {
10         // Métodos públicos
11     public Object leer( ) { return valorAlmacenado; }
12     public void escribir( Object x ) { valorAlmacenado = x; }
13
14         // Representación interna privada del dato
15     private Object valorAlmacenado;
16 }
```

Figura 6.2 Implementación de la clase concreta de celdas de memoria.

práctica colocar objetos inmutables en estas estructuras de datos, de tal forma que un agente externo no pueda cambiar el estado de un objeto que se encuentra dentro de una estructura de datos contenedora.

6.2 Las pilas

Una *pila* es una estructura de datos en la cual el acceso está limitado al elemento más recientemente insertado. Se comporta de forma muy parecida a una pila de facturas, de platos o de periódicos. El último elemento añadido a la pila es colocado en la cima, donde es accedido muy fácilmente, mientras que los elementos que llevan más tiempo son más difíciles de acceder. Por tanto, una pila es apropiada si sólo queremos acceder al elemento en la cima; el resto de elementos son inaccesibles (en principio).

En una pila, las tres operaciones naturales de `insertar`, `eliminar` y `buscar`, se renombran por `apilar`, `desapilar` y `cima`. Estas operaciones básicas se ilustran en la Figura 6.3. En la Figura 6.4 se muestra una interfaz Java para una pila abstracta. Se incluye un método `cimaYDesapilar`, que combina estas dos operaciones. En este ejemplo no aparece ninguna característica nueva de Java. La Figura 6.5 muestra cómo se utiliza una clase `Pila` y proporciona la salida correspondiente. Observe que una pila puede ser utilizada para dar la vuelta a una lista de elementos.

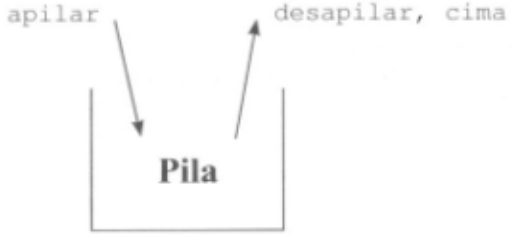


Figura 6.3 Modelo de una pila: la entrada a la pila se realiza con `apilar`, el acceso con `cima` y el borrado con `desapilar`.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Interfaz Pila
6  //
7  // ********************OPERACIONES PÚBLICAS*********************
8  // void apilar( x )           --> Inserta x
9  // void desapilar( )          --> Elimina el último elemento insertado
10 // Object cima( )             --> Devuelve el último elemento insertado
11 // Object cimaYDesapilar( )   --> Devuelve y elimina el último elemento
12 // boolean esVacia( )         --> Devuelve true si vacía; si no false
13 // void vaciar( )             --> Elimina todos los elementos
14 // ************************ERRORES*****************************
15 // cima, desapilar, o cimaYDesapilar sobre la pila vacía
16
17 public interface Pila
18 {
19     void    apilar( Object x );
20     void    desapilar( )        throws DesbordamientoInferior;
21     Object  cima( )             throws DesbordamientoInferior;
22     Object  cimaYDesapilar( )   throws DesbordamientoInferior;
23     boolean esVacia( );
24     void    vaciar( );
25 }
```

Figura 6.4 Interfaz para las pilas.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las pilas
5
6  public final class TestPila
7  {
8      public static void main( String [ ] args )
9      {
10         Pila p = new PilaVec( );
11
12         for( int i = 0; i < 5; i++ )
13             p.apilar( new Integer( i ) );
14
15         System.out.print( "Contenido:" );
16         try
17         {
18             for( ; ; )
19                 System.out.print( " " + p.cimaYDesapilar( ) );
20         }
21         catch( DesbordamientoInferior e ) { }
22
23         System.out.println( );
24     }
25 }
```

Figura 6.5 Programa ejemplo con pilas: la salida es `Contenido: 4 3 2 1 0`.

Cada operación de las pilas debería tardar una cantidad constante de tiempo en ejecutarse, independientemente del número de elementos apilados. De forma análoga, encontrar el periódico de hoy en una pila de periódicos es rápido, independientemente de lo alta que sea la pila. Sin embargo, el acceso arbitrario a una pila no se soporta de forma eficiente, por lo que no se cataloga como opción.

Lo que hace que las pilas sean útiles es que hay muchas aplicaciones en las que sólo es necesario acceder al elemento más recientemente insertado. Como ilustración, en la próxima sección se describe una utilización importante de las pilas en el diseño de compiladores.

6.2.1 Las pilas y los lenguajes de programación

Los compiladores comprueban los programas buscando errores sintácticos. A menudo, sin embargo, la falta de un símbolo (como por ejemplo la falta de un finalizador de comentario `*/` o `}`) causa que el compilador desparrame cientos de líneas de mensajes de error sin identificar el error real.

Una herramienta útil en esta situación es un programa que comprueba si todo está equilibrado: es decir, si para todo `{` existe un `}` correspondiente, todo `[` tiene un `]`, etc. La secuencia `[()]` es correcta, pero `[(])` no lo es, por lo que contar simplemente el número de apariciones de cada símbolo no es suficiente. (Suponga por ahora que estamos procesando una secuencia de tokens y no se preocupe de problemas tales como que la constante carácter `'{'` no necesita emparejarse con `'}'`.)

Una pila es útil para comprobar si hay símbolos desequilibrados, porque sabemos que cuando se encuentra un símbolo de terminación como `)`, debe emparejarse con el símbolo no cerrado `(` más recientemente visto.

Por tanto, colocando los símbolos de apertura en una pila, podemos fácilmente comprobar si un símbolo de cierre tiene sentido.

En concreto, tenemos el siguiente algoritmo:

1. Construir una pila vacía.
2. Leer símbolos hasta el final del fichero.
   a) Si el símbolo es un símbolo de apertura, apilarlo en la pila.
   b) Si es un símbolo de terminación y la pila está vacía, entonces producir un mensaje de error.
   c) En otro caso, desapilar de la pila. Si el símbolo desapilado no es el símbolo de apertura correspondiente, producir un mensaje de error.
3. Al finalizar el fichero, si la pila no está vacía, mostrar un error.

En la Sección 11.1 desarrollaremos este algoritmo para que trabaje sobre (casi) todos los programas Java. Detallaremos la devolución de errores así como el procesamiento de comentarios, cadenas, constantes de tipo carácter y secuencias de escape.

El algoritmo para comprobar si los símbolos están equilibrados sugiere una forma de implementar las llamadas a procedimientos. El problema es que cuando se hace una llamada a un nuevo método, todas las variables locales al método que realiza la llamada deben ser salvadas por el sistema; de otra forma, el nuevo método podría sobrescribir estas variables. Es más, la posición actual dentro de la rutina

llamante debe ser guardada de manera que el nuevo método sepa dónde volver cuando termine. Este problema es parecido al de los símbolos equilibrados porque una llamada a un método y el regreso de un método son esencialmente iguales que un paréntesis abierto y un paréntesis cerrado, así que se deberían poderse aplicar las mismas ideas. Efectivamente, esto es así. Como se discute en la Sección 7.3, la pila se utiliza en muchos lenguajes para implementar las llamadas a métodos.

Una aplicación final importante de las pilas es la evaluación de expresiones en lenguajes de programación. En la expresión `1 + 2 * 3`, en el momento en que nos encontramos el `*`, el operador `+` y los operandos `1` y `2`, ya han sido leídos. ¿Debe operar `*` sobre `2` o sobre `1 + 2`? Las reglas de precedencia nos dicen que `*` opera sobre `2`, que es el operando más recientemente visto. Después de leer el `3`, podemos evaluar `2 * 3`, obteniendo `6`, para después aplicar el operador `+`. Esto sugiere que los operandos deberían ser guardados en una pila (ya que el `+` es retenido hasta que el operador `*` de mayor precedencia se evalúa). Un algoritmo que utiliza esta estrategia es el algoritmo de *evaluación de expresiones con precedencia entre operadores*. Su funcionamiento se describe en la Sección 11.2.

6.3 Las colas

Otra estructura de datos simple son las *colas*. En muchos casos, es importante poder encontrar y/o eliminar el elemento más recientemente insertado. Pero en otras ocasiones, no sólo no es importante, sino que es la forma errónea de hacerlo. Por ejemplo, en un sistema multiprocesador, cuando los trabajos se mandan a una impresora, esperamos que el trabajo menos reciente o más viejo sea impreso antes. Esto no es solamente justo, sino que es necesario para garantizar que el primer trabajo no espera para siempre. En consecuencia, uno espera encontrar colas de impresión en todos los grandes sistemas.

Las operaciones básicas permitidas por las colas son:

- `insertar` — inserción al final de la línea,
- `quitarPrimero` — eliminación del elemento al frente de la línea, y
- `primero` — acceso al elemento en el frente de la línea.

La Figura 6.6 ilustra estas operaciones sobre colas. Históricamente, `quitarPrimero` y `primero` han sido combinadas en una sola operación. Seguimos aquí esta idea, de modo que `quitarPrimero` devuelve el primer elemento además de eliminarlo de la cola, si bien mantenemos también la operación de acceso `primero`.

La Figura 6.7 ilustra la interfaz de las colas, mientras que la Figura 6.8 muestra cómo se usan las colas, produciendo un ejemplo de salida. Ya que las operaciones de las colas están restringidas de forma similar a las de las pilas, es de esperar

Figura 6.6 Modelo de cola: la entrada se realiza con `insertar`, la salida con `primero` y la eliminación con `quitarPrimero`.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Interfaz Cola
6  //
7  // ******************OPERACIONES PÚBLICAS************************
8  // void insertar( x )       --> Inserta x
9  // Object primero( )        --> Devuelve el primer elemento
10 // Object quitarPrimero( )  --> Devuelve y elimina el primer elemento
11 // boolean esVacia( )       --> Devuelve true si vacía; si no false
12 // void vaciar( )           --> Elimina todos los elementos
13 // ***************************ERRORES*****************************
14 // primero y quitarPrimero sobre una cola vacía
15
16 public interface Cola
17 {
18     void    insertar( Object X );
19     Object  primero( ) throws DesbordamientoInferior;
20     Object  quitarPrimero( ) throws DesbordamientoInferior;
21     boolean esVacia( );
22     void    vaciar( );
23 }
```

Figura 6.7 Interfaz de las colas.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las colas
5
6  public final class TestCola
7  {
8      public static void main( String [ ] args )
9      {
10         Cola c = new ColaVec( );
11
12         for( int i = 0; i < 5; i++ )
13             c.insertar( new Integer( i ) );
14
15         System.out.print( "Contenido:" );
16         try
17         {
18             for( ; ; )
19                 System.out.print( " " + c.quitarPrimero( ) );
20         }
21         catch( DesbordamientoInferior e ) { }
22
23         System.out.println( );
24     }
25 }
```

Figura 6.8 Programa ejemplo con colas: la salida es `Contenido: 0 1 2 3 4`.

que también tarden un tiempo constante en ejecutarse. Así es en efecto. Todas las operaciones básicas sobre las colas tardan un tiempo $O(1)$. En los casos de estudio veremos varias aplicaciones de las colas.

6.4 Listas enlazadas

En una *lista enlazada*, los elementos se almacenan de forma no contigua, en vez de en un vector de posiciones de memoria consecutivas. Para conseguir esto, cada elemento se almacena en un *nodo* que contiene el objeto y una referencia al siguiente nodo en la lista, como se muestra en la Figura 6.9. En este marco, se mantienen referencias al primer y último nodo de la lista. Para ser más precisos, un nodo típico es el siguiente:

```
class NodoLista
{
    Object dato; // Algún elemento
    NodoLista siguiente;
}
```

En cualquier momento, podemos añadir un nuevo último elemento `x` haciendo lo siguiente:

```
ultimo.siguiente = new NodoLista( ); // odjunta un nuevo NodoLista
ultimo = ultimo.siguiente;           // Ajusta el último
ultimo.dato = x;                     // Coloca x en el nodo
ultimo.siguiente = null;             // Es el último; ajusta siguiente
```

Ahora, un elemento cualquiera no puede encontrarse con un único acceso; para ello debemos ir recorriendo la lista. Esto es similar a la diferencia entre acceder a un elemento en un disco compacto (un acceso) o en una cinta (secuencial). Aunque esto puede hacer pensar que las listas son menos atractivas que los vectores, tienen sus ventajas. Primero, una inserción en medio de la lista no requiere mover todos los elementos que se encuentran después del punto de inserción. Los movimientos de datos son muy costosos en la práctica, y las listas enlazadas permiten la inserción con sólo una cantidad constante de instrucciones de asignación.

Merece la pena hacer notar que si permitimos acceso sólo al `primero`, entonces tenemos una pila, y si permitimos inserciones sólo por el `ultimo` y accesos sólo por el `primero`, tenemos una cola.

Sin embargo, en estas situaciones necesitamos habitualmente operaciones más generales, tales como encontrar o eliminar cualquier elemento en la lista, guiados por su nombre. También necesitamos ser capaces de insertar un elemento en cualquier posición. Esto es mucho más que lo que permite una pila o una cola. La Figura 6.10 ilustra estas operaciones de las listas enlazadas.

Para acceder a los elementos en la lista, necesitamos una referencia al correspondiente nodo. Sin embargo, conceder este acceso viola claramente el principio de ocultamiento de la información. Necesitamos asegurar que cualquier acceso a

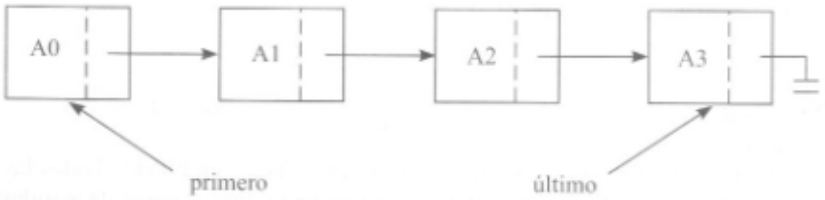


Figura 6.9 Una lista enlazada simple.

Figura 6.10 Modelo de lista enlazada: las entradas son arbitrarias, cualquier elemento puede salir y se soporta la iteración, aunque esta estructura de datos no es eficiente en tiempo.

la lista a través de una referencia es seguro. Para conseguir esto, definimos la lista en dos partes: una clase para las listas y una clase iteradora. La Figura 6.11 muestra la interfaz básica para las listas enlazadas, facilitando métodos que describen solamente el estado de la lista.

La Figura 6.12 define una clase iteradora utilizada en todos los accesos a la lista. Para ver cómo se utiliza, veamos el código estándar para imprimir todos los elementos de la lista. Si la lista estuviera almacenada en un vector contiguo, un código típico sería el siguiente:

```
// Recorre el vector v, mostrando cada elemento
for( int indice = 0; indice < tamanyo; indice++ )
    System.out.println( v[ indice ] );
```

En Java básico, el código para iterar a través de toda la lista enlazada es

```
// Recorre la lista unaLista, mostrando cada elemento
for( NodoLista p = unaLista.primero; p ! = null; p = p.siguiente )
    System.out.println( p.dato );
```

```
1  package EstructurasDatos;
2
3  // Interfaz Lista
4  //
5  // Acceso mediante la clase ListaIter
6  //
7  // ********************OPERACIONES PÚBLICAS*******************
8  // boolean esVacia( )  --> Devuelve true si vacía; si no, false
9  // void vaciar( )      --> Elimina todos los elementos
10 // ***********************ERRORES*****************************
11 // Ningún error especial
12
13 public interface Lista
14 {
15     boolean esVacia( );
16     void vaciar( );
17 }
```

Figura 6.11 Interfaz para la lista abstracta.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Interfaz ListaIter; mantiene la posición actual
6  //
7  // ******************OPERACIONES PÚBLICAS*********************
8  // void insertar( x )    --> Inserta x después de la posición actual
9  // void eliminar( x )    --> Elimina x
0  // boolean buscar( x )   --> Actualiza la posición actual accediendo a x
1  // void cero( )          --> Coloca actual antes que el primer elemento
2  // void primero( )       --> Coloca actual en el primer elemento
3  // void avanzar( )       --> Avanza
4  // boolean estaDentro( )--> true, si está en una posición valida
5  // Object recuperar      --> Devuelve el elemento actual
6  // ********************************ERRORES*********************************
7  // Excepciones lanzadas en acceso, inserción o eliminación ilegal.
8
9  public interface ListaIter
0  {
1      void    insertar( Object x )  throws ElementoNoEncontrado;
2      boolean buscar( Object x );
3      void    eliminar ( Object x ) throws ElementoNoEncontrado;
4      boolean estaDentro( );
5      Object  recuperar( );
6      void    cero( );
7      void    primero( );
8      void    avanzar( );
9  }
```

igura 6.12 Interfaz para el iterador de lista abstracto.

El mecanismo de iteración que Java usaría es similar al siguiente (ya que `ListaIter` es una interfaz, la `ListaIter` que sigue a `new` debería ser reemplazada por alguna clase que implementara a la interfaz `ListaIter`)

```
// Recorre laLista, utilizando la abstracción y el iterador
ListaIter iter = new ListaIter( laLista );
for( iter.primero( ); iter.estaDentro( ); iter.avanzar( ) )
    System.out.println( iter.recuperar( ) );
```

La inicialización anterior al bucle `for` se consigue llamando al constructor de `ListaIter`. El test utiliza el método `estaDentro` definido en la clase `ListaIter`. El método `avanzar` avanza al siguiente nodo de la lista enlazada. Podemos acceder al elemento actual llamando al método `recuperar` definido en `ListaIter`. El principio general es que garantizamos la seguridad, ya que todos los accesos a la lista se hacen a través de la clase `ListaIter`. Por último, indicaremos que podríamos tener varios iteradores recorriendo, simultánea pero independientemente, una única lista.

Para realizar esta tarea, la clase `ListaIter` debe mantener dos objetos internos. Primero, necesita una referencia al elemento «actual». Y en segundo lugar, necesita una referencia a la lista. Esta referencia es inicializada (por única vez) en el constructor. La Figura 6.13 muestra la interacción entre la lista y su iterador. Los detalles de la implementación de la lista enlazada se verán en el Capítulo 16. El uso de las técnicas con pilas y colas se verá en el Capítulo 15.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las listas
5
6  public final class TestLista
7  {
8      public static void main( String [ ] args )
9      {
10         Lista laLista   = new ListaEnlazada( );
11         ListaIter iter  = new ListaEnlazadaIter( laLista );
12
13             // Insertar repetidamente elementos por el principio
14         for( int i = 0; i < 5; i++ )
15         {
16             try
17               { iter.insertar( new Integer( i ) ); }
18             catch( ElementoNoEncontrado e ) { } // No puede ocurrir
19             iter.cero( );                  // Mueve iter al principio
20         }
21
22         System.out.print( "Contenido:" );
23         for( iter.primero( ); iter.estaDentro( ); iter.avanzar( ) )
24             System.out.print( " " + iter.recuperar( ) );
25         System.out.println( " fin" );
26     }
27 }
```

Figura 6.13 Ejemplo de uso de las listas: la salida es `Contenido: 4 3 2 1 0 fin.`

Aunque empezamos la discusión utilizando listas enlazadas, las interfaces de las Figuras 6.11 y 6.12 pueden utilizarse para cualquier lista, independientemente de la implementación subyacente. La interfaz no especifica que deban utilizarse listas enlazadas.

6.5 Árboles generales

Los *árboles* son una estructura de datos utilizada muy a menudo, formada por un conjunto de nodos y un conjunto de aristas que conectan pares de nodos. En este texto sólo se consideran *árboles con raíz*. Un árbol con raíz tiene las siguientes características:

- Un nodo es distinguido como la raíz.
- Todo nodo c, excepto la raíz, está conectado por medio de una arista a un único nodo p. p es el padre de c, y c es uno de los hijos de p.
- Hay un único camino desde la raíz a cada nodo. El número de aristas que deben atravesarse es la longitud del camino.

La Figura 6.14 muestra un árbol. El nodo raíz es A. Los hijos de A son B, C, D y E. Ya que A es la raíz, no tiene padre. El resto de nodos tiene padre. Por ejemplo, el padre de B es A. A los nodos que no tienen hijos se les llama *hojas*. Las hojas de este árbol son los nodos C, F, G, H, I y K. La longitud del camino de A hasta K es tres (aristas). La longitud del camino desde A a A es cero aristas. Pro-

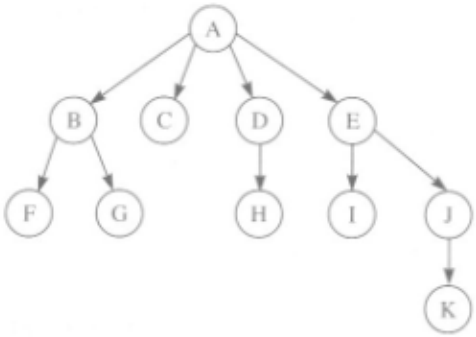


Figura 6.14 Un árbol.

piedades como ascendiente, descendiente o hermano, pueden también definirse con el significado usual. Así, los hermanos de *C* son los nodos *B*, *D* y *E*.

Los árboles son una estructura de datos fundamental en computación. Casi todos los sistemas operativos almacenan los ficheros en estructuras que son árboles o similares. Por ejemplo, bajo DOS, VMS o Unix, los directorios se almacenan en los nodos internos que no son hojas del árbol, mientras que el resto de ficheros se almacenan en las hojas. Atravesar una arista, es lo mismo que descender a un subdirectorio. Una arista especial que conecta de vuelta al padre (a saber, la entrada `..`) permite fácilmente la vuelta atrás en el árbol (aunque por crear ciclos, la entrada `..` hace que ésta estructura sea sólo *parecida* a un árbol, en vez de un árbol propiamente dicho).

Con esto en mente, podemos ver qué tipo de operaciones permiten los árboles. Queremos añadir, buscar y eliminar nodos del árbol. Como en el caso de las listas enlazadas, definiremos un iterador que nos permita acceder a los nodos, manteniendo la noción de nodo actual. En particular, debemos proporcionar métodos que nos permitan recorrer el árbol de manera ordenada. De esta forma, en la Figura 6.15 presentamos las interfaces para la clase de los árboles y su iterador, de forma análoga a como hicimos para las listas enlazadas. Las operaciones `irARaiz`, `primerHijo` y `siguienteHermano` son suficientes para permitirnos recorrer el árbol de diferentes maneras. En el Capítulo 17 describiremos una jerarquía de clases que permiten recorridos de árboles utilizando solamente los métodos `primerHijo`, `recuperar`, `esValido` y `siguienteHermano`. Esto incluye las clases `OrdenSim`, `PreOrden`, `PostOrden` y `PorNiveles`.

Una segunda aplicación de los árboles corresponde a los denominados *árboles de expresiones*, que se ilustran en la Figura 6.16. En el árbol sintáctico de una expresión, el valor de un nodo es el resultado de aplicar el operador en el nodo utilizando como operandos los valores de los hijos. Las hojas se evalúan a sí mismas. Por consiguiente, el árbol sintáctico de la expresión de la Figura 6.16 se evalúa a `(a+b)*(c-d)`. Los árboles sintácticos de expresiones, y el análisis sintáctico del árbol correspondiente, son estructuras de datos esenciales en las etapas de análisis sintáctico y generación de código de un compilador. Se darán más detalles en la Sección 11.2.

El árbol de expresión en la Figura 6.16 es un *árbol binario*, pues el número de hijos está limitado a como mucho dos por nodo. Un uso importante de los árboles binarios se examina en la próxima sección.

```
1  // Interfaz Arbol; los detalles son parecidos a los de Lista
2
3  public interface Arbol
4  {
5      Boolean esVacio( );
6      void    vaciar( );
7  }
8
9
10 // Interfaz ArbolIter; mantiene la posición actual
11
12 public interface ArbolIter
13 {
14     void    insertar( Object x ) throws ElementoNoEncontrado;
15     boolean buscar( Object x );
16     void    eliminar( Object x ) throws ElementoNoEncontrado;
17
18     void    irARaiz( );
19     void    primerHijo( );
20     void    siguienteHermano( );
21     boolean esValido( );
22     Object  recuperar( );
23 }
```

Figura 6.15 Interfaces para la clase abstracta de los árboles y su iterador.

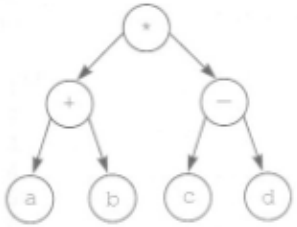


Figura 6.16 Árbol de la expresión `(a+b)*(c-d)`.

6.6 Árboles binarios de búsqueda

En la Sección 5.6, examinamos el problema de la búsqueda estática, y vimos que si los elementos se nos dan ordenados, podemos realizar la operación `buscar` en tiempo logarítmico, en el caso peor. Se trata de búsqueda estática, pues una vez que se nos dan los elementos, no podemos añadir ni eliminar ninguno.

Supongamos ahora que sí necesitamos insertar o eliminar elementos dinámicamente. Una estructura de datos que lo permite son los *árboles binarios de búsqueda*. La Figura 6.17 muestra las operaciones básicas permitidas con los árboles binarios de búsqueda. Su interfaz se muestra en la Figura 6.18. Observe que sólo objetos que implementen la interfaz `Comparable` se pueden almacenar en un árbol binario de búsqueda.

El conjunto de operaciones permitidas se extiende para permitir aplicaciones arbitrarias de `buscar` (por nombre) así como de `insertar` y `eliminar`. El método `buscar` devuelve una referencia al objeto que se ajusta al elemento buscado.

Figura 6.17 Modelo de árbol binario de búsqueda. La búsqueda binaria se generaliza para permitir inserciones y eliminaciones.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Interfaz ArbolBusqueda
7  //
8  // ******************OPERACIONES PÚBLICAS*********************
9  // void insertar( x )       --> Inserta x
10 // void eliminar( x )       --> Elimina x
11 // void eliminarMin( )      --> Elimina el menor elemento
12 // Comparable buscar( x )   --> Devuelve el elemento que ajusta con x
13 // Comparable buscarMin( )  --> Devuelve el menor elemento
14 // Comparable buscarMax( )  --> Devuelve el mayor elemento
15 // boolean esVacio( )       --> Devuelve true si vacío; si no false
16 // void vaciar( )           --> Elimina todos los elementos
17 // void imprimirArbol( )    --> Imprime el árbol ordenadamente
18 // ********************ERRORES*********************************
19 // Muchas rutinas lanzan ElementoNoEncontrado en varias
20 // condiciones degeneradas
21 // insertar lanza ElementoDuplicado si el elemento ya está en el árbol
22
23 public interface ArbolBusqueda
24 {
25     void       insertar( Comparable x ) throws ElementoDuplicado;
26     void       eliminar( Comparable x ) throws ElementoNoEncontrado;
27     void       eliminarMin( )           throws ElementoNoEncontrado;
28     Comparable buscar( Comparable x )   throws ElementoNoEncontrado;
29     Comparable buscarMin( )             throws ElementoNoEncontrado;
30     Comparable buscarMax( )             throws ElementoNoEncontrado;
31     boolean    esVacio( );
32     void       vaciar( );
33     void       imprimirArbol( );
34 }
```

Figura 6.18 Interfaz de árbol binario de búsqueda.

Este ajuste se basa en el método `compara` que debe ser implementado por los objetos de la clase `Comparable`. Si un elemento no se encuentra, `buscar` lanza una excepción. Ésta es una decisión de diseño; otra posibilidad habría sido devolver simplemente `null` cuando no se encuentra el elemento buscado. La diferencia básica de estos dos enfoques es que en el nuestro el programador debe explícitamente proporcionar código para el caso en el que la búsqueda falle. En el enfoque alternativo, si se devuelve `null` y no se hace ninguna comprobación, el resultado podría ser una `NullPointerException` en tiempo de ejecución. En términos de eficiencia, la versión con excepción puede ser menos eficiente, pero seguramente las consecuencias no se notarían, excepto quizá en alguna sección de código que se ejecutara muy a menudo. Sin embargo, al forzar al programador a tratar este caso, hemos hecho que el código sea menos susceptible a un error en tiempo de ejecución.

De la misma manera, la inserción de un elemento que ya se encuentra en el árbol se señala con la excepción `ElementoDuplicado`. De nuevo otras alternativas son posibles; una es permitir que la nueva inserción sobrescriba el valor almacenado. Esto es lo que se hace en la Sección 6.7, cuando se discuten las tablas hash.

Consideramos ahora un árbol binario de búsqueda que almacena cadenas de caracteres. Ya que sólo se pueden utilizar objetos `Comparable`, no podemos utilizar directamente el tipo `String`. De modo que definimos una clase `MiCadena`, mostrada en la Figura 6.19 (esta clase también implementa el interfaz `Hashable`, que se utiliza en la Sección 6.7). La Figura 6.20 muestra cómo funciona un árbol binario de búsqueda con objetos `MiCadena`.

Nuestra interfaz `ArbolBusqueda` también proporciona dos métodos adicionales: uno para buscar el elemento mínimo y otro para buscar el elemento máximo. Resulta que con un poco de trabajo adicional, también podemos soportar acceso al *K*-ésimo menor elemento, para un *K* cualquiera dado como parámetro. Esto se conoce como *búsqueda por posición en el orden*.

```
1  import Soporte.*;
2  import EstructurasDatos.*;
3
4  public final class MiCadena implements Comparable, Hashable
5  {
6      public MiCadena( String x )
7        { valor = x; }
8
9      public String toString( )
10       { return valor; }
11
12     public int compara( Comparable lder )
13       { return valor.compareTo( ((MiCadena) lder).valor ); }
14
15     public boolean menorQue( Comparable lder )
16       { return compara( rhs ) < 0; }
17
18     public boolean equals( Object lder )
19       { return valor.equals( ((MiCadena) lder).valor ); }
20
21     public int hash( int tamanyoTabla )
22       { return TablaExploracionCuadratica.hash( valor,
23                                                 tamanyoTabla ); }
24
25     private String valor;
26 }
```

Figura 6.19 Clase `MiCadena` para probar los programas de las Figuras 6.20 y 6.23.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar los árboles de búsqueda
5
6  public final class TestArbolBusqueda
7  {
8      public static void main( String [ ] args )
9  {
10         ArbolBusqueda a = new ArbolBinarioBusqueda ( );
11         MiCadena resultado = null;
12
13         try { a.insertar( new MiCadena( "Susana" ) ); }
14         catch( ElementoDuplicado e ) { } // No puede ocurrir
15
16         try
17         {
18            resultado =
19                 (MiCadena) a.buscar( new MiCadena( "Susana" ) );
20            System.out.print( "Encontrada " + resultado + ";" );
21         }
22         catch( ElementoNoEncontrado e )
23           { System.out.print( "Susana no encontrada;" ); }
24
25         try
26         {
27            resultado =
28                 (MiCadena) a.buscar( new MiCadena( "Juan" ) );
29            System.out.print( "Encontrado " + resultado + ";" );
30         }
31         catch( ElementoNoEncontrado e )
32           { System.out.print( " Juan no encontrado;" ); }
33
34          System.out.println( );
35     }
36 }
```

Figura 6.20 Programa ejemplo con árboles de búsqueda: la salida es `Encontrada Susana; Juan no encontrado;`.

Resumimos ahora los tiempos de ejecución de estas operaciones. Tenemos esperanzas de que el coste de los métodos `buscar`, `insertar` y `eliminar` sea, en el caso peor, $O(\log N)$, como sucede en el caso de la búsqueda binaria estática. Desgraciadamente, con la implementación más simple de los árboles binarios de búsqueda, esto no ocurre. El caso medio es en efecto logarítmico, pero el caso peor es $O(N)$, y puede presentarse con cierta frecuencia. Sin embargo, aplicando algunos trucos algorítmicos, podemos obtener una estructura algo más compleja, que sí tiene un coste $O(\log N)$ por operación.

¿Qué ocurre con los procedimientos `buscarMin`, `buscarMax` y el más general `buscarKesimo`? En el caso de la búsqueda binaria, son claramente operaciones constantes, porque indexamos un vector. En el Capítulo 18 veremos que estas operaciones tardan lo mismo en un árbol binario de búsqueda que la operación `buscar`, esto es, $O(\log N)$ en promedio, aunque $O(N)$ en el caso peor. Con suficiente cuidado, la cota en el caso peor puede ser reducida a $O(\log N)$. Como sugiere el nombre de la estructura, el árbol binario de búsqueda se implementa como un árbol binario, y por tanto requiere una sobrecarga de dos referencias por elemento. Una variante más elaborada que soporta accesos eficientes en el caso peor necesita más espacio adicional por elemento. El Capítulo 18 da más detalles sobre la implementación de los árboles de búsqueda binarios.

6.7 Tablas hash

Muchas aplicaciones requieren búsqueda dinámica basada sólo en un nombre. La aplicación clásica es la *tabla de símbolos* de un compilador. Mientras compila un programa, el compilador debe guardar los nombres (incluyendo tipos, ámbito y asignación de memoria) de todos los identificadores declarados. Cuando encuentra un identificador fuera de una instrucción de declaración, el compilador comprueba si ya ha sido declarado. Si es así, el compilador extrae la información correspondiente de la tabla de símbolos.

Ya que los árboles binarios de búsqueda soportan un acceso en tiempo logarítmico a elementos con nombre arbitrarios, ¿por qué necesitamos una nueva estructura de datos? La respuesta es que un árbol binario podría producir un coste lineal en tiempo por acceso, y asegurar un coste logarítmico requeriría algoritmos bastante sofisticados.

Las *tablas hash* son una estructura de datos que evitan este caso peor, consiguiéndo además operaciones en tiempo constante, casi con seguridad. Un rendimiento degenerado es posible, pero extremadamente improbable. Por tanto, en la práctica el tiempo de acceso a cualquier elemento no depende del número de elementos en la tabla. Las tablas hash también evitan repetidas llamadas a las rutinas de gestión de la memoria. Esto las hace también más rápidas en la práctica. Un beneficio adicional de las tablas hash es que, al contrario de los árboles binarios de búsqueda, no requieren que los objetos implementados implementen la interfaz `Comparable`.

Las operaciones permitidas se muestran en la Figura 6.21, y la interfaz se muestra en la Figura 6.22. Ahora, la inserción de un elemento repetido no produce el lanzamiento de una excepción. En vez de ello, el nuevo elemento reemplaza al viejo. Las tablas hash sólo funcionan con elementos que implementan la interfaz `Hashable`. La interfaz `Hashable` requiere una *función hash*, que convierte el objeto `Hashable` en un entero. Tiene la siguiente declaración:

```
// Devuelve un entero entre 0 y tamanyoTabla - 1
int hash( int tamanyoTabla );
```

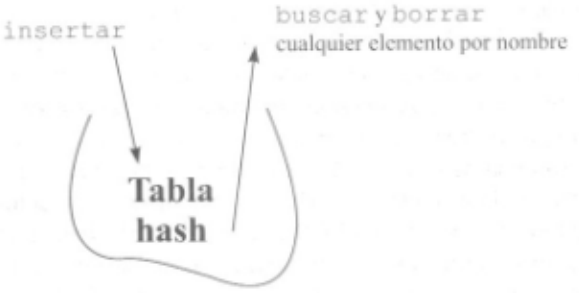


Figura 6.21 Modelo de las tablas hash: cualquier elemento con nombre puede ser accedido o borrado en un tiempo esencialmente constante.

```
1  package EstructurasDatos;
2
3  import Soporte.*;
4  import Excepciones.*;
5
6  // Interfaz TablaHash
7  //
8  // *********************OPERACIONES PÚBLICAS*******************
9  // void insertar( x )    --> Inserta x
10 // void eliminar( x )    --> Elimina x
11 // Hashable buscar( x ) --> Devuelve el elemento que ajusta con x
12 // void vaciar( )        --> Elimina todos los elementos
13 // ************************ERRORES*****************************
14 // buscar y eliminar lanzan ElementoNoEncontrado
15 // insertar sobrescribe el valor anterior en caso de repetición
16
17 public interface TablaHash
18 {
19     void     insertar( Hashable x );
20     void     eliminar( Hashable x ) throws ElementoNoEncontrado;
21     Hashable buscar( Hashable x )   throws ElementoNoEncontrado;
22     void     vaciar( );
23 }
```

Figura 6.22 Interfaz para la clase de tablas hash.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3
4  // Programa simple para probar las tablas hash
5
6  public final class TestTablaHash
7  {
8      public static void main( String [ ] args )
9      {
10         TablaHash h = new TablaExploracionCuadratica( );
11         MiCadena resultado = null;
12
13         h.insertar( new MiCadena( new String( "Susana" ) ) );
14
15         try
16         {
17             resultado = (MiCadena)
18                         h.buscar( new MiCadena( "Susana" ) );
19             System.out.println( "Encontrada " + resultado );
20         }
21         catch( ElementoNoEncontrado e )
22           { System.out.println( "Susana no se ha encontrado" ); }
23     }
24 }
```

Figura 6.23 Programa de ejemplo con tablas hash. La salida es `Encontrada Susana`.

(Los detalles de cómo implementar la función hash se verán en el Capítulo 19, donde se proporciona una función que funciona para valores de tipo `String`.) El método `equals` debe ser también sobrescrito. La Figura 6.19 muestra cómo los objetos de `MiCadena` implementan la interfaz `Hashable`, proporcionando las funciones `hash` y `equals`. En la Figura 6.23, se muestra un ejemplo del uso de tablas hash sobre objetos `MiCadena`.

Un uso común de las tablas hash lo representan los diccionarios. Un *diccionario* almacena objetos formados por una clave, por la cual se busca en el diccionario, y su definición, que es lo que se devuelve. Podemos utilizar una tabla hash para implementar un diccionario instanciándola de la siguiente manera:

1. El objeto almacenado es de una clase que almacena la clave y su definición.
2. Las funciones de igualdad y desigualdad y la función hash se basan solamente en la parte clave del objeto almacenado.
3. Una búsqueda se realiza construyendo un objeto `e` con la clave y realizando `buscar` en la tabla hash.
4. La definición se obtiene utilizando una referencia `f` a la que se le asigna el valor devuelto por `buscar`. El campo definición de `f` es lo que buscamos.

6.8 Colas de prioridad

Aunque los trabajos enviados a una impresora generalmente se colocan en una cola, ésta puede no siempre ser la mejor opción. Por ejemplo, un trabajo puede ser particularmente importante, por lo que querríamos ejecutarlo tan pronto como la impresora esté disponible. A la inversa, cuando la impresora termina un trabajo y están esperando varios trabajos de una página y un trabajo de 100 páginas, podría ser razonable imprimir el trabajo largo el último, aun cuando no sea el último trabajo enviado. (Desgraciadamente, la mayoría de los sistemas no hacen esto, lo que puede ser particularmente molesto en ocasiones.)

De la misma manera, en un entorno multiusuario, el planificador de tareas del sistema operativo debe decidir qué proceso ejecutará entre varios. Generalmente, a cada proceso sólo se le permite estar ejecutándose durante un periodo de tiempo fijo. Un posible algoritmo de gestión utiliza una cola. Los trabajos se colocan inicialmente al final de la cola. El planificador hará repetidamente lo siguiente: tomará el primer trabajo en la cola, lo ejecutará hasta que termine o agote su límite de tiempo, y lo volverá a colocar al final de la cola si no ha terminado. Generalmente, esta estrategia no es muy apropiada porque los trabajos cortos deben esperar mucho, por lo que parece que tardan mucho en ejecutarse. Claramente, los usuarios que están utilizando un editor no deberían ver con gran retraso los caracteres escritos. En consecuencia, los trabajos cortos (es decir, aquéllos que consumen pocos recursos) deberían tener preferencia sobre trabajos que ya han consumido gran cantidad de recursos. Pero también algunos trabajos de consumo intensivo, como por ejemplo los ejecutados por el administrador del sistema, podrían ser muy importantes, por lo que deberían también tener preferencia.

Si asignamos a cada trabajo un número para medir su prioridad, los valores menores (páginas impresas, recursos utilizados) asignarán una mayor importancia. Por tanto, queremos ser capaces de acceder al elemento menor de una colección

de elementos, y eliminarlo de la colección. Se obtienen así las operaciones `buscarMin` y `eliminarMin`. La estructura de datos que soporta estas operaciones es la denominada *cola de prioridad*. La Figura 6.24 ilustra las operaciones básicas de las colas de prioridad.

La Figura 6.25 muestra la interfaz de las colas de prioridad, y un programa ejemplo que ilustra su uso se muestra en la Figura 6.26. El programa ejemplo utiliza una cola de prioridad bastante sofisticada, denominada montículo de emparejamientos, que se discute en el Capítulo 22. Como es usual, uno no necesita saber nada acerca del montículo de emparejamientos, excepto que implementa la interfaz de las colas de prioridad.

Una vez más, debemos preguntar: ¿por qué no usar un árbol binario de búsqueda? Y una vez más, la respuesta es que los árboles binarios de búsqueda son innecesariamente potentes, tienen un rendimiento pobre en el caso peor, y requieren la sobrecarga de dos referencias a nodos por elemento. Utilizando un árbol binario de búsqueda sofisticado garantizaríamos un rendimiento logarítmico en el caso peor. Pero para lograrlo necesitaríamos un código elaborado que nos condu-

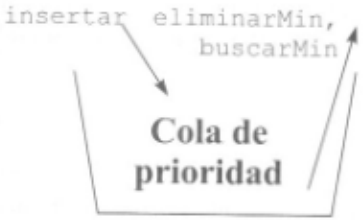


Figura 6.24 Modelo de cola de prioridad: sólo el elemento mínimo es accesible.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Interfaz ColaPrioridad
7  //
8  // *********************OPERACIONES PÚBLICAS*****************
9  // void insertar( x )        --> Inserta x
10 // Comparable buscarMin( )   --> Devuelve el elemento menor
11 // Comparable eliminarMin( ) --> Devuelve y elimina el elemento menor
12 // void vaciar( )            --> Elimina todos los elementos
13 // boolean esVacia( )        --> Devuelve true si vacía; si no, false
14 // *************************ERRORES****************************
15 // buscarMin y eliminarMin lanzan DesbordamientoInferior cuando vacía.
16
17 public interface ColaPrioridad
18 {
19     void       insertar( Comparable x );
20     Comparable buscarMin( )   throws DesbordamientoInferior;
21     Comparable eliminarMin( ) throws DesbordamientoInferior;
22     void       vaciar( );
23     boolean    esVacia( );
24 }
```

Figura 6.25 Interfaz para las colas de prioridad.

```
1  import EstructurasDatos.*;
2  import Excepciones.*;
3  import Soporte.*;
4
5  // Programa simple para probar las colas de prioridad
6  public final class TestColaPrioridad
7  {
8      public static void main( String [ ] args )
9      {
10         ColaPrioridad cp = new MonticuloEmparejamientos( );
11
12         cp.insertar( new MiEntero ( 4 ) );
13         cp.insertar( new MiEntero ( 2 ) );
14         cp.insertar( new MiEntero ( 1 ) );
15         cp.insertar( new MiEntero ( 3 ) );
16         cp.insertar( new MiEntero ( 0 ) );
17
18         System.out.print( "Contenido:" );
19         try
20         {
21             for( ; ; )
22                 System.out.print( " " + cp.eliminarMin( ) );
23         }
24         catch( DesbordamientoInferior e ) { }
25
26         System.out.println( );
27     }
28 }
```

Figura 6.26 Programa ejemplo con colas de prioridad. La salida es: `Contenido: 0 1 2 3 4.`

ciría a un rendimiento injustificadamente bajo en la práctica[1]. Ya que las colas de prioridad soportan únicamente las operaciones `buscarMin` y `eliminarMin`, podríamos esperar un rendimiento que fuera un compromiso entre el tiempo constante de las colas y el tiempo logarítmico de los árboles binarios de búsqueda.

Éste es, en efecto, el caso. Las colas de prioridad básicas soportan todas las operaciones en un tiempo logarítmico en el caso peor, no requieren el coste adicional de dos referencias a otros nodos, soportan la inserción en tiempo constante en promedio y son muy fáciles de implementar. La estructura resultante se conoce como *montículo binario* y es una de las estructuras más elegantes conocidas. El Capítulo 20 da detalles sobre la implementación de los montículos binarios.

El montículo de emparejamientos, utilizado en la Figura 6.26, es una alternativa complicada, útil cuando se necesita cierta funcionalidad adicional, incluyendo la habilidad de combinar eficientemente dos colas de prioridad separadas.

Una aplicación importante de las colas de prioridad se encuentra en el área de la *simulación dirigida por eventos*. Considere, por ejemplo, un sistema como un banco en el que los clientes llegan y esperan en fila hasta que una de las K ventanillas está disponible. La llegada de los clientes está gobernada por una función de distribución de probabilidad, así como el tiempo de servicio (el tiempo necesario para que el empleado proporcione el servicio completo a un cliente). Estamos intere-

[1] Sin embargo, los árboles de ensanchamiento, discutidos en el Capítulo 21, son una notable excepción en algun

sados en estadísticas como ¿cuánto tiempo debe esperar un cliente en promedio? o ¿cómo de larga podría ser la fila?

Con ciertas distribuciones de probabilidad y para ciertos valores de K, podemos calcular estos valores de forma exacta. Sin embargo, al hacerse K mayor, el análisis se hace considerablemente más difícil, por lo que hay que usar un computador para simular el funcionamiento del banco. De esta forma, los gerentes del banco pueden determinar cuántas ventanillas son necesarias para asegurar un servicio razonable. Una *simulación dirigida por eventos* consiste en el procesamiento de eventos. Aquí los dos eventos son: (1) la llegada de un cliente y (2) la salida de un cliente, dejando libre una ventanilla. En cualquier momento, tenemos una colección de eventos esperando a producirse. Para ejecutar la simulación, debemos determinar el *siguiente* evento que se producirá. Éste es el evento cuyo tiempo de acontecimiento es mínimo; por tanto, para procesar la lista de eventos eficientemente se utiliza una cola de prioridad que extrae el elemento con tiempo mínimo. En la Sección 13.2 se presenta una discusión completa y una implementación de la simulación dirigida por eventos.

Resumen

En este capítulo se han examinado las estructuras de datos básicas utilizadas en el resto del libro. Para cada una de ellas se ha proporcionado una interfaz e indicado cual debería ser el tiempo de ejecución de las operaciones. En la Parte IV daremos una implementación de dichas estructuras de datos que se ajuste a las cotas de tiempo afirmadas aquí. La Figura 6.27 resume los resultados que se obtendrán.

En el próximo capítulo estudiamos una herramienta muy importante para resolver problemas: la *recursión*. La recursión permite resolver de forma eficiente muchos problemas, utilizando algoritmos cortos y sencillos, y es fundamental para la implementación de los árboles binarios de búsqueda.

Estructura de datos	Acceso	Comentarios
Pilas	Sólo al más reciente, `cima`, $O(1)$	Muy, muy rápidas
Colas	Sólo al menos reciente, `quitarPrimero`, $O(1)$	Muy, muy rápidas
Listas enlazadas	Cualquier elemento	$O(N)$
Árboles de búsqueda	Cualquier elemento, por nombre o rango, $O(\log N)$	Caso medio; también puede lograrse en el caso peor
Tablas hash	Cualquier elemento, con nombre, $O(1)$	Casi seguro
Colas de prioridad	`buscarMin`, $O(1)$ `eliminarMin`, $O(\log N)$	`insertar` es $O(1)$ en promedio, $O(\log N)$ en el caso peor

Figura 6.27 Resumen de algunas estructuras de datos.

Elementos del juego

análisis sintáctico de precedencia de operadores Algoritmo que utiliza una pila para evaluar expresiones.

árbol Estructura de datos muy utilizada, formada por un conjunto de nodos y un conjunto de aristas que conectan pares de nodos. En todo este libro, asumimos que los árboles tienen raíz.

árbol binario Un árbol con, a lo sumo, dos hijos por nodo.

árbol binario de búsqueda Un árbol que permite la inserción, eliminación y búsqueda. También se puede utilizar para acceder al K-ésimo menor elemento. El coste en tiempo es logarítmico en promedio para una implementación simple, y logarítmico en el caso peor, con una implementación más cuidada.

árbol con raíz Árbol con un nodo designado como raíz.

árbol sintáctico de una expresión Un árbol utilizado en análisis sintáctico, en el cual el valor de un nodo es el resultado de aplicar el operador en el nodo, utilizando los valores de sus hijos como operandos.

clase iteradora Clase que permite acceder a una lista. La clase de la lista tiene métodos que reflejan el estado de la lista; el resto de operaciones están en la clase iterador.

cola Estructura de datos que limita el acceso al elemento insertado menos recientemente.

cola de prioridad Estructura de datos que soporta el acceso únicamente al menor elemento.

diccionario Almacena claves, que son las que se buscan en el diccionario, y sus correspondientes definiciones.

estructura de datos Una representación de los datos junto con unas operaciones permitidas sobre dichos datos. Las estructuras de datos nos permiten lograr la reutilización de componentes.

función hash Función que convierte un objeto `Hashable` en un entero.

hoja En un árbol, un nodo sin hijos.

longitud de un camino En un árbol, el número de aristas que hay que atravesar, desde la raíz, para alcanzar un nodo.

montículo binario Implementa sobre un vector las colas de prioridad con un tiempo logarítmico por operación.

pila Estructura de datos que restringe el acceso al elemento más recientemente insertado.

tabla de símbolos Estructura de datos utilizada por un compilador para almacenar los identificadores. Implementada generalmente con una tabla hash.

tabla hash Estructura de datos que soporta inserciones, eliminaciones y búsquedas, en tiempo constante en promedio.

Errores comunes

1. No documentar correctamente las características de la interfaz de una clase es un error grave.
2. Es un error acceder o eliminar de una pila, cola o cola de prioridad, vacía. El que implementa la clase debe asegurarse de detectar el error y lanzar una excepción. Las operaciones de acceso o eliminación están permitidas

en árboles o tablas hash vacías porque el acceso en un árbol o tabla hash vacía es un caso particular de búsqueda sin éxito.
3. Pueden ocurrir varios errores mientras se accede a una lista o a un árbol. Éstos deben ser señalizados por la clase.
4. La implementación de un objeto `Hashable` debe suministrar una buena función hash; en caso contrario, se deteriorará el rendimiento.
5. La función `equals` (que toma un `Object` como parámetro) debería escribirse para los objetos que se insertan en tablas hash o árboles de búsqueda.
6. Una cola de prioridad no es una cola. Sólo suena igual.

En Internet

Algunos de los programas de prueba de este capítulo, incluyendo la clase `MiCadena`, están en el directorio **Chapter06**. Por otra parte, las interfaces para las clases que serán implementadas en las Partes IV y V se encuentran en el directorio **DataStructures**, siendo parte del paquete `DataStructures`, traducido como `EstructurasDatos`. Se tratan, en concreto, de los siguientes:

Stack.java	La interfaz `Stack`, traducida por `Pila` en la Figura 6.4.
Queue.java	La interfaz `Queue`, traducida por `Cola` en la Figura 6.7.
List.java	La interfaz `List`, traducida por `Lista` en la Figura 6.11.
ListItr.java	La interfaz `ListItr`, traducida por `ListaIter` en la Figura 6.12.
SearchTree.java	La interfaz `SearchTree`, traducida por `ArbolBusqueda` en la Figura 6.18.
HashTable.java	La interfaz `HashTable`, traducida por `TablaHash` en la Figura 6.22.
PriorityQueue.java	La interfaz `PriorityQueue`, traducida por `ColaPrioridad` en la Figura 6.25.

Ejercicios

Cuestiones breves

6.1. Muestre los resultados de la siguiente secuencia de instrucciones: `insertar(4)`, `insertar(8)`, `insertar(1)`, `insertar(6)`, `eliminar()` y `eliminar()`, cuando `insertar` y `eliminar` son las operaciones básicas de

a) Las pilas.
b) Las colas.
c) Las colas de prioridad.

Problemas teóricos

6.2. Suponga que quiere soportar, exclusivamente, las tres operaciones: `insertar`, `buscarMax` y `eliminarMax`. ¿Cómo de rápido cree que se pueden implementar estas operaciones?

6.3. ¿Pueden todas las operaciones siguientes ser implementadas en tiempo logarítmico: `insertar`, `eliminarMin`, `eliminarMax`, `buscarMin` y `buscarMax`?

6.4. ¿Qué estructuras de datos de la Figura 6.27 conducirían a algoritmos de ordenación que se ejecutaran en un tiempo menor que cuadrático?

6.5. Muestre que es posible soportar simultáneamente las operaciones siguientes en tiempo constante: `apilar`, `cima`, y `buscarMin`. Observe que `eliminarMin` no forma parte del repertorio. *Pista*: mantenga dos pilas, una para almacenar los elementos y otra para almacenar los mínimos según van apareciendo.

6.6. Una cola doble permite inserciones y eliminaciones en ambos extremos de la línea. ¿Cuál cree que es el tiempo de ejecución por operación?

Problemas prácticos

6.7. Escriba una rutina que imprima en orden inverso los elementos de una lista enlazada. Para hacerlo, recorra la lista con una `ListaIter`, apilando cada elemento en una pila. Cuando alcance el fin de la lista enlazada, extraiga repetidamente la `cima` hasta que la pila se quede vacía.

6.8. Una eliminación en una lista enlazada o en un árbol de búsqueda binario, nos provoca el siguiente problema: cuando el elemento actual es borrado, ¿cuál se convierte en el nuevo elemento actual? Discuta varias alternativas.

6.9. Muestre como implementar eficientemente una pila utilizando una lista enlazada como atributo.

6.10. Muestre como implementar eficientemente una cola utilizando una lista enlazada como atributo y manteniendo un objeto `ListaIter` que apunta siempre al último elemento de la lista enlazada.

6.11. Partiendo de una clase de tablas hash, implemente una clase `Diccionario` que permita las operaciones de `insertar` y `buscar`. Para facilitar su tarea incluimos a continuación algunas de las líneas de esta clase.

```
public class Diccionario
{
      // Algunos de los métodos
    void insertar( Hashable clave, Object definicion );
    Object buscar( Hashable clave ) throws ElementoNoEncontrado;
}
```

Prácticas de programación

6.12. Una pila puede ser implementada utilizando un vector y manteniendo el tamaño actual. Los elementos de la pila se almacenan en posiciones consecutivas del vector, con el elemento de la cima siempre en la posición 0. Observe que éste no es el método más eficiente. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `apilar`, `desapilar` y `cima`.
b) ¿Cuál es el tiempo de ejecución en notación O de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.13. Una cola puede ser implementada utilizando un vector y manteniendo su tamaño actual. Los elementos de la cola se almacenan en posiciones consecutivas del vector, con el primer elemento siempre en la posición 0. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `primero`, `insertar` y `quitarPrimero`.
b) ¿Cuál es el tiempo de ejecución en notación O de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

Éste no es el método más eficiente de implementar una cola utilizando un vector. Trate de localizar la razón principal por la que alguna de las operaciones es ineficiente, y a partir de la idea de corregir esta situación proponga una implementación alternativa que resulte plenamente satisfactoria.

6.14. Las operaciones permitidas por un árbol de búsqueda pueden implementarse también con un vector, manteniendo su tamaño actual. Los elementos del vector se almacenan ordenados en posiciones consecutivas del vector. En consecuencia, `buscar` puede implementarse con una búsqueda binaria. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `insertar` y `eliminar`.
b) ¿Cuál es el tiempo de ejecución, en notación O, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.15. Una tabla hash puede implementarse utilizando un vector, manteniendo su tamaño actual. Los elementos del vector se almacenan en posiciones consecutivas del vector, pero no de forma ordenada. En lugar de ello, se van insertando en la siguiente posición libre del vector. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `insertar`, `eliminar` y `buscar`.
b) ¿Cuál es el tiempo de ejecución, en notación O, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.16. Una cola de prioridad puede implementarse utilizando un vector ordenado, como se indicó en el Ejercicio 6.14. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `eliminarMin` e `insertar`.
b) ¿Cuál es el tiempo de ejecución, en notación O, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.17. Una cola de prioridad puede implementarse almacenando los elementos en un vector sin ordenar e insertando elementos en la siguiente posición libre, como se indicó en el Ejercicio 6.15. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `eliminarMin` e `insertar`.
b) ¿Cuál es el tiempo de ejecución, en notación O, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.18. Añadiendo un atributo más a la clase de las colas de prioridad del Ejercicio 6.17, es posible implementar `insertar` y `buscarMin` en tiempo constante. El atributo extra almacenará la posición del vector donde se encuentra el mínimo. Sin embargo, `eliminarMin` seguirá siendo costosa. Se pide hacer lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `insertar` y `eliminarMin`.
b) ¿Cuál es el tiempo de ejecución, en notación O, de `eliminarMin`?
c) Escriba una implementación en Java de dichos algoritmos.

6.19. Una cola de prioridad doble permite accesos tanto al elemento mínimo como al máximo. En consecuencia, se permiten todas las operaciones siguientes: `buscarMin`, `eliminarMin`, `buscarMax` y `eliminarMax`. Discutir distintas implementaciones posibles basadas en un vector, haciendo para cada una de ellas lo siguiente:

a) Describa los algoritmos para llevar a cabo `buscarMin`, `eliminarMin`, `buscarMax` y `eliminarMax`.
b) ¿Cuál es el tiempo de ejecución, en notación O, de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

6.20. Un montículo de medianas soporta las siguientes operaciones: `insertar`, `buscarKesimo` y `eliminarKesimo`. Las dos últimas buscan y eliminan, respectivamente, el K-ésimo menor elemento. La implementación más simple mantiene los elementos del montículo ordenados en un vector. Se pide hacer lo siguiente:

a) Describa los algoritmos que pueden ser utilizados para soportar las operaciones de los montículos de medianas.
b) ¿Cuál es el tiempo de ejecución, en notación O, de cada uno de estos algoritmos?
c) Escriba una implementación en Java de dichos algoritmos.

Bibliografía

La bibliografía correspondiente a las estructuras de datos presentadas en este capítulo se proporciona en la Parte IV.

CAPÍTULO

7 Recursión

Un método parcialmente definido en términos de sí mismo recibe el nombre de *recursivo*. Como muchos otros lenguajes, Java soporta métodos recursivos. La recursión, consistente en el uso de métodos recursivos, es una herramienta de programación potente que en muchos casos puede producir algoritmos cortos y eficientes. Este capítulo estudia cómo funciona la recursión, proporcionando una visión de sus variaciones, sus limitaciones y algunos de sus numerosos usos. La discusión sobre la recursión comienza con un examen del principio matemático en el que se basa: la *inducción matemática*. Después se proporcionan ejemplos de métodos recursivos simples y se demuestra que éstos generan respuestas correctas.

En este capítulo veremos:

- Las cuatro reglas básicas de la recursión, y varios ejemplos sencillos.
- Aplicaciones numéricas de la recursión que nos conducen a la implementación de un algoritmo de encriptación.
- Una técnica general llamada *divide y vencerás*.
- Una técnica general llamada *programación dinámica* que es similar a la recursión pero que usa tablas en lugar de llamadas recursivas.
- Una técnica general llamada *vuelta atrás* que consiste en una búsqueda sistemática y exhaustiva.

7.1 ¿Qué es la recursión?

Un *método recursivo* es un método que, directa o indirectamente, se hace una llamada a sí mismo. Esto puede parecer un círculo vicioso: ¿cómo un método F puede resolver un problema llamándose a sí mismo? La clave está en que el método F se llama a sí mismo pero con instancias diferentes, más simples, en algún sentido adecuado. Aquí tiene algunos ejemplos:

- Los ficheros en un computador se almacenan generalmente en directorios. Los usuarios pueden crear directorios, que a su vez almacenen más ficheros y directorios. Suponga que deseamos examinar cada fichero de un directorio D, incluyendo todos los ficheros de sus subdirectorios (y sus subdirectorios, y así sucesivamente). Esto se puede hacer examinando recursivamente los ficheros de cada subdirectorio junto con todos los ficheros en el directorio D (este problema se estudia en el Capítulo 17).
- Supongamos que tenemos un diccionario grande. Las palabras en los diccionarios se definen en términos de otras palabras. Cuando buscamos el signifi-

cado de una palabra, no siempre entendemos la definición, por lo que debemos buscar algunas palabras de la propia definición. De la misma forma, puede que no entendamos algunas de ellas, en cuyo caso tendremos que continuar la búsqueda durante un rato. Como el diccionario es finito, eventualmente llegaremos a un punto en el que entendemos todas las palabras de una definición, con lo que entendemos dicha definición y podemos volver hacia atrás a través de las otras definiciones, o bien nos daremos cuenta de que las definiciones son circulares por lo que nos quedamos atascados, o bien alguna palabra que no entendemos no está definida en el diccionario. Nuestra estrategia recursiva para comprender las palabras es la siguiente: si conocemos el significado de una palabra, ya hemos terminado; en caso contrario, buscamos la palabra en el diccionario. Si entendemos todas las palabras de la definición, ya hemos terminado; en caso contrario, averiguamos el significado de la definición buscando recursivamente las palabras que no conocemos. Este procedimiento terminará si el diccionario está bien definido, pero puede ciclar indefinidamente si una palabra está definida circularmente.

- Habitualmente (partes de) los lenguajes de programación se definen de forma recursiva. Por ejemplo, una expresión (aritmética) es una variable primitiva, o una expresión con paréntesis, o la suma de dos expresiones, etc.

La recursión es una herramienta muy potente de resolución de problemas. Muchos algoritmos se pueden expresar más fácilmente usando una formulación recursiva. Y lo que es más, hay muchos problemas cuyas soluciones más eficientes usan esta formulación recursiva natural. Pero debemos ser cuidadosos para no crear una lógica circular que produzca bucles infinitos.

En este capítulo se estudian las condiciones generales que deben satisfacer los algoritmos recursivos, ofreciéndose varios ejemplos prácticos. También muestra que en ocasiones los algoritmos que se expresan de una forma natural en modo recursivo deben rescribirse sin recursión para conseguir mayor eficiencia.

7.2 Fundamentos: demostraciones por inducción matemática

En esta sección se estudia la técnica de demostración mediante *inducción matemática*. (A lo largo de este capítulo, se omitirá la palabra *matemática* al referirnos a esta técnica.) Las demostraciones por inducción se usan habitualmente para demostrar teoremas que cumplen los enteros positivos. Para ilustrar la técnica, comenzaremos por demostrar un teorema sencillo. Sería fácil demostrar este teorema usando otros métodos, pero sucede bastante a menudo que una prueba por inducción es el mecanismo más sencillo.

Para todo entero $N \geqslant 1$, la suma de los N primeros enteros, dada por $\sum_{i=1}^{N} i = 1 + 2 + \ldots + N$, es igual a $N(N+1)/2$.

Es fácil ver que el teorema es cierto para $N = 1$ porque tanto el lado izquierdo como el lado derecho se evalúan a 1. Un estudio más laborioso revelaría, por ejemplo, que es cierto para $2 \leqslant N \leqslant 10$. Sin embargo, el hecho de que el teorema se cumpla para todos los N para los cuales es fácil comprobarlo *a mano*, no implica que sea cierto para todo N. Consideremos, por ejemplo, los números de la forma $2^{2^k} + 1$. Los primeros cinco números (correspondientes a $0 \leqslant k \leqslant 4$) son 3, 5, 17, 257 y 65.537, los cuales son todos primos. De hecho, hubo un momento en el que se conjeturó que todos los números de esta forma debían ser primos. Pero ello no es cierto: se puede comprobar fácilmente con un computador que $2^{2^5} + 1 = 641 \times 6.700.417$. De hecho, fuera de los citados, no se conoce ningún otro primo de la forma $2^{2^k} + 1$.

Una prueba por inducción trabaja en dos pasos. Primero, al igual que antes, se demuestra que el teorema es cierto para los casos más sencillos. Después, se demuestra que si el teorema es cierto para ciertos casos, entonces se puede extender para incluir el siguiente caso. Por ejemplo, se demuestra que un teorema que es cierto para todo $1 \leqslant N \leqslant k$ también debe ser cierto para $1 \leqslant N \leqslant k + 1$. Una vez se muestra cómo extender el rango de casos ciertos, queda demostrado el resultado para todos los casos, ya que podemos extender el rango de los casos ciertos indefinidamente. Veamos la utilización de esta técnica para demostrar el Teorema 7.1.

Claramente, el teorema es cierto para $N = 1$. Supongamos que el teorema es cierto para todo $1 \leqslant N \leqslant k$. Entonces

$$\sum_{i=1}^{k+1} i = (k+1) + \sum_{i=1}^{k} i. \qquad \textbf{(7.1)}$$

Puesto que por hipótesis el teorema es cierto para k, podemos sustituir la suma del lado derecho de la Ecuación 7.1 por $k(k+1)/2$, obteniendo

$$\sum_{i=1}^{k+1} i = (k+1) + k(k+1)/2. \qquad \textbf{(7.2)}$$

Una manipulación algebraica del lado derecho de la Ecuación 7.2 produce ahora

$$\sum_{i=1}^{k+1} i = (k+1)(k+2)/2.$$

Esto confirma el teorema para el caso $k+1$. Luego, por inducción, el teorema es cierto para todos los enteros $N \geqslant 1$.

¿Por qué constituye esto una prueba? En primer lugar, el teorema es cierto para $N = 1$, al cual se le llama *caso base*. Se puede ver como la base de nuestra confianza en que el teorema es cierto en general. El caso base es el caso (o casos) sencillo que demostramos directamente. Una vez establecido el caso base, podemos suponer que el teorema es cierto para un k arbitrario, suposición que recibe el nombre de hipótesis de inducción. Entonces se demuestra que si el teorema es cierto para k, también es cierto para $k + 1$. En nuestro caso, aplicando el proceso, puesto que sabemos que el teorema es cierto para el caso base $N = 1$, obtenemos que es cierto para $N = 2$. Puesto que es cierto para $N = 2$, también debe serlo para

$N = 3$, y por tanto para $N = 4$, y así sucesivamente. De esta forma vemos que el teorema es cierto para todo entero positivo a partir de $N = 1$.

Apliquemos la inducción a un segundo problema no tan simple como el anterior. Consideramos la secuencia de números 1^2, $2^2 - 1^2$, $3^2 - 2^2 + 1^2$, $4^2 - 3^2 + 2^2 - 1^2$, $5^2 - 4^2 + 3^2 - 2^2 + 1^2$, etc. Cada miembro representa la suma con alternancia de signos de los N primeros cuadrados. La secuencia se evalúa a 1, 3, 6, 10 y 15. Parece que, en general, la suma será exactamente igual a la de los N primeros enteros, que, por el Teorema 7.1, es $N(N+1)/2$. El Teorema 7.2 demuestra este hecho.

La suma $\Sigma_{i=N}^{1}(-1)^{N-i}i^2 = N^2 - (N-1)^2 + (N-2)^2 \ldots$ *es igual a* $N(N+1)/2$.

La demostración es por inducción.

Caso base: claramente, el teorema es cierto para $N = 1$.

Hipótesis de inducción: supondremos que el teorema es cierto para k: $\Sigma_{i=k}^{1}(-1)^{k-i}i^2 = \dfrac{k(k+1)}{2}$.

Paso inductivo: debemos demostrar que el resultado es cierto para $k+1$; *es decir, que* $\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = \dfrac{(k+1)(k+2)}{2}$. *Escribimos*

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - k^2 + (k-1)^2 \ldots . \qquad \textbf{(7.3)}$$

Si rescribimos el lado derecho de la Ecuación 7.3 obtenemos

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - (k^2 - (k-1)^2 \ldots).$$

Esto permite una sustitución, obteniéndose

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - (\Sigma_{i=k}^{1}(-1)^{k-i}i^2) \qquad \textbf{(7.4)}$$

Aplicando la hipótesis de inducción, podemos sustituir el sumatorio del lado derecho de la Ecuación 7.4, obteniendo

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)^2 - k(k+1)/2. \qquad \textbf{(7.5)}$$

Una manipulación algebraica simple del lado derecho de la Ecuación 7.5 produce ahora

$$\Sigma_{i=k+1}^{1}(-1)^{k+1-i}i^2 = (k+1)(k+2)/2,$$

lo cual demuestra el teorema para $N = k+1$. *Luego, por inducción, el teorema es cierto para todo* $N \geqslant 1$.

7.3 Recursión básica

Las demostraciones por inducción nos muestran que si sabemos que una afirmación es cierta para los casos más sencillos, y podemos demostrar que la certeza para un caso implica la del siguiente, entonces sabemos que la afirmación es cierta para todos los casos.

En ocasiones las funciones matemáticas se definen recursivamente. Por ejemplo, si $S(N)$ denota la suma de los N primeros enteros, tenemos $S(1) = 1$, y podemos escribir $S(N) = S(N-1) + N$. Hemos así definido la función S en términos de una instancia más pequeña de sí misma. La definición recursiva de $S(N)$ es equivalente a la forma explícita $S(N) = N(N+1)/2$, excepto por el hecho de que la definición recursiva solamente vale para enteros positivos y su valor no puede calcularse de forma tan directa.

Algunas veces es más fácil definir una función recursivamente que hacerlo en forma explícita. En la Figura 7.1 se muestra una implementación directa de la función recursiva S. El caso $N = 1$ es el caso base, para el que sabemos que $S(1) = 1$. Este caso se trata en las líneas 5 y 6. Es el caso base de la recursión, pues es la instancia que se puede resolver sin llamadas recursivas. En otro caso, aplicamos la definición recursiva $S(N) = S(N-1) + N$, situada en la línea 8. Es difícil imaginar una forma más sencilla de implementar la función recursiva. Entonces, la pregunta natural es, ¿funciona esto realmente?

La respuesta es sí. Con las limitaciones que discutiremos en unos momentos, esta rutina funciona correctamente. Veamos cómo se evalúa la llamada a `s(4)`. Cuando se llama a `s(4)`, el test de la línea 5 falla, por lo que se ejecuta la línea 8, donde evaluamos `s(3)`. Como en el caso de otro método cualquiera, esto requiere una llamada a `s`, con lo que llegamos a la línea 5, donde el test falla, por lo que vamos a la línea 8. En este punto llamamos a `s(2)`, con lo que, de nuevo, llamamos a `s` siendo ahora `n` igual a 2. El test de la línea 5 vuelve a fallar, por lo que llamamos a `s(1)` en la línea 8. Ahora tenemos que `n` es igual a 1, con lo que `s(1)` devuelve 1. En este punto, puede continuar la evaluación de `s(2)`, sumando el valor de `s(1)` a 2, con lo que `s(2)` devuelve 3. Ahora continúa la evaluación de `s(3)` sumando el valor devuelto por `s(2)` a `n`, que vale 3, con lo que `s(3)` devuelve 6. Esto permite completar la llamada a `s(4)`, que devuelve 10 finalmente.

Observe que aunque parece que `s` se está llamando a sí misma, podemos interpretar que en realidad está llamando a un *clon* de sí misma. Dicho clon se puede ver como un método distinto que usa un parámetro diferente. En cada momento solamente hay un clon activo; el resto están suspendidos. Es trabajo del computador y no de uno llevar a cabo las tareas de bajo nivel. Es cierto que si hubiera

```
1 // Evalúa la suma de los n primeros enteros
2
3 public static long s( int n )
4 {
5     if( n == 1 )
6         return 1;
7     else
8         return s( n - 1 ) + n;
9 }
```

Figura 7.1 Evaluación recursiva de la suma de los N primeros enteros.

demasiadas, incluso para un computador, sería el momento de preocuparse. Esas cuestiones se discutirán más adelante en este mismo capítulo.

Si tenemos un caso base y nuestras llamadas recursivas progresan hacia el caso base, entonces terminaremos eventualmente. Con ello concluimos nuestras dos primeras reglas fundamentales de la recursión:

1. *Casos base:* se debe tener siempre al menos un caso base que pueda resolverse sin recursión.
2. *Progreso:* cualquier llamada recursiva debe progresar hacia un caso base.

Nuestra rutina tiene algunos problemas. Un primer problema lo constituiría la llamada a `s(0)`, para la cual el método se comporta de forma pobre[1]. Sin embargo, no debe verse como un error porque el caso recursivo de la definición de $S(N)$ no incluye los valores $N < 1$. Podríamos arreglar este problema extendiendo la definición de $S(N)$ para incluir $N = 0$. Puesto que en este caso no hay números que sumar, un valor natural para $S(0)$ sería 0. Esto además haría que la definición recursiva se pudiera aplicar a $S(1)$, ya que $S(0) + 1$ es 1. Para implementar este cambio, basta sustituir `1` por `0` en las líneas 5 y 6. Los N negativos también producirían comportamientos indeseables, que se podrían solventar de forma análoga a como hicimos para $N = 0$ (se deja como Ejercicio 7.2).

Un segundo problema es que cuando el parámetro `n` es grande, el programa puede abortar o no acabar. En nuestro sistema, por ejemplo, no se puede tratar $N \geqslant 9.410$.

Esto sucede porque, como hemos visto, la implementación de la recursión requiere un tratamiento de las llamadas recursivas pendientes, y para cadenas de recursión lo suficientemente largas, el computador simplemente se queda sin memoria dedicada al efecto. Esto se explicará con más detalle más adelante en el capítulo. Es también cierto que la rutina es más costosa que el bucle equivalente, pues el tratamiento de las llamadas recursivas lleva un cierto tiempo.

Queda claro pues que este ejemplo concreto no representa el mejor uso de la recursión, puesto que es muy sencillo resolver el problema sin ella. La mayor parte de los buenos usos de la recursión no agotarán la memoria del computador y sólo son ligeramente más lentos que las implementaciones no recursivas. Pero la recursión siempre nos conduce a producir un código más compacto.

7.3.1 Impresión de números en cualquier base

Un buen ejemplo de cómo la recursión simplifica la codificación de los procedimientos es la impresión de números. Supongamos que queremos imprimir un número N no negativo en base diez pero que no tenemos disponible un método de impresión de números, aunque sí podemos imprimir un dígito cada vez. Consideremos, por ejemplo, cómo imprimiríamos el número `1369`. Primero necesitaríamos imprimir un `1`, luego un `3`, después un `6` y finalmente un `9`. El problema es que obtener el primer dígito es un tanto costoso: dado un número `n`, necesitamos un bucle para determinar el primer dígito de `n`. Esto contrasta con la obtención del último dígito, que se puede calcular de forma inmediata como `n % 10` (que vale `n` para `n` menor que 10).

La recursión nos proporciona una solución elegante. Para imprimir `1369`, bastaría con imprimir `136` seguido del último dígito, `9`. Como hemos mencionado, es fácil obtener el último dígito usando el operador `%`. Imprimir el número entero exceptuando su último dígito es también sencillo, ya que corresponde a imprimir `n/10`, lo que se puede hacer mediante una llamada recursiva.

El método de la Figura 7.2 implementa esta rutina de impresión. Si `n` es menor que 10, entonces no se ejecuta la línea 6 y solamente se imprime el dígito `n%10`. En caso contrario, se imprime todos menos el último dígito y después se imprime éste.

Observe que tenemos un caso base (`n` es un entero de un solo dígito) y que todas las llamadas recursivas progresan hacia el caso base porque el argumento con el que se hace la llamada recursiva tiene un dígito menos. Por tanto, hemos satisfecho las dos primeras reglas fundamentales de la recursión.

Para que nuestra rutina de impresión sea útil más allá de la mera anécdota, la extendemos para que pueda imprimir el número en cualquier base entre 2 y 16[2]. Esta modificación se muestra en la Figura 7.3. Hemos introducido un vector de caracteres para hacer más sencilla la impresión de las cifras entre la `a` y la `f`, por lo que ahora la salida de cada dígito se realiza indexando el vector `tablaDigitos`. Obsérvese que la rutina `imprimeEntero` no es robusta, pues si `base` es mayor que 16, entonces el índice dentro de `tablaDigitos` estará fuera de rango. Por su parte, si `base` es 0, se producirá un error aritmético en la línea 9 al intentar realizar una división por 0.

```
1  // Imprime n como un número en base diez
2
3  public static void imprimeBaseDiez( int n )
4  {
5      if( n >= 10 )
6          imprimeBaseDiez( n / 10 );
7      System.out.print( '0' + n % 10 );
8  }
```

Figura 7.2 Rutina recursiva para imprimir N en base diez.

```
1  // Imprime n en cualquier base
2  // Asume que 2 <= base <= 16
3
4  final static String tablaDigitos = "0123456789abcdef";
5
6  public static void imprimeEntero( int n, int base )
7  {
8      if( n >= base )
9          imprimeEntero( n / base, base );
10     System.out.print( tablaDigitos.charAt( n % base ) );
11 }
```

Figura 7.3 Rutina recursiva para imprimir N en cualquier base entre 2 y 16.

[1] ada a `s(-1)`, y el programa falla eventualmente por haber demasiadas llamadas recursivas pendientes. Tenemo hay progreso hacia un caso base.

[2] El método `toString` de Java puede tomar cualquier base, pero muchos lenguajes no tienen esta capacid

La situación errónea más interesante aparece cuando `base` es 1. Cuando tal cosa sucede, la llamada recursiva de la línea 9 no progresa hacia el caso base, pues los dos parámetros de la llamada recursiva son idénticos a los de la llamada original. Por ello el sistema hará repetidas llamadas recursivas hasta que se quede sin espacio disponible, momento en el que terminará de forma poco elegante lanzando una excepción.

Podemos conseguir que la rutina sea más robusta añadiendo una comprobación explícita del valor de `base`. El problema con dicha estrategia es que la comprobación se realizaría en cada una de las llamadas recursivas a `imprimeEntero`, y no solamente en la primera llamada. Una vez que `base` es válida en la primera llamada, es una tontería volver a comprobarlo, pues su valor no cambia a lo largo de las llamadas recursivas, por lo que seguirá siendo válida. Una forma de evitar esta ineficiencia consiste en introducir una rutina guía. Dicha *rutina guía* comprueba la validez de `base` y tras ello llama a la rutina recursiva. En la Figura 7.4 se presenta el programa de impresión completo. El uso de rutinas guía sobre programas recursivos es una técnica habitual.

```
1      // Imprime n en cualquier base
2
3  static final String tablaDigitos = "0123456789abcdef";
4  private static final int maxBase = tablaDigitos.length( );
5
6      // Rutina recursiva
7  public static void imprimeEnteroRec( int n, int base )
8  {
9      if( n >= base )
10         imprimeEnteroRec( n / base, base );
11     System.out.print( tablaDigitos.charAt( n % base ) );
12 }
13
14     // Rutina guía
15 public static void imprimeEntero( int n, int base )
16 {
17     if( base <= 1 || base > maxBase )
18         System.err.println( "Base ilegal: " + base );
19     else
20     {
21         if( n < 0 )
22         {
23             n = -n;
24             System.out.print( "-" );
25         }
26         imprimeEnteroRec( n, base );
27     }
28 }
```

Figura 7.4 Programa robusto de impresión de números.

7.3.2 ¿Por qué funciona?

En esta sección se demuestra, de una forma rigurosa, que el algoritmo `imprimeBaseDiez` funciona correctamente. Nuestro objetivo es verificar que el algoritmo es correcto, por lo que asumiremos que no hemos cometido ningún error en su codificación.

El algoritmo `imprimeBaseDiez` de la Figura 7.2 imprime correctamente `n` en base 10.

Sea k el número de dígitos en base 10 de `n`. La demostración se realiza por inducción sobre k.

Caso base: si $k = 1$, no se realiza ninguna llamada recursiva, y la línea 7 produce correctamente como salida el único dígito de `n`.

Hipótesis de inducción: supongamos que `imprimeBaseDiez` funciona correctamente para todos los enteros de $k \geqslant 1$ dígitos.
Paso inductivo: veamos que a partir de estas hipótesis podemos probar la corrección del algoritmo para cualquier entero `n` de $k + 1$ dígitos. Puesto que $k \geqslant 1$, la condición de la instrucción `if` de la línea 5 se satisface para todo entero `n` de $k + 1$ dígitos.
Por hipótesis de inducción, la llamada recursiva de la línea 6 imprime los k primeros dígitos de `n`. Entonces la llamada de la línea 7 imprime el último dígito. Por tanto si cualquier entero de k dígitos se imprime correctamente, también se imprimirá correctamente uno de $k + 1$ dígitos. Concluimos entonces que `imprimeBaseDiez` funciona para todo k, y por tanto para todo `n`.

La demostración del Teorema 7.3 ilustra un principio importante. Cuando se diseña un algoritmo recursivo, siempre podemos asumir que las llamadas recursivas internas funcionan correctamente, y dicha suposición podrá utilizarse como hipótesis de inducción al hacer la demostración de la corrección.

A primera vista, tal asunción puede parecer extraña. Pero recuerde que al razonar estructuradamente sobre un programa complejo no recursivo, siempre hemos podido suponer que las llamadas a los métodos funcionan correctamente, y por tanto la suposición de que las llamadas recursivas funcionan no es muy diferente. Como cualquier método, una rutina recursiva necesita combinar las soluciones producidas por las llamadas a otros métodos para producir una solución, con la particularidad de que tales métodos pueden ser instancias más sencillas del método original.

Esta observación nos conduce a la tercera regla fundamental de la recursión:

3. *«Puede creerlo»:* asuma siempre que toda llamada recursiva interna funciona correctamente.

La Regla 3 nos dice que cuando diseñamos un método recursivo, no tenemos que intentar seguir el, posiblemente largo y tortuoso, camino de llamadas recursivas. Como vimos anteriormente, ésta puede ser una tarea tediosa que haría muy difícil el diseño y verificación. De hecho, en muchas ocasiones, aunque hagamos un buen uso de la recursión, dicha traza puede ser casi imposible de comprender. Intuitivamente, estamos dejando que la máquina se ocupe de las tareas de bajo nivel (las llamadas recursivas internas), ya que si nos ocupáramos nosotros directamente, el código resultante sería muchísimo más largo.

Este principio es tan importante que no nos cansaremos de repetirlo: *asuma siempre que toda llamada recursiva interna funciona correctamente.*

7.3.3 Cómo funciona

Recuerde que la implementación de la recursión requiere la realización de tareas de bajo nivel por parte del computador. Dicho de otra forma, la implementación de cualquier método requiere ciertas tareas de bajo nivel y un método recursivo no es en ello particularmente especial (excepto por el hecho de que puede sobrepasar las limitaciones del computador al llamarse a sí misma demasiadas veces).

Al igual que otros lenguajes como C++ y Ada, Java implementa los métodos mediante una *pila de registros de activación*. Podemos ver un registro de activación como un trozo de papel que contiene información relevante sobre el método, la cual incluye, por ejemplo, los valores de los parámetros y las variables locales. El contenido concreto de los registros de activación depende del sistema.

Se usa una pila de registros de activación porque la terminación de los métodos se produce en sentido inverso a la invocación de los mismos. Recuerde que las pilas son adecuadas para invertir secuencias. Lo más habitual es que la cima de la pila almacene el registro de activación del método actualmente activo. Cuando se llama al método *G*, se coloca un nuevo registro de activación en la cima de la pila, lo cual hace que *G* sea el método actualmente activo. Cuando termina un método, se desapila la cima de la pila y el registro de activación que queda en la cima contiene los valores previos.

Como ejemplo, en la Figura 7.5 se muestra una pila de registros de activación que aparecen en el curso de la evaluación de `s(4)`. En este punto, tenemos llamadas pendientes a `main`, `s(4)` y `s(3)`, y estamos procesando `s(2)`.

Se produce una sobrecarga en espacio debido a la memoria necesaria para almacenar un registro de activación para cada método actualmente activo. Así, en el ejemplo anterior en el que `s(9410)` aborta, el sistema tenía espacio reservado para unos 9.410 registros de activación. (Observe que `main` genera también un registro de activación.) La sobrecarga en tiempo producida al ejecutar una llamada a un método corresponde a la ejecución de las operaciones de apilar y desapilar sobre la pila interna.

La cercana relación entre recursión y pilas nos dice que los programas recursivos siempre se pueden implementar de forma iterativa con una pila explícita. Presumiblemente la pila almacenará objetos más pequeños que los registros de activación, dada la generalidad de éstos, por lo que podemos esperar razonablemente que se use algo menos de espacio. Además, el resultado sería ligeramente más rápido, pero el código notablemente más largo y complejo. Los modernos compiladores optimizantes han reducido los costes en tiempo asociados a la recursión a un grado tal que raramente vale la pena eliminar la recursión de una aplicación que la usa correctamente.

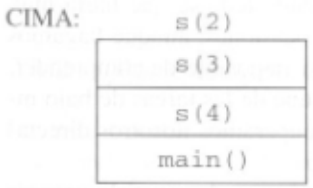


Figura 7.5 Pila de registros de activación.

7.3.4 Demasiada recursión puede ser peligrosa

En este libro hay muchos ejemplos que ilustran el poder de la recursión. Sin embargo, antes de examinar esos ejemplos, es importante darse cuenta de que la recursión no siempre es apropiada. Por ejemplo, el uso de la recursión en la Figura 7.1 es pobre, un bucle funcionaría igual de bien. El inconveniente práctico es que las llamadas recursivas consumen tiempo y limitan el valor de `n` para el cual se puede ejecutar el programa. Una buena regla es que nunca se debe usar la recursión en sustitución de un simple bucle.

Un problema mucho más serio aparece, por ejemplo, al intentar calcular los números de Fibonacci recursivamente. Los primeros *números de Fibonacci* $F_0, F_1, \ldots, F_i$ se definen de la siguiente forma: $F_0 = 0$ y $F_1 = 1$. El *i*-ésimo número de Fibonacci es igual a la suma de los dos anteriores, es decir $F_i = F_{i-1} + F_{i-2}$. A partir de esta definición, podemos determinar que la serie de números de Fibonacci continúa así: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Los números de Fibonacci tienen una enormidad de propiedades, que en muchos casos parecen surgir de forma inesperada. De hecho, la revista *The Fibonacci Quarterly*[3] existe solamente con el propósito de publicar teoremas que impliquen a los números de Fibonacci. Por ejemplo, la suma de los *N* primeros números de Fibonacci es uno menos que F_{N+2}, y la suma de los cuadrados de dos números de Fibonacci consecutivos es siempre otro número de Fibonacci (en el Ejercicio 7.9 se muestran otras igualdades interesantes).

Puesto que los números de Fibonacci se definen recursivamente, parece natural escribir una rutina recursiva para determinar F_N. Dicha rutina, mostrada en la Figura 7.6, funciona correctamente a nivel teórico, pero tiene un serio problema: en una máquina relativamente rápida tardamos varios minutos en calcular F_{40}. Esto es una cantidad absurda de tiempo considerando que el cálculo requerido precisa solamente de 39 sumas.

El problema subyacente es que esta rutina recursiva lleva a cabo multitud de cálculos repetidos. Para calcular `fib(n)`, calculamos recursivamente `fib(n-1)`. Cuando la llamada recursiva termina, calculamos `fib(n-2)` usando otra llamada recursiva. Pero nosotros ya habíamos calculado `fib(n-2)` al calcular `fib(n-1)`, luego la llamada a `fib(n-2)` es un cálculo repetido y, por ello, una pérdida de tiempo. De hecho se están haciendo dos llamadas a `fib(n-2)` en lugar de una.

```
1 // Calcula el n-ésimo número de Fibonacci
2 // Es un mal algoritmo
3 public static long fib( int n )
4 {
5     if( n <= 1 )
6         return n;
7     else
8         return fib( n - 1 ) + fib( n - 2 );
9 }
```

Figura 7.6 Rutina recursiva para el cálculo de los números de Fibonacci: una mala idea.

[3] *N. del T.:* Una publicación científica trimestral de alcance internacional "¿editorial?".

Normalmente, hacer dos llamadas a un método en lugar de una, doblaría el tiempo de ejecución de un programa, pero en este caso la situación es peor, porque tanto la llamada a `fib(n-1)` como las llamadas a `fib(n-2)` hacen a su vez llamadas a `fib(n-3)`, lo que significa que hay en realidad tres llamadas a `fib(n-3)`. A partir de ahí, la situación se vuelve aún peor: las llamadas a `fib(n-2)` y a `fib(n-3)` producen llamadas a `fib(n-4)`, cinco en total. Obtenemos por tanto un efecto acumulativo: cada llamada recursiva produce más y más trabajo redundante.

Sea $C(N)$ el número de llamadas a `fib` durante la evaluación de `fib(n)`. Claramente, $C(0) = C(1) = 1$. Para $N \geqslant 2$, hacemos la llamada a `fib(n)`, más todas las llamadas que se necesitan para evaluar `fib(n-1)` y `fib(n-2)`, recursiva e independientemente. Luego, $C(N) = C(N-1) + C(N-2) + 1$. Por inducción, podemos fácilmente probar que para $N \geqslant 3$ se tiene $C(N) = F_{N+2} + F_{N-1} - 1$. En consecuencia, el número de llamadas recursivas es todavía mayor que el número de Fibonacci que queremos calcular y por ello es exponencial. Por ejemplo, para $N = 40$, $F_{40} = 102.334.155$, mientras que el número total de llamadas recursivas es mayor que 300 millones. No se sorprenda si su programa no termina. En la Figura 7.7 se ilustra el crecimiento explosivo del número de llamadas recursivas.

Este ejemplo ilustra la cuarta y última regla básica de la recursión:

4. *Regla de interés compuesto:* nunca duplique trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

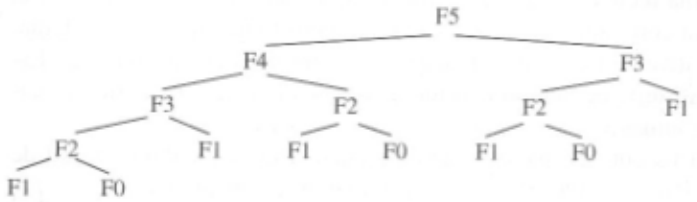


Figura 7.7 Traza del cálculo recursivo de los números de Fibonacci.

7.3.5 Ejemplos adicionales

Quizá la mejor forma de entender la recursión sea viendo algunos ejemplos significativos. En esta sección, presentamos cuatro ejemplos más de recursión. Los dos primeros se podrían implementar fácilmente de forma no recursiva, pero los dos últimos ponen al descubierto el poder de la recursión.

El factorial

Recuerde que $N!$ es el producto de los N primeros enteros, por lo que podemos escribir $N!$ como N veces $(N-1)!$. Combinado con el caso base $1! = 1$, ya tenemos todo lo necesario para una implementación recursiva, la cual se muestra en la Figura 7.8.

```
1 // Evalúa n!
2 public static long factorial( int n )
3 {
4     if( n <= 1 ) // caso base
5         return 1;
6     else
7         return n * factorial( n - 1 );
8 }
```

Figura 7.8 Implementación recursiva de la función `factorial`.

Búsqueda binaria

En la Sección 5.6.2 describimos la búsqueda binaria. Recuerde que en una búsqueda binaria, llevamos a cabo una búsqueda en un vector ordenado A examinando el elemento del centro. Si dicho elemento es el que buscamos, ya hemos terminado. En caso contrario, si el elemento que estamos buscando es menor que el elemento del centro, buscamos en la mitad izquierda del vector, y si es mayor, buscamos en la mitad derecha. Esto, siempre que el vector en cuestión no sea vacío, pues si lo es, entonces concluimos que el elemento buscado no se encuentra en el vector.

Esto se traduce directamente en el método recursivo de la Figura 7.9. El código ilustra la técnica en la que la rutina guía pública hace una llamada inicial a una rutina recursiva privada, y devuelve el resultado recibido (o relanza una excepción). Aquí, la guía inicializa los límites inferior y superior del fragmento del vector a `0` y `a.length-1`.

```
1  /**
2   * Lleva a cabo la búsqueda binaria estándar
3   * usando dos comparaciones por nivel.
4   * Esta es una guía que llama al método recursivo.
5   */
6  public static int busquedaBinaria( Comparable [ ] a,
7                                     Comparable x ) throws ElementoNoEncontrad
8  {
9      return busquedaBinaria( a, x, 0, a.length -1 );
10 }
11
12 /**
13  * Rutina recursiva oculta.
14  */
15 private static int busquedaBinaria( Comparable [ ] a,
16                                     Comparable x, int inf, int sup )
17                                                   throws ElementoNoEncontrad
18 {
19     if( inf > sup )
20         throw new ElementoNoEncontrado( "La búsqueda binaria falla" );
21     int med = ( inf + sup ) / 2;
22     if( a[ med ].compara( x ) < 0 )
23         return busquedaBinaria( a, x, med + 1, sup );
24     else if( a[ med ].compara( x ) > 0 )
25         return busquedaBinaria ( a, x, inf, med - 1 );
26     else
27         return med;
28 }
```

Figura 7.9 Rutina de búsqueda binaria usando recursión.

En el método recursivo, el caso base de las líneas 19 y 20 trata el caso del vector vacío. En caso contrario, seguimos las descripción dada, haciendo una llamada recursiva sobre el fragmento del vector apropiado (líneas 23 o 25) si no se ha encontrado ya el elemento en el centro. Cuando se encuentra el elemento, se devuelve su posición en la línea 27.

Observe que el tiempo de ejecución, en términos de O, no cambia con respecto a la implementación no recursiva, ya que estamos realizando esencialmente el mismo trabajo. En la práctica, el tiempo de ejecución será ligeramente superior debido a los costes ocultos de la recursión.

Dibujando una regla

La Figura 7.10 muestra el resultado de la ejecución de un programa Java que dibuja las marcas de una regla. Aquí consideramos el problema de dibujar una pulgada. En el medio se coloca la marca más larga. En la Figura 7.10, tanto a la izquierda como a la derecha de la marca central encontramos una versión miniaturizada de la regla. Esto sugiere un algoritmo recursivo que dibuja primero la línea central y después las mitades izquierda y derecha.

El Apéndice D proporciona los detalles del dibujo de líneas y figuras en Java. Basta decir aquí que el programador debe proporcionar un método `paint`, el cual puede llamar a otros métodos. El método `pintaRegla` de la Figura 7.11 es la rutina recursiva llamada desde `paint`. Usa el método `drawLine`, que es parte de la librería de Java. Este método pinta una línea desde un punto de coordenadas (x,y) a otro punto, donde las coordenadas son desplazamientos desde la esquina superior izquierda.

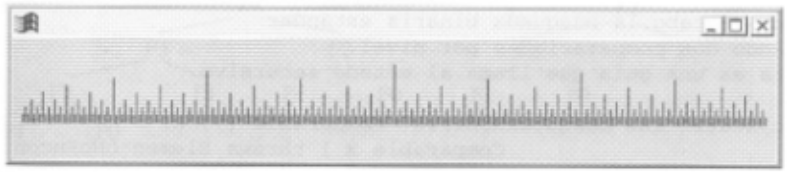


Figura 7.10 Una regla dibujada recursivamente.

```
1  private void
2  pintaRegla( Graphics g, int izdo, int dcho, int nivel)
3  {
4      if( nivel < 1 )
5          return;
6
7      int med = ( izdo + dcho ) / 2;
8
9      g.drawLine( med, 80, med, 80 - nivel * 5 );
10
11     pintaRegla( g, izdo, med - 1, nivel- 1 );
12     pintaRegla( g, med + 1, dcho, nivel - 1 );
13 }
```

Figura 7.11 Método recursivo para dibujar una regla.

Nuestra rutina dibuja marcas a `nivel` alturas diferentes; cada llamada recursiva es un nivel más profunda (en la Figura 7.10 hay ocho niveles). En las líneas 4 y 5 se trata el caso base. En la línea 9 se dibuja la línea central y después se dibujan recursivamente las dos miniaturas en las líneas 11 y 12. En el código disponible en Internet, incluimos código extra para ralentizar el proceso de dibujo, de forma que se pueda ir viendo el orden en que se pintan las líneas.

Estrella fractal

En la parte izquierda de la Figura 7.12 aparece aparentemente un complejo patrón, llamado *estrella fractal*, el cual sin embargo, se puede dibujar fácilmente usando recursión. El dibujo entero es inicialmente gris (no se muestra en la figura). El patrón se forma dibujando cuadrados blancos sobre el fondo gris. En la parte derecha de la Figura 7.12 se muestra el dibujo antes de añadir el último cuadrado, que va en el centro. Parece evidente que antes de añadir el último cuadrado del centro, se ha tenido que dibujar cuatro versiones en miniatura, una para cada uno de los cuadrantes, lo cual proporciona la información necesaria para derivar el algoritmo recursivo.

Igual que en el ejemplo anterior, el método `pintaFractal` será llamado desde `paint` y usa una rutina de la librería de Java, en este caso `fillRect`, que pinta un rectángulo cuya esquina superior izquierda y dimensiones deben especificarse. En la Figura 7.13 se muestra el código resultante. Los parámetros de `pintaFractal` incluyen el centro del fractal y la dimensión total. Con ellos podemos calcular en la línea 5 el tamaño del cuadrado grande del centro. Después de tratar el caso base en las líneas 7 y 8, calculamos los límites del rectángulo central. Entonces podemos dibujar los cuatro pequeños fractales en las líneas 17 a 20. Finalmente dibujamos el cuadrado central en la línea 23. Observe que el cuadrado se debe pintar tras las llamadas recursivas, ya que en caso contrario se obtendría un dibujo diferente (en el Ejercicio 7.28 se le pide que explique la diferencia).

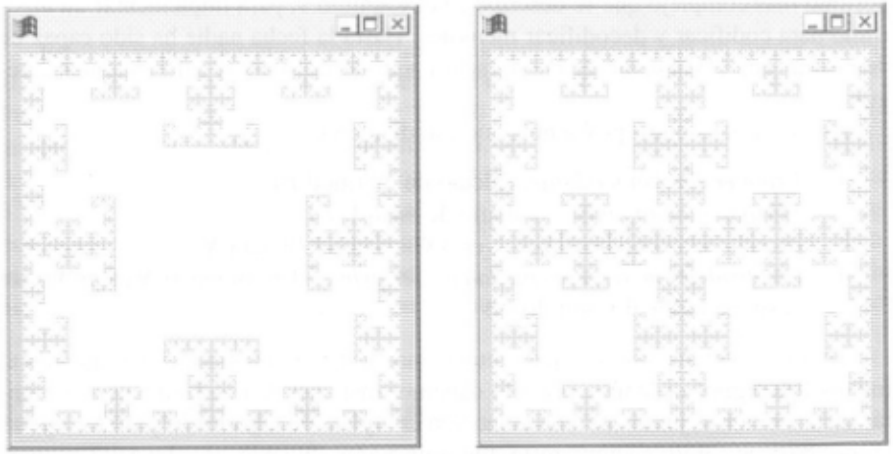


Figura 7.12 La figura de la izquierda es una estrella fractal dibujada por el programa de la Figura 7.13. La figura de la derecha es la misma estrella antes de añadir el último cuadrado.

```
1  // Pinta el dibujo de la Figura 7.12
2  private void pintaFractal( Graphics g, int xCentro,
3                             int yCentro, int cotaDim )
4  {
5      int lado = cotaDim / 2;
6
7      if( lado < 1 )
8          return;
9
10         // Cálculo de las esquinas
11     int izdo = xCentro - lado / 2;
12     int sup =  yCentro - lado / 2;
13     int dcho = xCentro + lado / 2;
14     int inf =  yCentro + lado / 2;
15
16         // Dibujo recursivo de los cuatro cuadrantes
17     pintaFractal( g, izdo, sup, cotaDim / 2 );
18     pintaFractal( g, izdo, inf, cotaDim / 2 );
19     pintaFractal( g, dcho, sup, cotaDim / 2 );
20     pintaFractal( g, dcho, inf, cotaDim / 2 );
21
22         // Dibuja el cuadro central, que se solapa con los cuadrantes
23     g.fillRect( izdo, sup, dcho - izdo, inf - sup );
24 }
```

Figura 7.13 Código para dibujar la estrella fractal de la Figura 7.12.

7.4 Aplicaciones numéricas

Esta sección estudia tres problemas que proceden de la teoría de números. La teoría de números se solía considerar una rama interesante pero inútil de las matemáticas. Pero, en los últimos 20 años, ha surgido una importante aplicación de la teoría de números: la seguridad de los datos. La discusión comienza con unos breves fundamentos matemáticos, para continuar con sendos algoritmos recursivos para resolver tres problemas. Podemos combinar estas rutinas, junto con un cuarto algoritmo más complejo que se describe en el Capítulo 9, para implementar un algoritmo para codificar y decodificar mensajes. Hasta la fecha nadie ha sido capaz de demostrar que el esquema de encriptación aquí descrito no es suficientemente seguro.

Estos son los cuatro problemas que estudiaremos:

1. *Exponenciación modular*: cálculo de $X^N (\text{mod } P)$.
2. *Máximo común divisor*: cálculo de $mcd(A, B)$.
3. *Inverso multiplicativo*: resolver $AX \equiv 1 (\text{mod } P)$ para X.
4. *Comprobación de la propiedad de ser primo*: determinar si N es primo (se pospone hasta el Capítulo 9).

Todos los enteros con los que pretendemos trabajar son grandes y requieren al menos 100 dígitos cada uno. Por ello debemos disponer de una forma de representar dichos enteros, junto con una colección completa de algoritmos para las operaciones básicas como la suma, resta, multiplicación, división, etc. Java 1.1 proporciona para este propósito una clase denominada `BigDecimal`. Implementarla de forma eficiente no es una tarea trivial, y de hecho existe una gran cantidad de literatura sobre el tema.

Los algoritmos aquí descritos tienen la propiedad de que cuando trabajan con ese tipo de objetos aún requieren una cantidad razonable de tiempo. Sin embargo, los métodos que escribimos funcionarán con el tipo `long`, de modo que podamos fácilmente comprobar el funcionamiento de los algoritmos básicos. Debido a que Java no permite la sobrecarga de operadores, sería necesario rescribir el código usando métodos con nombre, en lugar de operadores, si queremos usar objetos `BigDecimal`.

7.4.1 Aritmética modular

En los distintos problemas que abordamos en esta sección, al igual que sucedía en la implementación de la tabla hash de la Sección 6.7 y del Capítulo 19, se requiere el uso del operador de módulo en Java, el cual se denota mediante el operador `%`. Este operador calcula el resto de la división entre dos valores de tipo entero. Por ejemplo, `13%10` se evalúa a 3, al igual que `3%10` y `23%10`. Cuando calculamos el resto de una división por 10, el rango de posibles resultados está entre 0 y 9. Esto hace que el operador `%` sea útil cuando necesitamos generar enteros pequeños.

Si dos números A y B producen el mismo resto al dividirlos por N, decimos que son congruentes módulo N, lo que se denota por $A \equiv B (\text{mod } N)$. En tal caso, se cumplirá que N divide a $A - B$. Además, el recíproco también es cierto: si N divide a $A - B$, entonces $A \equiv B (\text{mod } N)$. Puesto que sólo hay N posibles restos — 0, 1, ..., $N - 1$ — decimos que los enteros se dividen en N clases de congruencia módulo N. En otras palabras, cada entero se puede colocar en una de las N clases, y aquellos que pertenecen a la misma clase son congruentes entre sí, módulo N. Hay tres propiedades importantes que usamos en nuestros algoritmos (dejamos su demostración como Ejercicio 7.10):

1. Si $A \equiv B (\text{mod } N)$, entonces para cualquier C, $A + C \equiv B + C (\text{mod } N)$.
2. Si $A \equiv B (\text{mod } N)$, entonces para cualquier D, $AD \equiv BD (\text{mod } N)$.
3. Si $A \equiv B (\text{mod } N)$, entonces para cualquier P, $A^P \equiv B^P (\text{mod } N)$.

Estos teoremas permiten que algunos cálculos se hagan con un menor esfuerzo. Por ejemplo, supongamos que queremos conocer el último dígito de 3333^{5555}. Puesto que este número tiene más de 15.000 dígitos, es demasiado costoso calcular directamente la respuesta. Sin embargo, lo que queremos calcular en realidad es $3333^{5555} (\text{mod } 10)$. Puesto que $3333 \equiv 3 (\text{mod } 10)$, basta con calcular $3^{5555} (\text{mod } 10)$. Puesto que $3^4 = 81$, sabemos que $3^4 \equiv 1 (\text{mod } 10)$, y elevando ambos lados a la potencia 1.388 obtenemos $3^{5552} \equiv 1 (\text{mod } 10)$. Si ahora multiplicamos ambos lados por $3^3 = 27$, obtenemos $3^{5555} \equiv 27 \equiv 7 (\text{mod } 10)$, completando así el cálculo. En la siguiente sección se muestra cómo generalizar este proceso.

7.4.2 Exponenciación modular

En esta sección se muestra cómo calcular de forma eficiente $X^N (\text{mod } P)$. Esto se puede hacer inicializando el resultado `tmp` a 1 y después multiplicando repetidamente el resultado `tmp` por X y aplicando el operador `%` tras cada multiplicación.

Aplicar el operador % de esta forma, en lugar de hacerlo después de la última multiplicación, hace que las multiplicaciones sean más sencillas, ya que `resultado tmp` se mantiene pequeño.

Tras N multiplicaciones, `resultado tmp` es la respuesta que estamos buscando. Sin embargo, si N es un número de 100 dígitos, la multiplicación es aún impracticable. De hecho, si N vale mil millones, resulta impracticable en cualquier máquina excepto en las más rápidas.

Un algoritmo más rápido se basa en la siguiente observación: si N es par entonces $X^N = (X \cdot X)^{\lfloor N/2 \rfloor}$ y si es impar entonces $X^N = X \cdot X^{N-1} = X \cdot (X \cdot X)^{\lfloor N/2 \rfloor}$. (Recuerde que $\lfloor X \rfloor$ es el mayor entero que es menor o igual que X.) Al igual que antes, aplique el operador % tras cada multiplicación.

El algoritmo recursivo de la Figura 7.14 representa una implementación directa de la estrategia. En las líneas 7 y 8 se trata el caso base: X^0 vale 1 por definición[4]. En la línea 10, hacemos una llamada recursiva basada en la primera igualdad del párrafo anterior. Si N es par, dicha fórmula calcula la respuesta deseada, y si es impar necesitamos multiplicarla por una N extra (usando después el operador %).

Este algoritmo es más rápido que el algoritmo más simple propuesto anteriormente. Si $M(N)$ es el número de multiplicaciones usadas por `potencia`, entonces $M(N) \leqslant M(\lfloor N/2 \rfloor) + 2$; pues si N es par, llevamos a cabo una multiplicación más las que se generan recursivamente, mientras que si N es impar, realizamos dos multiplicaciones más las que se ejecutan recursivamente. Puesto que $M(0) = 0$, podemos demostrar que $M(N) < 2 \log N$. El factor logarítmico se puede obtener sin necesidad de un cálculo directo aplicando el principio de sucesivas divisiones por la mitad (*véase* Sección 5.5), que nos da el número de llamadas recursivas a `potencia`. Además, el valor medio de $M(N)$ es $(3/2) \log N$, ya que en cada paso recursivo, es igualmente probable que N sea par o impar. Si N es un número de 100 dígitos, entonces, en el peor caso, sólo se necesitan 655 multiplicaciones (y sólo 500 en media).

```
1  /**
2   * Devuelve x^n (mod p)
3   * Suponiendo que p es positivo y potencia(0, 0, p) es 1
4   */
5  public static long potencia( long x, long n, long p )
6  {
7      if( n == 0 )
8          return 1;
9
10     long tmp = potencia( ( x * x ) % p, n / 2, p );
11     if( n % 2 != 0 )
12         tmp = ( tmp * x ) % p;
13
14     return tmp;
15 }
```

Figura 7.14 Rutina de exponenciación modular.

[4] de este algoritmo, $0^0 = 1$, N es no negativo y P es positivo.

7.4.3 Máximo común divisor e inversos multiplicativos

Considere dos enteros no negativos A y B. Su máximo común divisor, $mcd(A, B)$, es el mayor entero D que divide a A y a B. Por ejemplo, $mcd(70, 25)$ es 5.

Es fácil verificar que $mcd(A, B) \equiv mcd(A - B, B)$. Si D divide a A y a B, también debe dividir a $A - B$; y si D divide a $A - B$ y a B, entonces también debe dividir a A.

Esta observación nos lleva a un algoritmo simple en el que restamos repetidamente B a A, transformando así el problema en uno más pequeño. Eventualmente A se hará más pequeño que B, y entonces podemos intercambiar los papeles de A y B y continuar a partir de ahí. En algún punto, B se hará 0, y entonces, como sabemos que $mcd(A, 0) \equiv A$, y puesto que cada transformación preserva el mcd de los A y B originales, ya tenemos nuestra respuesta. A este algoritmo se le llama algoritmo de Euclides y se describió por primera vez hace 2.000 años. Aunque es correcto, no se puede usar para objetos de la clase `BigDecimal` porque se requeriría un enorme número de restas.

Una modificación que hace más eficiente al algoritmo se basa en el hecho de que restar repetidamente B a A hasta que A es más pequeño que B, es equivalente a convertir A en $A \bmod B$; es decir $mcd(A, B) \equiv mcd(B, A \bmod B)$. Esta definición recursiva, junto con el caso base, en el que $B = 0$, se usa directamente para obtener la rutina de la Figura 7.15. Para ver cómo funciona, observe que en el ejemplo anterior se produciría la siguiente secuencia de llamadas recursivas para deducir que el mcd de 70 y 25 es 5: $mcd(70, 25) \Rightarrow mcd(25, 20) \Rightarrow mcd(20, 5) \Rightarrow mcd(5, 0) \Rightarrow 5$.

El número de llamadas recursivas es proporcional al logaritmo de A, es decir, es del mismo orden de magnitud que el resto de rutinas estudiadas en esta sección. Esto se debe a que al hacer una llamada recursiva el argumento se reduce al menos a la mitad. La demostración se deja como Ejercicio 7.11.

El algoritmo de mcd se usa de forma implícita para resolver un problema matemático similar. La solución $1 \leqslant X < N$ a la ecuación $AX \equiv 1 \pmod N$ se llama *inverso multiplicativo* de A, módulo N. Supongamos también que $1 \leqslant A < N$. Como ejemplo, el inverso de 3 módulo 13 es 9: $3 \cdot 9 \bmod 13$ devuelve 1.

La posibilidad de calcular inversos multiplicativos es importante porque ecuaciones como $3i \equiv 7 \pmod{13}$ se resuelven fácilmente si conocemos el inverso multiplicativo. Este tipo de ecuaciones surgen en muchas aplicaciones, incluyendo el algoritmo de encriptación estudiado al final de esta sección. En el ejemplo ante-

```
1  /**
2   * Devuelve el máximo común divisor.
3   */
4  public static long mcd( long a, long b )
5  {
6      if( b == 0 )
7          return a;
8      else
9          return mcd(b, a % b);
10 }
```

Figura 7.15 Cálculo del máximo común divisor.

rior, si multiplicamos por el inverso de 3 (es decir, 9), obtenemos $i \equiv 63 (\text{mod } 13)$, por lo que $i = 11$ es una solución. Si

$$AX \equiv 1 (\text{mod } N)$$

entonces

$$AX + NY \equiv 1 (\text{mod } N)$$

es cierto para cualquier Y. Para algún Y, el lado izquierdo debe ser exactamente 1. Luego la ecuación

$$AX + NY = 1$$

tiene solución si y sólo si A tiene un inverso multiplicativo.

Dados A y B, mostramos cómo encontrar X e Y que cumplan

$$AX + BY = 1.$$

Supondremos, sin pérdida de generalidad, que $0 \leqslant |B| < |A|$. Extenderemos el algoritmo del *mcd* para calcular X e Y.

Consideremos primero el caso base $B \equiv 0$, para el cual tenemos que resolver $AX = 1$, lo que implica que tanto A como X valen 1, ya que si A no es 1, entonces no hay inverso multiplicativo. Una consecuencia de esto es que A tiene inverso multiplicativo módulo N si y sólo si $mcd(A, N) = 1$.

Si B no es 0, recuerde que $mcd(A, B) = mcd(B, A \bmod B)$. Sea $A = BQ + R$, donde Q es el cociente y R es el resto. La llamada recursiva es entonces $mcd(B, R)$. Supongamos que podemos resolver recursivamente

$$BX_1 + RY_1 = 1.$$

Puesto que $R = A - BQ$, tenemos

$$BX_1 + (A - BQ)Y_1 = 1,$$

lo que significa que

$$AY_1 + B(X_1 - QY_1) = 1.$$

Luego $X = Y_1$ y $Y = X_1 - \lfloor A/B \rfloor Y_1$ es una solución de $AX + BY = 1$. La codificación de este algoritmo aparece en la Figura 7.16 con el nombre de `mcdCompleto`. La rutina `inverso` simplemente llama a `mcdCompleto`, donde X e Y son variables estáticas. El único detalle pendiente es que el valor de X podría ser negativo, en cuyo caso la línea 35 de `inverso` lo convertirá en positivo. La demostración de la corrección se puede hacer por inducción.

```
1     // Variables internas de mcdCompleto
2  private static long x;
3  private static long y;
4
5  /**
6   * Sigue hacia atrás el algoritmo de Euclides para encontrar
7   * x e y tales que mcd(a, b) = 1,
8   * ax + by = 1,
9   */
10 private static void mcdCompleto( long a, long b )
11 {
12     long x1, y1;
13
14     if( b == 0)
15     {
16         x = 1;          // Si a != 1, no hay inverso
17         y = 0;          // Omitimos esta comprobación
18     }
19     else
20     {
21         mcdCompleto( b, a % b );
22         x1 = x; y1 = y;
23         x = y1;
24         y = x1 - (a / b) * y1;
25     }
26 }
27
28 /**
29  * Resuelve ax == 1 (mod n), suponiendo que mcd( a, n ) = 1.
30  * @return x.
31  */
32 public static long inverso( long a, long n )
33 {
34     mcdCompleto( a, n );
35     return x > 0 ? x : x + n;
36 }
```

Figura 7.16 Rutina para determinar el inverso multiplicativo.

7.4.4 El sistema de criptografía RSA

Durante siglos se pensó que la teoría de números era una rama de las matemáticas completamente inútil desde el punto de vista práctico, pero recientemente se ha convertido en un campo importante debido a sus aplicaciones a la criptografía.

El problema que consideramos es el siguiente: supongamos que Alicia quiere enviar un mensaje a Roberto, pero que a ella le preocupa que la transmisión sea interceptada. Por ejemplo, si la transmisión se realiza a través de una línea telefónica y el teléfono está pinchado, entonces alguien más puede estar escuchando el mensaje. Supondremos, eso sí, que incluso si hay escuchas en la línea telefónica, no se produce ningún tipo de daño sobre la señal; es decir, Roberto obtiene exactamente lo que le envía Alicia.

Una solución a este problema es usar un esquema de encriptación que consta de dos partes. Primero, Alicia *codifica* el mensaje y envía el resultado, el cual ya no se puede leer directamente. Cuando Roberto recibe la transmisión de Alicia, la *decodifica* para obtener el mensaje original. La seguridad del algoritmo se basa en que nadie más que Roberto podría llevar a cabo la decodificación del mensaje (ni siquiera Alicia, si no ha guardado el original).

Para aplicar el proceso, Roberto debe proporcionar a Alicia un método de encriptación que solamente él sepa cómo invertir. Este problema es un serio reto. Muchos de los algoritmos que han venido siendo propuestos pueden verse comprometidos por alguna técnica sutil para averiguar el código. Aquí se describe un método, llamado *sistema de encriptación RSA* (por las iniciales de sus autores) que es una implementación muy elegante de una estrategia de encriptación.

Nuestro objetivo es dar solamente una visión de alto nivel, mostrando cómo interactúan los métodos escritos en esta sección. Las referencias contienen punteros a descripciones más detalladas, así como a demostraciones de las propiedades clave de los algoritmos.

Observemos primero que un mensaje consta de una secuencia de caracteres y que cada carácter es una secuencia de bits; luego un mensaje es, al fin y al cabo, una secuencia de bits. Si partimos el mensaje en bloques de B bits, podemos interpretar el mensaje como una serie de números muy grandes, por lo que el problema básico se reduce a codificar y después decodificar números grandes.

Cálculo de las constantes RSA

El algoritmo RSA comienza exigiendo que el receptor del mensaje determine ciertas constantes adecuadas. Primero, se eligen aleatoriamente dos números primos grandes p y q, habitualmente de 100 dígitos o más cada uno. Para hacer manejable el ejemplo, tomaremos $p = 127$ y $q = 211$. Observe que Roberto, que es el receptor, es quien está haciendo este proceso. Como quiera que hay muchos números primos, Roberto puede continuar eligiendo pares de números primos hasta que uno de ellos cumpla el test de primalidad, que se discutirá en el Capítulo 9.

A continuación, Roberto calcula $N = pq$ y $N' = (p - 1)(q - 1)$. En nuestro ejemplo, obtenemos $N = 26.797$ y $N' = 26.460$. Roberto continúa eligiendo un $e > 1$ tal que $mcd(e, N') = 1$. En términos matemáticos, elige cualquier e que sea relativamente primo con N'. De nuevo, Roberto puede ir probando con distintos valores de e, usando la rutina de la Figura 7.15, hasta encontrar uno que satisfaga la propiedad. En particular, como cualquier primo e valdría, encontrar e es tan fácil como encontrar un número primo. En nuestro caso, $e = 13.379$ es una de las muchas opciones. A continuación, se calcula el inverso multiplicativo de e módulo N', d, usando la rutina de la Figura 7.16. En el ejemplo, $d = 11.099$.

Una vez que Roberto ha calculado todas estas constantes, hace lo siguiente: primero, destruye p, q y N'. La seguridad del sistema se podría ver comprometida si se cuenta con dichos valores. Entonces, Roberto le dice a todo el mundo que quiera enviarle un mensaje codificado cuáles son los valores de e y N, aunque guarda en secreto el valor de d.

Algoritmos de codificación y decodificación

Para codificar un entero M, el emisor calcula $M^e \pmod N$ y lo envía. En nuestro caso $M = 10.237$, luego el valor enviado es 8.422. Cuando se recibe un entero codificado R, todo lo que tiene que hacer Roberto es calcular $R^d \pmod N$. Para $R = 8.422$ se puede comprobar que se obtiene de nuevo el $M = 10.237$ original, lo cual no es en absoluto accidental. Tanto la codificación como la decodificación se pueden llevar a cabo usando la rutina de exponenciación modular de la Figura 7.14.

¿Por qué funciona el algoritmo? La elección de e, d y N garantiza (por medio de una demostración en teoría de números fuera del ámbito de este libro) que $M^{ed} \equiv M \pmod N$, siempre que N y M no tengan factores comunes. Puesto que los únicos factores de N son dos números primos de 100 dígitos, es prácticamente imposible que esto suceda[5]. Por tanto, la decodificación del texto codificado devuelve siempre el texto original.

Lo que hace que el sistema que hemos visto sea seguro es que en principio se necesita d para poder decodificar un mensaje. Ahora bien N y e determinan únicamente a d. Por ejemplo, si factorizamos N, obtenemos p y q y a partir de ellos se puede reconstruir d. El problema está en que factorizar números grandes es aparentemente muy difícil. En consecuencia, la seguridad del sistema RSA se basa en el hecho de que se piensa que factorizar números muy grandes es intrínsecamente muy difícil. Como quiera que hasta la fecha nadie ha sido capaz de factorizar de forma eficiente, aunque tampoco tenemos la prueba de que ello sea un hecho imposible, hasta el momento el método funciona bien.

Este esquema general recibe el nombre de *criptografía de clave pública*. Cualquiera que quiera recibir mensajes, publica información de codificación para todos aquellos que quieran usarla. En el sistema RSA, e y N se calcularían una vez por parte de cada receptor y se harían públicos en una lista.

El algoritmo RSA se usa ampliamente para implementar correo seguro, así como transacciones seguras en Internet. Cuando vea un candado cerrado (🔒) en la parte inferior de una página Web en Netscape Navigator, se está realizando una transacción segura usando criptografía. El método que se emplea realmente es más complejo que el aquí descrito. La razón que justifica dicho cambio de método es que el algoritmo RSA resulta lento cuando se trata de enviar mensajes bastante largos.

Un método más rápido es el llamado *DES*. A diferencia del algoritmo RSA, DES es un algoritmo de clave única, lo que significa que la misma clave sirve para codificar y decodificar. Esto es semejante al típico cerrojo en la puerta de nuestra casa. El problema con la clave única es que ambas partes tienen que compartir la clave única. ¿Cómo se asegura una parte de que la otra tiene la clave única? Esto se resuelve usando el algoritmo RSA. Una solución habitual es que, Alicia genera aleatoriamente una clave para la codificación DES. Con ella codifica el mensaje usando DES, lo cual es mucho más rápido que usar RSA. Transmite el mensaje codificado a Roberto. Para que Roberto pueda decodificar el mensaje, primero necesita obtener la clave DES. Puesto que dicha clave es relativamente corta, Alicia

[5] Es más probable que gane la lotería primitiva 13 semanas seguidas. Sin embargo, si esto ocurriera, el sistem
tido, porque el *mcd* será un factor de N.

puede usar RSA para codificar la clave DES, enviándola en una segunda transmisión a Roberto. Roberto decodifica la segunda transmisión, obteniendo así la clave DES, con la que se puede decodificar el mensaje original. Este tipo de protocolos con refuerzos forman la base de la mayor parte de las implementaciones prácticas de la codificación.

7.5 Algoritmos divide y vencerás

Una técnica importante de resolución de problemas que hace uso de la recursión es la técnica de *divide y vencerás*. Los algoritmos divide y vencerás son algoritmos recursivos que constan de dos partes:

- *Dividir*: se resuelven recursivamente problemas más pequeños (excepto, por supuesto, los casos base).
- *Vencer*: la solución al problema original se consigue a partir de las soluciones a los subproblemas.

Tradicionalmente, las rutinas en las que el algoritmo contiene al menos dos llamadas recursivas se llaman algoritmos divide y vencerás, al contrario que las rutinas cuyo texto contiene solamente una llamada recursiva. En consecuencia, las rutinas recursivas vistas hasta el momento en este capítulo, no son algoritmos divide y vencerás. Además, los subproblemas deben ser disjuntos (más exactamente, esencialmente sin superposiciones) para evitar los costes excesivos que ya se vieron en el cálculo recursivo de los números de Fibonacci. En esta sección presentaremos un ejemplo del paradigma divide y vencerás: veremos cómo usar la recursión para resolver el problema de la obtención de la subsecuencia de suma máxima. A continuación presentamos el análisis de esta solución recursiva, demostrando que su tiempo de ejecución es $O(N \log N)$. Aunque ya vimos un algoritmo lineal para resolver este problema, lo que presentamos ahora es un ejemplo representativo de una gran variedad de aplicaciones, incluyendo los algoritmos de ordenación, como mergesort y quicksort, estudiados en el Capítulo 8. Resulta por tanto muy importante estudiar esta técnica. Para terminar, discutiremos la forma general del tiempo de ejecución de una amplia variedad de algoritmos divide y vencerás.

7.5.1 El problema de la subsecuencia de suma máxima

Recordemos el problema de encontrar, en una secuencia de números, una subsecuencia contigua de suma máxima, que estudiamos en la Sección 5.3. Para facilitar la lectura, volvemos a incluir su enunciado a continuación:

PROBLEMA DE LA OBTENCIÓN DE LA SUBSECUENCIA DE SUMA MÁXIMA

Dada una sucesión de enteros (posiblemente negativos) $A_1, A_2, \ldots, A_N$, encontrar (e identificar la secuencia correspondiente a) el valor máximo de $\sum_{k=i}^{j} A_k$. Consideraremos que la subsecuencia contigua de suma máxima es la vacía, de suma cero, si todos los enteros son negativos.

Para resolver este problema se presentaron tres algoritmos de distintas complejidades. Uno de ellos era un algoritmo cúbico basado en una búsqueda exhaustiva: calculábamos la suma de cada posible subsecuencia y elegíamos la máxima. Se describió también una mejora con tiempo cuadrático que aprovechaba el hecho de que la suma de una nueva subsecuencia se puede calcular en tiempo constante a partir de la anterior. Puesto que tenemos $O(N^2)$ subsecuencias, ésta es la mejor cota que se puede conseguir usando una aproximación que examine directamente todas las subsecuencias. Vimos, por último, un algoritmo con coste en tiempo lineal que examinaba solamente unas pocas subsecuencias, y cuya corrección no era obvia.

Diseñaremos ahora un algoritmo divide y vencerás. Supongamos que la sucesión dada es $\{4, -3, 5, -2, -1, 2, 6, -2\}$. Dividiremos esta entrada en dos partes iguales, tal y como se muestra en la Figura 7.17. Entonces la subsecuencia de suma máxima puede aparecer en una de estas tres formas:

- *Caso* 1: está totalmente incluida en la primera mitad.
- *Caso* 2: está totalmente incluida en la segunda mitad.
- *Caso* 3: comienza en la primera mitad, pero termina en la segunda.

Mostraremos cómo encontrar los máximos valores de cada uno de estos tres casos, de una forma más eficiente que la que correspondería a una búsqueda exhaustiva.

Empezamos por el caso 3. Queremos evitar el bucle anidado generado al considerar independientemente todos los $N/2$ puntos de comienzo y los $N/2$ puntos de finalización. La idea es sustituir dos bucles anidados por dos bucles consecutivos. Los bucles consecutivos, cada uno de tamaño $N/2$, se combinan, requiriendo en total un tiempo lineal. Podemos hacer esto porque cualquier subsecuencia contigua que comience en la primera mitad y termine en la segunda, debe incluir el último elemento de la primera mitad y el primero de la segunda mitad.

La Figura 7.17 muestra que podemos calcular, para cada elemento de la primera mitad, la suma de la subsecuencia contigua que termina en el elemento situado más a la derecha. Hacemos esto con un recorrido de derecha a izquierda, partiendo del elemento situado entre las dos mitades. Análogamente, podemos calcular la suma de todas las subsecuencias contiguas que comiencen con el primer elemento de la segunda mitad. Entonces se puede combinar estas dos subsecuencias para formar la subsecuencia de suma máxima que cruza la línea divisoria. En el ejemplo de la Figura 7.17, la secuencia resultante va desde el primer elemento de la primera mitad hasta el penúltimo elemento de la segunda mitad. La suma total es la suma de las dos subsecuencias, $4 + 7 = 11$.

Primera mitad				Segunda mitad				
4	−3	5	−2	−1	2	6	−2	Valores
4*	0	3	−2	−1	1	7*	5	Sumas

Suma desde el centro
(*denota el máximo para cada mitad)

Figura 7.17 Dividiendo en dos el problema de la obtención de la subsecuencia máxima.

Este análisis nos muestra que el caso 3 se puede resolver en tiempo lineal. Pero, ¿qué pasa con los casos 1 y 2? Puesto que hay $N/2$ elementos en cada mitad, la aplicación de una búsqueda exhaustiva en cada mitad requeriría un tiempo cuadrático por cada mitad. Específicamente, lo único que hemos hecho es eliminar la mitad del trabajo, y la mitad de un tiempo cuadrático sigue siendo cuadrático. Pero podemos aplicar la misma estrategia de división por la mitad en los casos 1 y 2. Podemos continuar dividiendo hasta que sea imposible dividir más. Esto equivale, más concretamente, a resolver los casos 1 y 2 recursivamente. Como se demostrará más adelante en el capítulo, esto reducirá el tiempo de ejecución por debajo de cuadrático, pues los ahorros se acumulan a lo largo de la ejecución del algoritmo. Mostramos a continuación un esquema del algoritmo:

1. Calcular recursivamente la subsecuencia de suma máxima que está totalmente contenida en la primera mitad.
2. Calcular recursivamente la subsecuencia de suma máxima que está totalmente contenida en la segunda mitad.
3. Calcular, usando dos bucles consecutivos, la subsecuencia de suma máxima que comienza en la primera mitad pero termina en la segunda.
4. Elegir la mayor de las tres sumas.

El método Java resultante aparece en la Figura 7.18. Un algoritmo recursivo nos exige definir un caso base. Naturalmente, cuando el dato es un solo elemento, no usamos recursión.

A la llamada recursiva se le pasa el vector de entrada junto con los límites izquierdo y derecho, los cuales delimitan la porción de vector sobre la que se está operando. Una rutina guía de una línea inicializa los parámetros límite a 0 y $N-1$.

Las líneas 13 y 14 tratan el caso base. Si `izdo==dcho`, sólo tenemos un elemento que conforma la subsecuencia de suma máxima si es no negativo; en caso contrario, la secuencia vacía con suma 0 es máxima. Las líneas 16 y 17 llevan a cabo las llamadas recursivas. Podemos ver que las llamadas recursivas siempre se realizan sobre un argumento más pequeño que el original, por lo que hacemos progresos hacia el caso base. Las líneas 19 a 24 y 25 a 30 calculan las sumas máximas que tocan el borde central. La suma de estos dos valores es la suma máxima que se extiende por ambas mitades. La rutina trivial `max3` (que no se muestra) devuelve la mayor de las tres soluciones parciales.

7.5.2 Análisis de un algoritmo divide y vencerás sencillo

El algoritmo recursivo para la obtención de la subsecuencia de suma máxima usa un tiempo lineal para calcular la suma de la mayor secuencia que atraviesa el borde, y después hace dos llamadas recursivas. Las llamadas recursivas calculan una suma que atraviesa el borde central, después hacen más llamadas recursivas y así sucesivamente. El trabajo total invertido por el algoritmo es entonces proporcional al coste de las búsquedas realizadas por todas las llamadas recursivas.

La Figura 7.19 ilustra gráficamente cómo trabaja el algoritmo para $N=8$ elementos. Cada rectángulo representa una llamada a `maxSumRec`, y la longitud del rectángulo es proporcional al tamaño del subvector (y por tanto al coste del recorrido del subvector) sobre el que actúa la llamada. La llamada inicial se muestra en la primera línea. Observe que el tamaño del subvector es N, lo cual representa

```
1  /**
2   * Algoritmo de obtención de la subsecuencia de suma máxima.
3   * Encuentra la suma máxima de un subvector a[izdo..dcho].
4   * No intenta mantener la mejor secuencia.
5   */
6  private static int maxSumRec( int [ ] a, int izdo,
7                                int dcho )
8  {
9      int maxSumIzdaBorde = 0, maxSumDchaBorde = 0;
10     int sumIzdaBorde = 0, sumDchaBorde = 0;
11     int centro = ( izdo + dcho ) / 2;
12
13     if( izdo == dcho )   //Caso base
14         return a[ izdo ] > 0 ? a[ izdo ] : 0;
15
16     int maxSumIzda = maxSumRec( a, izdo, centro );
17     int maxSumDcha = maxSumRec( a, centro + 1, dcho );
18
19     for( int i = centro; i >= izdo; i-- )
20     {
21         sumIzdaBorde += a[ i ];
22         if (sumIzdaBorde > maxSumIzdaBorde)
23             maxSumIzdaBorde = sumIzdaBorde;
24     }
25     for( int j = centro + 1; j <= dcho; j++ )
26     {
27         sumDchaBorde += a[ j ];
28         if (sumDchaBorde > maxSumDchaBorde)
29             maxSumDchaBorde = sumDchaBorde;
30     }
31
32     return max3( maxSumIzda, maxSumDcha,
33                  maxSumIzdaBorde + maxSumDchaBorde );
34 }
35
36 // Rutina visible públicamente
37 public static int maxSubSum4( int [ ] a)
38 {
39     return maxSumRec( a, 0, a.length - 1 );
40 }
```

Figura 7.18 Algoritmo divide y vencerás para el problema de la obtención de la subsecuencia de suma máxima.

el coste del recorrido para el tercer caso. La llamada inicial hace entonces dos llamadas recursivas sobre dos subvectores de tamaño $N/2$. El coste de cada recorrido en el caso 3 es la mitad del coste original, pero puesto que hay dos llamadas recursivas, el coste combinado de las dos llamadas recursivas es también N. Cada una de esas dos instancias recursivas hacen a su vez dos llamadas recursivas, generando por tanto cuatro subproblemas que son una cuarta parte del problema original. Luego el total de los costes para el caso 3 es también N.

Eventualmente, llegaremos al caso base. Cada uno de ellos tiene tamaño 1, pero tenemos N. Por supuesto, no hay costes para el caso 3 en este caso, pero consideramos que la comprobación que determina si el único elemento es positivo o negativo tiene un coste de 1 unidad de tiempo. Por tanto, el coste total por nivel de recursión, mostrado en la Figura 7.19, es N. Cada nivel reduce a la mitad el tamaño

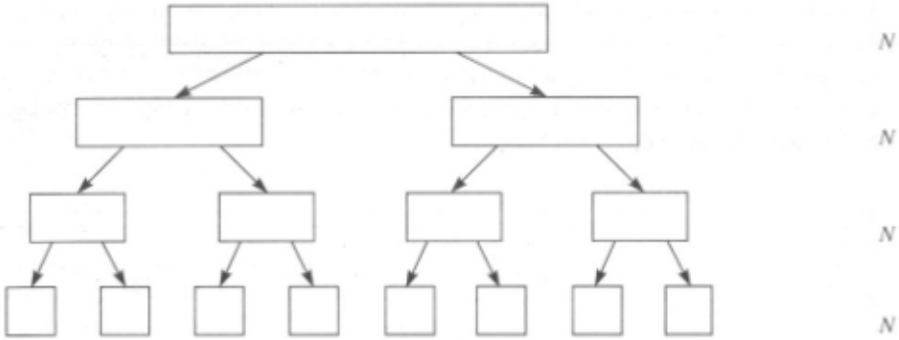


Figura 7.19 Traza de las llamadas recursivas del algoritmo de la subsecuencia contigua de suma máxima; $N = 8$ elementos.

del problema original, luego por el principio de las sucesivas divisiones por la mitad tenemos aproximadamente $\log N$ niveles. De hecho, el número de niveles es $1 + \lceil \log N \rceil$ (que vale 4 cuando N es igual a 8). Por tanto, el tiempo total de ejecución que esperamos es $O(N \log N)$.

Este análisis da una explicación intuitiva de por qué el tiempo de ejecución es $O(N \log N)$. Pero, en general, expandir un algoritmo recursivo para examinar su comportamiento es una mala idea, ya que viola la tercera regla de la recursión. A continuación presentaremos un tratamiento matemático más sistemático.

Sea $T(N)$ el tiempo que nos lleva resolver el problema de la obtención de la subsecuencia de suma máxima para un vector de tamaño N. Si $N = 1$, el programa tarda una cantidad de tiempo constante en ejecutar las líneas 13 a 14, la cual tomaremos como una unidad de tiempo. Luego $T(1) = 1$. En caso contrario, el programa debe hacer dos llamadas recursivas además del tiempo lineal que tarda en calcular la suma máxima para el caso 3. La sobrecarga constante es absorbida por el término $O(N)$. ¿Cuánto tardan las llamadas recursivas? Puesto que resuelven problemas de tamaño $N/2$, tenemos que cada una de ellas tarda $T(N/2)$ unidades de tiempo; por lo que el trabajo recursivo total es $2T(N/2)$. Esto genera las ecuaciones

$$T(1) = 1,$$
$$T(N) = 2T(N/2) + O(N).$$

Por supuesto, para que la segunda ecuación tenga sentido exacto, N debe ser una potencia de dos. En otro caso, en algún momento $N/2$ no será par. Una ecuación más precisa, válida para todo N, sería

$$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N).$$

No obstante, para simplificar los cálculos, supondremos que N es una potencia de 2 y reemplazamos el término $O(N)$ por N. Estas suposiciones son menores y no afectan al resultado O. En consecuencia, suponiendo que N es una potencia de 2, necesitamos obtener una solución explícita de $T(N)$ para

$$T(1) = 1,$$
$$T(N) = 2T(N/2) + N. \qquad \textbf{(7.6)}$$

Ésta es exactamente la ecuación que se ilustra en la Figura 7.19, luego ya sabemos que la respuesta ha de ser $N \log N + N$. Esto se puede ratificar fácilmente examinando unos pocos valores: $T(1) = 1$, $T(2) = 4$, $T(4) = 12$, $T(8) = 32$ y $T(16) = 80$. A continuación demostramos formalmente este resultado (Teorema 7.4) usando dos métodos diferentes.

Suponiendo que N es una potencia de 2, la solución a la ecuación $T(N) = 2T(N/2) + N$, con condición inicial $T(1) = 1$, es $T(N) = N \log N + N$. **Te**

Para N suficientemente grande, tenemos que $T(N/2) = 2T(N/4) + N/2$ por lo que podemos usar la Ecuación 7.6 con $N/2$ en lugar de N. En consecuencia, tenemos **De (M**

$$2T(N/2) = 4T(N/4) + N.$$

Sustituyendo esto en la Ecuación 7.6 obtenemos

$$T(N) = 4T(N/4) + 2N. \qquad \textbf{(7.7)}$$

Si usamos la Ecuación 7.6 para $N/4$ y multiplicamos por 4, obtenemos

$$4T(N/4) = 8T(N/8) + N,$$

que podemos sustituir en el lado derecho de la Ecuación 7.7 para obtener

$$T(N) = 8T(N/8) + 3N.$$

Continuando de esta forma, obtenemos

$$T(N) = 2^k T(N/2^k) + kN.$$

Finalmente, tomando $k = \log N$ (lo cual tiene sentido, puesto que $2^k = N$), obtenemos

$$T(N) = NT(1) + N \log N = N \log N + N.$$

Aunque este método de prueba parece funcionar bien, puede ser difícil aplicarlo en casos más complicados, pues tiende a producir ecuaciones muy largas. A continuación se presenta un segundo método que parece más sencillo, porque genera ecuaciones verticalmente, que se manipulan más fácilmente.

Dividimos la Ecuación 7.6 por N, generando una nueva ecuación básica: **D (M**

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1.$$

Esta ecuación es ahora válida para cualquier N que sea una potencia de 2, luego también podemos escribir las siguientes ecuaciones:

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1 \qquad (7.8)$$

$$\cdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

A continuación, sumamos todos los lados izquierdos de las Ecuaciones 7.8 y lo igualamos a la suma de todos los lados derechos. Observe que el término $T(N/2)/(N/2)$ aparece en ambos lados y por tanto se cancela. De hecho, prácticamente todos los términos aparecen en ambos lados y pueden cancelarse. Esto recibe el nombre de suma telescópica. Cuando se ha sumado y simplificado todo, el resultado final es

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

porque todos los demás términos se cancelan y hay $\log N$ *ecuaciones, de modo que todos los unos al final de esas ecuaciones se suman para producir* $\log N$. *Multiplicando por N en ambos lados, obtenemos la misma respuesta final que antes.*

Observe que si no dividiéramos por N al principio de la solución, la suma no sería telescópica. La decisión sobre la división necesaria para asegurar una suma telescópica requiere alguna experiencia y hace el método ligeramente más difícil de aplicar que la primera alternativa. Sin embargo, una vez ha encontrado el divisor correcto, la segunda alternativa tiende a generar un trabajo que cabe en una hoja de papel, dando lugar con ello a reducir los subsiguientes errores matemáticos. En contraste, el primer método es más una técnica de fuerza bruta.

Observe cuidadosamente que siempre que tenemos un algoritmo divide y vencerás que resuelve dos problemas de la mitad de tamaño con un coste lineal adicional, tendremos un tiempo de ejecución $O(N \log N)$. La siguiente sección examina lo que sucede en un entorno más general.

7.5.3 Una cota superior general para los tiempos de ejecución de los algoritmos divide y vencerás

En el análisis de la sección anterior veíamos que cuando se divide un problema en dos partes iguales, con un coste adicional $O(N)$, el resultado es un algoritmo con coste $O(N \log N)$. Pero, ¿qué pasa si dividimos el problema en tres partes del mismo tamaño con coste lineal adicional?, ¿o en siete partes con coste adicional cuadrático? (*Véase* el Ejercicio 7.17.) Esta sección proporciona una fórmula general para calcular el tiempo de ejecución de un algoritmo divide y vencerás. La fórmula requiere tres parámetros:

- A, que es el número de subproblemas.
- B, que es el factor de reducción del tamaño relativo de los subproblemas (por ejemplo, $B = 2$ representa que los subproblemas tienen la mitad del tamaño del problema original).
- k, que representa el hecho de que el coste adicional es $\Theta(N^k)$.

La demostración de la fórmula requiere cierta familiaridad con las sumas geométricas, pero no es necesario entender la demostración para utilizar la fórmula.

La solución de la ecuación $T(N) = AT(N/B) + O(N^k)$, *donde* $A \geq 1$ *y* $B > 1$, *es*

$$T(N) = \begin{cases} O(N^{\log_B A}) & si \quad A > B^k \\ O(N^k \log N) & si \quad A = B^k \\ O(N^k) & si \quad A < B^k \end{cases}.$$

Antes de demostrar el Teorema 7.5, veamos algunas aplicaciones del mismo. En el problema de la obtención de la subsecuencia de suma máxima, teníamos dos subproblemas, cada uno de ellos con la mitad de tamaño que el original, y un coste adicional lineal. Por tanto, los valores de los parámetros son $A = 2$, $B = 2$ y $k = 1$, y en consecuencia se aplica el segundo caso del Teorema 7.5, obteniendose así un valor $O(N \log N)$, que coincide con nuestros cálculos previos. Si, por ejemplo, resolvemos recursivamente tres problemas de la mitad del tamaño del problema original con un coste adicional lineal, tenemos que $A = 3$, $B = 2$ y $k = 1$, por lo que aplicaríamos la primera alternativa del teorema, obteniendo $O(N^{\log_2 3}) = O(N^{1.59})$. En este caso podemos observar que el coste adicional no contribuye al coste total del algoritmo. Cualquier sobrecarga menor que $O(N^{1.59})$ produciría el mismo tiempo de ejecución del algoritmo recursivo. Un algoritmo con tres subproblemas de la mitad de tamaño pero coste adicional cuadrático, tendría un tiempo de ejecución $O(N^2)$, pues se aplicaría el tercer caso. Ahora, la sobrecarga domina, una vez ésta excede el umbral $O(N^{1.59})$. En el umbral, la penalización es el factor logarítmico que aparece en el segundo caso. Pasemos ahora a demostrar el Teorema 7.5.

De forma similar a como hicimos en la prueba del Teorema 7.4, supondremos que N es una potencia de B. Por tanto, tendremos $N = B^M$. Entonces $N/B = B^{M-1}$ y $N^k = (B^M)^k = (B^k)^M$. Tomaremos $T(1) = 1$ e ignoramos el factor constante en $O(N^k)$. Entonces obtenemos la siguiente ecuación básica

$$T(B^M) = AT(B^{M-1}) + (B^k)^M.$$

Si dividimos en ambos lados por A^M, obtenemos la nueva ecuación básica

$$\frac{T(B^M)}{A^M} = \frac{T(B^{M-1})}{A^{M-1}} + \left(\frac{B^k}{A}\right)^M.$$

Si ahora escribimos esta ecuación para cada valor menor que M, obtenemos

$$\begin{aligned}
\frac{T(B^M)}{A^M} &= \frac{T(B^{M-1})}{A^{M-1}} + \left(\frac{B^k}{A}\right)^M \\
\frac{T(B^{M-1})}{A^{M-1}} &= \frac{T(B^{M-2})}{A^{M-2}} + \left(\frac{B^k}{A}\right)^{M-1} \\
\frac{T(B^{M-2})}{A^{M-2}} &= \frac{T(B^{M-3})}{A^{M-3}} + \left(\frac{B^k}{A}\right)^{M-2} \qquad (7.9) \\
&\cdots \\
\frac{T(B^1)}{A^1} &= \frac{T(B^0)}{A^0} + \left(\frac{B^k}{A}\right)^1
\end{aligned}$$

Si sumamos todas las ecuaciones de la Ecuación 7.9, sucede de nuevo que la mayoría de los términos del lado izquierdo se cancelan con los correspondientes en el lado derecho, produciendo

$$\begin{aligned}
\frac{T(B^M)}{A^M} &= 1 + \sum_{i=1}^{M} \left(\frac{B^k}{A}\right)^i \\
&= \sum_{i=0}^{M} \left(\frac{B^k}{A}\right)^i
\end{aligned}$$

de donde

$$T(N) = T(B^M) = A^M \sum_{i=0}^{M} \left(\frac{B^k}{A}\right)^i. \qquad (7.10)$$

Si $A > B^k$, entonces la suma es una serie geométrica con un radio menor que 1. Puesto que la suma de la correspondiente serie infinita converge a una constante, esta suma finita está también acotada por dicha constante, con lo que obtenemos la Ecuación 7.11.

$$T(N) = O(A^M) = O(N^{\log_B A}). \qquad (7.11)$$

Cuando $A = B^k$, cada término de la suma de la Ecuación 7.10 es 1. Puesto que la suma contiene $1 + \log_B N$ términos y $A = B^k$ implica que $A^M = N^k$, obtenemos

$$T(N) = O(A^M \log_B N) = O(N^k \log_B N) = O(N^k \log N).$$

Finalmente, si $A < B^k$, entonces los términos de la serie geométrica son mayores que 1 y podemos calcular la suma usando una fórmula estándar, obteniendo así

$$T(N) = A^M \frac{\left(\frac{B^k}{A}\right)^{M+1} - 1}{\frac{B^k}{A} - 1} = O\left(A^M \left(\frac{B^k}{A}\right)^M\right) = O((B^k)^M) = O(N^k)$$

lo cual demuestra el tercer caso del Teorema 7.5.

7.6 Programación dinámica

Todo problema que se puede expresar recursivamente en forma matemática también se puede expresar directamente como un algoritmo recursivo. En muchos casos, hacer esto produce una mejora significativa en el rendimiento con respecto a una búsqueda exhaustiva ingenua. Cualquier fórmula matemática recursiva se podría traducir directamente a un algoritmo recursivo, pero a menudo el compilador no hace justicia al algoritmo recursivo y se obtiene un programa ineficiente. Éste es el caso del cálculo de los números de Fibonacci descritos en la Sección 7.3.4. En tales casos, podemos rescribir el algoritmo recursivo como un algoritmo no recursivo que sistemáticamente guarda las respuestas de los subproblemas resueltos en una tabla. Una técnica que hace uso de esta estrategia es la *programación dinámica*. Esta técnica se ilustra con el siguiente problema:

PROBLEMA DEL CAMBIO DE MONEDAS

Para una divisa con monedas de C_1, C_2, ..., C_N (unidades), ¿cuál es el mínimo número de monedas que se necesitan para devolver K unidades de cambio?

La divisa española tiene monedas de 1, 5, 10 y 25 pesetas (ignoremos las monedas de 50 pesetas, por ser poco frecuentes). Podemos juntar 63 pesetas usando dos monedas de 25 pesetas, una moneda de 10 pesetas y tres de 1 peseta, lo que hace un total de 6 monedas. Cambiar monedas en esta divisa es sencillo: usamos repetidamente la moneda con mayor valor que tengamos disponible. Se puede demostrar que para la divisa española, esto siempre minimiza el número total de monedas usadas. Éste es un ejemplo de los llamados algoritmos devoradores, en los cuales se toma una decisión en cada fase que parece ser la apropiada, sin tener en cuenta futuras consecuencias. Esta estrategia de «coge todo lo que puedas ahora» es la fuente del nombre de esta clase de algoritmos. Resulta agradable que un problema se pueda resolver usando un algoritmo devorador, ya que entonces el

algoritmo se suele acercar bastante a nuestra intuición y además el código producido resulta comprensible. Desafortunadamente los algoritmos devoradores no siempre funcionan correctamente. Si la divisa española contuviera una moneda de 21 pesetas, entonces el algoritmo devorador seguiría dando una solución de seis monedas, cuando la solución óptima usa tres (de 21 pesetas).

La pregunta se convierte entonces en cómo resolver el problema para un conjunto de monedas arbitrario. Supongamos que siempre existe una moneda de 1 unidad, con lo que nos aseguramos de que el problema tiene siempre solución. Una estrategia simple para reunir K unidades consiste en usar la recursión como sigue:

1. Si con una sola moneda ya podemos devolver el cambio solicitado, ésta (1) es la cantidad mínima de monedas.
2. En caso contrario, para cada posible valor i podemos calcular de forma independiente el número mínimo de monedas que se necesitan para reunir i y $K - i$ unidades, y elegimos el i que minimice la suma de ambos.

Como ejemplo, una vez introducida la moneda de 21 unidades, veamos cómo podemos devolver 63 unidades de cambio. Claramente, una moneda no es suficiente. Podemos calcular independientemente el número de monedas requerido para devolver 1 unidad y 62 unidades de cambio (que son 1 y 4 respectivamente). Estos resultados se obtienen recursivamente, por lo que podemos asumirlos como óptimos (las 62 unidades se reúnen con dos monedas de 21 unidades y dos de 10 unidades). Obtenemos así una solución con cinco monedas. Si dividimos el problema en 2 y 61 unidades, las soluciones recursivas son respectivamente 2 y 4, dando un total de seis monedas. Continuamos probando todas las posibilidades, algunas de las cuales se muestran en la Figura 7.20. Eventualmente, llegamos a la división en 21 y 42 unidades, que se pueden obtener usando respectivamente 1 y 2 monedas, lo que hace un total de tres.

La última división probaría con 31 y 32 unidades. Podemos reunir 31 unidades con dos monedas y 32 con tres, lo que hace un total de cinco monedas. En consecuencia, el mínimo sigue siendo tres monedas.

De nuevo, cada uno de los subproblemas se está resolviendo recursivamente, lo cual genera de forma natural el algoritmo de la Figura 7.21. Si ejecutamos el algoritmo para cambiar cantidades pequeñas, funciona perfectamente, pero al igual que ocurría en el cálculo de los números de Fibonacci hay demasiado trabajo

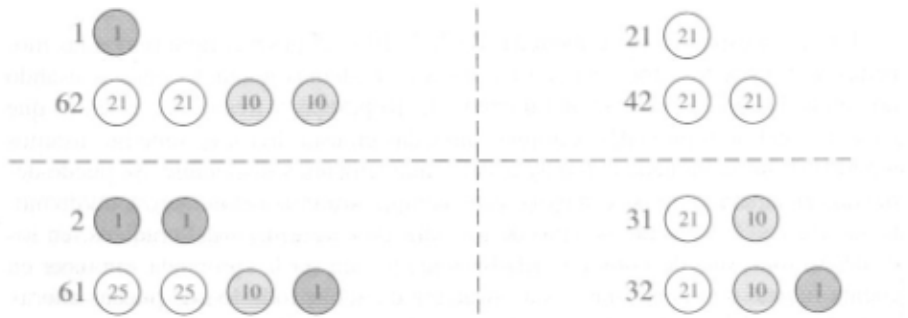


Figura 7.20 Algunos de los subproblemas resueltos recursivamente en la Figura 7.21.

```
1  // Devuelve el mínimo número de monedas para devolver un cambio
2  // Es un algoritmo recursivo simple muy ineficiente
3
4  public static int devolverCambio( int [ ] monedas, int cambio,
5                                    int monedasDistintas)
6  {
7      int minMonedas = cambio;
8
9          // Búsqueda de una moneda que reúna el cambio exacto
10     for( int i = 0; i < monedasDistintas; i++ )
11         if( monedas[i] == cambio )
12             return 1;
13
14         // No hay ninguna; se resuelve recursivamente
15     for( int j = 1; j <= cambio / 2; j++ )
16     {
17         int monedasActuales =
18           devolverCambio( monedas, j, monedasDistintas) +
19           devolverCambio( monedas, cambio - j, monedasDistintas );
20         if ( monedasActuales < minMonedas )
21             minMonedas = monedasActuales;
22     }
23
24     return minMonedas;
25 }
```

Figura 7.21 Método recursivo ineficiente para resolver el problema del cambio.

redundante. Como consecuencia, el algoritmo no terminará en una cantidad razonable de tiempo para el caso de las 63 unidades.

Un algoritmo alternativo consiste en reducir recursivamente el tamaño del problema especificando una primera moneda. Por ejemplo, para 63 unidades, podemos dar cambio de cada una de las formas siguientes, que se muestran en la Figura 7.22:

- Una moneda de 1 unidad más el número de monedas necesarios para cambiar 62 unidades, calculado de forma recursiva.
- Una moneda de 5 unidades más el número de monedas necesarios para cambiar 58 unidades, calculado de forma recursiva.
- Una moneda de 10 unidades más el número de monedas necesarios para cambiar 53 unidades, calculado de forma recursiva.
- Una moneda de 21 unidades más el número de monedas necesarios para cambiar 42 unidades, calculado de forma recursiva.
- Una moneda de 25 unidades más el número de monedas necesarios para cambiar 38 unidades, calculado de forma recursiva.

En lugar de generar 62 problemas recursivamente, como se hizo en la Figura 7.20, ahora solamente realizamos cinco llamadas recursivas, una por cada tipo distinto de moneda. A pesar de ello, una implementación recursiva ingenua seguiría siendo ineficiente porque repetiría cálculos. Por ejemplo, en el primer caso tenemos que resolver el problema de devolver 62 unidades de cambio. En este subproblema, una de las llamadas recursivas que se hacen elige una moneda de 10 unidades y resuelve recursivamente el problema para 52 unidades. Por otro lado, en el tercer caso tenemos que resolver el problema de devolver 53 unidades. Una

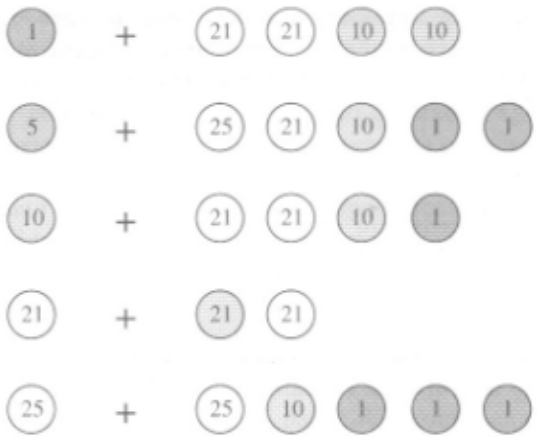


Figura 7.22 Algoritmo recursivo alternativo para el problema de la devolución de cambio.

de sus llamadas recursivas quita una unidad y resuelve recursivamente el problema para 52 unidades. Este trabajo redundante produce otra vez una gran cantidad de tiempo de ejecución. Sin embargo, si replanteamos radicalmente la forma de implementar el algoritmo, podemos hacer que el mismo sea razonablemente rápido.

El truco está en guardar las soluciones de los subproblemas en un vector. Esta técnica de programación dinámica forma la base de muchos algoritmos. Puesto que la solución a un problema grande depende solamente de las soluciones de problemas más pequeños, podemos ir calculando las soluciones óptimas correspondientes a cambiar 1 unidad, 2 unidades, 3 unidades y así sucesivamente. Esta estrategia se implementa en el programa de la Figura 7.23.

Comenzamos observando en la línea 11, que 0 unidades se pueden cambiar usando cero monedas. A continuación, a partir de la línea 13 vamos calculando el cambio óptimo para cada valor de `unidades` desde 1 hasta el valor solicitado `maxCambio`. El vector `ultimaMoneda` se usa para recordar cuál fue la última moneda usada para devolver el cambio óptimo en cada caso. Para calcular la forma óptima de devolver `unidades` unidades, probamos con cada tipo de moneda, como se indica en el bucle `for` de la línea 18. Si el valor de la moneda es mayor que la cantidad que queremos cambiar, entonces no hacemos nada. En caso contrario, en la línea 22 comprobamos si el número de monedas usado para resolver el subproblema restante más 1 (por la moneda actual) es menor que las usadas hasta el momento. Si es así, realizamos la actualización de `minMonedas` en las líneas 25 a 27. Cuando el bucle termina, en las líneas 30 y 31 se almacenan los valores obtenidos de `minMonedas` y `nuevaMoneda` en los vectores.

Al final del algoritmo, `monedasUsadas[maxCambio]` representa el mínimo número de monedas necesarias para cambiar `maxCambio` unidades. Volviendo hacia atrás a lo largo de `ultimaMoneda`, podemos averiguar las monedas necesarias para obtener la solución óptima. El tiempo de ejecución es el de los dos bucles `for` anidados, y por tanto es $O(NK)$, donde N es el número de monedas con distintos valores y K la cantidad (máxima) de cambio que queremos devolver.

```
1  // Algoritmo de programación dinámica para el problema de la devolución
2  // del cambio. Como resultado, el vector monedasUsadas se rellena con
3  // el mínimo número de monedas necesario para cambiar desde 0 hasta
4  // maxCambio unidades, y ultimaMoneda contiene una de las monedas
5  // necesarias para hacer el cambio.
6
7  public static void devolverCambio( int [ ] monedas,
8                                     int monedasDistintas, int maxCambio,
9                                     int [ ] monedasUsadas, int [ ] ultimaMoneda )
10 {
11     monedasUsadas[ 0 ] = 0; ultimaMoneda[ 0 ] = 1;
12
13     for( int unidades = 1; unidades <= maxCambio; unidades ++ )
14     {
15         int minMonedas = unidades;
16         int nuevaMoneda = 1;
17
18         for( int j = 0; j < monedasDistintas; j++ )
19         {
20             if( monedas[ j ] > unidades ) // No se puede usar la moneda j
21                 continue;
22             if( monedasUsadas[ unidades - monedas[ j ] ] + 1
23                                                     < minMonedas )
24             {
25                 minMonedas =
26                         monedasUsadas[ unidades - monedas[ j ] ] + 1;
27                 nuevaMoneda = monedas[ j ];
28             }
29         }
30         monedasUsadas[ unidades ] = minMonedas;
31         ultimaMoneda [ unidades ] = nuevaMoneda;
32     }
33 }
```

Figura 7.23 Algoritmo de programación dinámica para resolver el problema de la devolución de cambio que calcula el cambio óptimo para todas las cantidades desde 0 hasta `maxCambio` y mantiene información para construir la secuencia de monedas.

7.7 Algoritmos de vuelta atrás

Esta sección muestra la última aplicación de la recursión. Veremos, en particular, cómo escribir una rutina que haga que el computador seleccione un movimiento óptimo en el juego de las tres en raya. La clase `Mejor`, mostrada en la Figura 7.24, se usa para almacenar el movimiento óptimo devuelto por el algoritmo de selección. En la Figura 7.25 se muestra el esqueleto para una clase `TresEnRaya`. Esta clase tiene un atributo `tablero` que representa el estado actual del juego. Se especifican una serie de métodos, incluyendo rutinas para limpiar el tablero, para comprobar si una casilla está ocupada, para colocar una pieza en una casilla y para comprobar si alguno de los jugadores ha ganado. Los detalles de implementación se encuentran en el código correspondiente disponible en Internet.

```
1  final class Mejor
2  {
3      int fila;
4      int columna;
5      int valor;
6
7      public Mejor( int v )
8        { this( v, 0, 0 ); }
9      public Mejor( int v, int f, int c )
10       { valor = v; fila = f; columna = c; }
11 }
```

Figura 7.24 Clase para almacenar un movimiento evaluado.

La rutina interesante es la que decide, desde cualquier posición, cuál es el mejor movimiento. Ésta es la rutina `elegirMovimiento`. La estrategia general usa un algoritmo de vuelta atrás, es decir, un algoritmo que usa recursión para probar sistemáticamente todas las posibilidades.

La base para tomar esta decisión se encuentra en la rutina `estadoActual`, que se muestra en la Figura 7.26. Dicha rutina devuelve `HUMANO_GANA`, `COMPUTADORA_GANA`, `EMPATE` o `INCIERTO`, dependiendo de la posición sobre el tablero.

La estrategia que se usa habitualmente es la *estrategia minimax*, que considera que ambos jugadores juegan de forma óptima. El valor del estado es `COMPUTADORA_GANA` si mediante una jugada óptima el computador puede forzar una victoria. Si el computador puede forzar un empate pero no una victoria, el valor del estado es `EMPATE`; si la persona puede forzar su victoria el valor del estado es `HUMANO_GANA`. Puesto que queremos que el computador gane, consideraremos `HUMANO_GANA<EMPATE<COMPUTADORA_GANA`.

Para el computador, el valor del estado es el máximo de los valores de los estados que pueden resultar de hacer un movimiento. Supongamos que un movimiento lleva a un estado ganador, dos llevan a un empate y seis llevan a uno perdedor. Entonces el estado inicial es un estado ganador porque el computador puede conseguir la victoria, sin que su rival pueda evitarlo. El correspondiente movimiento que lleva al computador a ganar es el que ésta debería hacer. Para generar el comportamiento de la persona, obramos de forma análoga pero utilizando el mínimo en lugar del máximo.

Esto sugiere un algoritmo recursivo para determinar el valor de un estado. Una vez escrito éste, llevar cuenta del mejor movimiento es simple cuestión de almacenamiento. Si el estado es final (es decir, se ha obtenido tres en raya o el tablero está lleno), el valor del estado se obtiene de forma inmediata. En caso contrario, probamos recursivamente cada uno de los movimientos posibles, calculando el valor de cada estado resultante, y elegimos el máximo valor posible. La llamada recursiva requiere entonces que el jugador humano evalúe el estado. Para él, el valor del estado es el menor de los valores de los posibles estados siguientes, puesto que el jugador humano está tratando que el computador pierda. En consecuencia, el método recursivo `elegirMovimiento`, que se muestra en la Figura 7.27, toma un parámetro `lado`, que indica a quién le toca mover.

Las líneas 12 y 13 tratan el caso base de la recursión. Si tenemos una respuesta inmediata, podemos terminar. En caso contrario, en las líneas 15 a 22 determinamos los valores adecuados de un par de variables, atendiendo a qué jugador le toca

```
1  class TresEnRaya
2  {
3      public static final int HUMANO             = 0;
4      public static final int COMPUTADORA        = 1;
5      public static final int VACIA              = 2;
6
7      public static final int HUMANO_GANA        = 0;
8      public static final int EMPATE             = 1;
9      public static final int INCIERTO           = 2;
10     public static final int COMPUTADORA_GANA   = 3;
11
12         // Constructor
13     public TresEnRaya( )
14       { limpiaTablero( ); }
15
16         // Calcula el movimiento óptimo
17     public Mejor elegirMovimiento( int lado )
18       { /* Implementación en la Figura 7.27 */ }
19
20         // Calcula el valor estático del estado actual
21     private int valorEstado( )
22       { /* Implementación en la Figura 7.26 */ }
23
24         // Hace un movimiento, comprobando su legalidad
25     public boolean movimientoJuego( int lado, int fila, int columna)
26       { /* Implementación en el código disponible en Internet */ }
27
28         // Limpia el tablero de juego
29     public void limpiaTablero( )
30       { /* Implementación en el código disponible en Internet */ }
31
32         // Devuelve true si el tablero está lleno
33     public boolean tableroLleno( )
34       { /* Implementación en el código disponible en Internet */ }
35
36         // Devuelve true si algun jugador ha ganado
37     boolean hayGanador( int lado )
38       { /* Implementación en el código disponible en Internet */ }
39
40         // Hace un movimiento, posiblemente liberando una casilla
41     private void lugar( int fila, int columna, int ficha )
42       { tablero[ fila ][ columna ] = ficha; }
43
44         // Comprueba si una casilla está vacía
45     private boolean casillaVacia( int fila, int columna )
46       { return tablero[ fila ][ columna ] = VACIA; }
47
48     private int [ ] [ ] tablero = new int[ 3 ][ 3 ];
49 }
```

Figura 7.25 Esqueleto de la clase `TresEnRaya`.

mover. El código de las líneas 28 a 38 se ejecuta una vez para cada movimiento disponible. Intentamos mover en la línea 28, evaluamos recursivamente el movimiento en la línea 29 (guardando el valor obtenido), y finalmente deshacemos el movimiento en la línea 30. Las líneas 33 y 34 comprueban si éste es el mejor movimiento hasta la fecha. Si es así, ajustamos `valor` en la línea 36 y guardamos el movimiento en la línea 37. En la línea 41, devolvemos el valor del estado en un objeto `Mejor`.

```
    // Calcula el valor estático del estado actual
private int valorEstado( )
{
    return hayGanador( COMPUTADORA ) ? COMPUTADORA_GANA :
           hayGanador( HUMANO )      ? HUMANO_GANA :
           tableroLleno( )           ? EMPATE : INCIERTO;
}
```

Figura 7.26 Rutina para obtener el valor de un estado.

```
    // Calcula el movimiento óptimo
public Mejor elegirMovimiento( int lado )
{
    int op;            // El lado opuesto
    Mejor replica;     // Mejor réplica del oponente
    int evalSimple;    // Resultado de una evaluación simple
    Estado estadoActual = new Estado( tablero );
    int mejorFila = 0;
    int mejorColumna = 0;
    int valor;

    if( ( evalSimple = valorEstado( ) ) != INCIERTO )
        return new Mejor( evalSimple );

    if( lado == COMPUTADORA )
    {
        op = HUMANO; valor = HUMANO_GANA;
    }
    else
    {
        op = COMPUTADORA; valor = COMPUTADORA_GANA;
    }

    for( int fila = 0; fila < 3; fila++ )
        for( int columna = 0; columna < 3; columna++ )
            if( casillaVacia( fila, columna ) )
            {
                lugar( fila, columna, lado );       // Intenta mover
                replica = elegirMovimiento( op ); // Evalúa
                lugar( fila, columna, VACIA ); // Deshace el movimiento

                    // Actualiza si un jugador obtiene el mejor estado
                if( lado == COMPUTADORA && replica.valor > valor
                   || lado == HUMANO && replica.valor < valor )
                {
                    valor = replica.valor;
                    mejorFila = fila; mejorColumna = columna;
                }
            }

    return new Mejor( valor, mejorFila, mejorColumna );
}
```

Figura 7.27 Rutina recursiva para encontrar un movimiento óptimo en las tres en raya.

Aunque la Figura 7.27 resuelve de forma óptima las tres en raya, lleva a cabo excesivas búsquedas. En concreto, para elegir el primer movimiento en un tablero vacío, hace 549.946 llamadas recursivas (este número se obtiene ejecutando el programa). Usando algunos trucos algorítmicos, podemos calcular exactamente la misma información usando menos búsquedas. Uno de esos trucos se conoce como *poda alfa-beta*, y se describe en detalle en el Capítulo 10. La aplicación de la poda alfa-beta reduce el número de llamadas recursivas a sólo 18.297.

Resumen

Este capítulo ha examinado la recursión y ha demostrado que es una herramienta potente de resolución de problemas. A continuación aparecen las reglas fundamentales de la recursión, que uno nunca debería olvidar:

1. *Casos base*: se debe tener siempre al menos un caso base que pueda resolverse sin recursión.
2. *Progreso*: cualquier llamada recursiva debe progresar hacia un caso base.
3. *«Puede creerlo»:* suponga siempre que las llamadas recursivas internas funcionan correctamente.
4. *Regla de interés compuesto*: nunca duplique trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

La recursión tiene muchos usos, algunos de los cuales han sido discutidos en este capítulo. Se han estudiado tres importantes técnicas de diseño de algoritmos basadas en la recursión: divide y vencerás, programación dinámica y vuelta atrás.

El siguiente capítulo estudia el problema de la ordenación. El algoritmo de ordenación más rápido que se conoce es recursivo.

Elementos del juego

algoritmo de vuelta atrás Algoritmo que usa la recursión para probar sistemáticamente todas las posibilidades.

algoritmo devorador Algoritmo que toma decisiones locales óptimas en cada paso. Escribirlo suele ser simple. Sin embargo, no siempre funciona correctamente.

algoritmo divide y vencerás Tipo de algoritmo recursivo generalmente muy eficiente. La recursión es la parte «divide» y la combinación de las soluciones recursivas es el «vencerás».

caso base Instancia que se puede resolver sin hacer llamadas recursivas. Toda llamada recursiva debe hacer progresos hacia un caso base. También, en una prueba por inducción, es el caso que se puede demostrar directamente.

criptografía de clave pública Tipo de criptografía en la que cada emisor publica el código que otros pueden usar para enviar a dicho emisor mensajes codificados, pero guarda el código secreto de decodificación.

encriptación Esquema de codificación usado en la transmisión de mensajes para que no puedan ser leídos por otras personas.

estrategia minimax Estrategia usada para las tres en raya y otros juegos estratégicos que supone que ambos jugadores juegan de forma óptima.

hipótesis de inducción Hipótesis consistente en que un teorema es cierto para un caso arbitrario de partida. Se usa en las pruebas por inducción probando que a partir de dicha hipótesis se puede demostrar que el siguiente caso también es cierto.

inducción Técnica de demostración usada para demostrar teoremas que se cumplen para los enteros positivos.

inverso multiplicativo Solución $1 \leqslant X < N$ a la ecuación $AX \equiv 1 (\mathrm{mod}\ N)$.

máximo común divisor (mcd) El máximo común divisor de dos enteros es el mayor entero que los divide a ambos.

método recursivo Método que, directa o indirectamente, se hace una llamada a sí mismo.

números de Fibonacci Secuencia de números en la que el número *i*-ésimo es la suma de los dos anteriores. Los números de Fibonacci aparecen repetidamente en la literatura.

poda alfa-beta Mejora del algoritmo minimax.

programación dinámica Técnica que evita la explosión recursiva guardando respuestas en una tabla.

registro de activación Método usado para realizar las tareas de bajo nivel en muchos lenguajes de programación. Se suele usar una pila de registros de activación.

reglas de la recursión 1. *Casos base*: se debe tener siempre al menos un caso base que pueda resolverse sin llamadas recursivas. 2. *Progreso*: cualquier llamada recursiva debe progresar hacia un caso base. 3. *«Puede creerlo»:* suponga siempre que las llamadas recursivas funcionan correctamente. 4. *Regla de interés compuesto*: nunca duplique trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

rutina guía Rutina que comprueba la validez de los argumentos de partida y llama después a la rutina recursiva.

sistema de criptografía RSA Método popular de encriptación.

suma telescópica Desarrollos en los que se genera una gran cantidad de términos que luego se cancelan.

Errores comunes

1. El error más común cuando se usa la recursión es olvidar el caso base.
2. Asegúrese de que cada llamada recursiva hace progresos hacia un caso base, pues en caso contrario, la recursión no terminará.
3. Se debe evitar la superposición de llamadas recursivas, pues tiende a generar algoritmos exponenciales.
4. Usar la recursión en lugar de un simple bucle es un mal estilo de programación.
5. Los algoritmos recursivos se analizan usando fórmulas recursivas. No suponga alegremente que una llamada recursiva requiere un tiempo lineal.
6. Violar los derechos de propiedad (*copyright*) es también un error. RSA está patentado, aunque se permiten algunos usos libres. Consulte la bibliografía para una mayor información al respecto.

En Internet

Se proporciona todo el código que aparece en el capítulo, incluyendo un programa para el juego de las tres en raya. En el Capítulo 10 se discute un versión mejorada de este juego que usa estructuras de datos más sofisticados. Todas las rutinas están en el directorio **Chapter07,** salvo cuando se indique explícitamente otra cosa. A continuación se enumeran los nombres de los ficheros:

RecSum.java	Incluye la rutina de la Figura 7.1, con un `main` simple.
PrintInt.java	Incluye la rutina traducida en la Figura 7.4 por `ImprimeEntero`, la cual se usa para imprimir un número en cualquier base, y además una función `main`.
Factorial.java	Incluye la traducción de la rutina de la Figura 7.8 y una función `main` simple.
BinarySearch.java	Incluye la rutina traducida en la Figura 7.9 por `busquedaBinaria` y una función `main`.
Ruler.java	Incluye la rutina traducida en la Figura 7.11 por `pintaRegla`, lista para ser ejecutada. Contiene también código que ralentiza el proceso de dibujo.
FractalStar.java	Incluye la rutina traducida en la Figura 7.13 por `pintaFractal`, lista para ejecutarse. Contiene también código que ralentiza el proceso de dibujo.
Numerical.java	Incluye la versión inglesa de rutinas matemáticas de la Sección 7.4, junto con una rutina para comprobar si un número es primo, y una función `main`; todas ellas en el directorio **Supporting** (traducido por `Soporte`).
RSA.java	Incluye el algoritmo RSA.
MaxSum.java	Incluye las cuatro rutinas del problema de la obtención de la subsecuencia de suma máxima (en el directorio**Chapter05**).
MkChange.java	Incluye la rutina traducida en la Figura 7.23 por `devolverCambio` y una función `main` simple.
Best.java	Clase usada en el algoritmo de las tres en raya, aquí traducida por `Mejor`.
TicTacToe.java	Algoritmo de tres en raya básico (traducido aquí por `TresEnRaya`).
TicTacMain.java	Simple interfaz gráfica de usuario para el programa de las tres en raya.

Ejercicios

Cuestiones breves

7.1. ¿Cuáles son las cuatro reglas fundamentales de la recursión?

7.2. Modifique el programa de la Figura 7.1 de forma que si `n` es negativo, se devuelva cero. Haga el menor número de cambios.

7.3. A continuación aparecen cuatro alternativas para la línea 10 de la rutina `potencia`. Diga por qué cada una de ellas es errónea.

```
long tmp = potencia(x * x, n/2, p);
long tmp = potencia(potencia(x, 2, p), n/2, p);
long tmp = potencia(potencia(x, n/2, p), 2, p);
long tmp = potencia(x, n/2, p) * potencia(x, n/2, p) % p;
```

7.4. Muestre cómo se procesan las llamadas recursivas en el cálculo de $2^{63} \bmod 37$.

7.5. Calcule *mcd*(1995, 1492).

7.6. Roberto elige p y q iguales a 37 y 41, respectivamente. Determine valores aceptables para los parámetros restantes del algoritmo RSA.

7.7. Muestre que el algoritmo devorador para la devolución del cambio fallaría si las monedas de 5 pesetas no formaran parte de la divisa española.

Problemas teóricos

7.8. Demuestre por inducción la fórmula siguiente para F_N:

$$F_N = \frac{1}{\sqrt{5}}\left(\left(\frac{(1+\sqrt{5})}{2}\right)^N - \left(\frac{1-\sqrt{5}}{2}\right)^N\right)$$

7.9. Demuestre las siguientes identidades relativas a los números de Fibonacci:

a) $F_1 + F_2 + \ldots + F_N = F_{N+2} - 1$

b) $F_1 + F_3 + \ldots + F_{2N-1} = F_{2N}$

c) $F_0 + F_2 + \ldots + F_{2N} = F_{2N+1} - 1$

d) $F_{N-1}F_{N+1} = (-1)^N + F_N^2$

e) $F_1F_2 + F_2F_3 + \ldots + F_{2N-1}F_{2N} = F_{2N}^2$

f) $F_1F_2 + F_2F_3 + \ldots + F_{2N}F_{2N+1} = F_{2N+1}^2 - 1$

g) $F_N^2 + F_{N+1}^2 = F_{2N+1}$

7.10. Demuestre que si $A \equiv B \pmod N$, entonces para cualesquiera C, D y P, se cumple:

a) $A + C \equiv B + C \pmod N$

b) $AD \equiv BD \pmod N$

c) $A^P \equiv B^P \pmod N$

7.11. Demuestre que si $A \geq B$, entonces $A \bmod B < A/2$. (*Pista*: considere por separado los casos $B \leq A/2$ y $B > A/2$.) ¿Cómo nos ayuda esto a demostrar que el tiempo de ejecución de `mcd` es logarítmico?

7.12. Demuestre por inducción la fórmula que define el número de llamadas del método recursivo `fib` de la Sección 7.3.4.

7.13. Demuestre por inducción que si $A > B \geq 0$ y la llamada a `mcd(a, b)` lleva a cabo $k \geq 1$ llamadas recursivas, entonces $A \geq F_{k+2}$ y $B \geq F_{k+1}$.

7.14. Demuestre por inducción que en el algoritmo extendido del *mcd*, $|X| < B$ y $|Y| < A$.

7.15. Escriba un algoritmo alternativo para el cálculo del *mcd* basado en las siguientes observaciones (supuesto que $A > B$):

a) $mcd(A, B) = 2\, mcd(A/2, B/2)$ si A y B son pares.

b) $mcd(A, B) = mcd(A/2, B)$ si A es par y B impar.

c) $mcd(A, B) = mcd(A, B/2)$ si A es impar y B es par.

d) $mcd(A, B) = mcd((A + B)/2, (A - B)/2)$ si A y B son impares.

7.16. Resuelva la siguiente ecuación, suponiendo que $A \geq 1$, $B > 1$ y $P \geq 0$.

$$T(N) = AT(N/B) + O(N^k \log^P N)$$

7.17. El algoritmo de Strassen para multiplicar matrices lleva a cabo la multiplicación de dos matrices $N \times N$ a través de siete llamadas recursivas que calculan el producto de dos matrices $N/2 \times N/2$. El coste adicional es cuadrático. ¿Cuál es el tiempo de ejecución del algoritmo de Strassen?

Problemas prácticos

7.18. El método `imprimeEntero` de la Figura 7.4 trata de forma incorrecta el caso en el que `N= Integer.MIN_VALUE`. Explique por qué y proporcione una solución.

7.19. Escriba un método recursivo que devuelva el número de unos en la representación binaria de N. Use el hecho de que es igual al número de unos en la representación binaria de $N/2$, más 1 si N es impar.

7.20. Implemente de forma recursiva la búsqueda binaria con una comparación por nivel.

7.21. El algoritmo de obtención de la subsecuencia de suma máxima de la Figura 7.18 no devuelve la subsecuencia en cuestión, ni ningún otro valor a partir del cual podamos obtenerla fácilmente. Modifíquelo de forma que devuelva en un único objeto el valor de la subsecuencia máxima y los índices de la subsecuencia (de forma similar a como se hizo en `elegirMovimiento`). Después haga que la rutina guía determine los atributos estáticos, igual que en la Sección 5.3.

7.22. Proporcione un algoritmo para el problema del cambio que calcule el número de formas distintas de devolver K unidades de cambio.

7.23. El problema del *subconjunto suma* es el siguiente: dado un conjunto de N enteros, $A_1, A_2, \ldots, A_N$ y un entero K, determinar si hay un subconjunto de enteros que sumen exactamente K. Proporcione un algoritmo $O(NK)$ para resolver el problema.

7.24. Proporcione un algoritmo $O(2^N)$ para el problema del subconjunto suma descrito en el Ejercicio 7.23. *Pista*: use recursión.

7.25. Escriba sendas rutinas con las siguientes declaraciones:

```
public void permutacion( String str );
private void permutacion( char [ ] str, int inf, int sup );
```

La primera rutina es una guía que llama a la segunda e imprime todas las permutaciones de los caracteres del `String str`. Si `str` fuera `"abc"`, entonces las cadenas de salida serían `abc`, `acb`, `bca`, `cab` y `cba`. Use recursión en la segunda rutina.

7.26. Escriba un programa que calcule el máximo valor de N que puede manejar en su máquina el programa de la Figura 7.1. *Pista*: use la idea de la búsqueda binaria para determinar N; el mecanismo para decidir si una llamada a `s` tiene éxito consiste en comprobar si se ha lanzado alguna excepción de la clase `Throwable`.

7.27. Escriba un programa que dibuje la Figura 7.7.

7.28. Explique lo que sucede si en la Figura 7.13 dibujamos el cuadrado central antes de hacer las llamadas recursivas.

Prácticas de programación

7.29. Los coeficientes binomiales $C(N, k)$ se pueden definir recursivamente en la siguiente forma:

$$C(N, 0) = 1,\ C(N, N) = 1$$

y para

$$0 < k < N,\ C(N, k) = C(N - 1, k) + C(N - 1, k - 1)$$

Escriba un método para calcular los coeficientes binomiales y proporcione un análisis de su tiempo de ejecución. Escriba el método usando:

a) Recursión.
b) Programación dinámica.

7.30. Implemente el sistema de criptografía RSA usando una clase de librería `BigDecimal`.

7.31. Mejore la implementación de las tres en raya haciendo las rutinas de soporte más eficientes y la interfaz gráfica de usuario más elaborada que la que se proporciona en Internet.

7.32. Sea A una secuencia de N números distintos ordenados $A_1, A_2, \ldots, A_N$ con $A_1 = 0$. Sea B una secuencia de $N(N - 1)/2$ números, definidos por $B_{i,j} = A_i - A_j\ (i < j)$. Sea D la secuencia resultante de la ordenación de B. Tanto B como D pueden contener elementos repetidos. Por ejemplo, podríamos tener $A = 0, 1, 5, 8$, y entonces $D = 1, 3, 4, 5, 7, 8$. Haga lo siguiente:

a) Escriba un programa que construya D a partir de A. Esta parte es fácil.
b) Escriba un programa que construya una secuencia A correspondiente a D. Observe que A no es única. Use un algoritmo de vuelta atrás.

7.33. Considere una cuadrícula $N \times N$ en la que algunas casillas están ocupadas por círculos negros. Dos casillas ocupadas pertenecen al mismo grupo si comparten un lado común. En la Figura 7.28, hay un grupo de cuatro casillas ocupadas, tres grupos de dos casillas ocupadas y dos casillas individuales ocupadas. Supongamos que la cuadrícula se representa mediante un vector bidimensional. Escriba un programa que:

a) Calcule el tamaño del grupo al que pertenece una casilla ocupada.
b) Calcule el número de grupos distintos.
c) Liste todos los grupos.

7.34. Escriba un programa que expanda las directivas `#include` de los ficheros fuente escritos en C o C++. Para ello reemplace líneas de la forma

```
#include "nombrefichero"
```

por el contenido de `nombrefichero`.

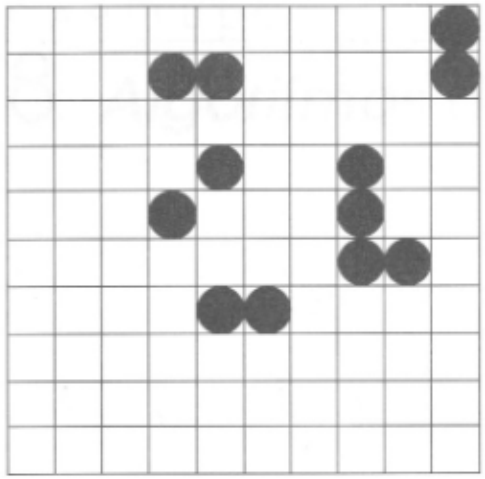


Figura 7.28 Cuadrícula del Ejercicio 7.33.

Bibliografía

Una gran parte de este capítulo está basada en [4]. En [1] se puede encontrar una descripción del algoritmo RSA junto con una prueba de corrección, y además un capítulo dedicado a la programación dinámica. El algoritmo RSA está patentado, y su uso comercial requiere el pago de una licencia; pero existe una implementación llamada *PGP (pretty good privacy)* que está disponible y cuyo uso no comercial está permitido sin necesidad de pago. En [3] se pueden encontrar más detalles sobre el particular. Los ejemplos de elaboración de los dibujos se han adaptado de [2].

1. T. H. Cormen, C. E. Leiserson, y R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Mass. (1990).
2. R. Sedgewick, *Algorithms in C++*, Addison-Wesley, Reading, Mass. (1992).
3. W. Stallings, *Protect Your Privacy: A Guide for PGP Users*, Prentice-Hall, Englewood Cliffs, NJ (1995).
4. M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1995).

CAPÍTULO

8 Algoritmos de ordenación

La ordenación es una aplicación fundamental en computación. La mayoría de los datos producidos por un programa están ordenados de alguna manera, y muchos de los cómputos son eficientes porque invocan internamente a un método de ordenación. En consecuencia, la ordenación es muy probablemente la operación más importante y mejor estudiada en computación.

Este capítulo estudia el problema de ordenar un vector de elementos, describiendo y analizando varios algoritmos de ordenación. Las ordenaciones que estudiaremos en este capítulo pueden realizarse enteramente en memoria principal, en la medida en que el número de elementos a ordenar sea moderadamente pequeño (del orden de unos pocos millones). Son también importantes las ordenaciones que no pueden realizarse en memoria principal y deben ser realizadas en disco o cinta. Este tipo de ordenación, denominada *ordenación externa*, se estudia en la Sección 20.6.

La discusión sobre la ordenación que aquí se presenta es una mezcla entre teoría y práctica. Incluye varios algoritmos con distintos rendimientos y muestra cómo el análisis del rendimiento de un algoritmo puede llevarnos a concebir una implementación que diste bastante de ser obvia.

En este capítulo veremos:

- Que la ordenación por inserción, mostrada ya en la Figura 4.18, se ejecuta en tiempo cuadrático.
- Cómo implementar Shellsort (la ordenación de Shell), un método de ordenación simple y eficiente, con un tiempo de ejecución subcuadrático.
- Cómo escribir los algoritmos un poco más complicados de quicksort y mergesort, que tienen un tiempo de ejecución en $O(N \log N)$.
- Que un algoritmo de ordenación de propósito general requiere $\Omega(N \log N)$ comparaciones.

8.1 ¿Por qué es importante la ordenación?

En la Sección 5.6 vimos cómo buscar en un vector ordenado es más sencillo que buscar en uno que no lo esté. Esto no es sólo cierto para algoritmos a ser ejecutados por máquinas, sino también para el proceder cotidiano de las personas. Por

ejemplo, encontrar el nombre de alguien en el listín telefónico es fácil, mientras que buscar un número de teléfono sin conocer el nombre de la persona a la que corresponde es, en la práctica, una tarea casi imposible. Como consecuencia, mucha de la información devuelta por un cómputo se presenta de forma ordenada de modo que su manejo resulte mucho más sencillo. Algunos ejemplos son:

- Las palabras de un diccionario están ordenadas (ignorando la distinción entre mayúsculas y minúsculas).
- Los ficheros de un directorio generalmente se muestran ordenados.
- El índice de un libro está ordenado (ignorando la distinción entre mayúsculas y minúsculas).
- El catálogo de una biblioteca está ordenado por autor y por título.
- El listado de cursos ofrecidos por una universidad se ofrece ordenado, primero por departamentos y después por número de curso, por ejemplo.
- Muchos bancos ofrecen estados de cuentas donde los cheques están ordenados por su número.
- En un periódico, el calendario de eventos está ordenado por fecha.
- Los discos compactos musicales están ordenados en una tienda de discos por el nombre del artista.
- En los programas impresos para ceremonias de graduación, los departamentos se listan en orden y los estudiantes de cada departamento también se muestran ordenados.

No es sorprendente por tanto, que gran parte del trabajo realizado en computación involucre la ordenación. Además, hay también usos indirectos de la ordenación. Por ejemplo, suponga que quisiéramos averiguar si en un vector existen elementos duplicados. La Figura 8.1 muestra un método sencillo del cual se ve inmediatamente que en el caso peor requiere un tiempo cuadrático. La ordenación nos proporciona un método alternativo: si ordenamos una copia del vector, los elementos duplicados aparecerán juntos y podrán ser detectados mediante un recorrido secuencial del vector. El tiempo de este algoritmo está dominado por la ordenación, por lo que si podemos ordenar en un tiempo subcuadrático, tendremos un algoritmo mejor que el de partida. Hay muchos otros algoritmos cuyo tiempo de ejecución se ve mejorado de forma significativa ordenando los datos inicialmente.

Una gran mayoría de los proyectos de programación utilizan la ordenación en algún momento, y en muchos casos, el coste de la ordenación determina su coste global. Por tanto, queremos ser capaces de implementar la ordenación de forma rápida y segura.

```
1  // Devuelve true si el vector contiene duplicados; false,
     en otro caso
2
3  public static boolean duplicados( Object [ ] a )
4  {
5      for( int i = 0; i < a.length; i++ )
6          for( int j = i + 1; j < a.length; j++ )
7              if( a[ i ].equals( a[ j ] ) )
8                  return true;   // Encontrado un duplicado
9
10         return false;          // No hay duplicados
11 }
```

Figura 8.1 Algoritmo simple de coste cuadrático para detectar duplicados.

8.2 Preliminares

Todos los algoritmos descritos en este capítulo son intercambiables. Todos reciben un vector que contiene los elementos, y sólo pueden ser ordenados objetos que implementen el interfaz `Comparable`. Por tanto, las únicas operaciones disponibles para obtener información sobre el orden relativo de los elementos son `menorQue` y `compara`. Los algoritmos que ordenan bajo estas condiciones se denominan *algoritmos de ordenación por comparación*. Todos los algoritmos de ordenación que veremos son métodos estáticos de la clase `Ordenacion` (en el paquete `EstructurasDatos`). En este capítulo, N es el número de elementos a ordenar.

8.3 Análisis de la ordenación por inserción y otras ordenaciones simples

El método de ordenación más simple, denominado *ordenación por inserción*, se discutió en la Sección 4.3.1. La implementación en la Figura 8.2 repite la presentada en la Figura 4.18. Debido a los bucles anidados, cada uno de los cuales puede realizar N iteraciones, el algoritmo de ordenación por inserción es $O(N^2)$. Es más, esta cota es alcanzable, ya que un vector de entrada ordenador en orden inverso requiere en efecto un tiempo cuadrático. Un cálculo preciso muestra que el test de la línea 10 puede ejecutarse, como mucho, $P + 1$ veces, por cada valor de P. Sumando sobre todos los posibles P, obtenemos un total de

$$\Sigma_{P=1}^{N-1} (P + 1) = 2 + 3 + 4 + \cdots + N = \Theta(N^2).$$

Por otro lado, si el vector de entrada ya viene ordenado, el tiempo de ejecución es $O(N)$ ya que el test del bucle `for` más interno falla inmediatamente. De hecho, si el vector de entrada está casi ordenado (el término *casi ordenado* será definido formalmente en breve), la ordenación por inserción se ejecutará rápidamente. Esto significa que el tiempo de ejecución no depende exclusivamente de la cantidad de

```
1  // ordenacionPorInsercion: ordena el vector a
2
3  public static void ordenacionPorInsercion( Comparable [ ] a )
4  {
5      for( int p = 1; p < a.length; p++ )
6      {
7          Comparable tmp = a[ p ];
8          int j = p;
9
10         for( ; j > 0 && tmp.menorQue( a[ j - 1 ] ); j-- )
11             a[ j ] = a[ j - 1 ];
12         a[ j ] = tmp;
13     }
14 }
```

Figura 8.2 Ordenación genérica por inserción.

datos, sino también del orden específico de estos datos. Debido a esta gran variación, merece la pena analizar el comportamiento en el caso medio. Resulta que el tiempo medio de la ordenación por inserción, al igual que el de otros algoritmos simples de ordenación, es $\Theta(N^2)$.

Una *inversión* en un vector de números es todo par (i, j) con $i < j$ y $A_i > A_j$. Por ejemplo, la secuencia {8, 5, 9, 2, 6, 3} contiene diez inversiones que corresponden a los pares (8, 5), (8, 2), (8, 6), (8, 3), (5, 2), (5, 3), (9, 2), (9, 6), (9, 3) y (6, 3). Puede observarse que el número de inversiones del vector de partida es igual al número de veces que se ejecuta la línea 11 de la Figura 8.2 al ordenarlo. Esto es así pues el efecto de la instrucción de asignación es intercambiar los elementos `a[j]` y `a[j-1]`. (En realidad, evitamos tantos intercambios utilizando una variable auxiliar, pero aún así se trata de un intercambio abstracto.) Intercambiar dos elementos que no están bien colocados elimina exactamente una inversión, y un vector ordenado no contiene ninguna inversión. Por tanto, si al principio del algoritmo tenemos I inversiones, debemos realizar I intercambios implícitos. Ya que el resto del trabajo realizado por el algoritmo es $O(N)$, el tiempo de ejecución de la ordenación por inserción es $O(I+N)$, donde I es el número de inversiones en el vector original. Luego el algoritmo de ordenación por inserción se ejecuta en tiempo lineal si el número de inversiones es $O(N)$.

Podemos calcular cotas precisas del tiempo de ejecución en el caso medio del algoritmo de ordenación por inserción, calculando el número medio de inversiones en el vector. Definir correctamente lo que significa «en el caso medio» no es fácil. Asumiremos que no hay elementos repetidos (si permitimos repeticiones, no está claro lo que significaría el número medio de repeticiones). Entonces podemos suponer, sin perdida de generalidad, que los datos de entrada son una cierta reorganización de los N primeros números naturales (ya que sólo el orden relativo entre ellos importa); estas reorganizaciones se denominan *permutaciones*. Podemos razonablemente asumir que todas estas permutaciones son equiprobables. Bajo estas hipótesis, podemos establecer el Teorema 8.1.

> *El número medio de inversiones de un vector de N números distintos es* $N(N-1)/4$.
>
> *Para cada vector de números A, denotemos por* A_r *el vector en orden inverso. Por ejemplo, el inverso del vector (1, 5, 4, 2, 6, 3) es (3, 6, 2, 4, 5, 1). Consideremos cualquier par de números* (x, y) *del vector, con* $y > x$. *Exactamente en uno de los vectores A o* A_r, *este par representa una inversión. El número total de pares en A y* A_r *es* $N(N-1)/2$. *Por tanto, en promedio tendremos la mitad de este número de inversiones, es decir,* $N(N-1)/4$.

Del Teorema 8.1 se deduce que la ordenación por inserción es cuadrática en promedio. El resultado también puede ser utilizado para proporcionar una cota inferior muy fuerte del coste de todo algoritmo de ordenación que sólo intercambie elementos adyacentes. Esto se expresa en el Teorema 8.2.

> *Todo algoritmo que ordene mediante intercambio de elementos adyacentes requiere, en promedio, un tiempo* $\Omega(N^2)$.
>
> *El número medio de inversiones del vector de partida es* $N(N-1)/4$. *Cada intercambio elimina exactamente una inversión, por lo que el número de intercambios necesarios es* $\Omega(N^2)$.

Éste es un ejemplo de *cálculo de una cota inferior*. Es válida no sólo para la ordenación por inserción, que realiza implícitamente intercambios de elementos adyacentes, sino también para otros algoritmos simples (no descritos aquí), como la ordenación por el método de la burbuja o la ordenación por selección. Más en general, es válida para toda una *clase* de algoritmos, incluyendo algunos que no se habrán descubierto aún: todos aquéllos que realizan sólo intercambios de elementos adyacentes.

Observe que la obtención experimental de resultados válidos para toda una clase de algoritmos requerirá, en general, la ejecución de todos los algoritmos de la clase. Esto no es posible en cuanto el número de posibles algoritmos sea infinito. De modo que cualquier resultado obtenido experimentalmente sólo se aplicaría a aquellos algoritmos comprobados. Esto hace que la confirmación experimental de la validez de una hipotética cota inferior sea mucho más difícil que la confirmación de la validez de una cota superior de un único algoritmo, a la que estamos acostumbrados. Un cómputo sólo podría refutar una conjetura sobre una cota inferior; pero, en general, nunca podría probarla.

Aunque la demostración de la cota inferior arriba obtenida es bastante simple, en general, probar una cota inferior es bastante más complicado que probar una cota superior. Los argumentos para probar una cota inferior son bastante más abstractos que sus contrapartidas para las cotas superiores.

La cota inferior obtenida nos indica que para que un algoritmo de ordenación sea subcuadrático, esto es, esté en $O(N^2)$, debe hacer comparaciones y, en particular, intercambiar elementos, que estén alejados entre sí. Un algoritmo de ordenación progresa eliminando inversiones. Para que se ejecute eficientemente, debe eliminar más de una inversión por intercambio.

8.4 Shellsort

El primer algoritmo que mejoró de forma sustancial la ordenación por inserción fue *Shellsort*. El algoritmo de ordenación de Shell, Shellsort, fue desarrollado en 1959 por Donald Shell. Aunque hoy en día no es el algoritmo conocido más rápido, sólo es algo más largo y complejo que la ordenación por inserción, lo que le hace el más simple de los algoritmos más rápidos.

La idea de Shell fue evitar gran cantidad de movimientos de datos comparando primero elementos que estuvieran muy separados, para después comparar elementos algo más cercanos, y así sucesivamente, reduciéndose gradualmente al método de inserción. Shellsort utiliza una secuencia $h_1, h_2, \ldots, h_t$, denominada *secuencia de incrementos*. Cualquier secuencia de incrementos es válida, con tal de que $h_1 = 1$, pero algunas elecciones son mejores que otras. Después de una fase utilizando un cierto incremento h_k, tendremos que $a[i] \leq a[i + h_k]$ para todo i para el

que dicha comparación tenga sentido. Esto es, todos los elementos separados por una distancia h_k están ordenados. Se dice entonces que el vector está *h_k-ordenado*.

Por ejemplo, la Figura 8.3 muestra un vector después de la aplicación de varias fases del algoritmo Shellsort. Después de una 5-ordenación, se garantiza que los elementos a distancia 5 están correctamente ordenados. En la figura, los distintos elementos a distancia 5 están sombreados de igual forma. Como puede verse, todos están ordenados entre sí. De igual forma, después de la 3-ordenación, se garantiza que los elementos a distancia 3 están correctamente ordenados entre sí. Una propiedad importante de Shellsort (que afirmaremos sin demostración) es que un vector h_k-ordenado sigue estándolo después de una h_{k-1}-ordenación. Si no fuera así, el algoritmo sería de poca utilidad ya que el trabajo realizado por las primeras fases sería deshecho por las siguientes.

En general, una h_k-ordenación requiere que para cada posición i en h_k, h_{k+1}, ..., $N-1$, se coloque el elemento en el sitio adecuado de entre las i, $i - h_k$, $i - 2h_k$ y así sucesivamente. Aunque ello no afecta a la implementación, un examen cuidadoso muestra que el efecto de una h_k-ordenación es realizar una ordenación por inserción en h_k subvectores independientes (mostrados con diferentes sombreados en la Figura 8.3). Por consiguiente, no debería sorprender que en la Figura 8.4, las líneas de la 9 a la 19 representen una *ordenación por inserción por intervalos*. En una ordenación por inserción por intervalos, después de ejecutar el bucle, los elementos del vector separados por una distancia de `intervalo` están ordenados. Por ejemplo, cuando el `intervalo` es 1, el bucle es idéntico, instrucción a instrucción, a la ordenación por inserción. En consecuencia, Shellsort también se conoce como *ordenación por disminución de intervalos*.

Es fácil mostrar varios hechos. Primero, cuando `intervalo` es 1, se garantiza que el bucle más interno ordena el vector `a`. Si `intervalo` nunca se hiciera 1, tendríamos algún vector de entrada que no podría ser ordenado correctamente. Por tanto, Shellsort ordena hasta que `intervalo` llega a valer 1, y tras ello podemos parar. La única cuestión pendiente es cómo elegir la secuencia de incrementos.

Shell sugirió empezar con `intervalo` igual a $N/2$, e ir dividiéndolo por la mitad hasta llegar a 1, momento en el que el algoritmo termina. Utilizando estos incrementos, Shellsort representa una mejora sustancial con respecto a la ordenación por inserción, a pesar de tener tres bucles `for` anidados en vez de dos, lo que habitualmente sería ineficiente. Alterando la secuencia de incrementos, uno puede seguir mejorando el rendimiento del algoritmo. En la Figura 8.5 se muestra una tabla de rendimientos de Shellsort con tres elecciones diferentes de secuencias de incrementos.

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
Después de la 5-ordenación	35	17	11	28	12	41	75	15	96	58	81	94	95
Después de la 3-ordenación	28	12	11	35	15	41	58	17	94	75	81	96	95
Después de la 1-ordenación	11	12	15	17	28	35	41	58	75	81	94	95	96

Figura 8.3 Shellsort después de cada paso, cuando la secuencia de incrementos es (1, 3, 5).

```
1  /**
2   * Shellsort, utilizando la secuencia sugerida por Gonnet.
3   * @param a un vector de elementos Comparable.
4   */
5  public static void shellsort( Comparable [ ] a )
6  {
7      for( int intervalo = a.length / 2; intervalo > 0;
8           intervalo = intervalo == 2 ? 1 : (int) ( intervalo / 2.2 ) )
9           for( int i = intervalo; i < a.length; i++ )
10          {
11              Comparable temp = a[ i ];
12              int j = i;
13
14              for( ; j >= intervalo &&
15                     temp.menorQue( a[ j - intervalo ] );
16                                                  j -= intervalo )
17                  a[ j ] = a[ j - intervalo ];
18              a[ j ] = temp;
19          }
20 }
```

Figura 8.4 Implementación de Shellsort que divide por 2,2 en vez de por 2.

8.4.1 Rendimiento de Shellsort

El tiempo de ejecución de Shellsort depende en gran medida de la secuencia de incrementos elegida. Las correspondientes demostraciones de los resultados precisos en cada caso pueden resultar bastante complicadas. El análisis en el caso medio de Shellsort es un problema abierto desde hace mucho tiempo, excepto para las secuencias de incrementos triviales.

N	Ordenación por inserción	Shellsort		
		Secuencia de Shell	Sólo intervalos impares	Dividiendo por 2,2
1.000	122	11	11	9
2.000	483	26	21	23
4.000	1.936	61	59	54
8.000	7.950	153	141	114
16.000	32.560	358	322	269
32.000	131.911	869	752	575
64.000	520.000	2.091	1.705	1.249

Figura 8.5 Tiempo de ejecución (en milisegundos) de la ordenación por inserción y de Shellsort con varias secuencias de incrementos.

Cuando se utilizan los incrementos de Shell, se puede probar que el caso peor es $O(N^2)$. Esta cota se alcanza cuando N es una potencia de 2, todos los elementos grandes están en posiciones pares del vector y todos los pequeños en posiciones impares. Cuando se alcanza la última iteración, todos los elementos grandes estarán todavía en las posiciones pares, y todos los pequeños en posiciones impares. Un cálculo del número de inversiones restante muestra que la última iteración requerirá un tiempo cuadrático. La afirmación de que éste es el peor caso se sigue del hecho de que una h_k-ordenación consiste en h_k ordenaciones por inserción de, aproximadamente, N/h_k elementos. Por consiguiente, el coste de cada iteración es $O(h_k(N/h_k)^2)$, o sea $O(N^2/h_k)$. Al sumar sobre todas las iteraciones, obtenemos $O(N^2\Sigma h_k)$. Ya que los incrementos generan, aproximadamente, una serie geométrica, la suma está acotada por una constante. El resultado es que el tiempo de ejecución es cuadrático en el caso peor. También se puede probar, a través de un argumento complejo, que cuando N es una potencia de 2, el tiempo medio de ejecución es $O(N^{3/2})$. Por tanto, en promedio, la secuencia de incrementos de Shell representa una mejora significativa sobre la ordenación por inserción.

Un cambio mínimo en la secuencia de incrementos puede evitar el caso peor cuadrático. Si al dividir `intervalo` por 2 el resultado es par, entonces le sumaremos 1 para hacerlo impar. En este caso se puede probar que el caso peor ya no es cuadrático, sino $O(N^{3/2})$. Aunque la demostración es complicada, se basa en que en esta nueva secuencia de incrementos, los incrementos consecutivos no comparten ningún factor común (mientras que para la secuencia de incrementos de Shell esto sí ocurría). Cualquier secuencia que cumpla esta propiedad (y cuyos incrementos decrezcan, aproximadamente, de forma geométrica) tendrá un tiempo de ejecución en el caso peor $O(N^{3/2})$[1]. Se desconoce el rendimiento en promedio del algoritmo con estos nuevos incrementos, pero basándose en simulaciones parece ser $O(N^{5/4})$.

Una tercera secuencia, que se comporta bien en la práctica, aunque se desconozca el fundamento teórico de ello, consiste en dividir por 2,2 en vez de por 2. Parece que con ello conseguimos un tiempo de ejecución en promedio por debajo de $O(N^{5/4})$ (quizás $O(N^{7/6})$), aunque esto está todavía por precisar. Con un número de elementos entre 100.000 y 1.000.000, típicamente se mejora el rendimiento entre un 25 y un 35 por ciento, aunque nadie sabe muy bien por qué. Una implementación de Shellsort con esta secuencia de incrementos se presenta en la Figura 8.4. El código complicado de la línea 8 es para evitar que `intervalo` se pueda hacer igual a 0. Si tal cosa ocurriera, el algoritmo no funcionaría correctamente, pues nunca se haría una 1-ordenación. La línea 8 asegura que cuando `intervalo` vale 2, pasa a valer 1, y no 0.

La tabla de la Figura 8.5 compara el rendimiento de la ordenación por inserción y el de Shellsort, con varias secuencias de intervalos. Estos resultados se obtuvieron en una máquina razonablemente rápida. El test claramente está predispuesto en contra de la secuencia de intervalos original, pues N se elige 125 veces una potencia de 2. Así el redondear hacia arriba hasta un número impar es particularmente beneficioso, especialmente cuando N se hace grande. Podríamos concluir fácilmente que Shellsort, incluso con la secuencia de intervalos más simple, proporciona una mejora significativa sobre la ordenación por inserción, con un coste adicional mínimo en lo que se refiere a la complejidad lógica del algoritmo. Un simple cambio en la secuencia de intervalos puede mejorar todavía más el rendimiento. Se pueden conseguir mejoras aún mayores (como se verá en el Ejercicio 8.18). Algunas de estas mejoras tienen un fundamento teórico, pero ninguna secuencia conocida mejora apreciablemente la mostrada en la Figura 8.4.

El rendimiento de Shellsort es bastante aceptable en la práctica, aun con valores de N de decenas de miles. La simplicidad del código hace que sea el algoritmo a elegir para ordenar entradas de tamaño moderadamente grande. Es también un buen ejemplo de algoritmo muy simple con un análisis extremadamente complejo.

[1] Aunque la demostración no es muy complicada, requiere una cierta habilidad matemática. Para una pista de la naturaleza involucrada, observe que restar 1, en vez de sumar 1, no funciona. Por ejemplo, si N es 186, la secuencia 9, 3, 1, en la que todos los incrementos comparten 3 como factor común.

8.5 Mergesort

En la Sección 7.5 vimos cómo la recursión puede utilizarse para desarrollar algoritmos subcuadráticos. En concreto, un algoritmo divide y vencerás en el cual se resuelven recursivamente dos problemas con la mitad de tamaño, con una sobrecarga $O(N)$, es un algoritmo $O(N \log N)$. Mergesort es uno de estos algoritmos, ofreciendo una mejor cota, al menos a nivel teórico, que las afirmadas para Shellsort.

El algoritmo mergesort consta de tres pasos:

1. Si el número de elementos a ordenar es 0 o 1, acabar.
2. Ordenar recursivamente las dos mitades del vector.
3. Mezclar las dos mitades ordenadas en un vector ordenado.

Para afirmar que el algoritmo es $O(N \log N)$, es suficiente demostrar que la mezcla ordenada de dos vectores ordenados puede realizarse en tiempo lineal. En la siguiente subsección veremos cómo mezclar dos vectores A y B, colocando el resultado en un tercer vector C. Después mostraremos una implementación sencilla de mergesort. La rutina de mezcla es la piedra angular de muchos algoritmos de ordenación externa. Esto se ilustrará en la Sección 20.6.

8.5.1 Mezcla lineal de vectores ordenados

El algoritmo de mezcla básico parte de dos vectores de entrada A y B, produciendo un vector de salida C, apoyándose en tres contadores *Acont*, *Bcont* y *Ccont*, inicializados al principio de los vectores respectivos. En cada paso el menor de los valores $A[Acont]$ y $B[Bcont]$ se copia en la próxima posición de C, actualizándose los contadores apropiados. Cuando uno de los vectores de entrada se acaba, el resto del otro vector se copia en C.

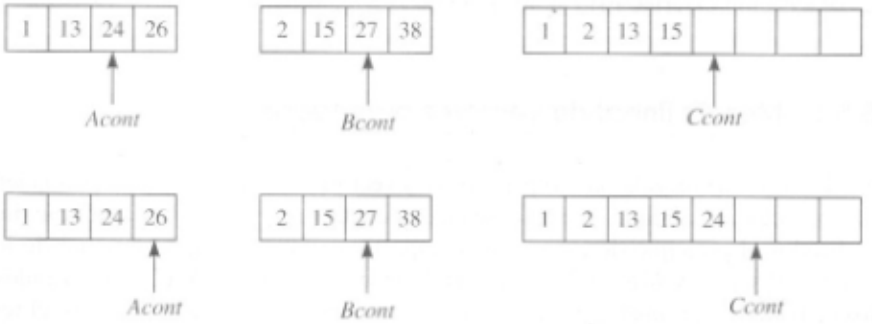
Veamos cómo funciona la rutina de mezcla sobre la siguiente entrada:

Cuando el vector *A* contiene los elementos 1, 13, 24 y 26, y *B* contiene los elementos 2, 15, 27, 38, el algoritmo se comporta de la siguiente manera: en primer lugar, se comparan los elementos 1 y 2, con lo que 1 se copia en *C*; tras ello se comparan los elementos 13 y 2:

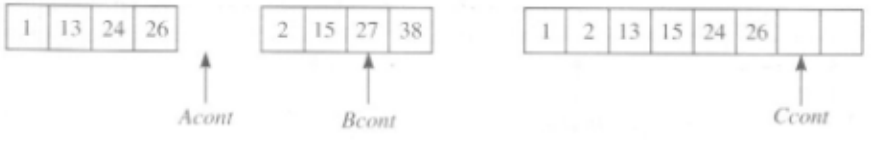
El 2 se añade a *C*, y se pasan a comparar los elementos 13 y 15:

El 13 se añade a *C*, y se comparan los elementos 24 y 15:

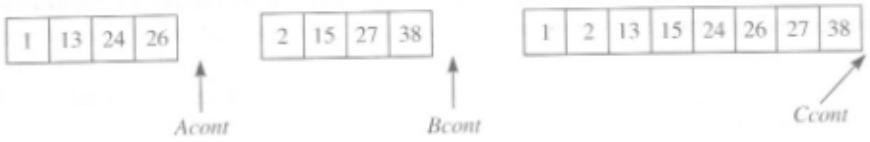
El proceso continúa comparando los elementos 26 y 27:



El elemento 26 se añade a *C*, con lo que el vector *A* se acaba:



Finalmente, el resto del vector *B* se copia en *C*:



El tiempo necesario para mezclar dos vectores ordenados es lineal pues cada comparación incrementa *Ccont* (limitando, por tanto, el número de comparaciones). Como resultado, un algoritmo divide y vencerás que utilice un procedimiento de mezcla lineal se ejecutará en el caso peor en un tiempo $O(N \log N)$. Éste también será el tiempo de ejecución en el caso promedio, e incluso en el caso mejor, ya que la mezcla es siempre lineal.

Como ejemplo de aplicación del algoritmo mergesort ordenemos el vector de 8 elementos (24, 13, 26, 1, 2, 27, 38, 15). Después de ordenar, recursivamente, los 4 primeros elementos y los 4 últimos, obtenemos (1, 13, 24, 26, 2, 15, 27, 38). Finalmente se mezclan las dos mitades, obteniendo el vector final (1, 2, 13, 15, 24, 26, 27, 38).

```
1  /**
2   * Método interno que realiza las llamadas recursivas.
3   * @param a un vector de elementos de tipo Comparable.
4   * @param vectorTemp un vector para colocar el resultado de la
      mezcla.
5   * @param izq el índice más a la izquierda del subvector.
6   * @param der el índice más a la derecha del subvector.
7   */
8  private static void mergeSort( Comparable [ ] a,
9                                 Comparable [ ] vectorTemp, int izq, int der )
10 {
11     if( izq < der )
12     {
13         int centro = ( izq + der ) / 2;
14         mergeSort( a, vectorTemp, izq, centro );
15         mergeSort( a, vectorTemp, centro + 1, der );
16         mezclar( a, vectorTemp, izq, centro + 1, der );
17     }
18 }
19
20 /**
21  * Algoritmo mergesort.
22  * @param a un vector de elementos de tipo Comparable.
23  */
24 public static void mergeSort( Comparable [ ] a )
25 {
26     Comparable [ ] vectorTemp = new Comparable[ a.length ];
27
28     mergeSort( a, vectorTemp, 0, a.length - 1 );
29 }
```

Figura 8.6 Rutinas básicas de `mergeSort`.

8.5.2 El algoritmo mergesort

La Figura 8.6 muestra una implementación muy sencilla de mergesort. El `mergeSort` público es un simple envoltorio que declara un vector temporal y llama al `mergeSort` recursivo con los límites del vector. La rutina `mezclar` sigue la descripción dada en la sección anterior. Utiliza la primera mitad del vector (indexado desde `izq` a `centro`) como A, la segunda mitad (indexada desde `centro+1` hasta `der`) como B y el vector auxiliar como C. La Figura 8.7 muestra la rutina `mezclar`. Por último, el vector auxiliar se copia en el vector original.

Aunque el tiempo de ejecución de mergesort es $O(N \log N)$, rara vez se utiliza para ordenaciones en memoria. Esto es debido a que la mezcla de dos subvectores ordenados utiliza un espacio adicional lineal, y el trabajo adicional debido a las copias hacia y desde el vector auxiliar al original, ralentizan considerablemente la ordenación. Las copias se pueden evitar intercambiando alternativamente los papeles de `a` y `vectorAux` en las llamadas recursivas.

```
1  /**
2   * Método interno que mezcla dos mitades ordenadas de un
      subvector.
3   * @param a un vector de elementos de tipo Comparable.
4   * @param vectorTemp un vector para colocar el resultado de la
      mezcla.
5   * @param posIzq el índice más a la izquierda del vector.
6   * @param porDer el índice de comienzo de la segunda mitad.
7   * @param posFin el índice más a la derecha del vector.
8   */
9  private static void mezclar( Comparable [ ] a,
10                              Comparable [ ] vectorAux, int posIzq,
11                              int posDer, int posFin )
12 {
13     int finIzq = posDer - 1;
14     int posAux = posIzq;
15     int numElementos = posFin - posIzq + 1;
16
17     // Bucle principal
18     while( posIzq <= finIzq && posDer <= posFin )
19         if( a[ posIzq ].menorQue( a[ posDer ] ) )
20             vectorAux[ posAux++ ] = a[ posIzq++ ];
21         else
22             vectorAux[ posAux++ ] = a[ posDer++ ];
23
24     // Copiar el resto de la primera mitad
25     while( posIzq <= finIzq )
26         vectorAux[ posAux++ ] = a[ posIzq++ ];
27
28     // Copiar resto de la segunda mitad
29     while( posDer <= posFin )
30         vectorAux[ posAux++ ] = a[ posDer++ ];
31
32     // Copiar el vector temporal en el original
33     for( int i = 0; i < numElementos; i++, posFin-- )
34         a[ posFin ] = vectorAux[ posFin ];
35 }
```

Figura 8.7 Rutina `mezclar`.

Una variación de mergesort se puede implementar de forma no recursiva. De cualquier forma, para aplicaciones serias de ordenación interna el algoritmo utilizado suele ser quicksort, descrito en la próxima sección.

8.6 Quicksort

Quicksort[2] es el algoritmo de ordenación más rápido conocido. Su tiempo de ejecución promedio es $O(N \log N)$. Su rapidez se debe principalmente a un bucle interno muy ajustado y altamente optimizado. Tiene un rendimiento cuadrático en el caso peor, pero este caso puede hacerse estadísticamente improbable con poco esfuerzo. El algoritmo quicksort es relativamente fácil de entender, así como de demostrar su corrección, ya que se basa en la recursión. Sin embargo su implementación requiere atención, pues cambios mínimos en el código pueden repercutir considerablemente en el tiempo de ejecución. Esta sección comienza describiendo el algoritmo en un sentido amplio, para proporcionar después un análisis que muestra sus tiempos de ejecución en el caso peor, en promedio, y en el caso mejor. Utilizaremos este análisis para fijar ciertas decisiones sobre la implementación en Java, atendiendo a ciertos detalles, tales como el tratamiento de elementos duplicados.

8.6.1 El algoritmo quicksort

El algoritmo básico *Quicksort*(S) es recursivo, consistiendo en los siguientes cuatro pasos:

1. Si el número de elementos de S es 0 o 1, terminar.
2. Escoger un elemento *cualquiera* v de S. Este elemento se denominará *pivote*.
3. Hacer una *partición* de $S - \{v\}$ (el resto de elementos de S) en dos grupos disjuntos: $I = \{x \in S - \{v\} \mid x \leqslant v\}$ y $D = \{x \in S - \{v\} \mid x \geqslant v\}$.
4. Devolver el resultado de *Quicksort*(I), seguido de v y seguido del resultado de *Quicksort*(D).

Hay que puntualizar una serie de cuestiones. En primer lugar, el caso básico de la recursión incluye la posibilidad de que S sea vacío. Esto es necesario, ya que las llamadas recursivas pueden generar subconjuntos vacíos. Segundo, el algoritmo permite que cualquier elemento sea usado como *pivote*. El pivote divide a los elementos del vector en dos grupos: los menores y los mayores que él. El análisis que realizaremos mostrará que algunas elecciones del pivote son mejores que otras. Por tanto, al proporcionar una implementación concreta, no utilizaremos un pivote cualquiera; trataremos de hacer una buena elección al respecto.

En el paso de *partición*, cada elemento de S, excepto el pivote, se coloca bien en I, que representa la parte izquierda del vector, o en D, que representa la parte derecha del vector. El objetivo es que los elementos menores que el pivote vayan a I, mientras que los elementos mayores van a D. Sin embargo, la descripción del

[2] N. del T.: Ordenación rápida, en inglés.

algoritmo no puntualiza qué hacer con los hipotéticos elementos iguales al pivote. Se permite que en cada caso un elemento tal vaya a cualquiera de los subconjuntos, especificándose únicamente que debe ir a uno sólo. Parte de una buena implementación en Java residirá en tratar este caso tan eficientemente como sea posible. Una vez más, el análisis nos permitirá tomar una decisión bien fundamentada.

La Figura 8.8 muestra el efecto de quicksort sobre un conjunto de números. Se toma como pivote (sin que haya ninguna razón especial para hacerlo así) el número 65. El resto de elementos en el conjunto se separan en dos subconjuntos de tamaño menor. Cada unos de estos subconjuntos se ordena recursivamente. Aplicando la tercera regla de la recursión, podemos suponer que este paso funciona correctamente. Por último, se obtiene de forma inmediata el resultado ordenado del conjunto original. En una implementación en Java, los elementos a ordenar en

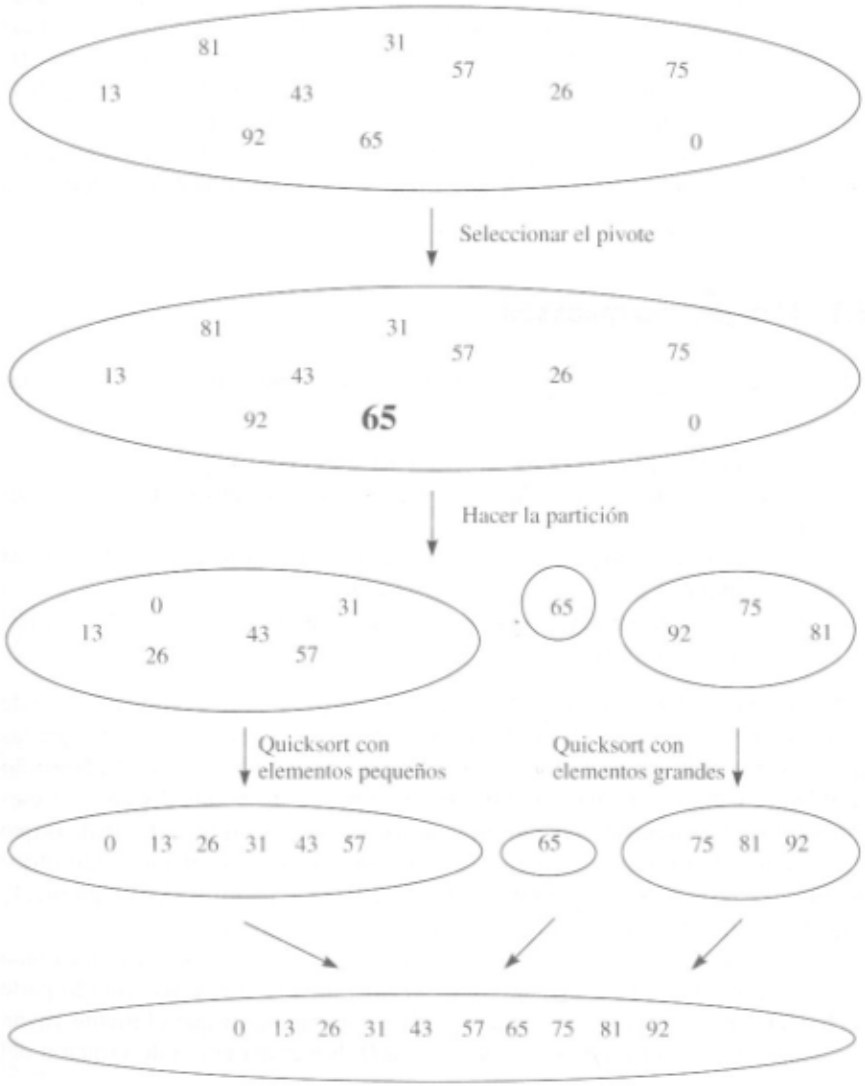


Figura 8.8 Los pasos de quicksort.

cada paso se almacenarían en una parte del vector delimitada por `ini` y `fin`. Después del paso de partición, el pivote terminaría en alguna posición `p` del vector y las llamadas recursivas se harían con los subvectores desde `ini` hasta `p-1` y desde `p+1` hasta `fin`.

La corrección del algoritmo se basa en los siguientes hechos:

- El subconjunto de elementos pequeños se ordena correctamente, en virtud de la recursión.
- El mayor elemento en el subconjunto de elementos pequeños no es mayor que el pivote, dado cómo se hace la partición.
- El pivote no es mayor que el menor elemento en el subconjunto de elementos grandes, dado cómo se hace la partición.
- El subconjunto de elementos grandes se ordena correctamente, en virtud de la recursión.

Aunque la corrección del algoritmo se establece fácilmente, no está claro por qué es más rápido que mergesort. Igual que mergesort, resuelve de manera recursiva dos subproblemas y requiere un cierto trabajo adicional (en el paso de partición). Además, al contrario de mergesort, no se garantiza que el tamaño de los subproblemas sea el mismo. Esto no es bueno para el rendimiento. Quicksort es más rápido que mergesort porque el paso de partición puede hacerse significativamente más deprisa que el paso de mezcla. En concreto, puede implementarse sin utilizar un vector auxiliar, y el código es muy compacto y eficiente. Esto compensa la no generación sistemática de subproblemas del mismo tamaño.

8.6.2 Análisis de quicksort

La descripción del algoritmo deja varias cuestiones sin contestar. ¿Cómo elegimos el pivote? ¿Cómo realizamos la partición? ¿Qué hacemos si encontramos un elemento igual al pivote? Todas estas cuestiones pueden afectar en gran medida al tiempo de ejecución del algoritmo. Realizaremos un análisis que nos ayude a decidir cómo deberíamos implementar los pasos no especificados de quicksort.

El caso mejor

El caso mejor para quicksort se presenta cuando el pivote divide al conjunto en dos subconjuntos de igual tamaño. En tal caso tenemos dos llamadas con un tamaño de la mitad y una sobrecarga adicional lineal, igual que en mergesort. En consecuencia, en este caso, el tiempo de ejecución es $O(N \log N)$. (No hemos probado formalmente que éste sea en efecto el mejor caso. Tal demostración es posible, aunque no la veremos aquí.)

El caso peor

Ya que los subconjuntos de igual tamaño son los mejores para quicksort, uno podría esperar que los de muy distintos tamaños sean malos. Y esto es, en efecto, lo

que ocurre. Supongamos que en cada paso de la recursión, el pivote es el menor elemento. En tal caso, el subconjunto I de elementos pequeños será vacío y el subconjunto D de elementos grandes contendrá todos los elementos, excepto el pivote. Habrá ahora que llamar recursivamente a quicksort con el subconjunto D. Supongamos que $T(N)$ es el tiempo de ejecución de quicksort con N elementos y que el tiempo de ordenar 0 o 1 elemento es solamente una unidad. Supongamos también que hacer la partición de N elementos requiere N unidades de tiempo. Entonces, para $N > 1$, obtenemos un tiempo de ejecución que satisface

$$T(N) = T(N-1) + N. \tag{8.1}$$

En otras palabras, la Ecuación 8.1 afirma que en este caso el tiempo que necesita quicksort para ordenar N elementos es igual al tiempo de ordenar recursivamente los $N-1$ elementos del subconjunto de elementos mayores más las N unidades del coste de realizar la partición. Esto asumiendo que en todos los pasos de la recursión somos tremendamente desafortunados y tomamos como pivote el menor elemento. Para simplificar el análisis, normalizamos eliminando los factores constantes. Podemos resolver la recurrencia desplegando repetidamente la Ecuación 8.1:

$$\begin{aligned} T(N) &= T(N-1) + N \\ T(N-1) &= T(N-2) + (N-1) \\ T(N-2) &= T(N-3) + (N-2) \\ &\cdots \\ T(2) &= T(1) + 2 \end{aligned} \tag{8.2}$$

Cuando sumamos por columnas todos los términos de la Ecuación 8.2, podemos cancelar términos de uno y otro lado, obteniendo

$$T(N) = T(1) + 2 + 3 + \cdots + N = N(N+1)/2 = O(N^2). \tag{8.3}$$

Este análisis corrobora la intuición de que una división desigual no es buena. Gastamos N unidades de tiempo para hacer la partición y después tenemos que hacer una llamada recursiva para $N-1$. Entonces gastamos $N-1$ unidades para hacer la partición de este subconjunto, sólo para hacer una llamada recursiva con $N-2$ elementos. En esa llamada gastamos $N-2$ unidades para realizar la partición, y así sucesivamente. El tiempo total para realizar todas las particiones a través de las llamadas recursivas coincide exactamente con el obtenido en la Ecuación 8.3. Esto nos dice que cuando implementemos la selección del pivote y el paso de partición, no debemos hacer nada que haga probable la obtención de subconjuntos de tamaño desigual.

Caso medio

Los dos primeros análisis nos dicen que el mejor y el peor caso son enormemente diferentes. Naturalmente, queremos saber qué ocurre en el caso medio. Podríamos esperar que ya que cada subproblema es en promedio la mitad que el original, la cota para el caso medio sea $O(N \log N)$. Esa expectativa, aunque correcta para la aplicación particular de quicksort que presentamos aquí, no constituye una demostración formal. Los análisis en el caso medio no pueden realizarse a la ligera. Supongamos, por ejemplo, que tenemos un algoritmo para elegir el pivote que garantiza seleccionar el menor o el mayor elemento, cada uno con probabilidad 1/2. Entonces el tamaño medio del subconjunto con elementos pequeños es aproximadamente $N/2$, así como el del subconjunto de elementos grandes (ya que cada uno de ellos tiene con igual probabilidad 0 o $N-1$ elementos). Pero el tiempo de ejecución de quicksort con esta selección del pivote será siempre cuadrático, ya que siempre se realiza una división pobre de los elementos. Debemos ser muy cuidadosos en cómo asignamos la etiqueta «en el caso medio». Siendo más precisos, podemos argumentar que el conjunto de elementos pequeños tendrá con igual probabilidad 0, 1, 2, ..., o $N-1$ elementos, lo que también es cierto para el subconjunto de elementos mayores. Partiendo de esta hipótesis, sí podemos probar que el tiempo de ejecución de quicksort en el caso medio sería $O(N \log N)$.

Puesto que el coste de quicksort para N elementos es igual a N unidades correspondientes a la partición, más el coste de dos llamadas recursivas, necesitamos determinar el coste medio de cada llamada recursiva. Si $T(N)$ representa el coste medio de quicksort para N elementos, el coste medio de cada llamada recursiva es igual a la media, sobre todos los posibles tamaños de los subproblemas, del coste medio de una llamada recursiva con un subproblema:

$$T(I) = T(D) = \frac{T(0) + T(1) + T(2) + \cdots + T(N-1)}{N} \tag{8.4}$$

La Ecuación 8.4 afirma que miramos cada uno de los costes de las posibles llamadas y después hacemos la media. Como tenemos dos llamadas recursivas, más un tiempo lineal para hacer la partición, obtenemos

$$T(N) = 2\left(\frac{T(0) + T(1) + T(2) + \cdots + T(N-1)}{N}\right) + N. \tag{8.5}$$

Para resolver la Ecuación 8.5, empezamos por multiplicar ambos lados por N, obteniendo

$$NT(N) = 2(T(0) + T(1) + T(2) + \cdots + T(N-1)) + N^2. \tag{8.6}$$

Ahora escribimos la Ecuación 8.6 para el caso $N-1$, con la idea de restarla de la anterior y así simplificar en gran medida la ecuación. Tenemos entonces

$$(N-1)T(N-1) = 2(T(0) + T(1) + T(2) + \cdots + T(N-2)) + (N-1)^2. \tag{8.7}$$

y si restamos la Ecuación 8.7 de la Ecuación 8.6, obtenemos

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2N - 1.$$

Si reorganizamos los términos y eliminamos el -1 de la derecha (no significativo), obtenemos

$$NT(N) = (N+1)T(N-1) + 2N. \tag{8.8}$$

Ahora tenemos una fórmula para $T(N)$ en términos de $T(N-1)$. De nuevo, la idea es desplegar, pero la Ecuación 8.8 no está en la forma adecuada para hacerlo. Si dividimos la Ecuación 8.8 por $N(N+1)$, obtenemos

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1}.$$

Ahora podemos desplegar:

$$\begin{aligned}
\frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2}{N+1} \\
\frac{T(N-1)}{N} &= \frac{T(N-2)}{N-1} + \frac{2}{N} \\
\frac{T(N-2)}{N-1} &= \frac{T(N-3)}{N-2} + \frac{2}{N-1} \qquad (8.9) \\
&\cdots \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2}{3}
\end{aligned}$$

Si sumamos todas las ecuaciones de la Ecuación 8.9, obtenemos

$$\begin{aligned}
\frac{T(N)}{N+1} &= \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N} + \frac{1}{N+1}\right) \\
&= 2\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N+1}\right) - \frac{5}{2} \qquad (8.10) \\
&= O(\log N)
\end{aligned}$$

La última línea de la Ecuación 8.10 se deduce del Teorema 5.5. Cuando multiplicamos ambos lados por $N+1$, obtenemos el resultado final

$$T(N) = O(N \log N). \qquad (8.11)$$

8.6.3 Seleccionando el pivote

Ahora que hemos establecido que el tiempo de ejecución en promedio de quicksort es $O(N \log N)$, nuestro objetivo central será asegurarnos que el peor caso no se presenta nunca. Realizando un análisis complejo, podemos calcular la desviación típica del tiempo de ejecución de quicksort. El resultado es que si partimos de una permutación aleatoria, es casi seguro que el tiempo de ejecución para ordenarla estará próximo al caso medio. Pero debemos tener cuidado de que las entradas degeneradas no lleven a tiempos de ejecución malos. Las entradas degeneradas incluyen, entre otros, el caso en el que los datos ya estén ordenados, y aquél en el que todos los elementos sean iguales. Como veremos más adelante, en ocasiones son los casos aparentemente más sencillos los que complican el algoritmo.

Una forma incorrecta

La elección más popular cuando no se cuenta con la información que lo desaconsejea, es tomar sin más el primer elemento (es decir, el elemento en la posición `ini`) como pivote. Esto es aceptable si la entrada es aleatoria, pero si la entrada ya está ordenada o está en orden inverso, entonces este pivote proporciona una división pobre, pues será un elemento extremo. Es más, este mal comportamiento se repetirá recursivamente. Como vimos anteriormente en este capítulo, invertiríamos un tiempo cuadrático en ¡no hacer nada! (el vector ya venía ordenado). Esto, huelga decirlo, es bastante embarazoso. En consecuencia, *nunca* use sin más el primer elemento como pivote.

Una elección segura

Una elección perfectamente razonable es elegir como pivote el elemento central, es decir, el elemento en la posición `(ini + fin)/2` del vector. Cuando la entrada ya está ordenada, esto nos da el pivote perfecto, en cada llamada recursiva. Por supuesto, podríamos construir un vector de entrada que forzaría un comportamiento cuadrático bajo esta estrategia (*véase* el Ejercicio 8.8), pero las posibilidades de que se ejecute aleatoriamente un caso que requiera incluso el doble del caso medio son astronómicamente pequeñas.

Partición con la mediana de tres

Elegir el elemento central como pivote evita los casos degenerados que representan entradas no aleatorias. Sin embargo, ésta es una elección pasiva. Es decir, no intentamos elegir un buen pivote; en vez de eso, evitamos únicamente elegir uno malo. La *partición con la mediana de tres* es un intento de seleccionar un pivote mejor que el central.

La mediana de un grupo de N elementos es el $\lceil N/2 \rceil$-ésimo menor elemento. La mejor elección del pivote sería claramente la mediana, pues garantizaría una división uniforme de los elementos. Desgraciadamente, calcular la mediana no es fácil y ralentizaría considerablemente a quicksort. Lo que queremos es obtener una buena estimación de la mediana sin gastar demasiado tiempo. Dicha estimación puede hacerse por muestreo: es decir, tomamos un subconjunto de los elementos a ordenar y buscamos su mediana. Éste es el método clásico utilizado en los sondeos de opinión. Cuanto mayor sea la muestra, más acertada será la estimación. Sin embargo, cuanto mayor sea la muestra más tardaremos en hacer los cálculos. Se ha probado que una muestra de tamaño tres proporciona una pequeña mejora en el caso promedio de quicksort, y también simplifica el código de parti-

ción resultante, eliminando algunos casos especiales. Se ha probado también que muestras de gran tamaño no mejoran el rendimiento de forma significativa, por lo que no merecen la pena.

Los tres elementos que utilizaremos en nuestra muestra son el primero, el central y el último. Por ejemplo, teniendo como entrada (8, 1, 4, 9, 6, 3, 5, 2, 7, 0), el elemento más a la izquierda es 8, el de más a la derecha es 0 y el del centro es 6; por tanto, el pivote escogido sería 6 (cuando hay dos elementos en el centro, se elige el de más a la izquierda). Observe que para vectores ya ordenados, mantenemos el elemento central como pivote, lo que vuelve a representar el mejor caso.

8.6.4 Estrategia de partición

Existen varias estrategias de partición utilizadas con asiduidad. Se sabe que la descrita en esta sección da buenos resultados. A continuación se discute la estrategia de partición más simple. Funciona en tres pasos. La Sección 8.6.6 muestra las mejoras obtenidas cuando se utiliza la mediana de tres.

El primer paso en el algoritmo de partición es quitar el pivote de en medio intercambiándolo con el último elemento. El resultado sobre nuestro ejemplo de entrada se muestra en la Figura 8.9. El pivote se muestra con el sombreado más oscuro, al final del vector.

Por ahora, asumiremos que todos los elementos son distintos y dejaremos para después qué hacer en presencia de duplicados. Como caso límite, nuestro algoritmo debería funcionar correctamente cuando *todos* los elementos son iguales.

En el segundo paso, utilizamos nuestra estrategia de partición para mover todos los elementos menores a la parte de la izquierda, y todos los elementos mayores a la parte de la derecha. En las Figuras de la 8.10 a la 8.15, las celdas blancas son celdas que sabemos que están bien colocadas. No se sabe todavía si las que están en penumbra están bien colocadas.

Buscamos de izquierda a derecha un elemento mayor. Para hacer esto utilizamos un contador `i`, inicializado a `ini`. Por su parte, buscamos de derecha a izquierda un elemento menor. Lo hacemos utilizando un contador `j`, inicializado a `fin-1`. La Figura 8.10 muestra que la búsqueda del elemento mayor para en el 8 y la búsqueda del menor para en el 2. Estas celdas han sido sombreadas ligeramente. Observe como hemos saltado el 7, porque sabemos que el 7 no es menor que el pivote por lo que está bien colocado. En consecuencia queda coloreado de blanco. Ahora tenemos en el lado izquierdo del vector, un elemento mayor que el pivote, el 8, y uno menor, el 2, en la parte derecha. Debemos intercambiarlos para colocarlos correctamente, como muestra la Figura 8.11.

Figura 8.9 Algoritmo de partición: el pivote, 6, se coloca al final.

Figura 8.10 Algoritmo de partición: `i` se para en el elemento mayor que el pivote, 8; `j` se para en el elemento menor que el pivote, 2.

Figura 8.11 Algoritmo de partición: los elementos mal colocados 8 y 2 son intercambiados.

Al continuar la ejecución del algoritmo, `i` se para en el elemento mayor que el pivote, 9, y `j` se para en el elemento menor, 5. Una vez más, los elementos que se saltan `i` y `j` durante el proceso están bien colocados. La Figura 8.12 muestra el resultado: los elementos de los extremos (sin incluir el pivote) están colocados correctamente.

Lo siguiente que hay que hacer es intercambiar los elementos indexados por `i` y `j`, como muestra la Figura 8.13. El recorrido continúa, con `i` parándose en el elemento mayor que el pivote, 9, y `j` parándose en el elemento menor, 3. Sin embargo, en este momento los índices `i` y `j` se han cruzado, por lo que un intercambio no tendría sentido. Esto se muestra en la Figura 8.14, en la cual vemos que el elemento indexado por `j` está colocado correctamente, por lo que no debe moverse.

En la Figura 8.14 se muestra que todos los elementos salvo el pivote están correctamente colocados. ¿No sería bonito que con un intercambio pudieramos ubicarlo correctamente, para así terminar? Pues sí, podemos. Todo lo que tenemos que hacer es intercambiar el pivote con el elemento de la posición `i`. El resultado se muestra en la Figura 8.15. El elemento apuntado por `i` era mayor que el pivote por lo que pasarlo a la última posición es acertado.

Observe que el algoritmo de partición no necesita memoria auxiliar y que cada elemento es comparado con el pivote exactamente una vez. Cuando se escribe el código, esto se traduce a un bucle interno muy ajustado.

Figura 8.12 Algoritmo de partición: `i` se para en el elemento mayor que el pivote 9; `j` se para en el elemento menor, 5.

Figura 8.13 Algoritmo de partición: los elementos mal colocados 9 y 5 son intercambiados.

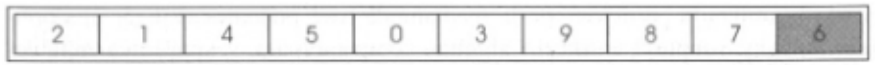


Figura 8.14 Algoritmo de partición: `i` se para en el elemento mayor que el pivote 9; `j` se para en el elemento menor, 3.

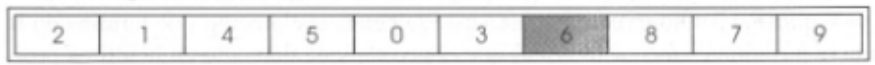


Figura 8.15 Algoritmo de partición: intercambio del pivote y el elemento en la posición `i`

8.6.5 Elementos iguales al pivote

Un detalle importante que debemos considerar es cómo tratar los elementos que son iguales que el pivote. Las preguntas que debemos hacernos son si `i` debería parar al encontrar un elemento igual al pivote, y si `j` debería parar al ver un elemento igual que el pivote. `i` y `j` deberían hacer lo mismo; de otra forma, el paso de partición sería desigual en ambos lados. Por ejemplo, si `i` parara y `j` no, todos los elementos iguales al pivote terminarían en la parte de la derecha.

Consideremos el caso en el que todos los elementos del vector son idénticos. Si paran tanto `i` como `j`, habrá muchos intercambios de elementos idénticos. Aunque esto parezca inútil, el efecto positivo que se consigue es que `i` y `j` se cruzan en la mitad del vector, por lo que, cuando el pivote es recolocado, la partición crea dos subconjuntos casi iguales. En consecuencia, se puede aplicar el análisis del caso mejor, y el tiempo de ejecución sería $O(N \log N)$.

Si no paran ni `i` ni `j`, entonces `i` termina en la última posición (suponiendo, por supuesto, que controlamos que no se salga de los límites) y no se realiza ningún intercambio. Esto parece bueno, dada la simetría, hasta que nos damos cuenta de que el pivote se mantendría en la última posición pues ésta es la celda apuntada por `i`. Así se obtendrían dos conjuntos completamente dispares, y el tiempo de ejecución sería el del caso peor, $O(N^2)$. El efecto es el mismo que el de utilizar el primer elemento del vector cuando éste está ordenado. Tardamos un tiempo cuadrático en no hacer nada.

Podemos concluir que es mejor hacer intercambios innecesarios y crear subconjuntos iguales, que arriesgarse a crear conjuntos desiguales. Por tanto, haremos que `i` y `j` paren cuando encuentre un elemento igual que el pivote. Ésta resulta ser la única de las cuatro posibilidades que no conduce a un tiempo cuadrático para este tipo de entrada.

A primera vista, parece que preocuparse por un vector con todos los elementos iguales es absurdo. Después de todo, ¿para qué querría alguien ordenar 5.000 elementos idénticos? Sin embargo, recuerde que quicksort es recursivo. Suponga que hay 100.000 elementos, de los cuales 5.000 son iguales. En algún momento quicksort podría hacer una llamada recursiva con sólo los 5.000 elementos idénticos. Y en tales ocasiones es importante asegurar que los 5.000 elementos se ordenan eficientemente.

8.6.6 Partición con la mediana de tres

Cuando se utiliza la partición con la mediana de tres, podemos realizar una optimización muy simple que ahorra algunas comparaciones y simplifica notablemente el código. En la Figura 8.16 volvemos a mostrar nuestro vector original.

La partición con la mediana de tres requiere que encontremos la mediana del primer elemento, el central y el último. La forma más fácil de hacerlo consiste en ordenarlos sobre las posiciones que ocupan en el vector. El resultado se muestra en la Figura 8.17. Observe el sombreado resultante: el elemento que termina en la primera posición es menor (o igual) que el pivote, y el elemento en la última posición es mayor (o igual) que el pivote. Partiendo de estos hechos concluimos lo siguiente:

8	1	4	9	6	3	5	2	7	0

Figura 8.16 Vector original.

0	1	4	9	6	3	5	2	7	8

Figura 8.17 Resultado de ordenar tres elementos (el primero, el central y el último).

0	1	4	9	7	3	5	2	6	8

Figura 8.18 Resultado del intercambio del pivote con el penúltimo elemento.

1. No deberíamos intercambiar el pivote con el elemento en la última posición. En vez de eso, deberíamos intercambiarlo con el que ocupa la penúltima posición. Esto se ilustra en la Figura 8.18.
2. `i` puede empezar en `ini+1` y `j` en `fin-2`.
3. Podemos estar seguros de que cuando `i` busca un elemento mayor parará, porque, en el peor caso, encontrará el pivote (parando al darse la igualdad).
4. Podemos estar seguros de que cuando `j` busca un elemento menor parará, porque, en el peor caso, encontrará el primer elemento (parando al darse igualdad).

Todas estas optimizaciones se incorporarán en nuestro código final del algoritmo en Java.

8.6.7 Vectores pequeños

Nuestra última optimización tiene que ver con los vectores pequeños. ¿Merece la pena utilizar una rutina tan potente como quicksort cuando sólo hay que ordenar diez elementos? Por supuesto, la respuesta es no. Una rutina simple, como la ordenación por inserción, será probablemente más rápida sobre vectores pequeños. La naturaleza recursiva de quicksort nos dice que se generarán muchas llamadas con subconjuntos pequeños. Por tanto, merece la pena preguntar por el tamaño del vector. Si es menor que un cierto límite, aplicaremos ordenación por inserción; en otro caso, seguiremos haciendo quicksort.

Se ha comprobado que diez elementos es un buen límite, aunque cualquier valor entre 5 y 20 produce resultados similares. El límite óptimo dependería de la máquina y del código máquina que se genere en cada caso. El utilizar un límite nos evita casos degenerados. Por ejemplo, no tiene sentido buscar la mediana de tres elementos cuando no hay ni siquiera tres elementos.

En el pasado se pensó que aún podía ser mejor dejar el vector algo desordenado no haciendo nada, de momento, cuando nos encontramos subvectores pequeños. Ya que la ordenación por inserción es muy eficiente para vectores casi ordenados, podríamos mostrar matemáticamente que una ejecución final de la ordenación por inserción para limpiar el vector sería más rápido que ejecutar todas

las llamadas a inserción con vectores pequeños. El ahorro correspondería a la sobrecarga producida por las llamadas al método de ordenación por inserción.

Hoy en día, las llamadas a métodos no son tan costosas como antes. Además, un segundo recorrido del vector para realizar la ordenación por inserción aporta un cierto coste. Debido a la técnica de gestión de memoria denominada *caching*, en los computadores actuales es mejor hacer la ordenación por inserción sobre los subvectores pequeños. Ello es debido a que los accesos a memoria localizados son más rápidos que los accesos no localizados; así, acceder a una misma posición de memoria dos veces casi seguidas en un recorrido, es más rápido que realizar un acceso en cada uno de los dos recorridos.

8.6.8 Rutina de quicksort en Java

La implementación concreta de quicksort se muestra en la Figura 8.19. El método `quicksort` visible, declarado en las líneas de la 46 a la 49, es simplemente un envoltorio que llama al método recursivo `quicksort`. En consecuencia, sólo discutiremos la implementación del método recursivo `quicksort`.

En la línea 7 se pregunta si el vector a ordenar es pequeño y se llama a la ordenación por inserción (no mostrada) cuando el tamaño del problema está por debajo de un valor concreto, dado por la constante `LIMITE`. En otro caso, se sigue con el procedimiento recursivo. Las líneas de la 12 a la 18 ordenan el primer elemento, el central y el último sobre sus propias posiciones. El elemento central se utiliza como pivote y en las líneas 21 y 22 se intercambia (utilizando `intercambiarReferencias`, que no se muestra) con el elemento en la última posición. Pasamos entonces a realizar la partición. Se inicializan los contadores `i` y `j` a una posición anterior a su valor inicial deseado, dado que los operadores prefijos de incremento y decremento los ajustarán inmediatamente en las líneas 28 y 30 antes de realizar ningún acceso al vector. Cuando el primer bucle `while` de la línea 28 termina, `i` estará apuntando a un elemento que es mayor, o posiblemente igual, que el pivote. De igual forma, cuando el segundo bucle termina, `j` apuntará a un elemento menor o igual que el pivote. Si `i` y `j` no se han cruzado, estos elementos son intercambiados, tras lo que se continúa el recorrido. En otro caso, se termina el recorrido, colocándose correctamente el pivote en la línea 39. La ordenación termina con las dos llamadas recursivas en las líneas 41 y 42.

Las operaciones fundamentales se realizan en las líneas 28 a 35. Los recorridos consisten en una serie de operaciones simples: incrementos, accesos al vector y comparaciones sencillas. Esto explica la rapidez de quicksort. Para asegurar la eficiencia de los bucles más internos, queremos estar seguros de que el intercambio de la línea 33 consta de las tres asignaciones esperadas y no incurre en la sobrecarga de una llamada a un método. Un buen compilador que optimice el código generado haría esto automáticamente, ya que todos los métodos en el paquete `Ordenacion` están declarados como `final`, lo que permite la optimización en línea. Sin embargo, podría ser necesario introducir manualmente el código, si el compilador no es capaz de realizar dicha optimización.

Aunque el código ahora parece sencillo, es importante no olvidar que ello se debe al análisis realizado antes de la implementación. Todavía hay algunas trampas escondidas (*véase* el Ejercicio 8.12). Quicksort es un ejemplo clásico de la utilización de un análisis previo para guiar la implementación del programa.

```
1  // Algoritmo quicksort que utiliza partición con
2  // la mediana de tres, y un límite para detectar vectores pequeños
3
4  private static void
5  quicksort( Comparable [ ] a, int ini, int fin )
6  {
7      if( ini + LIMITE > fin )
8          ordenacionPorInsercion( a, ini, fin );
9      else
10     {
11             // ordenar ini, medio, fin
12         int medio = ( ini + fin ) / 2;
13         if( a[ medio ].menorQue( a[ ini ] ) )
14             intercambiarReferencias( a, ini, medio );
15         if( a[ fin ].menorQue( a[ ini ] ) )
16             intercambiarReferencias ( a, ini, fin );
17         if( a[ fin ].menorQue( a[ medio ] ) )
18             intercambiarReferencias ( a, medio, fin );
19
20             // Colocar el pivote en la posición fin - 1
21         intercambiarReferencias ( a, medio, fin - 1 );
22         Comparable pivote = a[ fin - 1 ];
23
24             // Empezar la partición
25         int i, j;
26         for( i = ini, j = fin - 1; ; )
27         {
28             while( a[ ++i ].menorQue( pivote ) )
29                 ;
30             while( pivote.menorQue( a[ --j ] ) )
31                 ;
32             if( i < j )
33                 intercambiarReferencias ( a, i, j );
34             else
35                 break;
36         }
37
38             // Colocar el pivote
39         intercambiarReferencias ( a, i, fin - 1 );
40
41         quicksort( a, ini, i - 1 ); // Ordenar elementos pequeños
42         quicksort( a, i + 1, fin ); // Ordenar elementos grandes
43     }
44 }
45
46 public static void quicksort( Comparable [ ] a )
47 {
48     quicksort( a, 0, a.length - 1 );
49 }
```

Figura 8.19 Quicksort con partición con la mediana de tres y un límite para detectar vectores pequeños.

8.7 Selección rápida

Un problema muy relacionado con la ordenación es el problema de *selección*. En el mismo, dado un vector de N elementos, se debe encontrar el k-ésimo menor elemento. Un caso especial importante es el de encontrar la mediana, o sea el $N/2$-ésimo elemento. Obviamente, podemos hacerlo ordenando el vector, pero es de esperar que selección sea un proceso más rápido, al solicitarse una menor información. Esto es así, en efecto. Haciendo un pequeño cambio en quicksort, podemos resolver el problema de selección en tiempo lineal, en promedio. Llamamos a este algoritmo *selección rápida*. Los pasos que realiza *seleccionRápida*(S, k) son los siguientes:

1. Si el número de elementos en S es 1, entonces presumiblemente k también es 1, por lo que se puede devolver el único elemento en S.
2. Elegir un elemento v de S como el pivote.
3. Hacer una partición de $S - \{v\}$ en I y D, exactamente igual a como se hacía en quicksort.
4. Si k es menor o igual que el número de elementos en I, entonces el elemento que estamos buscando debe estar en I, por lo que se llama recursivamente a *seleccionRápida*(I, k). Si k es exactamente igual a uno más que el número de elementos de I, entonces el pivote es el k-ésimo menor elemento y se puede devolver como respuesta. En otro caso, el k-ésimo menor elemento estará en D. De nuevo podemos hacer una llamada recursiva y devolver el resultado obtenido.

La selección rápida precisa una única llamada recursiva, en comparación con las dos que hace quicksort. Pese a ello, el peor caso es idéntico al de quicksort, siendo, por tanto, cuadrático. Se presenta cuando la llamada recursiva se realiza con sólo un elemento menos. En tal caso, la selección rápida no ahorra gran cosa. Sin embargo, podemos mostrar que el tiempo es lineal en promedio, utilizando un análisis similar al de quicksort (*véase* el Ejercicio 8.9).

La implementación de la selección rápida es aún más simple de lo que nuestra descripción abstracta puede hacer pensar. Una implementación completa se muestra en la Figura 8.20. Excepto por el parámetro adicional k y la realización de una sola llamada recursiva, el algoritmo es idéntico al de quicksort. Obsérvese que cuando termina la ejecución del algoritmo, el k-ésimo menor elemento está en su posición correcta en el vector ordenado. Además, en las posiciones anteriores tendremos los $k - 1$ menores elementos del vector, aunque no ordenados, y en las $n - k$ últimas los más grandes. Únicamente hay que tener cuidado con el hecho de que al empezar a contar los índices en 0, el cuarto elemento, por ejemplo, estará en la posición tres, y no en la cuatro. Por otra parte, como el proceso se hace *in situ* el orden original del vector desaparece. Si no se desea este efecto lateral, debemos hacer que la rutina envoltorio pase una copia del vector.

Utilizando la partición con la mediana de tres se consigue que la probabilidad del caso peor sea casi despreciable. Eligiendo cuidadosamente el pivote, puede demostrarse que el caso peor nunca puede ocurrir, con lo que el tiempo de ejecución sería lineal aun en el peor de los casos. Sin embargo, el algoritmo resultante solamente tiene un interés teórico, pues la constante que oculta la notación O es mucho mayor que la constante del algoritmo ordinario que utiliza la partición con la mediana de tres.

```
1  // Método de selección interno que hace las llamadas recursivas.
2  // Coloca el k-ésimo menor elemento en a[k-1].
3  // La llamada inicial es seleccionRapida( a, 0, a.length, k );
4
5  private static void
6  seleccionRapida( Comparable [ ] a, int ini, int fin, int k )
7  {
8      if( ini + LIMITE > fin )
9          ordenacionPorInsercion( a, ini, fin );
10     else
11     {
12             // ordenar ini, medio, fin
13         int medio = ( ini + fin ) / 2;
14         if( a[ medio ].menorQue( a[ ini ] ) )
15             intercambiarReferencias( a, ini, medio );
16         if( a[ fin ].menorQue( a[ ini ] ) )
17             intercambiarReferencias ( a, ini, fin );
18         if( a[ fin ].menorQue( a[ medio ] ) )
19             intercambiarReferencias ( a, medio, fin );
20
21             // Colocar el pivote en la posición fin - 1
22         intercambiarReferencias ( a, medio, fin - 1 );
23         Comparable pivote = a[ fin - 1 ];
24
25             // Empezar la partición
26         int i, j;
27         for( i = ini, j = fin - 1; ; )
28         {
29             while( a[ ++i ].menorQue( pivote ) )
30                 ;
31             while( pivote.menorQue( a[ --j ] ) )
32                 ;
33             if( i < j )
34                 intercambiarReferencias ( a, i, j );
35             else
36                 break;
37         }
38
39             // Colocar el pivote
40         intercambiarReferencias ( a, i, fin - 1 );
41
42             // Recursión: sólo cambia esta parte
43         if( k < i )
44             seleccionRapida( a, ini, i - 1, k );
45         else if( k > i )
46             seleccionRapida( a, i + 1, fin, k );
47     }
48 }
```

Figura 8.20 Selección rápida con partición con la mediana de tres y un límite para detectar vectores pequeños

8.8 Una cota inferior para la ordenación

Aunque tenemos algoritmos de ordenación $O(N \log N)$, no está claro que esto sea lo mejor que podemos conseguir. En esta sección probamos que cualquier algoritmo de ordenación que sólo utilice comparaciones necesita $\Omega(N \log N)$ comparaciones (y por tanto al menos el mismo tiempo) en el caso peor. Esto significa que *cualquier algoritmo que ordene utilizando comparaciones entre elementos debe utilizar no menos de unas $N \log N$ comparaciones para alguna entrada*. Se puede utilizar una técnica similar para probar que esto también es cierto en promedio.

¿Todo método de ordenación debe basarse en comparaciones? La respuesta es no. Sin embargo, los algoritmos que no utilizan comparaciones sólo funcionan, en principio, con tipos restringidos, como los enteros. Aunque puede que a menudo sólo necesitemos ordenar enteros (*véase* el Ejercicio 8.13), un algoritmo de ordenación de propósito general no puede hacer asunciones tan restrictivas sobre la entrada. Sólo puede asumir la obvia de que si los elementos han de ser ordenados, cualesquiera dos elementos pueden ser comparados.

Probamos a continuación uno de los teoremas fundamentales de la computación (Teorema 8.3). Antes de comenzar recuerde que el producto de los N primeros enteros positivos es $N!$. La demostración es una demostración existencial, un tanto abstracta. Demuestra que siempre debe existir alguna entrada mala.

> *Cualquier algoritmo que ordene utilizando únicamente comparaciones entre elementos debe usar al menos $\lceil \log(N!) \rceil$ comparaciones, con alguna secuencia de entrada.*
>
> *Podemos considerar como posibles entradas las permutaciones de 1, 2, ..., N. Esto es así pues sólo importa el orden relativo de los elementos, no su valor concreto. Por tanto, el número de posibles entradas es el número de permutaciones diferentes de N elementos, que es exactamente N!. Denotemos por P_i el máximo número de permutaciones que pueden ser consistentes con la información obtenida después de que el algoritmo ha procesado i comparaciones. Sea F el número de comparaciones realizadas cuando termina la aplicación del algoritmo. Se tiene lo siguiente: (a) $P_0 = N!$ ya que todas las permutaciones son posibles antes de la primera comparación; (b) $P_F = 1$, ya que si fueran posibles más de una permutación, el algoritmo no podría terminar todavía, al no poder estar seguros de que haya producido el resultado correcto; (c) $P_i \geq P_{i-1}/2$, porque después de una comparación, cada permutación va a uno de estos dos grupos: el grupo de las aún posibles y el grupo de las que ya no son posibles. El mayor de estos dos grupos debe contener al menos la mitad de las permutaciones, de modo que si el resultado de la última comparación realizada corresponde a dicho grupo mayor, el tamaño del conjunto de permutaciones consistentes sólo podría dividirse por la mitad tras cada comparación. El efecto del algoritmo de ordenación es, por tanto, ir desde el estado P_0, en el cual son posibles todas las N! permutaciones, al estado final P_F, en el cual sólo es posible una permutación, con la restricción de que existe una entrada tal que en cada comparación, sólo se pueden eliminar la mitad de las permutaciones. Por el principio de sucesivas divisiones por la mitad, sabemos que al menos se necesitan $\lceil \log(N!) \rceil$ comparaciones para esa entrada.*

¿Cómo de grande es $\lceil \log(N!) \rceil$? Es aproximadamente $N \log N - 1{,}44N$.

Resumen

En la mayoría de las aplicaciones generales de ordenación interna, el método elegido es la ordenación por inserción, Shellsort o quicksort. La decisión de cuál usar depende del tamaño de la entrada.

La ordenación por inserción es apropiada para pequeñas cantidades de datos. Shellsort es una buena elección para ordenar cantidades moderadas de datos. Con una secuencia de incrementos apropiada, tiene un excelente rendimiento, necesitando sólo unas pocas líneas de código. Quicksort tiene el mejor rendimiento, pero es complicado de implementar. Asintóticamente, tiene un rendimiento cercano a $O(N \log N)$ cuando se usa una implementación cuidadosa. Hemos visto que dicho rendimiento es esencialmente tan bueno como cabe esperar. La Sección 20.5 estudia otro método interno muy común, *heapsort* (método del montículo).

Cuando queremos ordenar datos que no caben juntos en memoria principal, se necesitan técnicas diferentes. La técnica general se discute en la Sección 20.6. Utiliza el algoritmo `mezclar` descrito en la Sección 8.5.

Para probar y comparar los méritos de los diferentes algoritmos de ordenación, necesitamos ser capaces de generar entradas aleatorias. La aleatoriedad es un tema importante en general que se estudia en el capítulo siguiente.

Elementos del juego

algoritmo de ordenación basado en comparaciones Un algoritmo que toma decisiones sobre el orden utilizando sólo comparaciones.

demostración de la cota inferior de la ordenación Confirma que cualquier algoritmo de ordenación basado en comparaciones debe usar al menos unas $N \log N$ comparaciones, tanto en promedio como en el caso peor.

inversión Un par no ordenado de elementos de un vector. Utilizadas para medir el grado de desorden.

mergesort Método de ordenación que utiliza divide y vencerás para obtener una ordenación $O(N \log N)$.

ordenación por disminución de intervalos Otro nombre para Shellsort.

partición con la mediana de tres Se utiliza como pivote la mediana del primer elemento, el central y el último. Esto simplifica la fase de partición de quicksort.

partición Paso de quicksort que coloca cada elemento del vector, excepto el pivote, en uno de dos posibles grupos, uno formado por los elementos menores que el pivote y otro formado por los mayores que él.

pivote En quicksort, el elemento que sirve para dividir el vector en dos grupos: uno con los elementos menores que él y otro con los elementos mayores.

quicksort Un algoritmo divide y vencerás rápido, cuando se implementa apropiadamente. Es, de hecho, el algoritmo de ordenación basado en comparaciones más rápido en la práctica.

selección Proceso para buscar el k-ésimo menor elemento de un vector.

selección rápida Algoritmo utilizado para realizar una selección, similar a quicksort, pero con una sola llamada recursiva. El tiempo de ejecución en promedio es lineal.

Shellsort Algoritmo subcuadrático que funciona bien en la práctica y es simple de implementar. El rendimiento de Shellsort depende en gran media de la secuencia de incrementos en que se base. Conlleva un análisis desafiante, todavía no completamente resuelto.

Errores comunes

1. Los algoritmos de ordenación implementados en este capítulo trabajan a partir de la componente 0 del vector, y no de la 1.
2. Utilizar una secuencia de incrementos inadecuada es un error común al usar Shellsort. Asegúrese de que la secuencia acaba en 1, y evite secuencias de las que se sabe que producen malos resultados.
3. Quicksort tiene multitud de trampas. Los errores más comunes tienen que ver con entradas ordenadas, elementos duplicados y particiones degeneradas.
4. El algoritmo de ordenación por inserción es apropiado para pequeñas entradas. Pero no es bueno con entradas grandes.

En Internet

Todos los algoritmos de ordenación, y una implementación de la selección rápida, se encuentran en un único fichero, como parte del paquete `DataStructures`, traducido como `EstructurasDatos`.

DuplicateTest.java	Contiene el método de la Figura 8.1 con un programa de prueba, en el directorio **Chapter08**.
Sort.java	Traducida por `Ordenacion.java`, en el directorio **DataStructures.**
TestSort.java	Clase en el directorio **Chapter08** que prueba todos los métodos de ordenación.

Ejercicios

Cuestiones breves

8.1. Ordene la secuencia (8, 1, 4, 1, 5, 9, 2, 6, 5) utilizando los siguientes métodos:

a) Ordenación por inserción.
b) Shellsort, con los incrementos (1, 3, 5).
c) Mergesort.
d) Quicksort, utilizando el elemento central como pivote y sin límite (muestre todos los pasos).
e) Quicksort, utilizando la mediana de tres como pivote y un límite de tres.

8.2. Un algoritmo de ordenación es *estable* si los elementos iguales permanecen en su orden original. ¿Cuáles de los algoritmos de ordenación vistos en este capítulo son estables y cuáles no? ¿Por qué?

8.3. Explique por qué el algoritmo quicksort del texto es típicamente mejor que permutar aleatoriamente la entrada y elegir el elemento central como pivote.

Problemas teóricos

8.4. Cuando todos los elementos son iguales, ¿cuál es el tiempo de ejecución de los siguientes métodos?

a) Ordenación por inserción.
b) Shellsort.
c) Mergesort.
d) Quicksort.

8.5. Cuando la entrada está ya ordenada, ¿cuál es el tiempo de ejecución de los siguientes métodos?

a) Ordenación por inserción.
b) Shellsort.
c) Mergesort.
d) Quicksort.

8.6. Cuando la entrada está originalmente ordenada, pero en orden inverso, ¿cuál es el tiempo de ejecución de los siguientes métodos?

a) Ordenación por inserción.
b) Shellsort.
c) Mergesort.
d) Quicksort.

8.7. Suponga que intercambiamos los elementos `a[i]` y `a[i+k]`, que originalmente estaban fuera de sitio. Demuestre que con ello al menos se elimina una inversión, y como mucho $2k - 1$.

8.8. Construya una entrada que provoque el caso peor para quicksort en los siguientes casos:

a) Utilizando el elemento central como pivote.
b) Haciendo partición con la media de tres.

8.9. Demuestre que la selección rápida tiene un rendimiento lineal en promedio. Hágalo resolviendo la Ecuación 8.5, con la constante 2 sustituida por 1.

8.10. Utilizando la fórmula de Stirling, $N! \geqslant (N/e)^N \sqrt{2\pi N}$, deduzca una estimación para $\log(N!)$.

8.11. Demuestre que cualquier algoritmo basado en comparaciones, utilizado para ordenar cuatro elementos, necesita 5 comparaciones para alguna entrada. Demuestre después que existe un algoritmo capaz de ordenar secuencias de cuatro elementos cualesquiera utilizando a lo sumo 5 comparaciones.

Problemas prácticos

8.12. Un estudiante modifica la rutina de la Figura 8.19 realizando los siguientes cambios en las líneas de la 26 a la 31.

```
for( i = ini + 1, j = fin - 2; ; )
{
    while( a[ i ].menorQue( pivote ) )
        i++;
    while( pivote.menorQue( a[ j ] ) )
        j--;
}
```

¿Es el resultado equivalente a la rutina original?

8.13. Si se tiene información específica adecuada sobre los elementos a ordenar, la ordenación puede pasar a ser lineal. Demuestre que una colección de N enteros `short` puede ser ordenada en un tiempo lineal. *Pista*: mantenga un vector indexado desde 0 hasta 65.535.

8.14. Tenemos un vector que contiene N números. Queremos saber si contiene dos números cuya suma sea igual a un número dado K. Por ejemplo, si la entrada es (8, 4, 1, 6) y K es 10, la respuesta es sí (4 y 6). Un mismo número puede ser usado dos veces. Haga lo siguiente:

a) De un algoritmo $O(N^2)$ para resolver el problema.
b) De un algoritmo $O(N \log N)$ para resolver el problema. *Pista*: ordene primero los elementos. Después de hacer esto puede resolver el problema en tiempo lineal.
c) Escriba un programa en Java para cada una de dichas soluciones y compare sus tiempos de ejecución.

8.15. Repita el Ejercicio 8.14 para cuatro números. Intente diseñar un algoritmo $O(N^2 \log N)$. *Pista*: calcule todas las posibles sumas de dos elementos, ordénelas y continúe como en el Ejercicio 8.14.

8.16. Repita el Ejercicio 8.14 para tres números. Intente diseñar un algoritmo $O(N^2)$.

8.17. El Ejercicio 5.25 pedía buscar la única solución entera a la ecuación $A^5 + B^5 + C^5 + D^5 + E^5 = F^5$ que satisface $0 < A \leq B \leq C \leq D \leq E \leq F \leq N$, donde N es 75. Use las ideas del Ejercicio 8.15 para obtener relativamente deprisa una solución, ordenando todos los posibles valores de $A^5 + B^5 + C^5$ y $F^5 - (D^5 + E^5)$, y viendo después si algún número del primer grupo aparece también en el segundo grupo. En términos de N, ¿cuánto espacio y tiempo requiere su algoritmo?

Prácticas de programación

8.18. Compare el rendimiento de Shellsort con varias secuencias de incrementos, siguiendo el siguiente procedimiento: obtenga experimentalmente el tiempo medio para un cierto tamaño de entrada N, generando varias secuencias aleatorias de N elementos. Utilice la misma entrada para todas las secuencias de incrementos. En un test por separado obtenga el número medio de comparaciones entre objetos `Comparable`, y el número de asignaciones de referencias a objetos `Comparable`. Haga un gran número de pruebas, pero sin que sea necesario, pongamos, más de una hora de tiempo de CPU. Las secuencias de incrementos son las siguientes:

a) La secuencia original de Shell (dividir repetidamente por 2).
b) La secuencia original de Shell, sumando 1 si el resultado es distinto de cero, pero par.
c) La secuencia de Gonnet mostrada en el texto, en la que repetidamente se divide por 2,2.
d) Los incrementos de Hibbard: 1, 3, 7, ..., $2^k - 1$.
e) Los incrementos de Knuth: 1, 4, 13, ... $(3^k - 1)/2$.
f) Los incrementos de Sedgewick: 1, 5, 19, 41, 109, ...; cada término es bien de la forma $9 \cdot 4^k - 9 \cdot 2^k + 1$, o de la forma $4^k - 3 \cdot 2^k + 1$.

8.19. Implemente los métodos de ordenación de Shellsort y quicksort y compare sus tiempos de ejecución. Use las implementaciones del texto. Ejecútelos con los siguientes objetos:

a) Enteros.
b) Números reales de tipo `double`.
c) Cadenas de caracteres.

8.20. Escriba un método que elimine todos los elementos repetidos de un vector A de N elementos. Devuelva el número de elementos que permanecen en A. Su método debe ejecutarse en promedio en tiempo $O(N \log N)$, (utilice quicksort como paso de preprocesamiento).

8.21. El Ejercicio 8.2 estudia la ordenación estable. Escriba un método genérico que realice un quicksort estable. Para hacerlo, se crea un vector en el que cada entrada contenga una referencia a un elemento y su posición inicial en el vector. Se ordena este vector; si dos entradas tienen elementos de datos idénticos, se utiliza la posición inicial para romper el empate. Cuando el vector esté ordenado se reordena el vector original.

8.22. Escriba una utilidad simple de ordenación, `Ordenar`. El comando `Ordenar` toma como parámetro el nombre de un fichero, que contiene un elemento por línea. Por defecto, las líneas se consideran como cadenas de caracteres y se ordenan mediante el orden lexicográfico habitual (aunque distinguiendo entre mayúsculas y minúsculas). Añada dos opciones: la opción `-c` significa que no se distinga entre mayúsculas y minúsculas, y la opción `-n` significa que las líneas deben ser consideradas como enteros.

8.23. Suponga que tiene K ficheros ordenados. Cada línea contiene un objeto `Linea`, donde `Linea` es un interfaz que implementa el interfaz `Comparable` y especifica un método `obtenerLinea`, que lee la línea dentro del objeto.

a) Escriba un método `leerFichero` que lea los elementos de un solo fichero y los introduzca en una cola creada por el método. Se devolverá una referencia a dicha cola. Observe que los elementos en la cola estarán ordenados. La declaración es

```
Cola leerFichero( String nombreFichero );
```

b) Escriba un método denominado `mezclarDos` que tome dos colas y cree una tercera con el resultado de la mezcla. Debe devolver una referencia a la nueva cola creada, y haber vaciado las dos colas originales. La declaración es:

```
Cola mezclarDos( Cola c1, Cola c2 );
```

c) El algoritmo para producir un fichero ordenado es el siguiente. Declare una cola de colas `losElementos`. Llame a `leerFichero` con cada uno de los K ficheros e inserte los resultados devueltos en `losElementos`. Elimine dos elementos de `losElementos`, mézclelos, e introduzca el resultado en la cola, $K - 1$ veces. El resultado será que `losElementos` contendrá una única cola ordenada que contendrá todos los elementos de los ficheros. Escriba el correspondiente método.

d) Escriba un programa completo que implemente la mezcla de K ficheros ordenados.

e) Muestre que declarar `losElementos` como una pila de colas puede conducir a un mal rendimiento. *Pista*: considere el caso en el cual el primer fichero tiene N elementos y todos los demás tienen un solo elemento.

Bibliografía

La referencia clásica para los algoritmos de ordenación es [5]. Una referencia más reciente, completada con resultados puestos al día, es [3]. El algoritmo de Shellsort apareció por primera vez en [7]. Un estudio empírico de su tiempo de ejecución fue realizado en [8]. Quicksort fue descubierto por Hoare [4]. El artículo también incluye el algoritmo de selección rápida, y detalla muchas de las cuestiones de implementación. Un minucioso estudio del algoritmo de quicksort, incluyendo análisis para la variante con la mediana de tres, aparece en [6]. Una implementación detallada en C que incluye mejoras adicionales puede encontrarse en [1]. La cota inferior $\Omega(N \log N)$ para los algoritmos de ordenación basados en comparaciones se ha tomado de [2]. La presentación de Shellsort se ha adaptado de [9].

1. J. L. Bentley y M. D. McElroy, «Engineering a Sort Function», *Software-Practice and Experience* **23** (1993), 1249-1265.
2. L. R. Ford y S. M. Johnson, «A Tournament Problem», *American Mathematics Monthly* **66** (1959), 387-389.
3. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2.ª ed., Addison-Wesley, Reading, Mass. (1991).
4. C. A. R. Hoare, «Quicksort», *Computer Journal* **5** (1962), 10-15.
5. D. E. Knuth, *The Art of Computer Programming, Volume* 3: *Sorting and Searching*, 2.ª ed., Addison-Wesley, Reading, Mass. (1997).
6. R. Sedgewick, *Quicksort*, Garland Publishing, New York, NY (1978). (Originally presented as the author's Ph.D. thesis, Stanford University, 1975.)
7. D. L. Shell, «A High-Speed Sorting Procedure», *Communications of the ACM* **2** 7 (1959), 30-32.
8. M. A. Weiss, «Empirical Results on the Running Time of Shellsort», *Computer Journal* **34** (1991), 88-91.
9. M. A. Weiss, *Efficient C Programming*: *A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1995).

CAPÍTULO 9

Números aleatorios

En muchas ocasiones en la computación se requiere el uso de números aleatorios. Por ejemplo, la criptografía moderna, los sistemas de simulación y, sorprendentemente, incluso algunos algoritmos de búsqueda y ordenación, se basan en la generación de números aleatorios. Sin embargo, no es tan fácil implementar buenos generadores de números aleatorios. Por todo ello, en este capítulo estudiaremos la generación y el uso de números aleatorios.

Más concretamente, en este capítulo veremos:

- Cómo se generan números aleatorios.
- Cómo se generan permutaciones aleatorias.
- Cómo los números aleatorios nos permiten diseñar algoritmos eficientes, mediante el uso de una técnica conocida como *algoritmos aleatorios*.

9.1 ¿Por qué son necesarios los números aleatorios?

Los números aleatorios se emplean en numerosas aplicaciones. Esta sección discute varias de las más comunes.

Una importante aplicación de los números aleatorios es la comprobación de programas (*testing*). Supongamos, por ejemplo, que deseamos comprobar si los algoritmos de búsqueda diseñados en el Capítulo 8 son correctos. Es fácil probarlos tomando entradas con pocos datos, pero si deseamos comprobar su corrección para las entradas de gran tamaño para las que han sido diseñados, necesitamos suministrar un montón de datos. Generar una entrada de datos ordenada, como por ejemplo la secuencia 1, 2, ..., *N*, sería sencillo, pero con ello sólo comprobaríamos el buen funcionamiento de los algoritmos en un caso límite. Estos algoritmos necesitan tests generales, por medio de los cuales obtengamos comprobaciones más convincentes. Por ejemplo, podríamos probar los programas haciendo 5.000 ordenaciones sobre entradas de tamaño 1.000. Esto requiere la escritura de una rutina para generar datos arbitrarios, para lo cual se necesita a su vez, la generación de números aleatorios.

Continuando con el ejemplo, una vez que tenemos los datos generados aleatoriamente, ¿cómo podemos saber si el algoritmo ha funcionado correctamente? Una condición necesaria es que el resultado de la ordenación sea un vector ordenado en orden no decreciente, lo que puede ser comprobado en tiempo lineal. Pero, ¿cómo sabemos que los elementos de la salida son exactamente los elementos de la entrada? Una forma de comprobarlo es exigir que la entrada sea una permutación de los N primeros enteros. Recuérdese del Capítulo 8, que una *permutación* de 1, 2, ..., N es una secuencia de N enteros en la que cada uno de ellos aparece (exactamente) una vez. De este modo, no importa con qué permutación comencemos, el resultado de la ordenación debe ser la secuencia 1, 2, ..., N, lo que se comprueba fácilmente.

Además de ayudarnos a generar datos de entrada para comprobar la corrección de los programas, los números aleatorios nos son útiles para comparar la eficiencia de distintos algoritmos. Esto es debido, una vez más, al hecho de que pueden emplearse para generar multitud de entradas representativas.

Otra utilidad de los números aleatorios se encuentra en la simulación. Si deseamos conocer el tiempo medio que tarda un servidor (como el servicio de información de un banco) en procesar una secuencia de peticiones, podemos modelar el sistema empleando un computador. En esta simulación por computador, la secuencia de peticiones se genera empleando números aleatorios.

Aún queda otro uso importante de los números aleatorios, que corresponde a la técnica de los *algoritmos aleatorios*. En ellos se emplea un número aleatorio para determinar (de forma probabilística) el siguiente paso que será ejecutado por el algoritmo. La clase más común de estos algoritmos se basa en la selección (aleatoria) de una alternativa entre varias, más o menos indistinguibles. Por ejemplo, en los programas comerciales de ajedrez, el computador generalmente elige su primer movimiento de forma aleatoria, en lugar de jugar de forma determinista (es decir, en lugar de realizar siempre el mismo movimiento). Este capítulo examina muchos de los problemas que pueden resolverse de modo más eficiente empleando algoritmos aleatorios.

9.2 Generadores de números aleatorios

¿Cómo se generan números aleatorios? La verdadera aleatoriedad es imposible de alcanzar en un computador, ya que los números obtenidos dependen del algoritmo empleado en su generación y esto los convierte en no aleatorios. Ahora bien, generalmente, es suficiente producir *números pseudoaleatorios*, esto es, números que *parecen* aleatorios, en el sentido de que satisfacen muchas de las propiedades que cumplen los números aleatorios. Aunque esto es mucho más fácil de decir que de conseguirlo.

Supongamos que necesitamos simular el lanzamiento de una moneda. Una forma de hacerlo es examinar el reloj del sistema. Este reloj indica el número de segundos de la hora actual. Si esta cantidad es par podemos devolver 0 (cara), y si es impar podemos devolver 1 (cruz). El problema es que esta estrategia no funciona correctamente si necesitamos generar una secuencia de números aleatorios. En tal caso, lo más probable es que se genere una secuencia de todo ceros o todo unos y, evidentemente, esto dista mucho de ser aleatorio. Un segundo es, para el computador, mucho tiempo, de modo que probablemente el reloj no cambiará mientras la secuencia está generándose. Incluso si el tiempo se mide en milisegundos y el programa está haciendo más cosas entre las llamadas al generador, la secuencia de números generada no sería aleatoria, ya que el tiempo transcurrido entre las llamadas al generador sería prácticamente idéntico en cada invocación del programa, con lo que los resultados que se producirían al realizar distintas ejecuciones serían demasiado parecidas.

Lo que necesitamos es una *secuencia* de números pseudoaleatorios, es decir, una secuencia con las mismas propiedades que una secuencia aleatoria. Supongamos que necesitamos generar números aleatorios entre 0 y 999, uniformemente distribuidos. En una *distribución uniforme*, todos los números en el rango especificado tienen la misma probabilidad de ser elegidos. Muchas distribuciones pueden derivarse a partir de la distribución uniforme, por esta razón se considera en primer lugar. En el esquema de clase mostrado en la Figura 9.1 aparecen muchas de ellas. Las siguientes propiedades son verificadas por toda distribución uniforme de valores del intervalo 0 ... 999:

- El primer número puede ser cualquiera entre 0 y 999. Todos ellos son equiprobables.
- El número i-ésimo puede ser cualquiera entre 0 y 999. De nuevo, todos son equiprobables.
- La media de todos los números generados es 499,5.

Estas propiedades por sí mismas no son suficientemente restrictivas. Por ejemplo, podríamos generar el primer número examinando el reloj del sistema, cuya precisión es de un milisegundo, empleando el número de milisegundos. Los siguientes valores se irían obteniendo añadiendo una unidad al número anterior. Es claro que tras generar mil números, se verifican todas las propiedades anteriores. Sin embargo, no se cumplen otras más fuertes. Algunas de estas propiedades que también deben verificar los números aleatorios uniformemente distribuidos son las siguientes:

- La suma de dos números aleatorios generados consecutivamente debe ser par o impar con la misma probabilidad.
- Si se generan aleatoriamente mil números, algunos de ellos estarán duplicados (en concreto, aproximadamente 368 de ellos no aparecerán).

Nuestros valores no satisfacen estas propiedades. La suma de dos números consecutivos siempre es impar, y nuestra secuencia está libre de duplicaciones. Como consecuencia, nuestro generador de números pseudoaleatorios no ha pasado dos pruebas estadísticas. Es bien cierto que todos los generadores de este tipo acaban por no pasar algún test estadístico, aunque los mejores sí que pasan los tests que fallan los menos buenos. El Ejercicio 9.14 describe un test estadístico muy utilizado en la práctica.

En esta sección estudiaremos el generador uniforme más sencillo que satisface un número razonable de tests estadísticos. Desde luego no es el mejor generador posible, pero resulta apropiado para ser empleado en aplicaciones en las que se aceptan buenas aproximaciones de secuencias aleatorias. El método seguido es el *generador lineal de congruencias*, definido por primera vez en 1951. Dicho gene-

```
// Clase Random
//
// CONSTRUCCIÓN: con (a) ninguna inicialización o (b) un entero
//     que especifica el estado inicial del generador
//
// ******************OPERACIONES PÚBLICAS**********************
//     Devuelve un número aleatorio siguiendo una cierta
//     distribución
// int randomInt( )                    --> Uniforme, 1 a 2^31-1
// double randomReal( )                --> Uniforme, 0..1
// int randomInt ( int linf, int lsup ) --> Uniforme linf..lsup
// int poisson( double valorEsperado ) --> Poisson
// double negExp( double valorEsperado ) --> Exponencial
//     Un método estático relacionado:
// void permute( Object [ ] a )        --> Permutación aleatoria

/**
 * Clase de números aleatorios que emplea un
 * generador lineal de congruencias 31-bit.
 * Obsérvese que java.util contiene una clase Random,
 * por lo que se deben vigilar los conflictos de nombres.
 */
public class Random
{
    public Random( )
    { /* Figura 9.2 */ }
    public Random( int valorInicial )
    { /* Figura 9.2 */ }
    public int randomInt( )
    { /* Figura 9.3 */ }
    public double randomReal( )
    { return randomInt( ) / ( double ) M; }
    public int randomInt( int linf, int lsup )
    { /* Figura 9.8 */ }
    public int poisson( double valorEsperado )
    { /* Figura 9.5 */ }
    public double negExp( double valorEsperado )
    { /* Figura 9.6 */ }
    public static final void permute( Object [ ] a )
    { /* Figura 9.7 */ }

    private int estado;
}
```

Figura 9.1 Esquema de una clase generadora de números aleatorios.

rador es un generador de números aleatorios en el que son generados los valores $X_1, X_2, \ldots$, verificándose

$$X_{i+1} = AX_i (\text{mod } M). \tag{9.1}$$

Esta ecuación indica que el $(i+1)$-ésimo número se obtiene multiplicando el i-ésimo número por una cierta constante A, para calcular después el resto de dividir el resultado por M. En Java lo escribiríamos en la forma siguiente:

```
x[ i+1 ] = A * x[ i ] % M;
```

Los valores adecuados de las constantes A y M se concretarán en breve. Nótese que todos los números generados serán menores que M. Para iniciar la secuencia necesitamos un valor X_0. Este valor se conoce como *semilla*. Si $X_0 = 0$ la secuencia no sería aleatoria, ya que se generarían únicamente ceros. Pero si A y M son cuidadosamente elegidos, entonces cualquier semilla que cumpla $1 \leqslant X_0 < M$ es válida. En la Figura 9.2 se construye un objeto `Random` en el que se inicializa el valor de la semilla. Si M es primo entonces X_i no puede ser 0. En particular, si $M = 11$, $A = 7$ y la semilla $X_0 = 1$, los números generados son

7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2, ...

Generar un número cada segundo produce una secuencia repetitiva. En nuestro caso, la secuencia se repite después de $M - 1 = 10$ números. La longitud de la secuencia hasta que se encuentra un número generado anteriormente se conoce como *periodo* de la secuencia. El periodo obtenido con esta elección de A es, claramente, el mejor posible ya que todos los números no nulos menores que M son generados. (Necesariamente debemos tener un número repetido en la 11-ésima iteración.)

Si M es primo, distintas elecciones de A permiten generar un periodo completo de $M - 1$ valores. Esta clase de generadores reciben el nombre de *generadores lineales de congruencias con periodo completo*. Otras elecciones de A no generan un periodo completo. Por ejemplo, si $A = 5$ y $X_0 = 1$, la secuencia tiene como periodo 5, al obtenerse:

5, 3, 4, 9, 1, 5, 3, 4, ...

Si se elige como valor de M un primo 31-bit muy alto, el periodo es suficientemente alto para la mayoría de las aplicaciones. El primo $M = 2^{31} - 1 = 2.147.483.647$

```
/**
 * Construcción de un objeto Random cuyo
 * estado inicial se obtiene del reloj del sistema.
 */
public Random( )
{
    this( (int) ( System.currentTimeMillis( )
                  % Integer.MAX_VALUE ) );
}

/**
 * Construcción de un objeto Random
 * especificando un estado inicial.
 * @param valorInicial el estado inicial.
 */
public Random( int valorInicial )
{
    if( valorInicial < 0 )
        valorInicial += M;

    estado = valorInicial;
    if( estado == 0 )
        estado = 1;
}
```

Figura 9.2 Constructores de la clase `Random`.

es una elección usual. Para este número, $A = 48.271$ es uno de los muchos valores que permiten obtener un generador lineal de congruencias con periodo completo. Su utilización ha sido ampliamente estudiada, siendo recomendada por los expertos en este campo. Como mostraremos más tarde en este mismo capítulo, jugar con generadores de números aleatorios significa en muchos casos romper la aleatoriedad, por lo que conviene emplear la fórmula indicada, a menos que tengamos garantizado que es posible obrar de otra forma.

Esta rutina parece sencilla de implementar. Si `estado` representa el último valor calculado por la rutina `Random` entonces el nuevo valor de `estado` viene definido por

```
estado = ( A * estado ) % M;        // Incorrecto
```

Desafortunamente, si este cálculo se realiza con enteros 32-bit, es casi seguro que el producto provocará un error de desbordamiento. Aunque Java tiene definido un tipo `long` 64-bit, emplearlo es computacionalmente costoso. Si nos restringimos al tipo `int` 32-bit, podríamos argumentar que el desbordamiento, una vez anulado el correspondiente error, mantiene la aleatoriedad. Sin embargo, el desbordamiento es inaceptable, pues con él perdemos la garantía del periodo completo. Afortunadamente, para evitar el desbordamiento sólo es necesaria una pequeña modificación. En concreto, si Q y R son el cociente y el resto de M/A, podemos rescribir la Ecuación 9.1 en la forma

$$X_{i+1} = A(X_i(\text{mod } Q)) - R\lfloor X_i/Q \rfloor + M\delta(X_i) \qquad \textbf{(9.2)}$$

con lo que se verifica lo siguiente (*véase* el Ejercicio 9.5):

- El primer término siempre puede evaluarse sin desbordamiento.
- El segundo término puede evaluarse sin desbordamiento, siempre que $R < Q$.
- $\delta(X_i)$ se evalúa a 0 si el resultado de la diferencia entre los dos primeros términos es positiva; se evalúa a 1 si dicha diferencia es negativa.

Para los valores de M y A fijados, se tiene que $Q = 44.488$ y $R = 3.399$. Como consecuencia, $R < Q$ y una aplicación directa de (9.2) nos da una implementación de una clase de números aleatorios. El código que resulta se muestra en la Figura 9.3. En ella, la rutina `randomInt` devuelve el valor de `estado`.

En la Figura 9.1 aparecen dos métodos adicionales: uno de ellos genera aleatoriamente un real en el intervalo de 0 a 1, mientras que el segundo genera un entero aleatorio en el intervalo cerrado que se especifica como argumento (*véase* el Ejercicio 9.8).

Por último, la clase también incluye un generador para el caso en el que se necesiten números aleatorios no uniformemente distribuidos. Al respecto, en la Sección 9.3 se muestra el código de los métodos `poisson` y `negExp`.

Se puede estar tentado a asumir que cualquier sistema computacional tiene un generador de números aleatorios al menos tan bueno como el de la Figura 9.3. Tristemente, esto no es cierto. Muchas librerías poseen generadores basados en la función

$$X_{i+1} = (AX_i + C) \text{mod } 2^B$$

```
1  private static final int A = 48271;
2  private static final int M = 2147483647;
3  private static final int Q = M / A;
4  private static final int R = M % A;
5
6  /**
7   * Devuelve un entero pseudoaleatorio y cambia el
8   * estado interno.
9   * @return un entero pseudoaleatorio.
10  */
11 public int randomInt( )
12 {
13     int estadoTmp = A * ( estado % Q ) - R * ( estado / Q );
14     if( estadoTmp >= 0 )
15         estado = estadoTmp;
16     else
17         estado = estadoTmp + M;
18
19     return estado;
20 }
```

Figura 9.3 Generador de números aleatorios.

donde B es el número de bits de los enteros de la máquina y C es impar. Estas librerías, al igual que la rutina de la Figura 9.3, también devuelven directamente el nuevo valor calculado en `estado`, en lugar de (por ejemplo) un valor entre 0 y 1. Desafortunadamente, estos generadores siempre producen valores de X_i que se alternan entre pares e impares —propiedad indeseable en un generador—. Más en general, cada sufijo de k bits cicla con un periodo de 2^k, en el mejor de los casos. Muchos otros generadores de números aleatorios tienen ciclos bastante menores que el mostrado aquí. Éstos no son adecuados en aplicaciones que requieran secuencias largas de números aleatorios. La librería de Java tiene un generador de este tipo; sin embargo, emplea un generador lineal de congruencias de 48 bits, devolviendo sólo los primeros 32 bits, evitando así el problema que representa la ciclicidad de los bits de menor orden. En concreto, las constantes que utiliza son $A = 25.214.903.917$, $B = 48$ y $C = 13$[1].

Finalmente, podría parecer que es posible obtener un mejor generador de números aleatorios añadiendo una constante a la ecuación. Por ejemplo, podría pensarse que

$$X_{i+1} = (48.271X_i + 1) \text{ mod } (2^{31} - 1)$$

generará valores más aleatorios. Sin embargo, cuando usamos esta ecuación podemos observar que

$$(48.271 \cdot 179.424.105 + 1) \text{ mod } (2^{31} - 1) = 179.424.105.$$

De modo que, si la semilla es 179.424.105, el generador se bloquea en un ciclo de periodo 1. Esto muestra lo frágiles que pueden ser estos generadores.

[1] Éste es el mismo generador que el conocido como `drand48`, existente en los sistemas Unix.

9.3 Números aleatorios no uniformes

No todas las aplicaciones requieren números aleatorios uniformemente distribuidos. Por ejemplo, las calificaciones de los alumnos de un curso numeroso no suelen distribuirse de manera uniforme, sino que se ajustan a la clásica distribución en forma de campana, conocida formalmente como *distribución normal* o *de Gauss*. Un generador uniforme de números aleatorios puede emplearse para generar números que satisfacen otro tipo de distribuciones.

Una importante distribución no uniforme que se produce durante las simulaciones es la *distribución de Poisson*. Los sucesos que se producen bajo las siguientes circunstancias satisfacen la distribución de Poisson:

- La probabilidad de un suceso en una región pequeña es proporcional al tamaño de la región.
- La probabilidad de dos sucesos en una región pequeña es proporcional al cuadrado del tamaño de la región, con lo que normalmente es lo suficientemente pequeña como para poder ser ignorada.
- Los hechos consistentes en obtener k sucesos en una región determinada y obtener j sucesos en una región disjunta a la anterior son independientes. (Técnicamente, esto quiere decir que la probabilidad de que ambos sucesos se produzcan simultáneamente se calcula multiplicando las probabilidades de los eventos individuales.)
- La cantidad media de sucesos en una región de un tamaño dado es conocida.

Entonces, si el número medio de sucesos viene dado por la constante a, la probabilidad de que se presenten exactamente k sucesos es $a^k e^{-a}/k!$.

La distribución de Poisson normalmente se aplica a sucesos que tienen una probabilidad reducida de producirse. Por ejemplo, considérese el evento de tener un boleto ganador de la lotería primitiva, en el que la probabilidad de ganar el bote es de 1 entre 14 millones. Supondremos que los números escogidos al rellenar un boleto son, más o menos, aleatorios e independientes. Si una persona rellena 100 boletos, sus probabilidades de ganar son ahora de 1 entre 140.000 (han mejorado en un factor de 100), de modo que se verifica la primera condición. La probabilidad de que una persona tenga dos boletos ganadores es casi nula, por lo que también se cumple la segunda condición. Si alguna otra persona compra 10 boletos, sus probabilidades de ganar son de 1 entre 1.400.000 , y éstas son independientes de la primera persona, de modo que la tercera propiedad también se cumple. Supóngase que se venden 28 millones de boletos. La cantidad media de boletos ganadores en esta situación es 2 (ésta es la cifra que necesitamos para satisfacer la última condición). El número previsible de boletos ganadores es una variable aleatoria con un valor esperado de 2, que satisface la distribución de Poisson. De este modo, la probabilidad de que se hayan vendido exactamente k boletos

Boletos ganadores	0	1	2	3	4	5
Frecuencia	0,135	0,271	0,271	0,180	0,090	0,036

Figura 9.4 Distribución de los ganadores de la lotería cuando el número esperado de ganadores es 2.

ganadores es $2^k e^{-2}/k!$. Esto nos da la distribución de la Figura 9.4. En general, si el número esperado de boletos ganadores es a, entonces la probabilidad de que aparezcan k boletos ganadores es $a^k e^{-a}/k!$.

Para generar de forma aleatoria un entero siguiendo una distribución de Poisson con media a, podemos seguir la siguiente estrategia (su justificación matemática queda fuera de los objetivos del texto): se generan repetidamente números aleatorios en el intervalo (0, 1) distribuidos de manera uniforme hasta que su producto sea menor (o igual) que e^{-a}. Esto es lo que se hace en la rutina de la Figura 9.5.

Otra importante distribución no uniforme es la *distribución exponencial*, generada en la Figura 9.6. En esta distribución la media y la varianza coinciden. La distribución exponencial se emplea para modelar el tiempo que transcurre entre dos eventos aleatorios independientes, como en el ejemplo de simulación de la Sección 13.2.

Muchas otras distribuciones aparecen con frecuencia. Hemos tratado aquí de mostrar como una mayoría de ellas pueden obtenerse a partir de la distribución uniforme. Puede consultarse cualquier libro sobre probabilidades y estadística para conocer más detalles acerca de estas distribuciones.

```
1  /**
2   * Devuelve un entero usando una distribución de Poisson y
3   * cambia el estado interno.
4   * @param valorEsperado la media de la distribución.
5   * @return el int pseudoaleatorio.
6   */
7  public int poisson( double valorEsperado )
8  {
9      double limite = -valorEsperado;
10     double producto = Math.log( randomReal( ) );
11     int contador;
12
13     for( contador = 0; producto > limite; contador++ )
14         producto += Math.log( randomReal( ) );
15
16     return contador;
17 }
```

Figura 9.5 Generación de un número aleatorio según la distribución de Poisson.

```
1  /**
2   * Devuelve un valor de tipo double que sigue una distribución
3   * exponencial y cambia el estado interno.
4   * @param valorEsperado la media de la distribución.
5   * @return el double pseudoaleatorio.
6   */
7  public double negExp( double valorEsperado )
8  {
9      return - valorEsperado * Math.log( randomReal( ) );
10 }
```

Figura 9.6 Generación de un número aleatorio siguiendo una distribución exponencial.

9.4 Generación de una permutación aleatoria

Considérese el problema de simular un juego de cartas. La baraja tiene 52 cartas distintas. En el transcurso de una partida debemos generar cartas de la baraja sin repetir ninguna. Esto corresponde al proceso de barajar las cartas para después repartir cartas del mazo resultante. Naturalmente deseamos que el barajeo de las cartas sea justo, es decir, que cada una de las 52! posibles combinaciones tenga la misma probabilidad de aparecer tras barajar las cartas.

Este tipo de problemas precisa la generación de *permutaciones aleatorias*. Una permutación aleatoria emplea números aleatorios distintos para cada elemento. En general, el problema es el siguiente: generar una permutación aleatoria de 1, 2, ..., N. Todas las permutaciones deben ser equiprobables. La aleatoriedad de la permutación está limitada por la aleatoriedad del generador de números pseudoaleatorios. De este modo, que todas las permutaciones sean igualmente probables depende de que todos los números generados sean independientes y se distribuyan uniformemente. Las permutaciones aleatorias pueden generarse en tiempo lineal.

En la Figura 9.7 se muestra una rutina, `permute`, que sirve para generar una permutación aleatoria. Al efectó procedemos del siguiente modo: partimos de un vector en el que aparecen los elementos de 1, ..., N en un orden cualquiera. El bucle realiza la mezcla aleatoria. En cada iteración se intercambia `a[ j ]` con un cierto elemento del vector entre las posiciones 0 y `j` (en particular es posible que no se realice ningún intercambio, al escogerse el elemento `j`-ésimo). Tras ejecutar la rutina se genera una permutación aleatoria de 1, 2, ..., N.

Es evidente que `permute` genera permutaciones arbitrarias. Pero, ¿todas las permutaciones son igualmente probables? La respuesta es sí y no, a la vez. Si nos basamos en el algoritmo la respuesta es sí. Existen $N!$ permutaciones posibles, y el número de posibles salidas diferentes de las $N - 1$ llamadas a `randomInt` en la linea 11 es también $N!$. Esto es debido a que la primera llamada produce 0 o 1, luego tiene dos salidas. La segunda llamada produce 0, 1 o 2, luego tiene tres salidas. La llamada i-ésima tiene N salidas. El número total de salidas es el producto de todas estas posibilidades, ya que cada número aleatorio es independiente del resto. En consecuencia, basta demostrar que cada secuencia de números aleatorios corresponde a una y sólo una permutación. Esto puede demostrarse razonando *hacia atrás* (*véase* el Ejercicio 9.6.)

Sin embargo la respuesta real es no, pues todas las permutaciones no son equiprobables. Como sólo hay $2^{31} - 2$ estados iniciales posibles para el generador de

```
1  /**
2   * Reordena un vector de forma aleatoria.
3   * Los números aleatorios empleados dependen de la hora y el día.
4   * @param a el vector.
5   */
6  public static final void permute( Object [ ] a )
7  {
8      Random r = new Random( );
9
10     for( int j = 1; j < a.length; j++ )
11         Ordenacion.intercambioRef( a, j, r.randomInt( 0, j ) );
12 }
```

Figura 9.7 Rutina para generar permutaciones aleatorias.

números aleatorios, sólo pueden generarse $2^{31} - 2$ permutaciones distintas. Esto podría representar un problema en algunas aplicaciones. Por ejemplo, un programa que generara 1.000.000 permutaciones (dividiendo quizás el trabajo entre varios computadores) para medir la eficiencia de un algoritmo de ordenación, generará, con total seguridad, algunas permutaciones dos veces. Se necesitan generadores de números aleatorios mejores para hacer que la teoría y la práctica coincidan absolutamente.

Observe que substituir la llamada al método de intercambiar por la llamada `r.randomInt(0,a.length-1)` no funciona, incluso en el caso de tener tres elementos. Existen $3! = 6$ posibles permutaciones y el número de secuencias diferentes que puede calcularse con las tres llamadas de `randomInt` es $3^3 = 27$. Como 6 no es divisor de 27, unas permutaciones tienen mayor probabilidad de aparecer que otras.

9.5 Algoritmos aleatorios

Suponga que es un profesor que da clases de programación cada semana. Quiere asegurarse de que los estudiantes están escribiendo sus propios programas o, al menos, que entienden el código que presentan. Una solución es preguntar a los alumnos acerca de los programas cada día de clase. Sin embargo, estas preguntas reducen el tiempo disponible, por lo que sólo sería práctico cuestionar la mitad de los programas. Su problema es decidir cuándo preguntar a los estudiantes.

Si avisa con antelación de que va a preguntar, los alumnos tendrán la posibilidad de hacer trampas no realizando el 50 por ciento de los programas que no van a ser examinados. Alternativamente puede adoptar la estrategia de examinar un programa sí y otro no, sin avisar previamente, pero los alumnos podrían descifrar dicha estrategia rápidamente. Otra posibilidad es examinar a los alumnos sobre los programas que parecen más importantes, pero esto probablemente conducirá a patrones similares de año en año, por lo que de nuevo la estrategia resultaría predecible a partir del primer año.

Un método que parece eliminar estos problemas es lanzar una moneda. Es decir, hace un cuestionario para cada programa (redactar las preguntas no consume tanto tiempo como corregirlas), y al comienzo de la clase lanza una moneda para decidir si va a haber examen. De esta manera es imposible saber antes de cada clase si ese día habrá cuestionario. Además, este patrón no se repite cada año. Los estudiantes pueden ser examinados con un 50 por ciento de posibilidades, independientemente de lo sucedido antes. La desventaja de esta estrategia es que el año podría acabar sin que haya habido ningún examen. Aunque si se asume que los estudiantes tienen que realizar muchos programas, esto es improbable, a menos que la moneda esté trucada. Cada año el número esperado de exámenes es la mitad del número de programas implementados, y con una elevada probabilidad, el número de exámenes realizados será muy parecido a dicha cantidad.

Este ejemplo ilustra los llamados *algoritmos aleatorios*. Durante su ejecución se emplea, en alguna ocasión, un número aleatorio para tomar una decisión, en lugar de tomarlas siempre de forma determinista. La duración de la ejecución del algoritmo depende no sólo de los datos de entrada sino también de los números aletorios generados.

En el peor de los casos la duración de la ejecución es, casi siempre, la misma que la que la del algoritmo no aleatorizado. La diferencia más importante es que un algoritmo bien aleatorizado no tiene datos de entrada malos sino números aleatorios (para cada entrada en particular) inadecuados. Esto parece sólo una diferencia teórica, pero tiene una gran importancia, tal y como muestra el siguiente ejemplo.

Considérese el problema siguiente. Su jefe le pide escribir un programa que calcule la media de un conjunto de 1.000.000 de números. Debe entregar el programa y ejecutarlo con los datos de entrada que el jefe decida. Si aparece la respuesta correcta en unos pocos segundos (lo que se espera de un algoritmo lineal), su jefe estará muy satisfecho por el trabajo realizado y le premiará de algún modo. Pero si su programa no funciona o es demasiado lento, su jefe le despedirá por incompetente. Su jefe piensa que gana demasiado y está deseando tomar la segunda opción. ¿Qué debería hacer?

El algoritmo de selección rápida descrito en la Sección 8.7 parece ser la mejor solución al problema. Aunque el algoritmo (*véase* la Figura 8.20) es muy rápido en media, es cuadrático en el caso peor si el pivote es pobre. Usando la mediana de tres, tenemos garantizado que este caso peor no se dará nunca para los datos de entrada más comunes, al igual que para aquellos que ya están ordenados o tienen algún elemento duplicado. Sin embargo, aún existe un caso peor cuadrático mostrado en el Ejercicio 8.8. De modo que su malévolo jefe tras leer su programa, podría ver cómo se elige el pivote, y será capaz de construir el caso peor. Como consecuencia, será despedido.

Empleando números aleatorios puede garantizar estadísticamente la seguridad de su trabajo. Puede comenzar el algoritmo de selección rápida mezclando aleatoriamente la entrada empleando el programa de la Figura 9.7[2]. Como consecuencia, su jefe ha perdido la capacidad de especificar una mala secuencia de entrada. Cuando ejecute el algoritmo de selección rápida, éste procesará una entrada aleatoria, por lo que es de esperar un tiempo lineal en su ejecución. ¿Puede durar un tiempo cuadrático? La respuesta es sí. Es posible que la mezcla de cualquier entrada inicial devuelva el peor caso para la selección rápida, por lo que la secuencia resultante sería ordenada en tiempo cuadrático. Si es lo suficientemente desafortunado como para que suceda esto, perderá su trabajo (¡aunque su jefe no sea en principio tan malicioso!). Sin embargo, este suceso es estadísticamente imposible: para un millón de elementos, la probabilidad de tener que emplear tan sólo el doble del tiempo del que indica la media es tan pequeña que puede ser ignorada. Es mucho, pero que mucho más probable que su computador se equivoque o se estropee inoportunamente. Así que su puesto de trabajo estaría seguro.

En lugar de mezclar los datos, podemos obtener un resultado equivalente eligiendo el pivote de forma aleatoria en lugar de hacerlo de forma determinista. Elegimos aleatoriamente un elemento del vector y lo intercambiamos con el elemento en la posición `linf`. Elegimos de forma aleatoria otro elemento y lo intercambiamos con el elemento en la posición `lsup`. Escogemos un tercer elemento y lo intercambiamos con el situado en la posición media. Después el algoritmo continúa como de ordinario. Como en el algoritmo original, las particiones degeneradas

rarse de que el generador de números aleatorios es lo suficientemente aleatorio como para que su salida no pueluto por su jefe.

siguen siendo posibles, pero ahora aparecen como resultado de números aleatorios inadecuados, y no por culpa de malas entradas.

Comentamos a continuación las principales diferencias entre los algoritmos aleatorizados y los no aleatorizados. Hasta ahora hemos venido estudiando los algoritmos no aleatorizados. Cuando calculamos su tiempo medio de ejecución, suponemos que todas las entradas son equiprobables. Sin embargo, esta suposición podría no ser cierta ya que, por ejemplo, las entradas casi ordenadas son más habituales de lo esperado estadísticamente. Esto puede causar problemas para algunos algoritmos como el quicksort. Pero usando un algoritmo aleatorizado el tener unos datos de entrada u otros no es tan importante. En este caso lo que sí es importante es poder contar con verdaderos números aleatorios, ya que para obtener una *estimación* del tiempo de ejecución se hace una media de todos los comportamientos posibles correspondientes a los números aleatorios asociados a cualquier entrada. Empleando la selección rápida con elección aleatoria de pivotes (o un paso de preprocesamiento de la entrada) obtenemos un algoritmo $O(N)$. Esto significa que para cualquier entrada, incluyendo aquella que ya está ordenada, el tiempo de ejecución estimado, basándonos en el comportamiento estadístico de los números aleatorios, es $O(N)$. Una cota del tiempo estimado será habitualmente algo mayor que una cota del tiempo en el caso promedio ya que las suposiciones hechas para generar los números aleatorios son más débiles (números aleatorios versus entrada aleatoria), pero será menor que la correspondiente cota en el caso peor. Por otra parte, en muchas ocasiones las soluciones que tienen buenas cotas en el caso peor exigen con frecuencia un trabajo añadido para asegurar que no se produzca el caso peor. El algoritmo de selección con coste $O(N)$ en el caso peor es un resultado teórico formulable pero poco útil en la práctica.

Los algoritmos aleatorios se clasifican en dos grandes familias. Los primeros, como el arriba estudiado, siempre devuelven la respuesta correcta, aunque pueden tardar en hacerlo, dependiendo de los números aleatorios generados. Los segundos son los que estudiaremos en el resto de este capítulo. Este segundo tipo de algoritmos aleatorios se ejecutan en una cantidad fija de tiempo pero pueden cometer errores de forma aleatoria (presumiblemente con una probabilidad reducida). Estos errores se clasifican en *falsos positivos* o *falsos negativos*. Ésta es una técnica ampliamente aceptada en medicina. Los falsos positivos y los falsos negativos son bastante comunes: algunos tests médicos tienen tasas de error sorprendentemente elevadas. Más aún, para algunos tests los errores dependen de las características del individuo y no de las circunstancias aleatorias generadas, por lo que repetir el test producirá un nuevo resultado erróneo. En los algoritmos aleatorios podemos repetir el test con la misma entrada usando números aleatorios diferentes. Si ejecutamos el algoritmo diez veces y obtenemos diez positivos, y si obtener un solo falso positivo es un suceso improbable (por ejemplo una vez de cada cien), entonces la probabilidad de obtener consecutivamente diez positivos falsos (una vez en 100^{10}, o sea cien trillones) es prácticamente nula.

9.6 Test aleatorio de primalidad

Recordemos que en la Sección 7.4 se han descrito muchos algoritmos numéricos que vimos cómo pueden emplearse para implementar el esquema de encriptación

basado en el algoritmo RSA. Un paso importante en el algoritmo RSA es la producción de dos números primos p y q. Podemos encontrar un número primo probando reiteradamente números impares sucesivos hasta encontrar uno primo. Así que el punto crucial del algoritmo es comprobar si un número dado es primo.

El algoritmo más sencillo para comprobar si un número impar N es primo consiste en estudiar su *divisibilidad*. En este algoritmo, se utiliza el hecho de que un número mayor que 3 es primo si no es divisible entre ningún otro impar menor o igual que $\sqrt{N}$. En la Figura 9.8 se implementa directamente este método.

El test de la divisibilidad es razonablemente rápido para números pequeños (32-bit), pero no se puede emplear ni siquiera para el tipo `long` 64-bit, pues se necesitan probar cerca de $\sqrt{N}/2$ divisores, empleándose para ello una cantidad de tiempo del orden $O(\sqrt{N})$. Lo que deseamos es un test del mismo orden de magnitud que la rutina `potencia` de la Sección 7.4.2. Un teorema muy conocido que puede llevarnos a resolver la cuestión es el *Teorema pequeño de Fermat*. (En aras de la completitud del presente texto el resultado se demuestra en el Teorema 9.1, aunque no es necesario para entender la demostración del test de primalidad.)

> *(Teorema pequeño de Fermat): Si P es primo y $0<A<P$ entonces* $A^{P-1} \equiv 1 (\text{mod } P)$.
>
> *Considérese un k cualquiera verificando $1 \leqslant k < P$. Como P es primo y menor que A y k, $Ak \equiv 0 (\text{mod } P)$ es imposible. Considérese ahora cualquier $1 \leqslant i < j < P$. $Ai \equiv Aj (\text{mod } P)$ implicaría $A(j-i) \equiv 0 (\text{mod } P)$ pero esto es imposible por el argumento anterior, ya que $1 \leqslant j - i < P$. Además, cuando trabajamos* mod P, *la secuencia $A, 2A, ..., (P-1)A$ es una permutación de $1, 2, ..., P-1$. El producto de ambas secuencias* (mod P) *debe ser equivalente, derivándose entonces la equivalencia $A^{P-1}(P-1)! \equiv (P-1)! (\text{mod } P)$, de la que se sigue el teorema.*

Si el recíproco del *Teorema pequeño de Fermat* fuese cierto, entonces tendríamos un test de primalidad computacionalmente equivalente a la exponenciación modular (esto es, $O(\log N)$). Desafortunadamente, el resultado recíproco no es cierto. Se comprueba fácilmente que $2^{340} \equiv 1 (\text{mod } 341)$, pero 341 es factorizable (11×31).

```
1  // Devuelve true si el número impar n es primo
2
3  public static boolean esPrimo( long n )
4  {
5      for( long i = 3; i * i <= n; i += 2 )
6          if( n % i == 0 )
7              return false;      // No primo
8
9      return true;               // Primo
10 }
```

Figura 9.8 Test de primalidad basado en la divisibilidad.

Para obtener un test de primalidad necesitamos un teorema adicional: el Teorema 9.2.

> *Si P es primo y $X^2 \equiv 1 (\text{mod } P)$ entonces $X \equiv \pm 1 (\text{mod } P)$.*
>
> *Como $X^2 - 1 \equiv 0 (\text{mod } P)$ implica $(X-1)(X+1) \equiv 0 (\text{mod } P)$ y P es primo, $X - 1$ o $X + 1 \equiv 0 (\text{mod } P)$.*

Una combinación de los Teoremas 9.1 y 9.2 resulta útil. Sea A un entero entre 2 y $N-2$. Si calculamos $A^{N-1} (\text{mod } N)$ y el resultado no es 1, entonces N no puede ser primo, ya que de otro modo se contradice el Teorema pequeño de Fermat. Decimos entonces que A es un *testigo* de la factorizabilidad de N, ya que A demuestra que N es factorizable. Cada número compuesto N tiene algún testigo A, pero para algunos números, llamados *números de Carmichael*, estos testigos son difíciles de encontrar. Necesitamos tener la seguridad de que existe una elevada probabilidad de encontrar un testigo, sea cual sea la elección de N. Para mejorar nuestras posibilidades aplicamos el Teorema 9.2.

Durante el cálculo de A^i también calculamos $(A^{\lfloor i/2 \rfloor})^2$. Así tenemos $X = A^{\lfloor i/2 \rfloor}$ y $Y = X^2$. Obsérvese que X e Y se calculan automáticamente, como parte de la rutina `potencia`. Si Y es 1 y X no es $\pm 1 (\text{mod } N)$, entonces, por el Teorema 9.2, N no puede ser primo. Podemos devolver 0 como el valor de A^i cuando esto se detecte. Entonces N habrá fallado el test de primalidad, basándonos en el Teorema pequeño de Fermat.

La rutina `testigo`, mostrada en la Figura 9.9, calcula $A^i (\text{mod } P)$, salvo que se devuelva 0 cuando se detecta la contradicción del Teorema 9.2[3]. Si `testigo` no devuelve 1, entonces A es un testigo del hecho de que N no puede ser primo. En las líneas desde la 12 hasta la 14 se hace una llamada recursiva y se produce X. Entonces calculamos X^2, como en una computación normal de `potencia`. Comprobamos si el Teorema 9.2 se contradice, y se devuelve 0 si es así. En cualquier otro caso se completa la ejecución de `potencia`.

El único tema pendiente es la corrección. Si nuestro algoritmo decide que N es factorizable, entonces N debe serlo. Si N es factorizable, ¿es cierto que toda A con $2 \leqslant A \leqslant N-2$ es un testigo? Desafortunadamente, la respuesta es no. Esto significa que existen algunos valores de A que pueden falsear nuestro algoritmo haciéndonos decir que N es primo. De hecho, si se elige A de forma aleatoria, tenemos una probabilidad de 1/4 de fallar en la detección de un número factorizable, cometiendo así un error. Obsérvese que esto es cierto para cualquier N. Si sólo se tuviera esta conclusión en media para los distintos valores de N, entonces no podríamos obtener una rutina lo suficientemente buena. Si recuperamos la analogía con los tests médicos, nuestro algoritmo genera falsos positivos para cualquier N, a lo sumo el 25 por cierto del tiempo.

[3] Obsérvese que este pseudocódigo no funciona para `long`s grandes debido a los posibles errores de desforzar que los números testeados sean 32-bit.

```
/**
 * Método que implementa el test básico de primalidad.
 * Si testigo devuelve 1, n es factorizable.
 * Para ello se calcula a^i (mod n), buscándose a la vez
 * las raíces no triviales de 1.
 */
private static long testigo( long a, long i, long n )
{
    if( i == 0 )
        return 1;

    long x = testigo( a, i / 2, n );
    if( x == 0 )   // Si n es recursivamente factorizable, parar
        return 0;

    // n no es primo si encontramos una raíz no trivial de 1
    long y = ( x * x ) % n;
    if( y == 1 && x != 1 && x != n - 1 )
        return 0;

    if( i % 2 != 0 )
        y = ( a * y ) % n;
    return y;
}

public static final int INTENTOS = 5;

/**
 * Test aleatorio de primalidad.
 * El ajuste de INTENTOS aumenta el grado de confianza.
 * @param n el número a testear.
 * @return false si n no es primo (con seguridad).
 *     true si n seguramente es primo.
 */
public static boolean esPrimo( long n )
{
    Random r = new Random( );

    for( int contador = 0; contador < INTENTOS; contador++ )
        if( testigo( r.randomInt( 2, (int) n - 2 ),
                                    n - 1, n ) != 1 )
            return false;

    return true;
}
```

Figura 9.9 Test aleatorio de primalidad.

Esto no parece una estadística muy buena, ya que una tasa de error del 25 por ciento es ciertamente muy alta. Sin embargo, si usamos 20 valores independientes de A, entonces la probabilidad de que ninguno de ellos sea testigo de un número factorizable es $1/4^{20}$, que es cerca de una entre un millón de millones. Este grado de certeza es mucho más razonable, y además podría mejorarse tanto como deseásemos realizando más intentos. La rutina `esPrimo`, mostrada en la Figura 9.9, se limita a hacer cinco intentos.

Resumen

Este capítulo describe cómo se generan y utilizan los números aleatorios. El generador lineal de congruencias es una buena elección para aplicaciones sencillas, siempre que se elijan con cuidado los parámetros A y M. Podemos obtener números aleatorios que sigan distribuciones no uniformes, como la distribución de Poisson o distribuciones exponenciales, a partir de un generador de números aleatorios uniforme.

Los números aleatorios tienen múltiples aplicaciones. Algunas de ellas incluyen el estudio empírico de algoritmos, la simulación de sistemas de la vida real y el diseño de algoritmos que eviten de forma probabilística el caso peor. Los números aleatorios se emplean en otras partes del texto, especialmente en la Sección 13.2 y el Ejercicio 20.18.

Con este capítulo se concluye la Parte II del libro. La Parte III muestra algunas aplicaciones sencillas, comenzando con una discusión acerca de los juegos en el Capítulo 10, en el que ilustramos tres técnicas importantes de resolución de problemas.

Elementos del juego

falsos positivos y falsos negativos Errores cometidos aleatoriamente (con baja probabilidad) por algunos algoritmos aleatorios de duración fija.

algoritmo aleatorio Algoritmo que emplea números aleatorios en lugar de decisiones deterministas, para decidir el siguiente paso de la ejecución.

distribución de Poisson Distribución que modela el número de veces que se produce un suceso raro.

distribución exponencial Distribución empleada para modelar el tiempo transcurrido entre dos ocurrencias de sucesos aleatorios. Su media y su varianza coinciden.

distribución uniforme Distribución en la que todos los números dentro del rango especificado son equiprobables.

generador lineal de congruencias Un buen algoritmo para generar distribuciones uniformes.

generador lineal de congruencias con periodo completo Generador lineal de congruencias cuyo periodo es $M - 1$.

números pseudoaleatorios Números que verifican muchas de las propiedades de los números aleatorios. Es difícil encontrar buenos generadores de números pseudoaleatorios.

periodo Un generador de números aleatorios con periodo P genera la misma secuencia aleatoria después de P iteraciones.

permutación aleatoria Recolocación aleatoria de N elementos. Puede generarse en tiempo lineal generando un número aleatorio por cada elemento.

permutación Una permutación de 1, 2, ..., N es una secuencia de N enteros que incluye cada elemento de 1, 2, ..., N exactamente una vez.

semilla Es el estado inicial de un generador de números aleatorios.

Teorema pequeño de Fermat Establece que si P es primo y $0 < A < P$ entonces $A^{P-1} \equiv 1 (\text{mod } P)$. Es una condición necesaria, pero no suficiente, para establecer la primalidad de un número.

testigo de composicionalidad Valor de A que prueba que un número no es primo, empleando el Teorema pequeño de Fermat.

trial división Algoritmo sencillo para comprobar la primalidad de un número. Es rápido para números pequeños (32-bit) pero no puede aplicarse para números grandes.

Errores comunes

1. La utilización de una semilla nula producirá números aleatorios inadecuados.
2. Los usuarios sin experiencia podrían reinicializar la semilla con un cierto valor prefijado antes de generar una nueva permutación aleatoria. Esto provocaría que se generara la misma permutación en distintas ocasiones, lo que probablemente no se pretendía.
3. Muchos generadores de números aleatorios son particularmente malos. En algunas aplicaciones que requieren largas secuencias de números aleatorios, incluso el generador lineal de congruencias es insatisfactorio.
4. Se sabe que, bajo ciertos criterios de aleatoriedad, los bits de menor orden de los generadores lineales de congruencias no son aleatorios. En consecuencia, debe evitarse su uso. Por ejemplo, `randomInt()%2` es una mala táctica para simular el lanzamiento de una moneda.
5. Cuando se generan números aleatorios en un intervalo determinado, un error muy común es manejar valores ligeramente fuera de los límites e incluso permitir la generación de valores fuera del intervalo, o no permitir que el valor más pequeño se genere con probabilidad ajustada.
6. Muchos generadores de permutaciones aleatorias no generan todas las permutaciones posibles con la misma probabilidad. Como se ha comentado en el texto, nuestro algoritmo está limitado por el generador de números aleatorios.
7. Cambiar los valores de un generador de números aleatorios equivale, muy probablemente, a debilitar sus propiedades estadísticas.

En Internet

Todo el código que aparece en este capítulo forma parte del paquete `Supporting` y puede encontrarse en el directorio **Supporting**. Éstos son los nombres de los ficheros:

Random.java	Contiene la clase `Random`.
Numerical.java	Contiene la rutina de test de primalidad de la Figura 9.9 además de las rutinas matemáticas de la Sección 7.4.

Ejercicios

Cuestiones breves

9.1. Con el generador de números aleatorios descrito en el texto, determine los primeros 10 valores de `estado`, suponiendo que su valor inicial es 1.

9.2. Muestre el resultado de ejecutar el algoritmo del test de primalidad para $N = 561$, con valores de A entre 2 y 5 (ambos inclusive).

9.3. Si se venden 42.000.000 boletos de la lotería primitiva, ¿cuál es el número esperado de ganadores? ¿Qué probabilidad hay de que el premio quede desierto?, ¿y las probabilidades de que haya exactamente un ganador?

9.4. ¿Por qué no puede emplearse una semilla nula en el generador lineal de congruencias?

Problemas teóricos

9.5. Demuestre que la Ecuación 9.2 es equivalente a la Ecuación 9.1 y que el programa resultante, presentado en la Figura 9.3, es correcto.

9.6. Complete la prueba de que cada permutación obtenida en la Figura 9.7 es equiprobable.

9.7. Supóngamos que tenemos una moneda sesgada, de modo que al lanzarla la probabilidad de que salga cara es p y la probabilidad de que salga cruz es $1 - p$. Muestre cómo podría diseñarse un algoritmo que emplee dicha moneda para generar un 0 o un 1, con la misma probabilidad.

Problemas prácticos

9.8. Escriba los métodos `randomReal` y `randomInt`. Después, implemente un programa que llame a `randomInt` 1.000 veces para generar números entre 1 y 1.000. ¿Satisface los tests estadísticos más fuertes de la Sección 9.2?

9.9. Ejecute un millón de veces el generador de Poisson de la Figura 9.5 tomando como valor esperado 2. ¿Concuerda la distribución obtenida con la Figura 9.4?

9.10. Considere una elección entre dos candidatos en la que el ganador ha obtenido una fracción p del voto. Si el recuento de votos se realiza secuencialmente, ¿cuál es la probabilidad de que el ganador esté siempre en cabeza (o empatado con su rival) durante todo el recuento? Este problema se conoce con el nombre del *problema de la votación*. La respuesta es p. Escriba un programa que lo verifique. *Indicación*: Simule una elección con 10.000 votantes. Genere aleatoriamente vectores de $10000p$ unos y $1000(1 - p)$ ceros. Después estudie secuencialmente si la diferencia entre ceros y unos no es nunca negativa (ello deberá ocurrir $100p$ de cada cien veces).

Prácticas de programación

9.11. Un algoritmo alternativo para generar permutaciones es rellenar el vector desde `a[0]` hasta `a[n-1]` del siguiente modo: para rellenar `a[i]` se gene-

ran números aleatorios hasta que aparezca uno que no haya sido usado previamente. Para realizar esta comprobación puede emplearse un vector de booleanos. Evalue el tiempo esperado de ejecución (¡ojo! tiene truco) y escriba un programa que compare el tiempo real de ejecución de la rutina, el tiempo obtenido por la estimación y la rutina de la Figura 9.7.

9.12. Supongamos que se quiere generar una variación aleatoria de N elementos distintos dentro del rango 1, 2, ..., M. (El caso $M=N$ corresponde a la generación de permutaciones.) El algoritmo de Floyd para conseguirlo es el siguiente: vamos generando de forma recursiva una variación de $N-1$ elementos distintos dentro del rango 1, 2, ..., $M-1$. Tras hacerlo generamos un entero aleatorio entre 1 y M. Si el número no está todavía en la permutación se añade el mismo, y si ya está se añade M. Debe hacer lo siguiente:

a) Pruebe que este algoritmo no genera elementos repetidos.
b) Pruebe que cada variación posible aparece de forma equiprobable.
c) Dé una implementación recursiva del algoritmo.
d) Dé una implementación iterativa del algoritmo.

9.13. Un *paseo aleatorio* en dos dimensiones es el siguiente juego, que se desarrolla en el sistema de coordenadas x, y. Se comienza en el origen (esto es, en (0, 0)). Cada iteración consiste en dar un paso, aleatoriamente elegido entre los siguientes: una unidad hacia arriba, hacia abajo, hacia la derecha o hacia la izquierda. El paseo termina cuando el caminante regresa al origen. Puede probarse que esto acaba ocurriendo más tarde o más temprano con probabilidad 1 en dos dimensiones. Sin embargo, si generalizamos el problema al espacio de tres dimensiones, la probabilidad pasa a ser menor que 1. Escriba un programa que realice 100 paseos independientes y calcule el número de pasos dados en cada dirección.

9.14. Un test estadístico muy sencillo, a la vez que efectivo, es el *test de la χ-cuadrado*. Supóngase que generamos, de forma supuestamente uniforme, N valores entre M posibles (por ejemplo, generamos N números entre 1 y M, ambos inclusive). El número de apariciones de cada valor es una variable aleatoria de media $\mu = N/M$. Para que el test, que comprueba la uniformidad, sea significativo debe verificarse $\mu > 10$. Sea f_i el número de veces que i es generado. Entonces se calcula el valor χ-cuadrado $V = \Sigma(f_i - \mu)^2/u$. El resultado debe estar próximo a M. Si el resultado está lejos de M demasiadas veces (en concreto, si lo supera en más de $2\sqrt{M}$, en uno de cada diez intentos), entonces el generador no ha pasado el test. Implemente este test y ejecútelo sobre su implementación del método `randomInt` (con `linf` = 1 y `lsup` = 100).

Bibliografía

En [3] puede encontrarse amplia información sobre los generadores de números aleatorios. El algoritmo de las permutaciones se debe a R. Floyd y puede encontrarse en [1]. El test de primalidad aleatorio se ha tomado de [2] y [4]. En cualquier buen libro de probabilidad y estadística se puede encontrar información adicional sobre los números aleatorios.

1. J. Bentley, «Programming Pearls», *Communications of the ACM* **30** (1987), 754-757.
2. G. L. Miller, «Riemann's Hypothesis and Tests for Primality», *Journal of Computer and System Science* **13** (1976), 300-317.
3. S. K. Park y K. W. Miller, «Random Number Generators: Good Ones Are Hard to Find», *Communications of the ACM* **31** (1988) 1192-1201. (See also *Technical Correspondence* in **36** (1993) 105-110.)
4. M. O. Rabin, «Probabilistic Algorithms for Testing Primality», *Journal of Number Theory* **12** (1980), 128-138.

Parte III

Aplicaciones

CAPÍTULO

10 Juegos y diversión

Este capítulo introduce tres técnicas algorítmicas importantes, mostrando su uso en la implementación de dos programas que resuelven problemas recreativos. El primer problema es una sopa de letras que involucra la búsqueda de palabras en un vector bidimensional de caracteres. El segundo es el problema de llevar a cabo una jugada óptima en el juego de las tres en raya.

En este capítulo veremos:

- Cómo usar el algoritmo de búsqueda binaria, modificando el de la Figura 5.12, para incorporar información procedente de búsquedas sin éxito y resolver casos particulares grandes de un problema de búsqueda de palabras en menos de 1 segundo.
- Cómo usar el algoritmo de la *poda alfa-beta* para acelerar el algoritmo recursivo de la Sección 7.7.
- Cómo usar las tablas hash para incrementar la velocidad del algoritmo de las tres en raya.

10.1 Sopas de letras

La entrada para el problema de la *sopa de letras* es un vector bidimensional de caracteres y una lista de palabras. El objetivo es encontrar las palabras en la matriz. Estas palabras pueden aparecer en horizontal, vertical o diagonal en cualquiera de los dos sentidos (hay un total de ocho direcciones). Como ejemplo, la sopa mostrada en la Figura 10.1 contiene las palabras `lado`, `los`, `pez` y `cual`. La palabra `lado` comienza en la fila 0, columna 0, es decir, en el punto (0, 0), y se extiende hasta el (0, 3); la palabra `los` va desde (0, 0) hasta (2, 0); `pez` desde (3, 0) hasta (1, 2); y `cual` desde (3, 3) hasta (0, 0).

	0	1	2	3
0	l	a	d	o
1	o	a	z	a
2	s	e	u	m
3	p	t	r	c

Figura 10.1 Ejemplo de sopa de letras.

10.1.1 Teoría

Hay varios algoritmos ingenuos que se pueden usar para resolver el problema de la sopa de letras. El más directo es el que hace uso de la fuerza bruta:

```
para cada palabra P en la lista de palabras
    para cada fila F
        para cada columna C
            para cada dirección D
                comprobar si P aparece en la fila F, columna C
                en la dirección D
```

Puesto que hay 8 direcciones, este algoritmo requiere $8PFC$ comprobaciones. Para las sopas habituales de las revistas, donde encontramos 40 palabras o más en una matriz de 16 filas por 16 columnas, esto supone unas 80.000 comprobaciones, lo cual no es un gran problema para una máquina moderna. Pero si consideramos una variación en la que solamente se proporciona la sopa y la lista de palabras posibles viene dada por aquellas que aparecen en el diccionario, entonces el número de palabras se elevaría hasta 40.000 en lugar de 40, generándose unos 80 millones de comprobaciones. Si se dobla el tamaño de la matriz tendríamos 320 millones de comprobaciones, lo que deja de ser un cálculo trivial. Queremos un algoritmo que pueda resolver una sopa de letras de este tamaño en aproximadamente un segundo. Para ello consideremos el siguiente algoritmo alternativo:

```
para cada fila F
    para cada columna C
        para cada dirección D
            para cada longitud de palabra L
                comprobar si los L caracteres a partir de la
                            posición (F, C) en la dirección
                            D forman una palabra
```

Este algoritmo modifica el bucle para evitar la búsqueda de cada palabra de la lista. Supongamos que la longitud de las palabras está limitada a 20 caracteres. En este caso, el número de comprobaciones realizadas por el algoritmo es $160FC$. Para una sopa de 32 filas por 32 columnas, esto supone unas 160.000 comprobaciones. El problema está ahora en que tenemos que decidir si una palabra pertenece a la lista de palabras. Si usamos una búsqueda lineal perdemos todo lo ganado. Pero si usamos una buena estructura de datos, podemos esperar que la búsqueda sea eficiente. Si la lista de palabras está ordenada, lo que es de esperar en un diccionario, entonces podemos usar una búsqueda binaria (mostrada en la Figura 5.12) con lo que para cada comprobación se realizan $\log P$ comparaciones entre cadenas. Para 40.000 palabras, esto significa realizar unas 16 comparaciones por comprobación, dando un total de menos de 3 millones de comparaciones. Esto se puede hacer con toda seguridad en unos pocos segundos. Este algoritmo es mejor que el anterior en un factor de 100.

Este algoritmo se puede mejorar aún más. Supongamos que estamos buscando en alguna dirección y nos damos cuenta de que hemos leído la secuencia de caracteres qx. El diccionario no contiene ninguna palabra comenzando por qx. ¿Vale entonces la pena seguir ejecutando el bucle interno? La respuesta es obviamente no. Si detectamos una secuencia que no es prefijo de ninguna palabra del diccionario, podemos cambiar inmediatamente de dirección. Este algoritmo se expresa de la siguiente forma utilizando pseudocódigo:

```
para cada fila F
    para cada columna C
        para cada dirección D
            para cada longitud de palabra L
                comprobar si los L caracteres a partir de la
                            posición (F, C) en la dirección
                            D forman una palabra
                si no forman un prefijo,
                    salir;      // del bucle más interno
```

El único detalle algorítmico pendiente de tratar es la implementación de la comprobación de prefijo. Es decir, suponiendo que la secuencia de caracteres actual no pertenece a la lista de palabras, ¿cómo podemos decidir si es un prefijo de alguna palabra de la lista? La respuesta es bastante simple. El algoritmo de búsqueda binaria de la Figura 5.12 reduce el dominio de búsqueda a un objeto y entonces mira si coincide con el elemento buscado. Supongamos que en lugar de hacer la comprobación, simplemente devolvemos la posición y dejamos que el usuario use la información. Entonces, por supuesto, es fácil para el que hace la llamada comprobar si el elemento en la posición devuelta coincide o no con el

```
1  /**
2   * Lleva a cabo la búsqueda binaria para el problema de la
3   * sopa de letras usando una comparación por nivel.
4   * Devuelve la última posición examinada, la cual o bien
5   * contiene a x, o bien x es un prefijo suyo, o bien
6   * no hay ninguna palabra de la que x es prefijo.
7   */
8  private static int busquedaPrefijo( String [ ] a,
9                                      String x, int n )
10 {
11     int inf = 0;
12     int sup = n - 1;
13
14     while( inf < sup )
15     {
16         int med = ( inf + sup ) / 2;
17         if( a[ med ].compareTo( x ) < 0 )
18             inf = med + 1;
19         else
20             sup = med;
21     }
22
23     return inf;
24 }
```

Figura 10.2 Búsqueda binaria modificada para devolver el último punto de la búsqueda.

elemento buscado. Si no es así, también es fácil ver si la secuencia de caracteres buscada es un prefijo de alguna palabra de la lista, ya que en tal caso debe ser un prefijo de la cadena situada en la posición devuelta (en el Ejercicio 10.3 se le pide que demuestre esto). En la Figura 10.2 mostramos este algoritmo modificado llamado `busquedaPrefijo`.

10.1.2 Implementación en Java

Nuestra implementación sigue la descripción del algoritmo casi al pie de la letra. Diseñamos una clase llamada `BusquedaPalabras` para almacenar la sopa y la lista de palabras así como los correspondientes canales de entrada. El esqueleto de la clase se muestra en la Figura 10.3. Por simplicidad, asumimos un límite de

```
1  // Clase para resolver el problema de la sopa de letras
2  //
3  // CONSTRUCTOR: sin inicialización
4  // ********************OPERACIONES PÚBLICAS*********************
5  // int resolverSopa( ) --> Imprime todas las palabras encontradas
6  //                         en la sopa; devuelve el número de palabras
7
8  public class BusquedaPalabras
9  {
10     private static final int MAX_FILAS    =     64;
11     private static final int MAX_COLUMNAS =     64;
12     private static final int MAX_PALABRAS = 100000;
13
14     public BusquedaPalabras( )
15       { /* Figura 10.4 */ }
16     public int resolverSopa( )
17       { /* Figura 10.8 */ }
18
19     private int filas;
20     private int columnas;
21     private int numPalabras;
22     private String [ ] palabras = new String[ MAX_PALABRAS ];
23     private BufferedReader sopaStream;
24     private BufferedReader palabraStream;
25     private char sopa[ ][ ] =
26                   new char[ MAX_FILAS ][ MAX_COLUMNAS ];
27     private BufferedReader in = new
28          BufferedReader( new InputStreamReader( System.in ) );
29
30     private static int busquedaPrefijo( String [ ] a,
31                                         String x, int n )
32       { /* Figura 10.2 */ }
33     private BufferedReader abreFichero( String mensaje )
34       { /* Figura 10.5 */ }
35     private void leePalabras( )
36       { /* Figura 10.6 */ }
37     private void leeSopa( )
38       { /* Figura 10.7 */ }
39     private int resolverDireccion( int filaBase, int colBase,
40                                    int filaDelta, int colDelta )
41       { /* Figura 10.9 */ }
42 }
```

Figura 10.3 Esqueleto de la clase `BusquedaPalabras`.

64 filas y columnas para la sopa, y un tamaño máximo de diccionario de 100.000 palabras, dejando el problema de suprimir estas restricciones para el Ejercicio 10.6. La parte pública de la clase consta de un constructor y un único método `resolverSopa`. La sección privada incluye los atributos y las rutinas de soporte.

La Figura 10.4 proporciona el código del constructor, que simplemente abre y lee los dos ficheros correspondientes a la sopa de letras y a la lista de palabras. La rutina de soporte `abreFichero`, mostrada en la Figura 10.5, solicita repetidamente el nombre de un fichero hasta que la apertura tiene éxito. La rutina

```
1  /**
2   * Constructor para la clase BusquedaPalabras. Solicita
3   * el nombre de los ficheros de la sopa y del diccionario.
4   */
5  public BusquedaPalabras( )
6  {
7      sopaStream = abreFichero( "Introduzca fichero de la sopa" );
8      palabraStream = abreFichero( "y el del diccionario " );
9      leeSopa( );
10     leePalabras( );
11 }
```

Figura 10.4 Constructor de `BusquedaPalabras`.

```
1  /**
2   * Imprime un mensaje y abre un fichero.
3   * Lo intenta repetidas veces hasta que consigue abrirlo.
4   * El programa termina cuando se alcanza el final de fichero.
5   */
6  private BufferedReader abreFichero( String mensaje )
7  {
8      String nombreFichero = "";
9      FileReader fichero;
10     BufferedReader fichEntrada = null;
11
12     do
13     {
14         System.out.println( mensaje + ": " );
15
16         try
17         {
18             nombreFichero = in.readLine( );
19             if( nombreFichero == null )
20                 System.exit( 0 );
21             fichero = new FileReader( nombreFichero );
22             fichEntrada = new BufferedReader( fichero );
23         }
24         catch( IOException e )
25           { System.err.println( "No se puede abrir " + nombreFichero ); }
26     } while( fichEntrada == null );
27
28     System.out.println( "Abierto " + nombreFichero );
29     return fichEntrada;
30 }
```

Figura 10.5 Rutina `abreFichero` para abrir los ficheros de la sopa de letras y de la lista de palabras.

`leePalabras`, mostrada en la Figura 10.6, lee la lista de palabras. La mayor parte del código está relacionada con la comprobación de errores: no queremos leer más de `MAX_PALABRAS` palabras, y queremos asegurarnos de que la lista de palabras está ordenada. Análogamente, la rutina `leeSopa`, mostrada en la Figura 10.7, lee la sopa de letras y también se encarga de la comprobación de errores. Necesitamos asegurarnos de que podemos manejar sopas vacías, y queremos avisar al usuario si la sopa no es rectangular. En interés de la mayor brevedad posible, se han omitido algunas de las comprobaciones que deberían haberse realizado. En el Ejercicio 10.1 se le pide que averigue lo que falta.

La rutina `resolverSopa` de la Figura 10.8 anida los bucles que recorren las filas, las columnas y las direcciones, y llama a la rutina privada `resolverDireccion` para cada una de las posibilidades. El valor devuelto es el número de palabras encontrado. Una dirección viene dada por una dirección en la columna y una dirección en la fila. Por ejemplo, el sur viene indicado por `dc = 0` y `df = 1`, y el noreste por `dc = 1` y `df = -1`. Las variables `dc` y `df` toman valores entre `-1` y `1`, pero no pueden tomar simultáneamente el valor `0`. Ahora, lo que queda por escribir es `resolverDireccion`, que aparece en la Figura 10.9.

La rutina `resolverDireccion` construye una cadena partiendo de la fila y la columna base y avanzando en la dirección apropiada.

```
1  /**
2   * Rutina para leer el diccionario.
3   * Se muestra un mensaje de error si no está ordenado.
4   * Se hace una comprobación para no sobrepasar MAX_PALABRAS.
5   */
6  private void leePalabras( )
7  {
8      numPalabras = 0;
9
10     try
11     {
12         while( ( palabras[ numPalabras ] =
13                         palabraStream.readLine( ) ) != null )
14         {
15             if( numPalabras != 0 && palabras[ numPalabras ].
16               compareTo( palabras[ numPalabras - 1 ] ) < 0 )
17             {
18                 System.err.println( "El diccionario no está " +
19                                     "ordenado - saliendo" );
20                 continue;
21             }
22             else if( ++numPalabras >= MAX_PALABRAS )
23                 break;
24         }
25
26         if( palabraStream.ready( ) )
27             System.err.println( "Aviso: no se han leido los " +
28                                 "datos - incremente MAX_PALABRAS" );
29     }
30     catch( IOException e ) { }
31 }
```

Figura 10.6 Rutina `leePalabras` para leer la lista de palabras.

```
1  /**
2   * Rutina para leer la sopa de letras.
3   * Comprueba que la sopa es rectangular.
4   * No se comprueba si se excede la capacidad.
5   */
6  private void leeSopa( )
7  {
8      String unaLinea;
9
10     try
11     {
12         unaLinea = sopaStream.readLine( );
13         if( unaLinea == null )
14         {
15             filas = 0;
16             return;
17         }
18         columnas = unaLinea.length( );
19         for( int i = 0; i < columnas; i++ )
20             sopa[ 0 ][ i ] = unaLinea.charAt( i );
21
22         for( filas = 1;
23              ( unaLinea = sopaStream.readLine( ) ) != null;
24                          filas++ )
25         {
26             if( unaLinea.length( ) != columnas )
27                 System.err.println( "La sopa es incorrecta" );
28
29             for( int i = 0; i < columnas; i++ )
30                 sopa[ filas ][ i ] = unaLinea.charAt( i );
31         }
32     }
33     catch( IOException e ) { }
34 }
```

Figura 10.7 Rutina `leeSopa` para leer la sopa de letras.

```
1  /**
2   * Rutina para resolver una sopa de letras.
3   * Lleva a cabo comprobaciones en todas la direcciones.
4   * @return número de palabras encontradas
5   */
6  public int resolverSopa( )
7  {
8      int numPal = 0;
9
10     for( int f = 0; f < filas; f++ )
11         for( int c = 0; c < columnas; c++ )
12             for( int df = -1; df <= 1; df++ )
13                 for( int dc = -1; dc <= 1; dc++ )
14                     if( df != 0 || dc != 0 )
15                         numPal += resolverDireccion( f,
16                                                      c, df, dc );
17
18     return numPal;
19 }
```

Figura 10.8 Rutina `resolverSopa` que busca en todas las direcciones a partir de un punto.

```
1  /**
2   * Busca en la sopa a partir de un punto y en una dirección.
3   * @return número de palabras encontradas
4   */
5  private int resolverDireccion( int filaBase, int colBase,
6                                 int filaDelta, int colDelta )
7  {
8      String secuencia = "";
9      int numPal = 0;
10     int resultadoBusqueda;
11
12     secuencia += sopa[ filaBase ][ colBase ];
13
14     for( int i = filaBase + filaDelta, j = colBase + colDelta;
15              i >= 0 && j >= 0 && i < filas && j < columnas;
16              i += filaDelta, j += colDelta )
17     {
18         secuencia += sopa[ i ][ j ];
19         resultadoBusqueda = busquedaPrefijo( palabras,
20                                              secuencia, numPal );
21
22         if( !palabras[ resultadoBusqueda ].
23                                 startsWith( secuencia ) )
24           break;
25
26         if( palabras[ resultadoBusqueda ].equals( secuencia ) )
27         {
28             numPal++;
29             System.out.println( "Encontrada " + secuencia +
30                     " de " + filaBase + " " + colBase +
31                     " a " + i + " " + j );
32         }
33     }
34     return numPal;
35 }
```

Figura 10.9 Implementación de una búsqueda individual.

Asumimos que las palabras de una letra no están permitidas (pues de estarlo se contarían ocho veces). En las líneas 15 y 16, extendemos la cadena mientras comprobamos que no rebasamos los límites de la sopa. En la línea 18 añadimos el siguiente carácter y en las líneas 19 y 20 llevamos a cabo una búsqueda binaria. Si la cadena leída hasta el momento no es un prefijo de ninguna palabra, podemos parar de mirar en esa dirección. En caso contrario, sabemos que continuaremos haciéndolo, tras comprobar, en la línea 26, si ya se ha leído una palabra de la lista. La línea 34 devuelve el número de palabras encontradas finalmente. En la Figura 10.10 se muestra un programa `main` simple.

```
1    // main simple
2  public static void main( String [ ] args )
3  {
4      BusquedaPalabras p = new BusquedaPalabras( );
5      System.out.println( "Resolviendo..." );
6      p.resolverSopa( );
7  }
```

Figura 10.10 Rutina `main` simple para el problema de la sopa de letras.

10.2 El juego de las tres en raya

Recordemos el algoritmo simple de la Sección 7.7 que permitía al computador elegir el movimiento óptimo en el juego de las tres en raya. La estrategia conocida como *minimax* consistía en lo siguiente:

1. Si el estado es *terminal* (es decir, se puede evaluar inmediatamente), devolvemos su valor.
2. En caso contrario, si le toca mover al computador, devolvemos el valor máximo de todos los estados que se pueden alcanzar después de hacer un movimiento. Los valores alcanzados se calculan de forma recursiva.
3. En caso contrario, es el turno de la persona. Devolvemos entonces el mínimo de todos los estados alcanzables después de hacer un movimiento. De nuevo, los valores alcanzados se calculan de forma recursiva.

10.2.1 Poda alfa-beta

Aunque la estrategia minimax proporciona un movimiento óptimo para las tres en raya, lleva a cabo una gran cantidad de búsquedas. En concreto, para decidir el primer movimiento hace alrededor de medio millón de llamadas recursivas. Esto sucede porque hace más búsquedas de las imprescindibles. Supongamos que el computador está considerando cinco movimientos: C_1, C_2, C_3, C_4 y C_5. Supongamos que la evaluación recursiva de C_1 revela que C_1 fuerza un empate. A continuación se evalúa C_2. En este momento, llegamos a un estado en el que le toca mover al humano. Supongamos que en respuesta a C_2, el humano puede considerar H_{2A}, H_{2B}, H_{2C} y H_{2D}, y además que una evaluación de H_{2A} muestra que se ha forzado un empate. Automáticamente se ha visto que C_2 es un empate en el mejor de los casos e incluso posiblemente una derrota para el computador (porque se supone que el humano juega óptimamente). Puesto que necesitamos mejorar C_1, no tenemos que evaluar H_{2B}, H_{2C} ni H_{2D}. Decimos que H_{2A} es una *refutación*, en el sentido de que C_2 no es un movimiento mejor que el que ya hemos visto. En consecuencia devolvemos que C_2 es un empate y mantenemos a C_1 como el mejor movimiento hasta el momento, lo cual se muestra en la Figura 10.11.

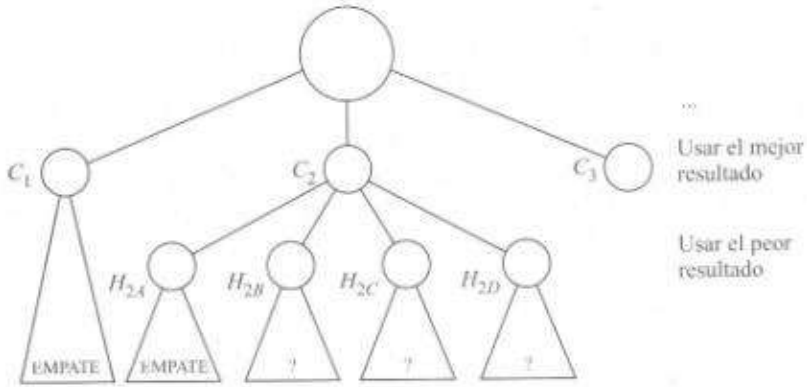


Figura 10.11 Poda alfa-beta: después de que se haya evaluado H_{2A}, C_2, que es el menor de los H_2, es un empate en el mejor de los casos, por lo que no puede ser una mejora respecto a C_1. Por tanto, no necesitamos evaluar H_{2B}, H_{2C} y H_{2D}, y se puede pasar directamente a C_3.

No necesitamos evaluar cada nodo completamente; para algunos nodos basta con encontrar una refutación. Esto significa que algunos bucles pueden terminar antes de tiempo. En concreto, cuando el humano evalúa una posición, como C_2, encontrar una refutación es tan bueno como el mejor movimiento absoluto. La misma lógica se aplica al computador. En cualquier punto, `alfa` es el valor que el humano ha de rechazar, y `beta` es el valor que el computador ha de rechazar. Cuando el humano hace una búsqueda, cualquier movimiento menor que `alfa` es equivalente a `alfa`; si la búsqueda la realiza el computador entonces cualquier movimiento mayor que `beta` es equivalente a `beta`. Esta estrategia recibe el nombre de *poda alfa-beta*.

Como se muestra en la Figura 10.12, la poda alfa-beta se obtiene realizando solamente unos pocos cambios en `elegirMovimiento`. Tanto `alfa` como `beta` se pasan como parámetros adicionales. Inicialmente, se llama a `elegirMovimiento` con valores `HUMANO_GANA` y `COMPUTADORA_GANA` para `alfa` y `beta` respectivamente. Las líneas 16 a 20 reflejan un cambio en la inicialización de `valor`. La evaluación de un movimiento es sólo ligeramente más compleja que la original de la Figura 7.27. La llamada recursiva de la línea 29 incluye los parámetros `alfa` y `beta`, cuyos valores se ajustan en las línea 36 o 38, cuando ello es necesario. El otro cambio aparece en la línea 42, que provoca la terminación por anticipado cuando se encuentra una refutación.

Para beneficiarse lo máximo posible de las ventajas de la poda alfa-beta, los programas de juegos normalmente intentan aplicar heurísticas para colocar los mejores movimientos al comienzo de la búsqueda. Esto produce más poda que la esperada en una búsqueda aleatoria de estados. En la práctica, la poda alfa-beta limita la búsqueda a solamente $O(\sqrt{N})$ nodos, donde N es el número de nodos que se examinarían sin poda alfa-beta. Esto supone una gran cantidad de ahorro. El ejemplo de las tres en raya no es ideal al respecto porque los valores son demasiado similares entre sí, pero aun así la búsqueda inicial se reduce a unos 18.000 estados.

10.2.2 Tablas de transposición

Otra práctica empleada normalmente consiste en usar una tabla para guardar todos los estados que ya han sido evaluados. Por ejemplo, en el curso de la búsqueda del primer movimiento, el programa examinará los estados mostrados en la Figura 10.13. Si se guardan los valores de los estados, la segunda vez que aparezca uno de ellos no será necesario calcular su valor de nuevo, lo que hace que dicho estado se convierta en terminal. La estructura que almacena los estados recibe el nombre de *tabla de transposición*, la cual se implementa usando una tabla hash. En muchos casos, la aplicación de esta técnica, análoga a la implementación de la programación dinámica que vimos en la Sección 7.6, puede ahorrar una gran cantidad de cálculos.

Para implementar la tabla de transposición, definimos primero una clase `Estado`, mostrada en la Figura 10.14, que se usa para almacenar cada estado y su valor calculado. Esta clase implementa el interfaz de `Hashable`. También proporcionamos un constructor que se puede inicializar con un tablero (vector bidimensional).

```
1      // Encuentra el movimiento óptimo
2  private Mejor elegirMovimiento( int lado, int alfa, int beta )
3  {
4      int op;               // El oponente
5      Mejor replica;        // Mejor réplica del oponente
6      int evalSimple;       // Resultado de una evaluación inmediata
7      int mejorFila = 0;
8      int mejorColumna = 0;
9      int valor;
10
11     if( ( evalSimple = valorEstado( ) ) != INCIERTO )
12         return new Mejor( evalSimple );
13
14     if( lado == COMPUTADORA )
15     {
16         op = HUMANO; valor=alfa;
17     }
18     else
19     {
20         op = COMPUTADORA; valor = beta;
21     }
22
23 Exterior:
24     for( int fila = 0; fila < 3; fila++ )
25        for( int columna = 0; columna < 3; columna++ )
26           if( casillaVacia ( fila, columna ) )
27           {
28               lugar( fila, columna, lado );
29               replica = elegirMovimiento( op, alfa, beta );
30               lugar( fila, columna, VACIA );
31               if( lado == COMPUTADORA && replica.valor > valor
32                  || lado == HUMANO && replica.valor < valor )
33               {
34                     // Se ha encontrado el mejor movimiento
35                  if( lado == COMPUTADORA )
36                     alfa = valor = replica.valor;
37                  else
38                     beta = valor = replica.valor;
39
40                  mejorFila = fila;
41                  mejorColumna = columna;
42                  if( alfa >= beta )
43                     break Exterior; // Refutación
44               }
45           }
46
47     return new Mejor( valor, mejorFila, mejorColumna );
48 }
```

Figura 10.12 Rutina `elegirMovimiento` para calcular el movimiento óptimo usando poda alfa-beta.

Los cambios necesarios en la clase `TresEnRaya` son mínimos y se muestran en la Figura 10.15. Se añade un nuevo atributo en la línea 3 y se modifica la declaración de `elegirMovimiento`. Ahora pasamos `alfa` y `beta` (igual que hicimos en la poda alfa-beta) y también la `profundidad` de la recursión, que es inicialmente cero. La llamada inicial a `elegirMovimiento` aparece en las líneas 7 y 8.

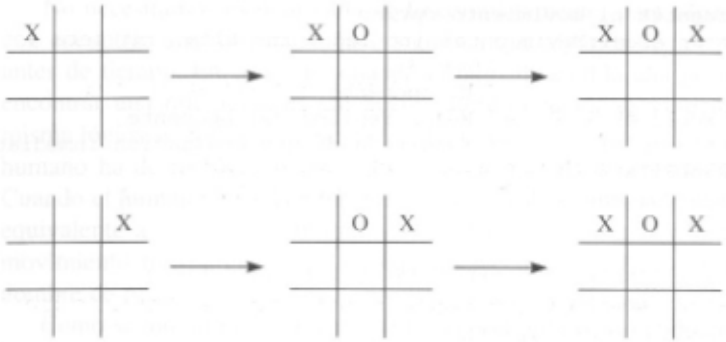


Figura 10.13 Dos búsquedas que llegan a estados idénticos.

```
1  final class Estado implements Hashable
2  {
3      int [ ][ ] tablero;
4      int valor;
5
6      Estado( int elTablero[ ][ ] )
7      {
8          tablero = new int[ 3 ][ 3 ];
9          for( int i = 0; i < 3; i++ )
10             for( int j = 0; j < 3; j++ )
11                 tablero[ i ][ j ] = elTablero[ i ][ j ];
12     }
13
14     public boolean equals( Object lder )
15     {
16         for( int i = 0; i < 3; i++ )
17             for( int j = 0; j < 3; j++ )
18                 if( tablero[ i ][ j ] !=
19                         ((Estado)lder).tablero[ i ][ j ] )
20                     return false;
21         return true;
22     }
23
24     public int hash( int tamTablero )
25     {
26         int valorHash = 0;
27
28         for( int i = 0; i < 3; i++ )
29             for( int j = 0; j < 3; j++ )
30                 valorHash = valorHash * 4 + tablero[ i ][ j ];
31
32         return valorHash % tamTablero;
33     }
34 }
```

Figura 10.14 Clase `Estado`.

Controlamos la profundidad porque no vale la pena incluir todos los estados en la tabla de transposición. La sobrecarga de mantener la tabla sugiere que los estados cercanos a los casos base no deberían guardarse por las siguientes razones:

```
1  class TresEnRaya
2  {
3      private TablaHash transposicion =
4                              new TablaExploracionCuadratica( );
5      publicMejor elegirMovimiento( int lado )
6      {
7          return elegirMovimiento( lado, HUMANO_GANA,
8                                   COMPUTADORA_GANA, 0 );
9      }
10     ...
11 }
```

Figura 10.15 Cambios en la clase `TresEnRaya` para incorporar la tabla de transposición y la poda alfa-beta.

- Hay demasiados estados.
- La idea de la poda alfa-beta y de la tabla de transposición es reducir los tiempos de búsqueda evitando llamadas lo más pronto posible en el juego: ahorrarnos una llamada muy profunda en la búsqueda no reduce un gran número de estados a examinar ya que en cualquier caso dicha llamada solamente examinará unos pocos estados.

Las Figuras 10.16 y 10.17 muestran el nuevo `elegirMovimiento`. En la línea 8 declaramos un objeto `Estado` llamado `estadoActual`. En su momento este objeto se colocará en la tabla de transposición. La variable `profTabla` nos dirá la profundidad hasta la que permitimos guardar estados en la tabla de transposición. Experimentando, averiguamos que 5 es un valor óptimo. Permitir que se guarden estados examinados en `profundidad` 6 es perjudicial, debido a que el coste extra que supone mantener la tabla de transposición no es compensado por el menor número de estados examinados.

Las líneas 17 a 28 son nuevas. Si estamos en la primera llamada a `elegirMovimiento`, inicializamos la tabla de transposición. En caso contrario, si estamos en una profundidad apropiada, comprobamos si ya se ha evaluado el estado, y en tal caso devolvemos su valor. El código tiene dos trucos. En primer lugar, solamente hemos de buscar en la tabla de transposición si `profundidad` es mayor que 3, como se observa en la Figura 10.13. La otra diferencia aparece de la línea 62 en adelante. Inmediatamente antes de devolver el resultado, almacenamos el valor del estado en la tabla de transposición.

El uso de la tabla de transposición en este algoritmo de las tres en raya elimina alrededor de la mitad de los estados a tener en cuenta, con solamente un pequeño coste para las operaciones de la tabla de transposición. Esto significa que prácticamente se dobla la velocidad del programa.

10.2.3 El ajedrez

En un juego complejo como el ajedrez o el juego Go, no es factible recorrer todo el camino hasta los nodos terminales: algunas estimaciones nos dicen que hay unos 10^{100} estados legales en el ajedrez, y ni todos los trucos del mundo reducirían esta cantidad a un nivel manejable. En este caso, tenemos que parar la búsqueda a partir de una cierta profundidad de la recursión. Los nodos en los que se termina la exploración se convierten en nodos terminales, los cuales se evalúan

```
1     // Encuentra el movimiento óptimo
2  private Mejor elegirMovimiento( int lado, int alfa, int beta,
3                                  int profundidad )
4  {
5     int op;              // El oponente
6     Mejor replica;       // Mejor réplica del oponente
7     int evalSimple;      // Resultado de una evaluación inmediata
8     Estado estadoActual = new Estado( tablero );
9     int profTabla = 5;// Máxima profundidad a guardar
10    int mejorFila = 0;
11    int mejorColumna = 0;
12    int valor;
13
14    if( ( evalSimple = valorEstado( ) ) != INCIERTO )
15       return new Mejor( evalSimple );
16
17    if( profundidad == 0 )
18       transposicion.vaciar( );
19    else if( profundidad >= 3 && profundidad <= profTabla )
20    {
21       try
22       {
23          Estado buscaValor = (Estado)
24                              transposicion.buscar( estadoActual );
25          return new Mejor( buscaValor.valor );
26       }
27       catch( ElementoNoEncontrado e ) { } /* Evaluado debajo */
28    }
29
30    if( lado == COMPUTADORA )
31    {
32       op = HUMANO; valor = alfa;
33    }
34    else
35    {
36       op = COMPUTADORA; valor = beta;
37    }
38
39 Exterior:
40    for( int fila = 0; fila < 3; fila++ )
41       for( int columna = 0; columna < 3; columna++ )
42          if( casillaVacia ( fila, columna ) )
43          {
44             lugar( fila, columna, lado );
45             replica = elegirMovimiento( op, alfa, beta,
46                                         profundidad + 1 );
47             lugar( fila, columna, VACIA );
```

Figura 10.16 Algoritmo de las tres en raya con poda alfa-beta y tabla de transposición (parte 1).

mediante una función que trata de estimar el valor del estado. Por ejemplo, en un programa de ajedrez, la función de evaluación calcula la cantidad relativa y fuerza de las piezas, así como otros factores relacionados con la posición.

Los computadores son especialmente hábiles en movimientos que implican profundas combinaciones que terminan en un intercambio de material, debido a que es sencillo evaluar la fuerza de las piezas. Sin embargo, extender la profundidad de la búsqueda solamente un nivel, supone un incremento de la velocidad de procesamiento de un factor de seis (porque el número de estados se incrementa en un factor de 36). Cada nivel extra de búsqueda refuerza en gran manera la habilidad del programa sólo hasta un cierto límite, que parece haber sido ya alcanzado por los mejores programas. Por otra parte, los computadores no son tan buenas en juegos posicionales más estáticos en los que se requieren evaluaciones más sutiles y un mayor conocimiento del juego. Sin embargo, esto sólo se pone en evidencia cuando el computador está compitiendo con un oponente muy hábil. Los populares programas de ajedrez son hoy en día mejores que la mayoría de los jugadores.

```
48                if( lado == COMPUTADORA && replica.valor > valor
49                    || lado == HUMANO && replica.valor < valor )
50                {
51                   if( lado == COMPUTADORA )
52                      alfa = valor = replica.valor;
53                   else
54                      beta = valor = replica.valor;
55
56                   mejorFila = fila; mejorColumna = columna;
57                   if( alfa >= beta )
58                      break Exterior; // Refutación
59                }
60             }
61
62       // Inserta en la tabla de transposición
63    estadoActual.valor = valor;
64    if( profundidad <= profTabla )
65       transposicion.insertar( estadoActual );
66
67    return new Mejor( valor, mejorFila, mejorColumna );
68 }
```

Figura 10.17 Algoritmo de las tres en raya con poda alfa-beta y tabla de transposición (parte 2).

En 1997, el programa *Deep Blue*, fue capaz de derrotar al campeón de ajedrez en un encuentro de seis partidos, usando una enorme capacidad computacional (evaluando unos 200 millones de movimientos por segundo).

Resumen

Este capítulo ha proporcionado una aplicación de la búsqueda binaria y de algunas técnicas algorítmicas que son usadas habitualmente en programas de juegos como el ajedrez, las damas y Otelo. Los mejores programas para estos juegos tienen el nivel de los mejores expertos en ellos. Sin embargo, el juego Go parece ser demasiado complejo para ser bien jugado a base de búsquedas por parte de un computador.

Elementos del juego

estrategia minimax Estrategia recursiva que permite al computador seleccionar un movimiento óptimo en el juego de las tres en raya.

poda alfa-beta Técnica usada para reducir el número de estados evaluados en una búsqueda minimax. Alfa es el valor que el humano ha de rechazar, y beta el valor que el computador ha de rechazar.

posición terminal Posición en un juego que se puede evaluar inmediatamente.

refutación Movimiento del contrario que demuestra que el movimiento original en examen no es una mejora de otros considerados previamente. Si encontramos una refutación, no tenemos que examinar más movimientos por debajo del nivel original y las llamadas recursivas en ejecución terminan.

sopa de letras Programa que lleva a cabo la búsqueda de palabras en un vector bidimensional de letras. Las palabras pueden estar orientadas en ocho direcciones.

tabla de transposición Tabla hash que almacena estados previamente evaluados.

Errores comunes

1. Cuando use una tabla de transposición, limite el número de posiciones almacenadas para evitar agotar la memoria.
2. Es importante comprobar que se cumplen cada una de nuestras suposiciones. Por ejemplo, en la sopa de letras, compruebe que el diccionario está ordenado. Es un error bastante común olvidar esta comprobación.

En Internet

Se proporciona el código completo de los dos ejemplos estudiados, aunque el interfaz del juego de las tres en raya deja bastante que desear. Ambos se encuentran en el directorio **Chapter10**. A continuación se enumeran los nombres de los ficheros:

WordSrch.java	Contiene el algoritmo de la sopa de letras. Se ha traducido aquí por `BusquedaPalabras`.
Best.java	Contiene la clase `Best`, traducida por `Mejor`, que forma parte del algoritmo de las tres en raya.
TicTacToe.java	Contiene una clase `TicTacToe`, traducida por `TresEnRaya`. Utiliza la poda alfa-beta y una tabla de transposición.
TicTacMain.java	Contiene una simple interfaz gráfica de usuario para las tres en raya.

Ejercicios

Cuestiones breves

10.1. ¿Qué comprobaciones de error faltan en la Figura 10.7?

10.2. Para la situación presentada en la Figura 10.18, determine:

a) ¿Cuál de las respuestas al movimiento C_2 es una refutación?

b) ¿Cuál es el valor del estado?

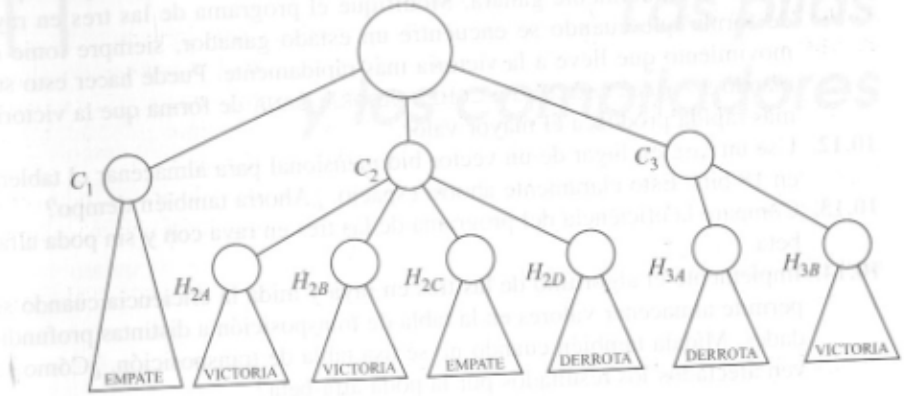


Figura 10.18 Ejemplo de poda alfa-beta para el Ejercicio 10.2.

Problemas teóricos

10.3. Verifique que si `x` es un prefijo de alguna palabra de un vector `a`, entonces `x` es un prefijo de la palabra en la que termina la búsqueda binaria.

10.4. Explique cómo cambia el tiempo de ejecución del algoritmo de la búsqueda de palabras cuando sucede lo siguiente:

a) Se dobla el número de palabras.

b) Se doblan simultáneamente el número de filas y columnas.

10.5. Describa el efecto de escribir un método `Equals` en lugar de un método `equals`, para la clase `Estado`.

Problemas prácticos

10.6. Elimine las restricciones impuestas al tamaño de la sopa de letras y al del diccionario, presentes en la implementación de la clase `BusquedaPalabras`.

10.7. Sustituya la búsqueda binaria por una secuencial en el problema de la búsqueda de palabras. ¿Cómo afecta esto a la eficiencia?

10.8. Compare la eficiencia del algoritmo de búsqueda de palabras con y sin la búsqueda prefijo.

10.9. Implemente una buena interfaz gráfica de usuario para el programa de las tres en raya. Su programa debe trabajar como una aplicación y como un applet.

10.10. Un programa para las tres en raya sin interfaz gráfica de usuario necesita código adicional para leer los movimientos y mostrar el tablero. Escriba una interfaz amigable al usuario.

10.11. Incluso cuando el computador tiene un movimiento que proporciona una victoria inmediata, puede que no lo realice si detecta que otro movimiento también le llevará a ganar. Algunos de los primeros programas de ajedrez tenían el problema de que entraban en una repetición de estado

cuando detectaban una victoria segura, permitiendo al contrario proclamar un empate. En las tres en raya esto no es un problema porque el programa eventualmente ganará. Modifique el programa de las tres en raya de forma que cuando se encuentre un estado ganador, siempre tome el movimiento que lleve a la victoria más rápidamente. Puede hacer esto sumando `9 - profundidad` a `COMPUTADORA_GANA` de forma que la victoria más rápida produzca el mayor valor.

10.12. Use un `int` en lugar de un vector bidimensional para almacenar el tablero en 18 bits. Esto claramente ahorra espacio. ¿Ahorra también tiempo?

10.13. Compare la eficiencia del programa de las tres en raya con y sin poda alfa-beta.

10.14. Implemente el algoritmo de las tres en raya y mida la eficiencia cuando se permite almacenar valores en la tabla de transposición a distintas profundidades. Mídala también cuando no se usa tabla de transposición. ¿Cómo se ven afectados los resultados por la poda alfa-beta?

Prácticas de programación

10.15. Escriba un programa para jugar a las cuatro en raya en un tablero de cinco por cinco. ¿Puede realizar la búsqueda hasta los nodos terminales?

10.16. El juego del boggle consiste en una matriz de letras y una lista de palabras. El objetivo es encontrar palabras en la matriz con la restricción de que dos letras seguidas en las palabras deben estar adyacentes en la matriz (es decir, hacia el norte, sur, este u oeste) y de que cada letra de la matriz sólo se puede usar una vez por palabra. Escriba un programa para jugar al boggle.

10.17. Escriba un programa para jugar a MAXIT. El tablero es una matriz $N \times N$ de números colocados aleatoriamente al principio del juego. Se designa una posición como posición inicial actual. Dos jugadores se van alternando. En cada turno, un jugador debe seleccionar un elemento de la matriz en la fila o columna actuales. El valor de la posición seleccionada se suma a la puntuación del jugador, y dicha posición se convierte en la actual y no se puede elegir de nuevo. Los jugadores se alternan hasta que se han seleccionado todos los elementos de la fila y columna actuales, en cuyo momento termina el juego ganando el jugador con la mayor puntuación.

10.18. Otelo jugado sobre un tablero de 6 por 6 conduce a victoria segura a las fichas negras. Demuestre esto escribiendo un programa. ¿Cuál es la puntuación final si los dos jugadores juegan de forma óptima?

Bibliografía

Si está interesado en juegos de computador, un buen punto de partida es el siguiente artículo, contenido en un número especial dedicado exclusivamente al tema. Encontrará gran cantidad de información y referencias a otros trabajos sobre ajedrez, damas y otros juegos de computador.

1. K. Lee y S. Mahajan, «The Development of a World Class Othello Program», *Artificial Intelligence* **43** (1990), 21-36.

CAPÍTULO

11 Las pilas y los compiladores

Las pilas se utilizan muy a menudo en los compiladores. Este capítulo presenta dos componentes simples de un compilador: un analizador sintáctico de símbolos equilibrados y una calculadora sencilla. El objetivo consiste en presentar algoritmos simples que utilicen pilas para, a través de ellos, ver cómo se utilizan las estructuras de datos descritas en el Capítulo 6.

En este capítulo veremos:

- Cómo utilizar una pila para comprobar si los símbolos de una expresión están equilibrados.
- Cómo utilizar una máquina de estados para hacer el análisis sintáctico de símbolos equilibrados.
- Cómo usar un análisis sintáctico con precedencia de operadores para evaluar expresiones infijas en un programa que implemente una calculadora sencilla.

11.1 Analizador de símbolos equilibrados

Como se discutió en la Sección 6.2, los compiladores comprueban los programas buscando errores sintácticos. Sin embargo, en ocasiones la falta de un símbolo (como la falta de un finalizador de comentario `*/` o `}`) puede causar que el compilador desparrame cientos de líneas de errores sin llegar a identificar el error real. Una herramienta útil para ayudar a la depuración de mensajes de error del compilador es un programa que compruebe si los símbolos están equilibrados. En otras palabras, toda `{` debe tener asociada una `}`, todo `[` un `]`, y así sucesivamente. Sin embargo, contar simplemente el número de apariciones de cada símbolo no es suficiente. Por ejemplo, la secuencia `[()]` es correcta, pero `[(])` no lo es.

11.1.1 El algoritmo básico

En esta situación es útil una pila porque sabemos que cuando encontramos un símbolo de terminación como `)`, debe emparejarse con el `(` anterior más reciente aún no emparejado. Por tanto, colocando los símbolos de apertura en una pila, pode-

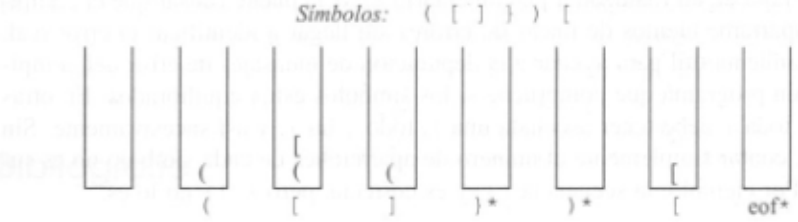
mos fácilmente comprobar si la aparición de un símbolo de terminación es o no oportuna. En concreto, tenemos el siguiente algoritmo:

1. Inicializar una pila vacía.
2. Leer símbolos hasta el final del fichero.
   a) Si el símbolo es de apertura, apilarlo en la pila.
   b) Si el símbolo es de terminación y la pila está vacía, producir un error.
   c) En otro caso, desapilar la cima de la pila. Si el símbolo desapilado no es el correspondiente símbolo de apertura, producir un error.
3. Al finalizar el fichero, si la pila no está vacía, producir un error.

En este algoritmo, ilustrado en la Figura 11.1, los símbolos cuarto, quinto y sexto generan errores. La) es un error porque el símbolo desapilado es (, por lo que se detecta un mal emparejamiento. El) provoca un error porque la pila está vacía, de modo que no existe el correspondiente (. Finalmente, el [provoca un error, que se detecta cuando finaliza el fichero y la pila no está vacía.

Para que esto funcione con programas en Java, necesitamos considerar todos los contextos en los cuales los paréntesis, las llaves y los corchetes no tienen por qué estar emparejados. Por ejemplo, no deberíamos considerar un paréntesis como un símbolo cuando aparece dentro de un comentario, de una cadena de caracteres constante o es una constante carácter. Necesitamos, por tanto, rutinas que se salten los comentarios, las cadenas constantes y los caracteres constantes. En Java es bastante complejo detectar una constante carácter debido a las secuencias de escape, por lo que simplificaremos las hipótesis de trabajo. El objetivo será diseñar un programa que funcione para la mayor parte de entradas comunes.

Para que el programa sea útil, no sólo debe mostrar los fallos de emparejamiento, sino también identificar la línea donde aparecen. En consecuencia, llevaremos cuenta del número de línea donde se encuentra cada símbolo leído. Cuando se detecta un error, componer el mensaje apropiado siempre resulta complicado. Si hay una } de más, ¿significa que sobra la }, o que falta una {? Mantendremos el tratamiento de errores tan simple como sea posible. Irremediablemente es posible que una vez que se muestra un error el programa se desoriente y empiece a mostrar muchos errores. Por tanto, sólo debemos, en principio, buscar la causa del primer error detectado en cada zona del texto examinado. A pesar de todas estas limitaciones, el programa que desarrollamos resulta bastante útil.




```
Errores (indicados con *):
    } cuando se esperaba un )
    ) con ningún símbolo de apertura con el que emparejarlo
    [ sin pareja al terminar la entrada
```

Figura 11.1 Operaciones sobre la pila en el algoritmo de comprobación de símbolos equilibrados.

11.1.2 Implementación

La Figura 11.2 muestra la clase `AnalizadorJava`, que realiza todo el trabajo. Además del constructor, la única otra rutina visible públicamente es `comprobarEquilibrados`, que aparece en la línea 26. El resto son rutinas auxiliares o atributos de la clase. Empezamos describiendo los atributos de la clase.

```
1  // Clase AnalizadorJava: comprobación de símbolos equilibrados
2  //
3  // CONSTRUCCIÓN: con un objeto PushbackReader
4  // ********************OPERACIONES PÚBLICAS*********************
5  // int comprobarEquilibrados( ) --> Imprime los errores y
6  //                                  devuelve el número de errores
7  // ***************************ERRORES*************************
8  // Se producen errores al comprobar comentarios o cadenas
9
10 import java.io.*;
11 import EstructurasDatos.*;
12 import Excepciones.*;
13 import Soporte.*;
14
15 class AnalizadorJava
16 {
17     public AnalizadorJava( PushbackReader cadenaEntrada )
18     {
19         errores = 0;
20         car = '\0';
21         lineaActual = 1;
22         entrada = cadenaEntrada;
23         tokensPendientes = new PilaVec( );
24     }
25
26     public int comprobarEquilibrados( )
27       { /* Figura 11.8 */ }
28
29     private PushbackReader entrada; // La cadena de entrada
30     private char car;               // Carácter actual
31     private int lineaActual;        // Línea actual
32     private Pila tokensPendientes;  // Símbolos de apertura
                                       pendientes
33     private int errores;            // Número de errores encontrados
34
35     private boolean siguienteCar( )
36       { /* Figura 11.4 */ }
37     private void devolverCar( )
38       { /* Figura 11.4 */ }
39     private void saltarComentario( int comienzo )
40       { /* Figura 11.5 */ }
41     private void saltarCadena( char tipoCadena )
42       { /* Figura 11.6 */ }
43     private boolean siguienteSimbolo( )
44       { /* Figura 11.7 */ }
45     private void procesarBarra( )
46       { /* Figura 11.7 */ }
47     private void comprobarPareja( Simbolo simAper, Simbolo simFin )
48       { /* Figura 11.9 */ }
49 }
```

Figura 11.2 Esqueleto de la clase para el programa de comprobación de símbolos equilibrados

`entrada` es un objeto de la clase `PushbackReader` y se inicializa en el constructor. Un objeto de la clase `PushbackReader` es como un `BufferedReader`, excepto por el hecho de que también proporciona un método `unread`. El carácter actual que está siendo explorado se almacena en `car`, y el número de línea actual se almacena en `lineaActual`. El algoritmo para comprobar símbolos equilibrados necesita que coloquemos los símbolos de apertura en una pila. Para poder imprimir diagnósticos, almacenamos un número de línea con cada símbolo, como se muestra en la clase `Simbolo` de la Figura 11.3. La pila se llama `tokensPendientes`, y se declara en la línea 32. Finalmente, en la línea 33 se declara un entero para contar el número de errores encontrados.

El constructor, mostrado en las líneas de la 17 a la 24, inicializa el contador de errores a 0 y el número de línea actual a 1, y da valor a la referencia `PushbackReader`. El resto de atributos, a saber, `car` y `tokensPendientes`, se inizializan adecuadamente. En el caso de `tokensPendientes`, se utiliza un constructor sin parámetros para crear una pila vacía.

Podemos pasar a examinar algunas de las rutinas auxiliares. Muchas de ellas tienen que ver con el tratamiento de la línea actual, y con la diferenciación de símbolos que representan tokens de apertura o cierre de aquellos que están dentro de comentarios, cadenas o constantes de caracteres. Todo esto se engloba dentro del denominado *análisis léxico*. La Figura 11.4 muestra un par de rutinas, `siguienteCar` y `devolverCar`. `siguienteCar` lee el siguiente carácter, lo asigna a `car`, y actualiza `lineaActual` cuando se trate de un salto de línea. Finalmente, devuelve `false` cuando se alcanza el final de fichero. El procedimiento complementario, `devolverCar`, coloca de nuevo el carácter (`car`) en la cadena de entrada, y decrementa `lineaActual` cuando ello es oportuno. Es claro que `devolverCar` debería llamarse sólo una vez entre dos llamadas a `siguienteCar`. Ya que es una rutina privada, no nos preocupamos de abusos por parte de la clase usuaria.

Devolver caracteres a la línea de entrada es una técnica común en el análisis sintáctico. En muchos casos, se han leído demasiados caracteres y es preciso deshacer el proceso de lectura. En nuestro caso, esto ocurre cuando procesamos una `/`. Debemos ver si el siguiente carácter corresponde al token de comienzo de comentario. Sin embargo, cuando ello no es así, no podemos ignorar este último carácter leído, pues podría ser un símbolo de apertura o cierre, o unas comillas. Por tanto, debemos actuar como si todavía no hubiera sido leído.

```
1  /**
2   * Símbolo que representa el objeto que se apilará.
3   */
4  class Simbolo
5  {
6      char token;
7      int  laLinea;
8
9      Simbolo( char tok, int linea )
10     {
11         token = tok;
12         laLinea = linea;
13     }
14 }
```

Figura 11.3 Clase de objetos que se insertan en la pila.

```
1  /**
2   * siguienteCar da valor a car basándose en el siguiente carácter
3   * en la línea de entrada. devolverCar devuelve el carácter a la
4   * cadena. Sólo debe utilizarse una vez después de siguienteCar.
5   * Ambas rutinas actualizan lineaActual cuando ello es necesario.
6   */
7  private boolean siguienteCar( )
8  {
9      try
10     {
11         int valorLeido = entrada.read( );
12         if( valorLeido == -1 )
13             return false;
14         car = (char) valorLeido;
15         if( car == '\n' )
16             lineaActual++;
17         return true;
18     }
19     catch( IOException e )
20       { return false; }
21 }
22
23 private void devolverCar( )
24 {
25     if( car == '\n' )
26         lineaActual--;
27     try
28       { entrada.unread( (int) car ); }
29     catch( IOException e ) { }
30 }
```

Figura 11.4 Rutina `siguienteCar` para leer el siguiente carácter, actualizar `lineaActual` cuando ello es necesario y devolver `true` cuando no estamos al final del fichero. Rutina `devolverCar` para devolver `car` a la entrada y actualizar `lineaActual` cuando ello es necesario.

La siguiente rutina a comentar es `saltarComentario`, mostrada en la Figura 11.5. El propósito de esta rutina es saltarse los caracteres de un comentario, para que después se siga leyendo la entrada a partir del siguiente carácter tras la finalización del comentario. Las cosas se complican por el hecho de que los comentarios pueden empezar bien por `//`, en cuyo caso el comentario termina con la línea, o con `/*`, en cuyo caso `*/` termina el comentario. Cuando estamos en el primer caso, leemos repetidamente el siguiente carácter, hasta que se alcanza el final del fichero (en cuyo caso, la primera parte del operador `&&` falla) o el final de línea. Tras hacerlo acabamos. Observe que el número de línea es actualizado automáticamente por `siguienteCar`. El caso `/*` se procesa a partir de la línea 17. La posible aparición de `/**` se trata antes de llamar a esta rutina (*véase* la Figura 11.7, líneas 17 y 18).

La rutina `saltarComentario` utiliza una *máquina de estados* simplificada. La máquina de estados es una técnica común en el análisis de símbolos. En cada momento la máquina se encuentra en un determinado estado; cada nuevo carácter de entrada, nos lleva a un nuevo estado. Si todo va bien acabamos por alcanzar un estado en el cual el símbolo ha sido reconocido.

```
1  /**
2   * Precondición: Estamos a punto de procesar un comentario; hemos
3   *               encontrado ya el token de comienzo de comentario.
4   * Postcondición: La entrada seguirá leyendo justo a partir
5   *                del token de finalización de comentario
6   */
7  private void saltarComentario( int comienzo )
8  {
9      if( comienzo == SLASH_SLASH )
10     {
11         while( siguienteCar( ) && ( car != '\n' ) )
12             ;
13         return;
14     }
15
16         // Buscar la secuencia */
17     boolean estado = false; // true si hemos encontrado *
18
19     while( siguienteCar( ) )
20     {
21         if( estado && car == '/' )
22             return;
23         estado = ( car == '*' );
24     }
25     errores++;
26     System.out.println( "¡Comentario sin terminar!" );
27 }
```

Figura 11.5 Rutina `saltarComentario`, para colocarse después de un comentario ya empezado.

En la rutina `saltarComentario`, en cada momento se habrán tratado 0, 1 o 2 caracteres del finalizador `*/`, correspondientes a los estados 0, 1 y 2. Si ya se han leído los dos caracteres, podemos terminar. Por tanto, dentro del bucle sólo podemos estar en los estados 0 o 1, ya que si estamos en el estado 1 y encontramos una `/`, terminamos inmediatamente. En consecuencia, el estado puede representarse mediante una variable booleana que es cierta cuando la máquina está en el estado 1. Si no acabamos, volvemos al estado 1 cuando encontramos un `*`, o al estado 0, en otro caso. Esto se hace en la línea 23.

Si nunca se encuentra el token de finalización de comentario, `siguienteCar` acabará devolviendo `false` y el bucle `while` terminará, provocando un mensaje de error. `saltarCadena`, mostrada en la Figura 11.6, es similar. Aquí, el parámetro es el carácter de comienzo de cadena, que puede ser `"` o `'`. En cada caso, debemos examinar el carácter que termina la cadena. Además, debemos estar preparados para tratar el carácter `\`; en otro caso, el programa produciría errores cuando se ejecutara, por ejemplo, sobre su propio código. Vamos, pues, tratando caracteres. Cuando el carácter leído es una comilla de cierre, hemos terminado. Si encontramos un fin de línea, tenemos una constante carácter o cadena sin terminar. Por último, si encontramos una `\`, leemos el siguiente carácter sin examinarlo.

Una vez escrita la rutina de salto, es fácil escribir `siguienteSimbolo`. Si el carácter actual es `/`, llamamos a `procesarBarra` para leer el siguiente carácter

```
1  /**
2   * Precondición: Estamos a punto de procesar una cadena; ya hemos
3   *               encontrado la primera comilla.
4   * Postcondición: La entrada seguirá leyendo justo a partir
5   *                de la comilla de cierre correspondiente.
6   */
7  private void saltarCadena( char tipoCadena )
8  {
9      while( siguienteCar( ) )
10     {
11         if( car == tipoCadena )
12             return;
13         if( car == '\n' )
14         {
15             errores++;
16             System.out.println( "Falta cerrar cadena en la línea " +
17                                 lineaActual );
18             return;
19         }
20         else if( car == '\\' )
21             siguienteCar( );
22     }
23 }
```

Figura 11.6 Rutina `saltarCadena`, para colocarnos justo después de una constante carácter o cadena ya empezada.

comprobando si hemos encontrado un comentario; si no, deshacemos la segunda lectura. Si tenemos una comilla, llamamos a `procesarCadena`. Si tenemos un símbolo de apertura o cierre, podemos acabar. En otro caso, seguimos leyendo hasta que se acabe la entrada o se encuentre un símbolo de apertura o cierre. La rutina completa se muestra en la Figura 11.7.

`comprobarEquilibrados` está implementado en la Figura 11.8. Sigue la descripción del algoritmo casi al pie de la letra. Los símbolos de apertura se apilan en la pila junto con el número de línea actual. Cuando se encuentra un símbolo de cierre y la pila está vacía, el símbolo es inesperado; en otro caso, eliminamos el elemento de la cima y comprobamos si el símbolo de apertura que estaba en la pila casa con el símbolo de finalización que se acaba de leer. Esto se hace en la rutina `comprobarPareja`, mostrada en la Figura 11.9. Una vez alcanzado el final del fichero, todos los símbolos en la pila no tendrían pareja; éstos, de existir alguno, son mostrados en la salida por el bucle `while` que empieza en la línea 41. Se devuelve el número total de errores detectados.

Observe que la implementación actual permite varias llamadas a `comprobarEquilibrados`. Sin embargo, si la entrada no se reinicializa externamente, lo único que ocurrirá es que se detectará inmediatamente el fin de fichero, acabando inmediatamente. La Figura 11.10 muestra que lo que se espera es que se cree un objeto de la clase `AnalizadorJava` y se llame a `comprobarEquilibrados`. En nuestro ejemplo, si no hay argumentos en la línea de comandos, un objeto de la clase `PushbackReader` se asocia con `System.in`. En otro caso, se utilizan repetidamente objetos de la clase `PushbackReader` asociados a los ficheros indicados en la lista de argumentos suministrada en la línea de comandos.

```
/**
 * Después de encontrar la barra /, tratar el siguiente
 * carácter. Si comienza un comentario, tratarlo;
 * en caso contrario, devolver el carácter si no se trata
 * del carácter de nueva línea
 */
private static final int SLASH_SLASH = 0;
private static final int SLASH_STAR = 1;

private void procesarBarra( )
{
    if( siguienteCar( ) )
    {
        if( car == '*' )
        {
            // comentario para Javadoc
            if( siguienteCar( ) && car != '*' )
                devolverCar( );
            saltarComentario( SLASH_STAR );
        }
        else if( car == '/' )
            saltarComentario( SLASH_SLASH );
        else if( car != '\n' )
            devolverCar( );
        }
    }

    /**
     * Llegar al siguiente símbolo de apertura o cierre.
     * Devolver false si se acaba el fichero.
     * Saltar comentarios y constantes carácter o cadena
     */
    private boolean siguienteSimbolo( )
    {
        while( siguienteCar( ) )
        {
            if( car == '/' )
                procesarBarra( );
            else if( car == '\'' || car == '"' )
                saltarCadena( car );
            else if( car == '(' || car == '[' || car == '{' ||
                     car == ')' || car == ']' || car == '}' )
                return true;
        }
        return false; // Final de fichero
}
```

Figura 11.7 Rutina `siguienteSimbolo`, para saltar comentarios y cadenas, y devolver el siguiente símbolo de apertura o cierre.

```
/**
 * Imprime un mensaje de error en caso de detectar desequilibrios.
 * @return número de errores detectados.
 */
public int comprobarEquilibrados ( )
{
    Simbolo pareja = null;

    errores = 0;
    lineaActual = 1;
    while( siguienteSimbolo( ) )
    {
        char ultimoCar = car;
        Simbolo ultimoSimbolo = new Simbolo( ultimoCar,
                                             lineaActual );

        switch( ultimoCar )
        {
          case '(': case '[': case '{':
            tokensPendientes.apilar( ultimoSimbolo );
            break;
          case ')': case ']': case '}':
            try
            {
                pareja = (Simbolo)
                         tokensPendientes.cimaYDesapilar( );
                comprobarPareja( pareja, ultimoSimbolo );
            }
            catch( DesbordamientoInferior e )
            {
                errores++;
                System.out.println( "Inesperado " + ultimoCar +
                    " en línea " + lineaActual );
            }
            break;
          default: // no puede ocurrir
            break;
        }
    }

    while( !tokensPendientes.esVacia( ) )
    {
        errores++;
        try
          { pareja = (Simbolo) tokensPendientes.cimaYDesapilar( ); }
        catch( DesbordamientoInferior e ) { } // No puede ocurrir
        System.out.println( pareja.token + " no emparejado " +
                            " en la línea " + pareja.laLinea );
    }
    return errores;
}
```

Figura 11.8 `comprobarEquilibrados`, el algoritmo principal

```
// Imprime un mensaje de error si simAper no casa con simFin.
// Actualiza los errores.
private void comprobarPareja( Simbolo simAper, Simbolo simFin )
{
    if( simAper.token == '(' && simFin.token != ')' ||
        simAper.token == '[' && simFin.token != ']' ||
        simAper.token == '{' && simFin.token != '}' )
    {
        System.out.println( "Encontrado " + simAper.token +
          " en la línea " + lineaActual +"; no casa con " +
          simFin.token + " en la línea " + simAper.laLinea );
        errores++;
    }
}
```

Figura 11.9 Rutina `comprobarPareja`, para comprobar que el símbolo de terminación casa con el símbolo de apertura.

```
// Rutina principal del comprobador de símbolos desequilibrados.
public static void main( String [ ] args )
{
    AnalizadorJava p;

    if( args.length == 0 )
    {
        p = new AnalizadorJava( new PushbackReader(
                new InputStreamReader( System.in ) ) );
        if( p.comprobarEquilibrados( ) == 0 )
            System.out.println( "¡No hay errores!" );
        return;
    }

    for( int i = 0; i < args.length; i++ )
    {
       try
       {
          FileReader f = new FileReader( args[ i ] );

          System.out.println( args[ i ] + ": " );
          p = new AnalizadorJava( new
                                  PushbackReader( f ) );
          if( p.comprobarEquilibrados( ) == 0 )
              System.out.println( " ...sin errores" );
          f.close( );
       }
       catch( IOException e )
         { System.err.println( e + args[ i ] ); }
    }
}
```

Figura 11.10 Rutina `main`, con argumentos de la línea de comandos

11.2 Una calculadora sencilla

Algunas de las técnicas utilizadas en el diseño de compiladores pueden utilizarse a pequeña escala en la implementación de una calculadora de bolsillo típica. Las calculadoras evalúan expresiones infijas, como `1+2`, que consisten en un operador binario junto con argumentos a su izquierda y a su derecha. Este formato, bastante fácil de evaluar en un principio, puede hacerse más complejo. Considere la expresión

```
1 + 2 * 3
```

Matemáticamente, la expresión se evalúa a `7`, porque el operador de multiplicación tiene mayor precedencia que la suma. Sin embargo, algunas calculadoras darían como respuesta `9`. Esto ilustra que una simple evaluación de izquierda a derecha no es suficiente. No podemos empezar evaluando `1+2`. Considere también las expresiones

```
10 - 4 - 3
2 ^ 3 ^ 3
```

donde `^` es el operador de exponenciación. ¿Qué resta y qué exponenciación debe evaluarse primero? Las restas se evalúan de izquierda a derecha, dando como resultado `3`. Por el contrario, las potencias se evalúan, generalmente, de derecha a izquierda, por lo que la expresión reflejaría 2^{3^3} en lugar de $(2^3)^3$. Es decir, la resta asocia de izquierda a derecha, mientras que la exponenciación asocia de derecha a izquierda. Todas estas posibilidades sugieren que la evaluación de una expresión como

```
1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2
```

representaría un serio reto.

Si los cálculos se realizan con matemática entera (es decir, redondeando hacia abajo en la división), la respuesta sería `-8`. Para probar esto, introducimos paréntesis para ilustrar el orden correcto de los cálculos:

```
(1 - 2) - ( ( ( 4 ^ 5 ) * 3 ) * 6 ) / ( 7 ^ ( 2 ^ 2 ) )
```

Aunque los paréntesis desambiguan el orden de evaluación, es difícil argumentar que hacen más claro el mecanismo de evaluación. Resulta que una forma de expresiones diferente, denominada *expresión postfija*, proporciona un mecanismo directo de evaluación. Las siguientes secciones muestran cómo funciona. En la primera estudiamos la notación postfija, mostrando cómo las expresiones escritas con esta notación pueden evaluarse con un simple recorrido de izquierda a derecha. La siguiente sección muestra cómo las expresiones originales, que utilizan la notación habitual infija, se pueden convertir a notación postfija. Finalmente, se presenta un programa Java, que evalúa expresiones infijas que contengan operadores aditivos, multiplicativos y de exponenciación, así como paréntesis. Se utiliza un algoritmo de evaluación de expresiones con precedencia entre operadores.

11.2.1 Máquinas postfijas

Una *expresión postfija* está formada por una serie de operadores y operandos. Se evalúa utilizando una *máquina postfija*, en la siguiente forma: cuando se encuentra un operando, se apila en la pila; cuando se encuentra un operador, el número apropiado de operandos son desapilados de la pila, se evalúa la operación, y el resultado se apila de nuevo en la pila. Para operadores binarios, que son los más comunes, dos operandos son desapilados. Cuando la expresión postfija completa ha sido procesada, el resultado debería ser el único elemento en la pila. La notación postfija es una forma natural de evaluar expresiones pues con ella no son necesarias reglas de precedencia.

Presentamos a continuación un ejemplo sencillo. Consideremos la expresión postfija

```
1 2 3 * +
```

La evaluación procede de la siguiente manera: el `1`, el `2` y el `3` son apilados, en ese orden, en la pila. Para procesar el `*`, se desapilan los dos elementos superiores de la pila: esto es, el `3` y después el `2`. Observe que el primer elemento desapilado se convierte en el parámetro derecho del operador, y el segundo en el izquierdo; por tanto, los parámetros se obtienen de la pila en orden inverso al natural. Para la multiplicación, esto no importa, pero para la resta y la división, desde luego que sí. El resultado de la multiplicación es `6`, y es apilado en la pila. En este momento la cima de la pila es un `6`; y debajo hay un `1`. Para procesar el `+`, se desapilan el `6` y el `1`, y su suma, `7`, se apila. En este punto, la expresión se ha leído completamente, y la pila tiene sólo un elemento. Por tanto, la respuesta final es `7`.

Toda expresión infija válida puede convertirse a notación postfija. Por ejemplo, la expresión infija vista antes puede escribirse en notación postfija como

```
1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -
```

La Figura 11.11 muestra los pasos utilizados por la máquina postfija para evaluarla. Cada paso involucra un apilamiento. Por tanto, ya que hay 9 operandos y 8 operadores, hay 17 pasos y 17 apilamientos. Claramente, el tiempo necesario para evaluar una expresión postfija es lineal.

El punto que falta por estudiar es un algoritmo que convierta notación infija en notación postfija. Una vez tengamos uno, tendremos un algoritmo para evaluar expresiones infijas.

11.2.2 Conversión de notación infija a postfija

El principio básico involucrado en el algoritmo de *análisis sintáctico de expresiones con precedencia entre operadores*, que convierte una expresión infija en una postfija, es el siguiente. Cuando se encuentra un operando, podemos pasarlo inmediatamente a la salida. Sin embargo, cuando encontramos un operador, no podemos pasarlo todavía a la salida, pues al efecto debemos esperar a encontrar su segundo operando. Por consiguiente, debemos guardarlo temporalmente en una estructura adecuada. Si consideramos una expresión como

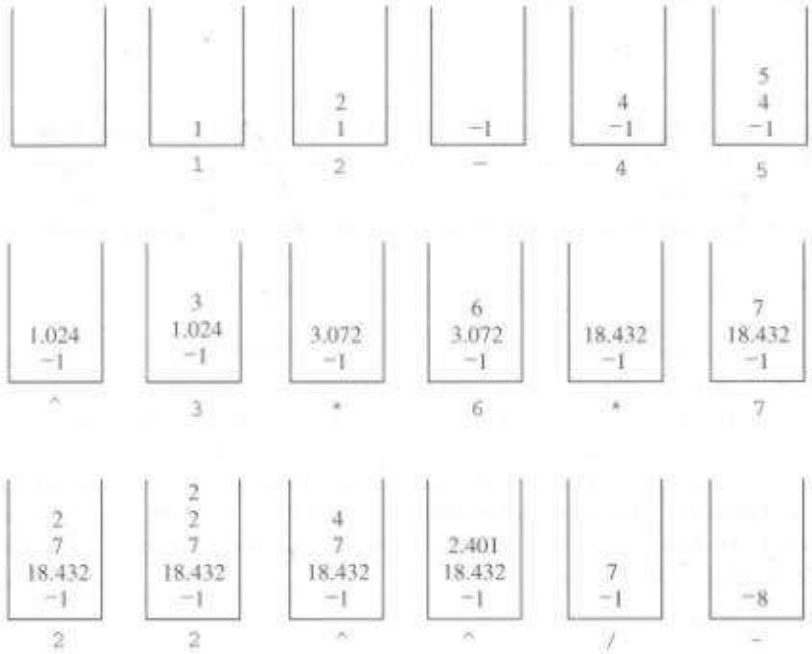


Figura 11.11 Pasos para la evaluación de una expresión postfija.

```
1 + 2 * 3 ^ 4
```

la cual en notación postfija es

```
1 2 3 4 ^ * +
```

observamos que en ocasiones los operadores pueden aparecer en orden inverso a como aparecían en la expresión infija. Por supuesto, esto sólo es cierto si la precedencia de los operadores involucrados crece al ir de izquierda a derecha. Aún así, el hecho comentado sugiere que una pila es la estructura apropiada para almacenar los operadores pendientes. Siguiendo esta lógica, cuando encontramos un operador, debe ser colocado de alguna manera en la pila. El resto del algoritmo tiene que ver con cuándo los operadores entran y salen de la pila.

Consideramos ahora otra expresión infija más simple:

```
2 ^ 5 - 1
```

Cuando alcanzamos el operador `-`, el `2` y el `5` se han incorporado ya a la salida, y `^` está en la pila. Ya que `-` tiene menor precedencia que `^`, `^` tiene que aplicarse al `2` y al `5`. Por lo que debemos desapilar `^` y, en general, cualquier otro operador

que hubiese en la cima de la pila con mayor precedencia que -. En consecuencia, la expresión resultante postfija es

```
2 5 ^ 1 -
```

En general, cuando procesamos un operador de la entrada, debemos sacar de la pila aquellos operadores que deban ser procesados atendiendo a las reglas de precedencia y asociatividad.

Un segundo ejemplo, algo más complejo, lo representa la expresión infija

```
3 * 2 ^ 5 - 1
```

Cuando se alcanza el operador ^, el 3 y el 2 han sido incorporados a la salida, y el * está en la pila. Ya que ^ tiene mayor precedencia que *, no se desapila nada, apilándose encima el ^. Tras ello, el 5 se muestra inmediatamente, para luego encontrarnos con el -. Las reglas de precedencia nos dicen que desapilemos el ^, seguido del *. En este punto, no hay nada más que desapilar, por lo que dejamos de hacerlo, apilando el -. Después incorporamos a la salida el 1. Cuando se alcanza el final de la expresión, se deben sacar el resto de los operadores en la pila. La expresión postfija resultante es

```
3 2 5 ^ * 1 -
```

Antes de resumir el algoritmo, deben responderse algunas preguntas. En primer lugar, si el símbolo actual es un + y la cima de la pila es un +, ¿debería desapilarse el + de la pila, o debería quedarse ahí? La respuesta se obtiene decidiendo si el + encontrado en la entrada nos indica que el + de la pila cuenta ya con sus operandos. Ya que el + asocia de izquierda a derecha, la respuesta es afirmativa. Sin embargo, si se tratara del operador ^, que asocia de derecha a izquierda, la respuesta sería negativa. Por tanto, cuando nos topamos con dos operadores de igual precedencia, hay que examinar las normas de asociatividad para decidir correctamente, como muestran los siguientes ejemplos:

Expresión infija	Expresión postfija	Asociatividad
2 +3 + 4	2 3 + 4 +	Asociatividad por la izquierda: el + en la entrada es menor que el + de la pila.
2 ^ 3 ^ 4	2 3 4 ^ ^	Asociatividad por la derecha: el ^ en la entrada es mayor que el ^ de la pila.

¿Qué ocurre con los paréntesis? Un paréntesis izquierdo puede considerarse como un operador de máxima precedencia cuando está en la entrada, pero de precedencia mínima cuando está en la pila. En consecuencia, el paréntesis izquierdo de la entrada, se apilará siempre sin más. Cuando encontremos un paréntesis derecho en la entrada, desapilaremos hasta encontrar el correspondiente paréntesis izquierdo. Por otra parte, en la salida no aparecen paréntesis.

Presentamos ahora un resumen de los diferentes casos del algoritmo de análisis sintáctico de expresiones con precedencia de operadores. Todo lo que se desapila es mostrado en la salida, a excepción de los paréntesis.

- *Operandos*: Pasan inmediatamente a la salida.
- *Paréntesis derecho*: Desapilar símbolos hasta encontrar un paréntesis izquierdo.
- *Operador*: Desapilar todos los símbolos hasta que encontremos un símbolo de menor precedencia o un símbolo de igual precedencia con asociatividad por la derecha. Apilar entonces el operador encontrado.
- *Fin de la entrada*: Desapilar el resto de símbolos en la pila.

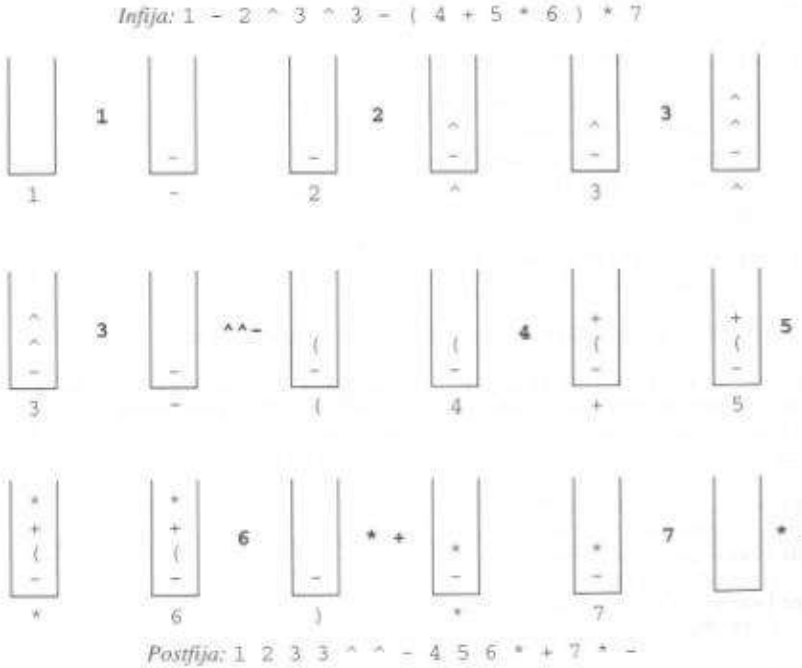
Como ejemplo, la Figura 11.12 muestra cómo el algoritmo procesa la expresión

```
1 - 2 ^ 3 ^ 3 - ( 4 + 5 * 6 ) * 7
```

Debajo de cada pila se muestra el símbolo leído. A la derecha, en negrita, se muestra el estado de la salida.

11.2.3 Implementación

Contamos ahora con el fundamento teórico necesario para implementar una calculadora simple. Nuestra calculadora soportará sumas, restas, multiplicaciones, divisiones y exponenciaciones. Desarrollaremos una clase denominada `Evaluador`. Haremos antes una hipótesis simplificadora: no se permitirán números negativos. La



que hubiese en la cima de la pila con mayor precedencia que -. En consecuencia, la expresión resultante postfija es

```
2 5 ^ 1 -
```

En general, cuando procesamos un operador de la entrada, debemos sacar de la pila aquellos operadores que deban ser procesados atendiendo a las reglas de precedencia y asociatividad.

Un segundo ejemplo, algo más complejo, lo representa la expresión infija

```
3 * 2 ^ 5 - 1
```

Cuando se alcanza el operador ^, el 3 y el 2 han sido incorporados a la salida, y el * está en la pila. Ya que ^ tiene mayor precedencia que *, no se desapila nada, apilándose encima el ^. Tras ello, el 5 se muestra inmediatamente, para luego encontrarnos con el -. Las reglas de precedencia nos dicen que desapilemos el ^, seguido del *. En este punto, no hay nada más que desapilar, por lo que dejamos de hacerlo, apilando el -. Después incorporamos a la salida el 1. Cuando se alcanza el final de la expresión, se deben sacar el resto de los operadores en la pila. La expresión postfija resultante es

```
3 2 5 ^ * 1 -
```

Antes de resumir el algoritmo, deben responderse algunas preguntas. En primer lugar, si el símbolo actual es un + y la cima de la pila es un +, ¿debería desapilarse el + de la pila, o debería quedarse ahí? La respuesta se obtiene decidiendo si el + encontrado en la entrada nos indica que el + de la pila cuenta ya con sus operandos. Ya que el + asocia de izquierda a derecha, la respuesta es afirmativa. Sin embargo, si se tratara del operador ^, que asocia de derecha a izquierda, la respuesta sería negativa. Por tanto, cuando nos topamos con dos operadores de igual precedencia, hay que examinar las normas de asociatividad para decidir correctamente, como muestran los siguientes ejemplos:

Expresión infija	Expresión postfija	Asociatividad
2 +3 + 4	2 3 + 4 +	Asociatividad por la izquierda: el + en la entrada es menor que el + de la pila.
2 ^ 3 ^ 4	2 3 4 ^ ^	Asociatividad por la derecha: el ^ en la entrada es mayor que el ^ de la pila.

¿Qué ocurre con los paréntesis? Un paréntesis izquierdo puede considerarse como un operador de máxima precedencia cuando está en la entrada, pero de precedencia mínima cuando está en la pila. En consecuencia, el paréntesis izquierdo de la entrada, se apilará siempre sin más. Cuando encontremos un paréntesis derecho en la entrada, desapilaremos hasta encontrar el correspondiente paréntesis izquierdo. Por otra parte, en la salida no aparecen paréntesis.

Presentamos ahora un resumen de los diferentes casos del algoritmo de análisis sintáctico de expresiones con precedencia de operadores. Todo lo que se desapila es mostrado en la salida, a excepción de los paréntesis.

- *Operandos*: Pasan inmediatamente a la salida.
- *Paréntesis derecho*: Desapilar símbolos hasta encontrar un paréntesis izquierdo.
- *Operador*: Desapilar todos los símbolos hasta que encontremos un símbolo de menor precedencia o un símbolo de igual precedencia con asociatividad por la derecha. Apilar entonces el operador encontrado.
- *Fin de la entrada*: Desapilar el resto de símbolos en la pila.

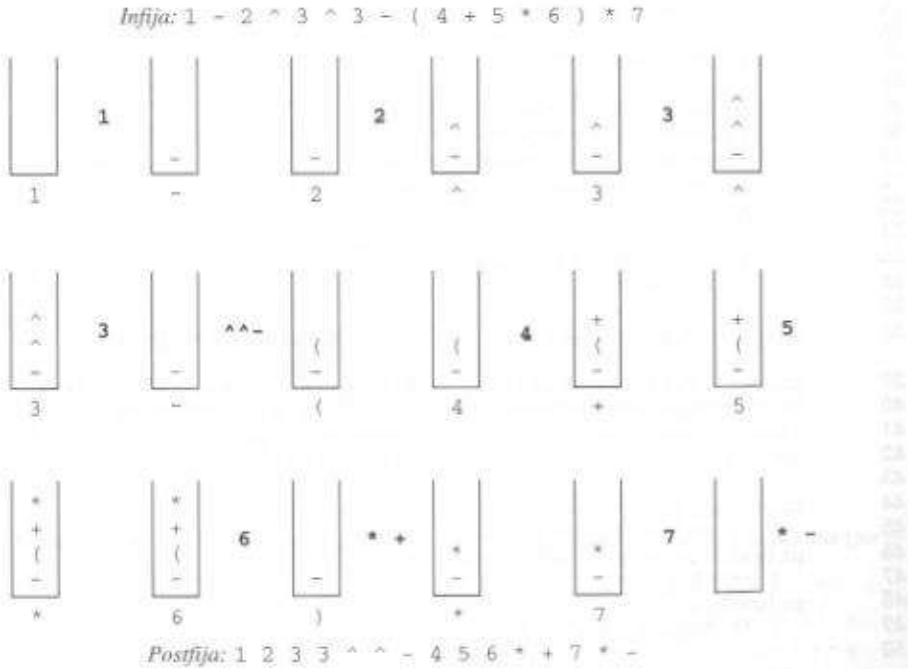
Como ejemplo, la Figura 11.12 muestra cómo el algoritmo procesa la expresión

```
1 - 2 ^ 3 ^ 3 - ( 4 + 5 * 6 ) * 7
```

Debajo de cada pila se muestra el símbolo leído. A la derecha, en negrita, se muestra el estado de la salida.

11.2.3 Implementación

Contamos ahora con el fundamento teórico necesario para implementar una calculadora simple. Nuestra calculadora soportará sumas, restas, multiplicaciones, divisiones y exponenciaciones. Desarrollaremos una clase denominada `Evaluador`. Haremos antes una hipótesis simplificadora: no se permitirán números negativos. La



distinción entre el operador menos binario y el menos unario requiere un trabajo extra en la rutina de análisis. También complica las cosas, al deber ser tratado como un operador que no es binario. Es cierto que la incorporación de operadores unarios no es demasiado difícil; sin embargo, el código extra no ilustra ningún concepto nuevo, por lo que se deja como ejercicio.

La Figura 11.13 muestra el esqueleto de la clase `Evaluador`, utilizada para leer una cadena de la entrada. El algoritmo básico de evaluación requiere dos pilas. La

```
1  // Clase Evaluador: evalúa expresiones infijas
2  //
3  // CONSTRUCCIÓN: con un valor de tipo String
4  //
5  // ******************OPERACIONES PÚBLICAS*********************
6  // long obtenerValor( ) --> Devuelve el valor de una expresión infija
7  // ************************ERRORES*****************************
8  // Se realiza alguna comprobación de errores
9
10 import EstructurasDatos.*;
11 import Excepciones.*;
12 import java.io.*;
13 import java.util.StringTokenizer;
14
15 public class Evaluador
16 {
17     static final int EOL      = 0;
18     static final int VALOR    = 1;
19     static final int PAREN_A  = 2;
20     static final int PAREN_C  = 3;
21     static final int EXP      = 4;
22     static final int MULT     = 5;
23     static final int DIV      = 6;
24     static final int MAS      = 7;
25     static final int MENOS    = 8;
26
27     public Evaluador( String s )
28     {
29         pilaOper     = new PilaVec( );
30         pilaPostfija = new PilaVec( );
31         entrada = new StringTokenizer( s, "+*-/^() ", true );
32         pilaOper.apilar( new Integer( EOL ) );
33     }
34
35     public long obtenerValor( )
36       { /* Figura 11.15 */ }
37
38     private Pila pilaOper;     // Pila de operadores para la
                                  conversión
39     private Pila pilaPostfija; // Pila para la máquina postfija
40     StringTokenizer entrada;   // La cadena de caracteres
41     long valorActual;          // Operando actual
42     int ultimoToken;           // Último token leído
43
44     private int obtenerToken( )
45       { /* Figura 11.16 */ }
46     private void opBinario( int opCima )
47       { /* Figura 11.18 */ }
48     private void procesarToken( )
49       { /* Figura 11.20 */ }
50 }
```

Figura 11.13 Esqueleto de la clase `Evaluador`.

primera pila se utiliza para evaluar la expresión infija y generar la expresión postfija. Es la pila de operadores declarada en la línea 38. En vez de mostrar directamente por la salida la expresión postfija, mandamos cada símbolo a la máquina postfija, según se van generando. Necesitamos, por tanto, una segunda pila, que almacena operandos. Ésta es la pila de la máquina postfija, declarada en la línea 39. Observe que si no tuviéramos un mecanismo de programación genérica, tendríamos problemas, pues las dos pilas almacenan elementos de diferente tipo. La Figura 11.14 contiene dos métodos para acceder a las dos pilas, y realizar las conversiones de tipos necesarias. El resto de atributos son un objeto de la clase `StringTokenizer` utilizado para recorrer la entrada, y atributos para almacenar el token actual y, cuando el token es un operando, el valor del operando.

Las líneas de la 17 a la 25 generan una serie de constantes. El constructor, mostrado en las líneas de la 27 a la 33, construye las dos pilas y el objeto de la clase `StringTokenizer`. Los parámetros para realizar la división en tokens indican los símbolos considerados como delimitadores. Indican también que estos delimitadores son tokens que deben tratarse (en vez de meros delimitadores que se saltan)[1].

El único método visible públicamente es `obtenerValor`. Mostrado en la Figura 11.15, `obtenerValor` lee repetidamente un token y lo procesa hasta que detecta el final de línea. En ese momento, el elemento en la cima de la pila es la respuesta. La Figura 11.16 muestra la rutina `obtenerToken`. Comenzamos saltando todos los blancos (llamando recursivamente a `obtenerToken`). Si no hemos alcanzado el final de la línea, comprobamos si tenemos alguno de los operadores de un carácter, y si es así, devolvemos el token apropiado. En otro caso, se alcanza la línea 24, y como quiera que esperamos que lo que tengamos sea un operando, lo introducimos en `valorActual`.

```
1  /**
2   * Método interno que oculta la conversión de tipos.
3   */
4  private long cimaYDesapilarPostfija( ) throws
   DesbordamientoInferior
5  {
6      return ( (Long) ( pilaPostfija.cimaYDesapilar( ) ) ).
7                                                longValue( );
8  }
9
10 /**
11  * Otro método interno que oculta la conversión de tipos.
12  */
13 private int cimaPilaOper( ) throws DesbordamientoInferior
14 {
15     return ( (Integer) ( pilaOper.cima( ) ) ).intValue( );
16 }
```

Figura 11.14 Métodos internos para ocultar la conversión de tipos en las pilas genéricas.

[1] El Apéndice C.3.2 describe con más detalle la clase `StringTokenizer`.

```
/**
 * Rutina pública que realiza la evaluación. Examina la
 * máquina postfija para ver si aparece un único resultado,
 * y si es así, lo devuelve; en otro caso, se produce un error.
 */
public long obtenerValor( )
{
    long elResultado = 0;

    do
    {
        ultimoToken = obtenerToken( );
        procesarToken( );
    } while( ultimoToken != EOL );

    try
      { elResultado = cimaYDesapilarPostfija( ); }
    catch( DesbordamientoInferior e )
    {
        System.err.println( ";Falta un operando!" );
        return 0;
    }

    if( !pilaPostfija.esVacia( ) )
        System.err.println( "Aviso: falta un operador!" );

    return elResultado;
}
```

Figura 11.15 Rutina `obtenerValor`, para leer y procesar tokens y devolver el elemento en la cima de la pila.

```
/**
 * Encuentra el siguiente token, saltando blancos, y lo devuelve.
 * Para un token VALOR, coloca el valor en valorActual.
 * Imprime un mensaje de error si no se reconoce la entrada.
 */
private int obtenerToken( )
{
    String s = "";

    try
      { s = entrada.nextToken( ); }
    catch( java.util.NoSuchElementException e )
      { return EOL; }

    if( s.equals( " " ) ) return obtenerToken( );
    if( s.equals( "^" ) ) return EXP;
    if( s.equals( "/" ) ) return DIV;
    if( s.equals( "*" ) ) return MULT;
    if( s.equals( "(" ) ) return PAREN_A;
    if( s.equals( ")" ) ) return PAREN_C;
    if( s.equals( "+" ) ) return MAS;
    if( s.equals( "-" ) ) return MENOS;

    try
      { valorActual = Long.parseLong( s ); }
    catch( NumberFormatException e )
    {
        System.err.println( "Error" );
        return EOL;
    }
    return VALOR;
}
```

Figura 11.16 Rutina `obtenerToken`, que devuelve el siguiente token de la cadena

Las Figuras 11.17 y 11.18 muestran las rutinas necesarias para implementar la máquina postfija. `obtenerCima` devuelve y elimina el elemento de la cima de la pila postfija. La rutina `opBinario` aplica `opCima` (que se espera sea el elemento de la cima de la pila de operadores) a los dos elementos de la cima de la pila postfija y los reemplaza por el resultado obtenido. También desapila de la pila de operadores, dado que se ha completado el procesamiento de `opCima`.

```
/*
 * cimaYDesapilar de la pila de la máquina postfija; devuelve el
 * resultado. Si la pila está vacía, se genera un error.
 */
private long obtenerCima( )
{
    try
      { return cimaYDesapilarPostfija( ); }
    catch( DesbordamientoInferior e )
      { System.err.println( "Falta un operando" ); }
    return 0;
}
```

Figura 11.17 Rutina `obtenerCima`, que devuelve el elemento en la cima de la pila postfija y lo elimina.

```
/**
 * Procesa un operador tomando dos elementos de la pila
 * postfija, aplicando el operador, y apilando el resultado.
 * Imprime un error si faltan paréntesis derechos o se divide por 0.
 */
private void opBinario( int opCima )
{
    if( opCima == PAREN_A )
    {
        System.err.println( "Paréntesis desequilibrados" );
        try
          { pilaOper.desapilar( ); }
        catch( DesbordamientoInferior e ) { } // No puede ocurrir
        return;
    }
    long lder = obtenerCima( );
    long lizq = obtenerCima( );

    if( opCima == EXP )
        pilaPostfija.apilar( new Long( potencia( lizq, lder ) ) );
    else if( opCima == MAS )
        pilaPostfija.apilar( new Long( lizq + lder ) );
    else if( opCima == MENOS )
        pilaPostfija.apilar( new Long( lizq - lder ) );
    else if( opCima == MULT )
        pilaPostfija.apilar( new Long( lizq * lder ) );
    else if( opCima == DIV )
        if( lder != 0 )
            pilaPostfija.apilar( new Long( lizq / lder ) );
        else
        {
            System.err.println( "División por cero" );
            pilaPostfija.apilar( new Long( lizq ) );
        }
    try
      { pilaOper.desapilar( ); }
    catch( DesbordamientoInferior e ) { }
}
```

Figura 11.18 Rutina `opBinario` para aplicar `opCima` a la pila postfija.

La Figura 11.19 declara una *tabla de precedencias*, que almacena la precedencia de operadores y es utilizada para decidir qué operadores se han de eliminar de la pila. Se introduce en memoria cuando se carga la clase `Evaluador`. Los operadores se listan en el mismo orden que las constantes básicas de la clase.

Queremos asignar un número a cada nivel de precedencia. Cuanto mayor es el número, mayor es la precedencia. Podríamos limitarnos a asignar a los operadores aditivos la precedencia 1, a los multiplicativos la precedencia 3, a la exponenciación la precedencia 5, y a los paréntesis la precedencia 99. Sin embargo, también necesitamos tener en cuenta la asociatividad. Para lograrlo, asignamos a cada operador un número que representa su precedencia cuando es un símbolo de la entrada, y otro que representa su precedencia cuando es un operador de la pila. Un operador que asocia por la izquierda tiene como precedencia en la pila una unidad más que la original, mientras que un operador que asocie por la derecha tiene precedencia en la entrada una unidad más que la original. Por tanto, la precedencia de un operador + que esté en la pila es 2.

Una consecuencia de esta regla es que cualquiera de los dos operandos que originalmente tengan diferente precedencia, siguen estando ordenados de la misma forma. Sin embargo, si encontramos sendos operadores aditivos en la pila y como símbolo de la entrada, el operador en la cima de la pila tendrá mayor precedencia, y por tanto será desapilado. Esto es lo que queremos para los operadores que asocian por la izquierda.

De forma similar, si un ^ está en la pila de operadores y también en la entrada, el operador de la pila tendrá menor precedencia, por lo que no será desapilado, lo que es correcto para un operador que asocia por la derecha. El token `VALOR` nunca se apila, por lo que su precedencia carece de significado. El token de fin de línea

```
1  class Precedencia
2  {
3      int simboloEntrada;
4      int cimaPila;
5
6      Precedencia( int simEnt, int simCima )
7      {
8          simboloEntrada = simEnt;
9          cimaPila       = simCima;
10     }
11 }
12         // Esto es parte de la clase Evaluador
13         // tablaPrec genera el orden de evaluación de los Tokens
14     static Precedencia [ ] tablaPrec = new Precedencia[ 9 ];
15     static
16     {
17         tablaPrec[ 0 ] = new Precedencia(   0, -1 ); // EOL
18         tablaPrec[ 1 ] = new Precedencia(   0,  0 ); // VALOR
19         tablaPrec[ 2 ] = new Precedencia( 100,  0 ); // PAREN_A
20         tablaPrec[ 3 ] = new Precedencia(   0, 99 ); // PAREN_C
21         tablaPrec[ 4 ] = new Precedencia(   6,  5 ); // EXP
22         tablaPrec[ 5 ] = new Precedencia(   3,  4 ); // MULT
23         tablaPrec[ 6 ] = new Precedencia(   3,  4 ); // DIV
24         tablaPrec[ 7 ] = new Precedencia(   1,  2 ); // MAS
25         tablaPrec[ 8 ] = new Precedencia(   1,  2 ); // MENOS
26     }
```

Figura 11.19 Tabla de precedencias utilizada para evaluar una expresión infija.

tiene la menor precedencia, de modo que se colocará en la pila (lo que se hace en el constructor) para que actúe como centinela. Si se trata como un operador que asocia por la derecha, se manejará en el caso de operadores generales.

El método que falta es `procesarToken`, mostrado en la Figura 11.20. Cuando encontramos un operando, se apila en la pila postfija. Si encontramos un paréntesis derecho, desapilamos y procesamos repetidamente el operador en la cima de la pila de operadores hasta que aparezca el paréntesis izquierdo (líneas de la 19 a la 21). El paréntesis izquierdo se desapila en la línea 23. (Observe que el test de la línea 22 se utiliza para evitar desapilar el centinela cuando falta el paréntesis izquierdo.) En caso contrario, tenemos el caso general de operador, que se describe con el código de las líneas de la 29 a la 34. En la Figura 11.21 se muestra una rutina `main` sencilla: repetidamente lee una línea de la entrada, instancia un objeto de la clase `Evaluador`, y calcula su valor.

```
1  /**
2   * Después de leer un token, se utiliza el algoritmo de análisis de
3   * expresiones con precedencia de operadores para evaluarlo.
4   * Se detecta la falta de paréntesis abiertos.
5   */
6  private void procesarToken( )
7  {
8      int opCima;
9
10     try
11     {
12         switch( ultimoToken )
13         {
14           case VALOR:
15             pilaPostfija.apilar( new Long( valorActual ) );
16             return;
17
18           case PAREN_C:
19             while( ( opCima = cimaPilaOper( ) ) != PAREN_A
20                    && opCima != EOL )
21                 opBinario( opCima );
22             if( opCima == PAREN_A )
23                 pilaOper.cima( ); // Eliminar paréntesis abierto
24             else
25                 System.err.println( "Falta paréntesis abierto" );
26             break;
27
28           default: // Caso de operador general
29             while( tablaPrec[ ultimoToken ].simboloEntrada <=
30                    tablaPrec[ opCima = cimaPilaOper( ) ].
31                                                    cimaPila )
32                 opBinario( opCima );
33             if( ultimoToken != EOL )
34                 pilaOper.apilar( new Integer( ultimoToken ) );
35             break;
36         }
37     }
38     catch( DesbordamientoInferior e ) { } // No puede ocurrir
39 }
```

Figura 11.20 Rutina `procesarToken` para procesar `ultimoToken` utilizando el algoritmo de análisis de expresiones con precedencia de operadores.

```
/**
 * Rutina main sencilla, aunque un tanto chapucera
 */
public static void main( String [ ] args )
{
    String cad;
    BufferedReader in = new BufferedReader( new
                        InputStreamReader( System.in ) );

    try
    {
        System.out.println( "Introduzca expresiones," +
                            " una por línea:" );
        while( ( cad = in.readLine( ) ) != null )
        {
            System.out.println( "Leída: " + cad );
            Evaluador ev = new Evaluador( cad );
            System.out.println( ev.obtenerValor( ) );
            System.out.println( "Introduzca otra expresión:" );
        }
    }
    catch( IOException e ) { }
}
```

Figura 11.21 Una rutina `main` simple, para evaluar expresiones repetidamente.

11.2.4 Árboles sintácticos de expresiones

La Figura 11.22 muestra un ejemplo de *árbol sintáctico de una expresión*. Las hojas del árbol sintáctico de una expresión son operandos, como constantes o nombres de variables, mientras que el resto de nodos contienen operadores. Este árbol particular es binario, porque todos los operadores que aparecen en la expresión lo son. Aunque éste es el caso más simple, es posible que los nodos tengan más de dos hijos, así como un único hijo, como ocurre en el caso del operador menos unario.

El árbol sintáctico de una expresión se evalúa aplicando el operador en la raíz a los valores obtenidos evaluando de forma recursiva los subárboles izquierdo y derecho. En el ejemplo anterior, el hijo izquierdo se evalúa a `(a+b)` y el hijo derecho a `(a-b)`. Por tanto, el árbol completo representa `((a+b)*(a-b))`. Es evidente que podemos producir una expresión infija (sobrecargada de paréntesis) produ-

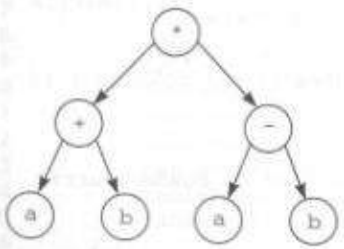


Figura 11.22 Árbol sintáctico de la expresión `(a + b) * (a - b)`.

ciendo recursivamente la expresión parentizada izquierda, incorporando entonces el operador en la raíz, y produciendo finalmente la expresión parentizada derecha. Esta estrategia general (izquierda, nodo, derecha) se denomina *recorrido en orden simétrico*. Este tipo de recorrido es fácil de recordar por el tipo de expresiones que produce.

Una segunda estrategia consiste en imprimir recursivamente el subárbol izquierdo, seguido del subárbol derecho, y terminando con el operador. Obtenemos de este modo la expresión en forma postfija. Este recorrido se denomina *recorrido en postorden del árbol*. Una tercera estrategia para recorrer el árbol tiene como resultado la expresión prefija. Todas estas estrategias se discutirán en el Capítulo 17. Los árboles sintácticos de expresiones (y sus generalizaciones) son estructuras de datos útiles en el diseño de compiladores, pues permiten manejar globalmente las expresiones, teniendo en cuenta su estructura lógica. Esto hace más sencilla la generación de código, y en algunos casos facilita significativamente los esfuerzos de optimización.

Es de interés también la construcción del árbol sintáctico de una expresión a partir de su forma infija. Como ya se ha visto, siempre se puede convertir una expresión infija en una postfija, por lo que es suficiente ver cómo se puede construir un árbol sintáctico a partir de la expresión postfija. Y esto es sencillo. Se mantiene una pila de (referencias a) árboles. Cuando nos encontramos un operando, creamos un árbol con un único nodo y lo apilamos en la pila. Cuando encontramos un operador, los dos árboles en la cima de la pila se desapilan y combinan. En el nuevo árbol, la raíz es el operador, el hijo derecho es el primer árbol desapilado y el hijo izquierdo el segundo árbol desapilado. El resultado se apila de nuevo en la pila. Éste es, esencialmente, el mismo algoritmo que utilizábamos para la evaluación postfija, sustituyendo el cálculo de operadores binarios por la creación de árboles.

Resumen

Este capítulo examina dos usos de las pilas en el área general de lenguajes de programación y diseño de compiladores. Nos enseña que las pilas son muy potentes, aunque sean una estructura bastante simple. Las pilas pueden utilizarse para comprobar si una secuencia de símbolos está correctamente equilibrada. El algoritmo resultante consume un tiempo lineal y, lo que es igualmente importante, sólo hace un recorrido secuencial de la entrada. El análisis de expresiones con precedencia de operadores es una técnica que puede utilizarse para analizar expresiones infijas. También necesita un tiempo lineal y un solo recorrido de la entrada, utilizando dos pilas. Aunque las pilas almacenan valores de diferente tipo, el mecanismo de tipos genéricos nos permite utilizar una única implementación de las pilas para ambos tipos de objetos.

Elementos del juego

análisis de operadores con precedencia Un algoritmo que convierte una expre-

análisis léxico El proceso de reconocimiento de tokens en una cadena de símbolos.

árbol sintáctico de una expresión Un árbol en el cual las hojas contienen operandos y el resto de nodos contiene operadores.

expresión infija Una expresión en la cual cada operador binario tiene sus argumentos a su izquierda y a su derecha. Cuando hay varios operadores, la precedencia y la asociatividad determinan en qué orden deben procesarse.

expresión postfija Una expresión que puede ser evaluada por la máquina postfija sin utilizar ninguna regla de precedencia.

máquina de estados Técnica común utilizada para analizar símbolos. En cada momento, la máquina está en un determinado estado. Cada carácter de entrada nos conduce a un nuevo estado. Cuando todo va bien, la máquina de estados acaba por alcanzar un estado en el cual un símbolo ha sido reconocido.

máquina postfija Máquina utilizada para evaluar una expresión postfija. El algoritmo que utiliza es el siguiente: los operandos se apilan en una pila y al encontrarnos con un operador desapilamos sus operandos y apilamos el resultado. Al final de la evaluación, la pila debería contener exactamente un elemento, que representa el resultado.

tabla de precedencias Una tabla utilizada para decidir qué operadores deben eliminarse de la pila de operadores. Los operadores que asocian por la izquierda tienen como precedencia en la pila de operadores una unidad más que su precedencia como símbolo de la entrada. Los operadores que asocian por la derecha se comportan justo al revés.

Errores comunes

1. Los errores en la entrada deben tratarse con el máximo cuidado. Es un error grave de programación ser negligente en este área.
2. En la rutina de comprobación de símbolos equilibrados, el tratamiento incorrecto de las comillas es un error habitual.
3. En el algoritmo de conversión de notación infija a postfija, la tabla de precedencias debe reflejar la precedencia y asociatividad correctas.

En Internet

Las dos aplicaciones están disponibles en el directorio **Chapter11**. Recomendamos que descargue el programa de comprobación de símbolos equilibrados; puede ayudarle a depurar programas en Java. Los nombres de los ficheros son:

JavaAnalizer.java	Traducido por `AnalizadorJava.java`, contiene el programa de comprobación de símbolos equilibrados. En el directorio **Part3** puede encontrarse una versión que incluye un generador de referencias cruzadas (*véase* Capítulo 12).
Evaluator.java	Traducido por `Evaluador.java`, contiene el programa para evaluar expresiones.

Ejercicios

Cuestiones breves

11.1. Muestre el resultado de ejecutar el programa de comprobación de símbolos desequilibrados sobre las siguiente entradas:

a) `)`
b) `( )`
c) `[ [ [`
d) `) (`
e) `[ ) ]`

11.2. Muestre las siguientes expresiones en notación postfija:

a) `1 + 2 - 3 ^ 4`
b) `1 ^ 2 - 3 * 4`
c) `1 + 2 * 3 - 4 ^ 5 + 6`
d) `( 1 + 2 ) * 3 - ( 4 ^ ( 5 - 6 ) )`

11.3. Para la expresión infija `a +b ^ c * d ^ e ^ f - g - h / ( i + j )`, haga lo siguiente:

a) Muestre cómo el algoritmo de análisis de expresiones con precedencia de operadores genera la correspondiente expresión postfija.
b) Muestre cómo la máquina postfija evalúa la expresión postfija resultante.
c) Dibuje el árbol sintáctico correspondiente.

Problemas teóricos

11.4. Para el programa de comprobación de símbolos equilibrados, explique cómo imprimir un mensaje de error que refleje la causa probable de cada error posible.

11.5. Explique, en términos generales, cómo se incorporarían operadores unarios al evaluador de expresiones. Asuma que los operadores unarios preceden a su operando, y que tienen máxima precedencia. Incluya una descripción de cómo serían reconocidos por la máquina de estados.

Problemas prácticos

11.6. Utilizar para la exponenciación el operador `^` puede confundir a los programadores Java (pues en dicho lenguaje se utiliza para denotar el operador O-exclusivo bit a bit). Rescriba la clase `Evaluador` tomando como operador de exponenciación `**`.

11.7. El evaluador infijo acepta ciertas expresiones ilegales, en las cuales los operadores están mal colocados. Se pide hacer lo siguiente:

a) ¿Cómo se evaluaría `1 2 3 + *`?
b) ¿Cómo podemos detectar estas ilegalidades?
c) Modifique la clase `Evaluador` para que las detecte.

11.8. Modifique el evaluador de expresiones de modo que admita números negativos en la entrada.

11.9. Implemente un evaluador completo de expresiones de Java. Trate todos los operadores de Java que puedan aceptar constantes y tengan sentido aritmético.

11.10. Implemente un evaluador de expresiones Java que incluya variables. Suponga que existen como mucho 27 variables, es decir, de la a a la z, y que se puede asignar un valor a una variable con el operador = de mínima precedencia (como en Java).

11.11. Escriba un programa que lea una expresión infija y genere una expresión postfija.

11.12. Escriba un programa que lea una expresión postfija y genere una expresión infija.

11.13. Diseñe un applet que implemente una calculadora que utilice el teclado para la entrada.

11.14. Escriba un applet que ilustre cómo las dos pilas van cambiando durante la ejecución del evaluador de expresiones infijas.

11.15. Escriba una aplicación Java que proporcione una GUI con el programa de comprobación de símbolos desequilibrados. Utilice una caja de diálogo de ficheros, y coloque la salida en un área de texto.

Bibliografía

El algoritmo de análisis de expresiones con precedencia de operadores utilizado para convertir una expresión infija en una postfija fue descrito por primera vez en [3]. [1] y [2] son dos buenos libros sobre la construcción de compiladores.

1. A. V. Aho, R. Sethi, y J. D. Ullman, *Compiler Design: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass. (1986).
2. C. N. Fischer y R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings, Redwood City, Calif. (1991).
3. R. W. Floyd, «Syntactic Analysis and Operator Precedence», *Journal of the ACM* **10:3** (1963), 316-333.

CAPÍTULO

12 Utilidades

En este capítulo se discuten dos usos de las estructuras de datos en el marco de las denominadas *utilidades*: la compresión de datos y la generación de referencias cruzadas. La compresión de datos es una técnica muy importante en computación. Se emplea para reducir el tamaño de los ficheros guardados en disco (aumentando de esta forma su capacidad) y para aumentar la tasa de efectividad en la transmisión de datos mediante módems (al transmitirse menos datos repetidos). En la actualidad, prácticamente todos los módems realizan algún tipo de compresión de datos. Las tablas de referencias cruzadas constituyen una técnica de búsqueda y ordenación que se usa, por ejemplo, para hacer el índice de un libro.

En este capítulo veremos:

- La exposición de un algoritmo de compresión de datos denominado *algoritmo de Huffman*, y una descripción de cómo puede implementarse dicho algoritmo (su implementación completa queda fuera de los objetivos del texto).
- Una implementación de un programa de generación de referencias cruzadas que enumera, de forma ordenada, todos los identificadores que aparecen en un programa Java, indicando los números de línea donde aparece cada uno.

12.1 Compresión de ficheros

El conjunto de caracteres ASCII está formado por unos 100 caracteres imprimibles. Para distinguir internamente estos caracteres se necesitan $\lceil \log 100 \rceil = 7$ bits. Siete bits permiten la representación de 128 caracteres, de modo que el conjunto de caracteres ASCII contiene además algunos caracteres no imprimibles. Se añade un octavo bit para poder realizar tests de paridad. Centrémonos, no obstante, en el hecho de que si el cardinal de un conjunto de caracteres es C, entonces se necesitan $\lceil \log C \rceil$ bits para una codificación de longitud fija estándar.

Supongamos que tenemos un fichero que contiene sólo caracteres *a*, *e*, *i*, *s*, *t*, además de espacios en blanco (*sp*) y caracteres de nueva línea (*nl*). Supongamos que la frecuencia de cada carácter es 10, 15, 12, 3, 4, 13 y 1, respectivamente. Como se muestra en la tabla de la Figura 12.1, este fichero ocuparía 174 bits, ya que contiene 58 caracteres y en este marco cada carácter necesitaría sólo 3 bits para ser representado

Carácter	Código	Frecuencia	Bits totales
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
Total			**174**

Figura 12.1 Un esquema de codificación estándar.

En la vida real, los archivos pueden ser bastante grandes. Muchos ficheros de gran tamaño son salida de algunos programas, y normalmente existe una gran disparidad entre el número de apariciones de los caracteres más frecuentes y los menos frecuentes. Por ejemplo, muchos ficheros de datos de gran tamaño tienen una enorme cantidad de dígitos, blancos y caracteres de nueva línea, sin embargo tienen pocas *q* o *x*.

Existen muchas situaciones en las que sería muy deseable reducir el tamaño de un fichero. Por ejemplo, como el espacio en disco es precioso en prácticamente cualquier máquina, disminuir el espacio necesario para almacenar los ficheros aumentaría la capacidad efectiva del disco. Cuando los datos se transmiten a través de las líneas telefónicas mediante un módem, la tasa de efectividad en la transmisión aumenta si puede reducirse la cantidad de datos transmitidos. Reducir la cantidad de bits necesarios para la representación de datos recibe el nombre de *compresión*. Realmente, la compresión se realiza en dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión). Una estrategia sencilla explicada en este capítulo permite ahorrar el 25 por ciento del espacio en ficheros típicos de gran tamaño, y hasta un 50 o 60 por ciento en muchos archivos de datos. Extensiones de esta técnica permiten obtener aún mejores resultados en la compresión.

La estrategia general es permitir que la longitud de la codificación cambie de carácter a carácter, haciendo que los caracteres más frecuentes tengan códigos más cortos. Nótese que cuando todos los caracteres aparecen con frecuencias similares, no podemos esperar un gran ahorro.

12.1.1 Códigos sin prefijos

El código binario de la Figura 12.1 se puede representar mediante el árbol binario de la Figura 12.2. En este árbol, los caracteres se almacenan sólo en las hojas. Los códigos de cada carácter se encuentran recorriendo el árbol desde la raíz hasta alcanzar dicho carácter, empleando un 0 para indicar la rama izquierda y un 1 para indicar la rama derecha. Por ejemplo, a *s* se llega yendo primero a la izquierda, y luego dos veces a la derecha. En consecuencia, su código es 011. Esta estructura recibe el nombre de *trie*[1] *binario* (pronunciado en inglés «trai»). Si cada carácter

[1] Existir una traducción habitual de este término hemos optado por conservar su nombre en inglés.

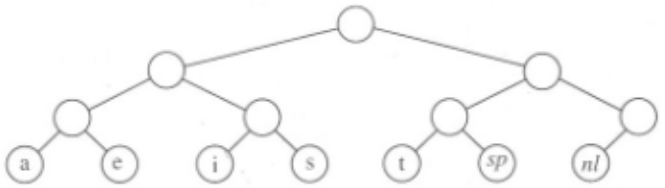


Figura 12.2 Representación mediante un árbol de la codificación original.

c_i se encuentra a profundidad d_i y aparece f_i veces en un fichero entonces el *coste* del código del mismo es igual a $\Sigma d_i f_i$.

Puede obtenerse una codificación mejor que la mostrada en la Figura 12.2 teniendo en cuenta que *nl* es hijo único. Colocándolo en un nivel superior (en el lugar de su padre) obtenemos un nuevo árbol mostrado en la Figura 12.3. Dicho árbol tiene un coste de 173 bits pero sigue estando lejos de ser óptimo.

Nótese que el árbol de la Figura 12.3 es un *árbol completo*, esto es, todos los nodos o son hojas o tienen dos hijos. Una codificación óptima cumple trivialmente esta propiedad; en caso contrario, como ya hemos visto, los nodos que tienen un único hijo pueden subirse un nivel.

El que los caracteres se sitúen sólo en las hojas es importante, pues así cualquier secuencia de bits se decodifica sin ambigüedad posible. Consideremos por ejemplo la cadena codificada 0100111100010110001000111. La Figura 12.3 muestra que 0 y 01 no son códigos de ningún carácter, pero 010 es el código de *i*. A continuación sigue 011, que representa una *s*. El código 11 siguiente es un carácter de nueva línea (*nl*). El resto de la cadena es *a*, *sp*, *t*, *i*, *e* y *nl*.

Los códigos de los caracteres pueden tener longitudes diferentes, y ningún código de carácter es un prefijo de cualquier otro código. Esta técnica de codificación se conoce como *código sin prefijos*. Recíprocamente, si permitimos que un carácter esté en un nodo que no sea una hoja, en general no será posible garantizar que la decodificación no sea ambigua.

Combinando estos hechos, concluimos que nuestro problema básico se reduce a encontrar el árbol completo de coste mínimo (aplicando la definición dada de coste), en el que todos los caracteres se encuentren en las hojas. La Figura 12.4 muestra el árbol óptimo para nuestro alfabeto de muestra. Como puede verse en la Figura 12.5, esta codificación requiere sólo 146 bits. Nótese que siempre existen varias codificaciones óptimas. En concreto, a partir de cada una podemos encontrar otras distintas, intercambiando los hijos de algunos nodos (los que gustemos) del árbol.

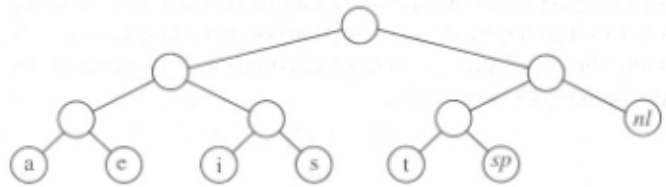


Figura 12.3 Un árbol de codificación un poco mejor.

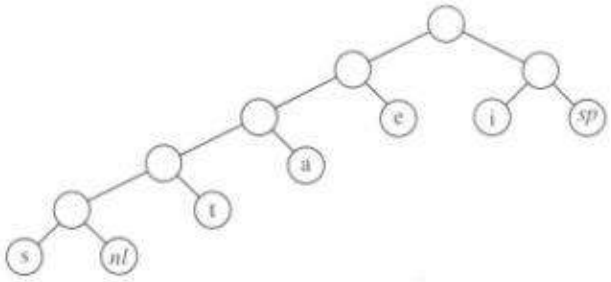


Figura 12.4 Árbol que genera el código óptimo sin prefijos.

12.1.2 Algoritmo de Huffman

¿Cómo se construye el árbol óptimo de codificación? El algoritmo que implementa el sistema de codificación se debe a Huffman, quien lo dio en 1952, y se conoce comúnmente como *algoritmo de Huffman*.

A lo largo de esta sección consideramos que el número de caracteres a codificar es C. El algoritmo de Huffman es el siguiente: se manejará un bosque de árboles. El *peso* de un árbol es igual a la suma de las frecuencias de sus hojas. Se seleccionan los dos árboles, T_1 y T_2, de menor peso y se forma un nuevo árbol cuyos subárboles son T_1 y T_2. Este proceso se repite $C - 1$ veces. Al empezar el algoritmo se tienen C árboles con un único nodo (un árbol por cada carácter). Al final del proceso sólo queda un árbol, que es justamente el árbol óptimo de Huffman. El Ejercicio 12.4 pide al lector que demuestre la corrección de este algoritmo.

Carácter	Código	Frecuencia	Bits totales
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
Total			**146**

Figura 12.5 Código sin prefijos óptimo.

Un ejemplo ilustrará el funcionamiento del algoritmo. La Figura 12.6 muestra el bosque de partida; el peso de cada árbol se indica con un número de tamaño pequeño al lado de la raíz. Se mezclan los dos árboles de menor peso, por medio de la nueva raíz $T1$, para así generar el bosque de la Figura 12.7. Hemos hecho de forma arbitraria que s sea el hijo izquierdo del nuevo árbol, pero la elección contraria sería perfectamente válida. El peso total del nuevo árbol es la suma de los pesos de los árboles que lo generan, esto es 4.

Figura 12.6 Configuración inicial del algoritmo de Huffman.

Figura 12.7 El algoritmo de Huffman tras la primera mezcla.

En este momento tenemos seis árboles, y de nuevo se seleccionan los dos de menor peso: $T1$ y t. Se mezclan en un nuevo árbol, con raíz $T2$ y peso 8. Esto se muestra en la Figura 12.8. El tercer paso mezcla $T2$ y a, creando $T3$, con peso $10 + 8 = 18$. La Figura 12.9 muestra el resultado de esta operación.

Figura 12.8 El algoritmo de Huffman tras la segunda mezcla.

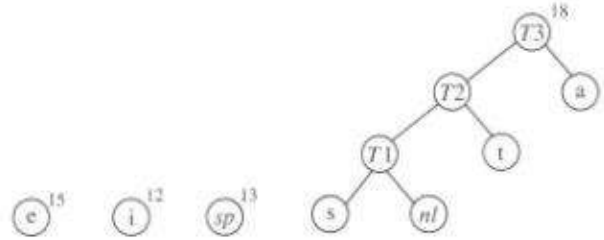


Figura 12.9 El algoritmo de Huffman tras la tercera mezcla.

Después de haber completado la tercera mezcla, los dos árboles de menor peso son los árboles con un único nodo correspondientes a los caracteres i y blanco. La Figura 12.10 muestra cómo dichos árboles se combinan en uno nuevo, con raíz $T4$. El quinto paso consiste en mezclar los árboles con raíz e y $T3$, ya que éstos son ahora los de menor peso. El resultado de este paso se muestra en la Figura 12.11.

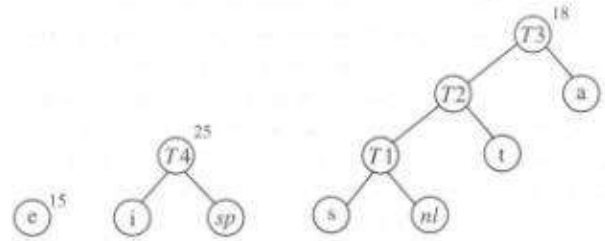


Figura 12.10 El algoritmo de Huffman tras la cuarta mezcla.

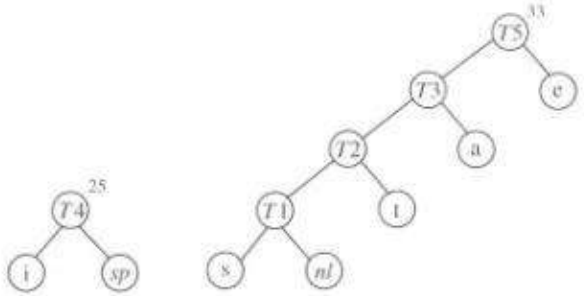


Figura 12.11 El algoritmo de Huffman tras la quinta mezcla.

Por último, se obtiene un árbol óptimo, el mostrado anteriormente en la Figura 12.4, combinando los dos árboles que nos quedan. La Figura 12.12 muestra el árbol óptimo, con raíz *T*6.

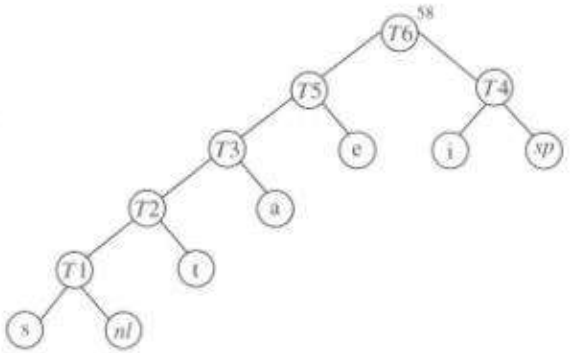


Figura 12.12 El algoritmo de Huffman tras la mezcla final.

12.1.3 La fase de codificación

Una vez que el árbol ha sido construido, debemos decidir el código de cada carácter. Esto puede hacerse si cada nodo del árbol tiene información acerca de su padre, e indica si es su hijo derecho o izquierdo. Si tenemos *C* caracteres, entonces el árbol tiene $2C - 1$ nodos. En esta fase podemos emplear un vector con una entrada por cada nodo. Dichas entradas están compuestas por el peso, el padre y la información de si el nodo es hijo derecho o izquierdo.

La tabla de codificación para nuestro árbol de Huffman se muestra en la Figura 12.13. Cada carácter no está almacenado en el vector sino que se emplea como índice de la tabla (así `'a'` indexa la posición 97, que es su código ASCII, `'b'` indexa la posición 98, y así sucesivamente). Por ejemplo, el padre de *nl* es *T*1 y *nl* es su hijo derecho. Podemos escribir el código de *nl* generando recursivamente el código de *T*1 y luego el tipo de hijo que es. Nótese que *T*6 tiene a 0 como padre. Esto nos indica cuándo finalizar la recursión. Como el rango de los códigos ASCII se extiende desde 0 hasta 127, el vector tendrá desde la casilla 0 hasta la 254, para poder representar el hecho de que, además de las hojas, se necesitan, a lo sumo, $C - 1$ símbolos internos.

	Carácter	Peso	Padre	Tipo de hijo
0	a	10	9	1
1	e	15	11	1
2	i	12	10	0
3	s	3	7	0
4	t	4	8	1
5	*sp*	13	10	1
6	*nl*	1	7	1
7	T1	4	8	0
8	T2	8	9	0
9	T3	18	11	0
10	T4	25	12	1
11	T5	33	12	0
12	T6	58	0	

Figura 12.13 Tabla de codificación (los números de la izquierda son los índices del vector).

12.1.4 La fase de decodificación

Más tarde o más temprano, cuando queramos utilizar su información, necesitaremos descomprimir un fichero comprimido. Para hacerlo, debemos incluir cierta información extra en el fichero comprimido. Obviamente, deseamos incluir tan poca información como sea posible. Supongamos que estamos empleando el conjunto de caracteres ASCII.

Una posibilidad es almacenar la frecuencia de cada carácter. Esto requiere 128 enteros. Seguramente será suficiente hacerlo con enteros 24-bit, es decir, enteros 3-byte (probablemente, los enteros 16-bit no basten ya que es posible, si el fichero es grande, que algún carácter aparezca más de 65.535 veces). En total se necesitan 384 bytes para el almacenamiento del fichero.

Una alternativa es almacenar la información del tipo de hijo y el padre de cada nodo. Como cada padre es un nodo completamente nuevo, su valor es un número entre 128 y 254. Como consecuencia, necesitamos 7 bits para almacenarlo. De hecho, podemos usar un carácter de 8 bits: sumando 128 al valor representado por los 7 bits menores obtendremos la posición del padre (la raíz puede usar la posición 255 como su padre). Esto deja libre el octavo bit (es decir, el más significativo) para representar el tipo de su hijo. Como consecuencia, se necesitan 255 bytes para almacenar la información auxiliar de la codificación.

12.1.5 Consideraciones prácticas

Antes de iniciar la compresión de datos, se necesita contabilizar la frecuencia de cada carácter. Si el fichero es lo suficientemente pequeño para poder ser almacenado en la memoria principal, podemos leer el fichero volcándolo en un vector, calcular las frecuencias de los caracteres y aplicar el algoritmo. Pero, ¿qué sucede si es demasiado grande?

Es obvio que no deseamos leer el fichero dos veces; la E/S a través de disco es extremadamente lenta. Una posibilidad es dividir el fichero en bloques que puedan ser almacenados en la memoria principal, y comprimir cada uno de ellos por separado. Muchos computadores modernas pueden almacenar muchos megabytes, así que no parece excesivo tener que almacenar repetidas tablas de códigos con 255 bytes.

Una alternativa es asumir que el fichero de entrada es «típico». Por ejemplo, es bien conocida la distribución de los caracteres en los programas implementados en Java. Una ventaja de este método es que no necesitamos almacenar en cada fichero la tabla de códigos. Esto es particularmente atractivo cuando los ficheros a almacenar no son muy grandes. Otras muchas posibilidades han sido exploradas, al igual que otros esquemas de compresión más complejos que funcionan mejor que la codificación de Huffman. *Véase* las referencias bibliográficas para una información más completa.

12.2 Generador de referencias cruzadas

En esta sección diseñaremos un programa que examina un código Java y muestra todos los identificadores del mismo, junto a los números de línea en los que aparecen. Además, los identificadores se muestran ordenados alfabéticamente. Esto se conoce como *generación de referencias cruzadas*. Una aplicación en el terreno de la compilación es enumerar, para cada función, los nombres de las funciones que invoca directamente.

Sin embargo, éste es un problema general que aparece en muchos otros contextos. Por ejemplo, generaliza la creación del índice de un libro. Otro uso, la revisión ortográfica, se describe en el Ejercicio 12.16. Un revisor ortográfico detecta las palabras mal escritas en un documento, que se muestran junto a los números de líneas donde aparecen. Esto evita imprimir varias veces la misma palabra incorrecta y nos indica dónde se han producido los errores.

12.2.1 Ideas básicas

Cada identificador y el número de línea donde aparece se almacenarán en un objeto del tipo `IdNodo`. Mantendremos una colección ordenada de `IdNodos`. Cuando el fichero fuente se haya leído, nos desplazaremos a través de la colección, mostrando los identificadores y sus números de líneas correspondientes.

Una posibilidad es emplear una lista enlazada de objetos `IdNodo`. Cuando se lee un identificador, debemos comprobar si ya está en la lista. Si es así, añadimos la línea actual al objeto `IdNodo` referenciado mediante una llamada a `encontrar`. Si no se encuentra en la lista, se crea un nuevo objeto `IdNodo` con el identificador y el número de línea, y se inserta en la lista enlazada. Después de que la entrada completa ha sido leída, podemos recorrer la lista enlazada y mostrar su contenido. Sin embargo, una lista enlazada ordenada tiene un comportamiento asintótico pobre, por lo que en su lugar emplearemos un árbol binario de búsqueda y haremos un recorrido en orden simétrico para obtener la salida final.

Cada objeto `IdNodo` almacena un identificador y los números de línea en las que aparece. El identificador se representa mediante un `String`. Los números de línea pueden representarse con una cola de enteros. Esto resulta adecuado ya que cuando mostramos los números de línea, aparecerán en el mismo orden en el que fueron recorridas, esto es, en orden creciente. Así cada `IdNodo` almacena una referencia a una cola a la que se le asigna memoria cuando se declara el `IdNodo`. La implementación se describe en la siguiente subsección.

12.2.2 Implementación en Java

La Figura 12.14 muestra la clase `IdNodo` (que sólo contiene componentes de acceso amistoso). Un `IdNodo` consiste en una cadena de caracteres y una referencia a una cola de enteros. El campo `String` llamado `palabra` se declara en la línea 9; `lineas`, que es la referencia a la cola de enteros, se declara en la lí-

```
1  /**
2   * Objeto básico, que se almacenará en un árbol de búsqueda,
3   * para el generador de referencias cruzadas.
4   * Implementa el interfaz Comparable, implementando
5   * compara y menorQue.
6   */
7  class IdNodo implements Comparable
8  {
9      String palabra;  // Un identificador
10     Cola lineas;     // Líneas donde aparece
11
12         // Constructor
13     IdNodo( String laPalabra, int lineaActual )
14     {
15         palabra = new String( laPalabra );
16         lineas = new Cola( );
17         lineas.insertar( new Integer( lineaActual ) );
18     }
19
20         // Métodos de ordenación
21     public boolean menorQue( Comparable lder )
22     {
23         return palabra.compareTo( ( (IdNodo) lder ).palabra ) < 0;
24     }
25
26     public int compara( Comparable lder )
27     {
28         return palabra.compareTo( ( (IdNodo) lder ).palabra );
29     }
30 }
```

Figura 12.14 Clase `IdNodo`.

nea 10. El constructor implementado en las líneas 13 a 18, inicializa un `IdNodo` con un `String` y un número de línea. El componente `String` se inicializa en la forma habitual. `lineas` es inicializado creando una cola en la que se inserta `lineaActual`. El resto de la clase `IdNodo` implementa el interfaz `Comparable`.

```
1  // Interfaz AnalizadorJava: genera referencias cruzadas
2  //
3  // CONSTRUCCIÓN: con un objeto BufferedReader
4  // ******************OPERACIONES PÚBLICAS*******************
5  // void generaReferenciaCruzada( ) --> El nombre lo dice todo .
6  // ************************ERRORES*************************
7  // Comprobación de errores sobre comentarios y comillas
8
9  import java.io.*;
10 import EstructurasDatos.*;
11 import Excepciones.*;
12 import Soporte.*;
13
14 /**
15  * Clase que genera referencias cruzadas para programas Java.
16  */
17 class AnalizadorJava
18 {
19     public AnalizadorJava( PushbackReader cadenaEntrada )
20     {
21         errores = 0;
22         car = '\0';
23         lineaActual = 1;
24         entrada = cadenaEntrada;
25         actualIdNodo = new IdNodo( "", 1 );
26     }
27
28     public void generaReferenciaCruzada( )
29       { /* Figura 12.19 */ }
30
31     private IdNodo actualIdNodo;
32
33     private static int esIdChar( ch )
34       { /* Figura 12.16 */ }
35     private String tomaString( )
36       { /* Figura 12.17 */ }
37     private int tomaSiguienteId( )
38       { /* Figura 12.18 */ }
39
40        // Los siguientes son los mismos que los de la Figura 11.2
41     private PushbackReader entrada; // La cadena entrada
42     private char car;               // Carácter actual
43     private int lineaActual;        // Linea actual
44     private int errores;            // Número de errores encontrados
45     private boolean siguienteCar( )
46     private void devolverCar( )
47     private void saltarComentario( int comienzo )
48     private void saltarCadena( char tipoCadena )
49     private void procesarBarra( )
50 }
```

Figura 12.15 Esquema de la clase para el generador de referencias cruzadas.

La clase de la Figura 12.15 es similar a la mostrada en la Figura 11.2, que era parte de un programa de equilibrado de símbolos (el código en Internet es una versión en la que se mezclan estas dos clases en una sola). Muchos de los métodos ya han sido explicados, por lo que no repetimos sus implementaciones. No se emplea herencia. La posibilidad de crear una clase abstracta y dos clases derivadas se deja como ejercicio al lector (Ejercicio 12.13).

Las nuevas rutinas de compilación tratan el reconocimiento de identificadores. En la línea 31, se declara un objeto `IdNodo`. Éste almacenará el identificador que está siendo procesado. Esto es debido a que como manejamos un árbol de objetos `IdNodo`, necesitamos un `IdNodo` como parámetro de las rutinas `buscar` e `insertar`.

Un identificador de Java es una secuencia de caracteres alfanuméricos y guiones, con la restricción de que el primer carácter no puede ser un dígito. La rutina de la Figura 12.16 comprueba si el carácter dado puede formar parte de un identificador. El método `tomaString` de la Figura 12.17 asume que el primer carácter de un identificador ya ha sido leído, y almacenado en el atributo `car`. Esta rutina lee repetidamente caracteres hasta que se encuentra uno que no puede formar parte de los identificadores. En ese momento, volvemos al carácter anterior y devolvemos un `String`[2].

El método `tomaSiguienteId`, mostrado en la Figura 12.18, es similar a la rutina de la Figura 11.7. La diferencia es que aquí, en la línea 15, si se detecta el primer carácter de un identificador, rellenamos `actualIdNodo.palabra` con dicho identificador.

Una vez escritas todas las rutinas auxiliares, podemos implementar el método principal, `generaReferenciaCruzada`, que se muestra en la Figura 12.19. En las líneas 6 y 7 se define un árbol binario de búsqueda de objetos `IdNodo`. En las líneas desde la 10 hasta la 28 se procesa el fichero fuente. En las líneas 14 y 15, se ejecuta `buscar` para cada identificador, con el fin de saber si ya ha sido encontrado antes. Si `buscar` tiene éxito, en las líneas 16 y 17 insertamos la línea actual en la cola correspondiente. En caso contrario, se trata de un nuevo identificador, que añadimos en el árbol de búsqueda, en las líneas 21 a 26.

Recuérdese que el método `insertar` necesita un objeto `IdNodo`. Por tanto, llamando al constructor de la Figura 12.14 creamos un nuevo nodo, sin nombre. Colocaremos en el árbol una referencia a dicho `IdNodo`. Esto significa que en el `IdNodo` insertado en el árbol, su campo `lineas` referencia a la cola.

```
1  /**
2   * Devuelve true si car puede formar parte de un identificador Java
3   */
4  private boolean esIdCar( char car )
5  {
6      return car == '_' || Character.isUpperCase( car )
7                        || Character.isLowerCase( car )
8                        || Character.isDigit( car );
9  }
```

Figura 12.16 Rutina para comprobar si un carácter puede formar parte de un identificador.

[2] Java 1.1 tiene predefinidos métodos para comprobar si un carácter puede formar parte de un identificador; véase el Apéndice C.1.1.

```
1  /**
2   * Devuelve un identificador de la entrada
3   * El primer carácter ya ha sido leído en car
4   */
5  private String tomaString( )
6  {
7      String tmpString = "";
8
9      for( tmpString += car; siguienteCar( ); tmpString += car )
10         if( !esIdCar( car ) )
11         {
12             devolverCar( );
13             break;
14         }
15
16     return tmpString;
17 }
```

Figura 12.17 Rutina que devuelve un `String` leído de la entrada.

```
1  /**
2   * Devuelve el siguiente identificador, saltándose los comentarios,
3   * las constantes String y las constantes de tipo carácter.
4   * Coloca el identificador en actualIdNodo.palabra y devuelve false
5   * cuando se alcanza el final de la entrada.
6   */
7  private boolean tomaSiguienteId( )
8  {
9      while( siguienteCar( ) )
10     {
11         if( car == '/' )
12             procesarBarra( );
13         else if( car == '\'' || car == '"' )
14             saltarCadena( car );
15         else if( !Character.isDigit( car ) &&
16                  esIdCar( car ) )
17         {
18             actualIdNodo.palabra = tomaString( );
19             return true;
20         }
21     }
22     return false; // Final de fichero
23 }
```

Figura 12.18 Rutina para asignar a `actualIdNodo.palabra` el siguiente identificador.

Una vez que hemos construido el árbol de búsqueda, sólo tenemos que recorrerlo en orden simétrico. Para hacerlo, usamos la clase `OrdenSim` descrita en la Sección 17.4.2. El iterador visita los nodos del árbol binario en el orden adecuado. Se emplea de forma análoga a `ListaIter`. El iterador se declara en la línea 33, mientras que en la línea 34 se emplean los métodos estándar de iteración de la clase `OrdenSim`. En la línea 36, `esteNodo` es una referencia al objeto `IdNodo` almacenado en el nodo actual del árbol de búsqueda. En las líneas 39 y 40, se imprime la palabra y el primer número de línea, a la vez que éste último se saca de la cola (tenemos garantizado que la cola no está vacía). Mientras que la cola no esté vacía,

```
1  /**
2   * Muestra las referencias cruzadas.
3   */
4  public void generaReferenciaCruzada( )
5  {
6      ArbolBinarioBusqueda losIdentificadores =
7                                      new ArbolBinarioBusqueda( );
8
9          // Inserta los identificadores en el árbol de búsqueda
10     while( tomaSiguienteId( ) )
11     {
12         try
13         {
14             IdNodo esteNodo = (IdNodo)
15                     losIdentificadores.buscar( actualIdNodo );
16             esteNodo.lineas.insertar(
17                     new Integer( lineaActual ) );
18         }
19         catch( ElementoNoEncontrado e )
20         {
21             try
22             {
23                 losIdentificadores.insertar( new IdNodo
24                     ( actualIdNodo.palabra, lineaActual ) );
25             }
26             catch( ElementoDuplicado ex ) { } // No puede ocurrir
27         }
28     }
29         // Iteración a lo largo del árbol y salida
30         // de los identificadores y sus números de línea.
31     try
32     {
33         OrdenSimBus itr = new OrdenSimBus( losIdentificadores );
34         for( itr.primero( ); itr.esValido( ); itr.avanzar( ) )
35         {
36            IdNodo esteNodo = (IdNodo) itr.recuperar( );
37
38            // Muestra el identificador y la primera línea donde aparece
39            System.out.print( esteNodo.palabra + ": " +
40                   (Integer) esteNodo.lineas.quitarPrimero( ) );
41
42                // Imprime el resto de líneas donde aparece
43            while( !esteNodo.lineas.esVacia( ) )
44                System.out.print( ", " + (Integer)
45                        esteNodo.lineas.quitarPrimero( ) );
46            System.out.println( );
47         }
48     }
49     catch( ElementoNoEncontrado e ) { } // Árbol vacío
50     catch( Desbordamiento e ) { }       // No puede suceder
51 }
```

Figura 12.19 Algoritmo principal para generar referencias cruzadas[3].

imprimimos reiteradamente sus elementos gracias al bucle de las líneas 43 a 45. La instrucción de la línea 46 imprime una línea en blanco. No implementamos un programa `main`, ya que es prácticamente idéntico al de la Figura 11.10.

[3] *N. del T.*: La clase `OrdenSimBus` que aparece en la línea 33, es similar a la del Capítulo 17 `OrdenSim`. Sólo en que una árboles binarios de búsqueda, en lugar de árboles binarios no ordenados.

Resumen

En este capítulo hemos presentado las implementaciones de dos importantes aplicaciones. La compresión de ficheros es una destacada técnica que permite aumentar tanto la capacidad efectiva del disco como la velocidad efectiva del módem. Éste es un campo de investigación muy activo. La técnica sencilla presentada aquí, conocida como algoritmo de Huffman, consigue, habitualmente, una compresión del 25 por ciento de los ficheros de texto. Otros algoritmos y extensiones del algoritmo de Huffman obtienen todavía mejores resultados. El problema de las referencias cruzadas es un problema general, que tiene numerosas aplicaciones.

Elementos del juego

algoritmo de Huffman Algoritmo que construye un código sin prefijos óptimo. Se basa en la mezcla repetida de dos árboles de peso mínimo.

árbol completo Árbol cuyos nodos o bien son hojas o bien tienen dos hijos.

código sin prefijos Codificación en la que ningún código de carácter es un prefijo de otro código de carácter. En un trie, esto está garantizado cuando los caracteres se encuentran sólo en las hojas. Un código sin prefijos puede decodificarse sin ambigüedad ninguna.

compresión Es el hecho de reducir la cantidad de bits necesarios para la representación de datos. Se compone de dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión).

generador de referencias cruzadas Programa que enumera los identificadores de un programa y los números de línea en las que aparecen. Es una aplicación muy común, similar a la creación de un índice.

trie binario Estructura de datos en la que una rama izquierda representa un 0 y una rama derecha representa un 1. El camino a cada nodo indica su codificación.

Errores comunes

1. Cuando se implementa la compresión de datos no debe leerse el fichero de entrada más de una vez.
2. Es un error habitual emplear demasiada memoria para almacenar la tabla de la compresión. Esto limita el porcentaje de compresión que puede conseguirse en el proceso.

En Internet

El generador de referencias cruzadas está disponible en el directorio **Chapter12**. El nombre del fichero es:

JavaAnalyzer.java Es la versión inglesa de `AnalizadorJava`. El programa que incluye la rutina para el equilibrado de símbolos del Capítulo 11 se encuentra en el directorio **Part3**.

Ejercicios

Cuestiones breves

12.1. Muestre el árbol de Huffman que resulta de la siguiente distribución de símbolos de puntuación y dígitos: dos puntos (100), espacio (605), nueva línea (100), coma (705), 0 (431), 1 (242), 2 (176), 3 (59), 4 (185), 5 (250), 6 (174), 7 (199), 8 (205) y 9 (217).

12.2. Muchos sistemas tienen ya implementado un programa de compresión. Comprima distintos tipos de ficheros para determinar la tasa media de compresión de su sistema. ¿Cuál debe ser el tamaño de los ficheros para que la compresión resulte rentable?

12.3. ¿Qué sucede si se emplea un fichero comprimido con el algoritmo de Huffman para transmitir datos a través de la línea telefónica y se pierde un bit accidentalmente? ¿Qué se podría hacer al respecto?

Problemas teóricos

12.4. Pruebe la corrección del algoritmo de Huffman, siguiendo los siguientes pasos:

a) Muestre que no existe ningún nodo con un único hijo.
b) Muestre que los dos caracteres con menor frecuencia deben ser los dos nodos más profundos del árbol.
c) Muestre que los caracteres de dos nodos cualesquiera situados a la misma profundidad pueden intercambiarse sin afectar a la optimalidad.
d) Emplee inducción: según vamos mezclando árboles, considere como nuevo conjunto de caracteres uno abstracto cuyos elementos están asociados a las raíces de los árboles generados.

12.5. ¿Bajo qué condiciones el árbol de Huffman de caracteres ASCII genera un código de dos bits para un determinado carácter?, ¿y un código 20-bit?

12.6. Muestre que si los símbolos nos vienen dados ordenados por frecuencia, el algoritmo de Huffman puede implementarse en tiempo lineal.

Problemas prácticos

12.7. En el generador de referencias cruzadas use una lista ordenada en lugar de un árbol binario de búsqueda. ¿Cómo afecta este cambio a la eficiencia?

12.8. La desventaja de emplear una cola para almacenar los números de línea del generador de referencias cruzadas, es que la cola se vacía durante el proceso. Esto podría representar un problema en algunos casos. En lugar de guardar los números en una cola, use una lista enlazada. Implemente las siguientes estrategias y compare su eficiencia:

a) Reemplace la operación de `insertar` por una inserción al final de la lista. Para hacerlo debe recorrer dicha lista.
b) Guarde la lista enlazada y el objeto `ListaIter` que apunta al último elemento de la lista. La inserción al final de la lista enlazada debe ser más eficiente que en el apartado (*a*).

12.9. Combine los ejercicios 12.7 y 12.8, de modo que el generador de referencias cruzadas emplee una lista enlazada ordenada para almacenar la información de las palabras, la cual consiste en un `String`, una lista de números de línea y un `ListaIter` que apunta al final de la lista.

12.10. Modifique `IdNodo`, en el generador de referencias cruzadas, añadiendo un método `toString` que muestre el nombre y los números de línea de `IdNodo`. Simplifique el método `generaReferenciaCruzada` en la forma adecuada. Este empleo de `toString` es engañoso, ya que normalmente este método no altera el estado del objeto sobre el que actúa. Sin embargo, aquí vacía la cola de números de línea del objeto. ¿Qué puede hacerse para solucionar esta incongruencia?

12.11. Si una palabra aparece dos veces en la misma línea, el generador de referencias guardará dicho número de línea dos veces. Modifique el algoritmo, de modo que los duplicados sólo se enumeren una vez.

12.12. Modifique el generador de referencias cruzadas de modo que si una palabra aparece en líneas consecutivas, se especifique un rango. Por ejemplo,

```
if: 2, 4, 6-9, 11
```

Prácticas de programación

12.13. Descompoga el `AnalizadorJava` en tres clases: una clase base abstracta que recoja la funcionalidad común, y dos clases derivadas `TestBalance` y `GeneraReferenciaCruzada`.

12.14. Añada una componente GUI al generador de referencias cruzadas que incluya una opción para consultas avanzadas, como mostrar nombres específicos almacenados en el árbol binario de búsqueda (en lugar de mostrar todo su contenido).

12.15. Genere el índice de un libro. El fichero de entrada consiste en un conjunto de entradas del índice. Cada línea está formada por la cadena `IX:`, seguida de una entrada del índice escrita entre llaves y de un número de página también entre llaves. Cada símbolo `!` en una entrada de índice representa un nivel inferior. Los símbolos `|(` representan el inicio de un rango, y `|)` representa el final del rango. En algunas ocasiones, este rango puede ser la misma página. En ese caso, sólo debe indicar un único número de página. En otro caso, no debe colapsar o expandir los rangos indicados. Como ejemplo, la Figura 12.20 muestra un posible fichero de entrada y la Figura 12.21 muestra la salida correspondiente.

```
IX: {Series|(}                          {2}
IX: {Series!geométricas|(}              {4}
IX: {Constante de Euler}                {4}
IX: {Series!geométricas|)}              {4}
IX: {Series!aritméticas|(}              {4}
IX: {Series!aritméticas|)}              {5}
IX: {Series!armónicas|(}                {5}
IX: {Constante de Euler}                {5}
IX: {Series!armónicas|)}                {5}
IX: {Series|)}                          {5}
```

Figura 12.20 Entrada sencilla del Ejercicio 12.15.

```
Constante de Euler: 4, 5
Series: 2-5
  aritméticas: 4-5
  geométricas: 4
  armónicas: 5
```

Figura 12.21 Salida sencilla del Ejercicio 12.15.

12.16. Implemente un revisor ortográfico empleando una tabla hash. Asuma que el diccionario está divido en dos fuentes: un gran diccionario ya creado y un segundo fichero. Debe mostrar todas las palabras mal escritas y los números de línea en las que aparecen (observe que llevar la cuenta de estas palabras y los números de línea equivale a la generación de referencias cruzadas). Además, para cada palabra incorrecta debe enumerar todas las palabras en el diccionario que se obtienen mediante la aplicación de alguna de las siguientes reglas:

a) Añadir un carácter.
b) Eliminar un carácter.
c) Cambiar caracteres adyacentes.

12.17. Implemente un applet interactivo que muestre como se construye el árbol de Huffman.

12.18. Implemente un programa completo para la compresión de ficheros. No se olvide de incluir un algoritmo para la decompresión. Para ello, implemente una clase `BitStream`. Observe que si el número total de bits en el código de salida no es un múltiplo exacto del número de bits en un byte, pueden aparecer bits perdidos en el resultado comprimido. Debe definirse un mecanismo para ignorar dichos bits basura.

Bibliografía

El artículo original sobre el algoritmo de Huffman es [3]. Algunas variaciones se discuten en [2] y [4]. Otro esquema de compresión muy conocido es la *codificación Ziv-Lempel*, descrito en [7] y [6]. Trabaja generando series de códigos de longitud fija. Normalmente, deberíamos generar 4.096 códigos 12-bits para representar las subcadenas más frecuentes del fichero. [1] y [5] son buenos resúmenes de los esquemas de compresión más comunes.

1. T. Bell, I. H. Witten, y J. G. Cleary, «Modelling for Text Compression», *ACM Computing Surveys* **21** (1989) 557-591.
2. R. G. Gallager, «Variations on a Theme by Huffman», *IEEE Transactions on Information Theory* **IT-24** (1978), 668-674.
3. D. A. Huffman, «A Model for the Construction of Minimum Redundancy Codes», *Proceedings of the IRE* **40** (1952), 1098-1101.
4. D. E. Knuth, «Dynamic Huffman Coding», *Journal of Algorithms* **6** (1985), 163-180.
5. D. A. Lelewer y D. S. Hirschberg, «Data Compression», *ACM Computing Surveys* **19** (1987), 261-296.
6. T. A. Welch, «A Technique for High-Performance Data Compression», *Computer* **17** (1984), 8-19.
7. J. Ziv y A. Lempel, «Compression of Individual Sequences via Variable-Rate Coding», *IEEE Transactions on Information Theory* **IT-24** (1978), 530-536.

CAPÍTULO

13 Simulación

Un uso importante del computador es el de la *simulación*. En una simulación, el computador emula el funcionamiento de un sistema real y recoge estadísticas. Como ejemplo, puede que queramos simular el funcionamiento de un banco con k ventanillas para determinar cuál es el valor mínimo de k con el que se consigue un tiempo de servicio razonable. Usar un computador proporcionaría grandes ventajas. En primer lugar, la información se recogería sin molestar a los clientes. En segundo lugar, una simulación por computador puede ser mucho más rápida que la implementación real debido a la velocidad del mismo. En tercer lugar, se podría replicar fácilmente la simulación. En muchos casos, la elección de una estructura de datos apropiada nos puede ayudar a mejorar la eficiencia de la simulación.

En este capítulo veremos:

- Cómo simular un juego modelado sobre el *problema Josephus*.
- Cómo simular el comportamiento de un banco de módems.

13.1 El problema Josephus

El *problema Josephus* es el siguiente juego. Se sientan N personas, numeradas de 1 a N, formando un círculo. Se pasa una patata caliente empezando en la persona 1. Después de pasar M veces la patata, se elimina a la persona que tiene la patata, el círculo se estrecha y el juego continúa, tomando la patata la persona sentada después de la que ha sido eliminada. La persona que se mantiene hasta el final gana. Es bastante habitual considerar que M es una constante del juego, aunque se puede usar un generador de números aleatorios para cambiar M tras cada eliminación.

El problema Josephus surgió en el primer siglo A.C. en una cueva de una montaña en Israel, donde los celotes judíos estaban siendo asediados por soldados romanos. El historiador Josephus se encontraba entre ellos. Para consternación de Josephus, los celotes votaron hacer un pacto de suicidio en lugar de rendirse a los romanos. Él sugirió el juego aquí mencionado. La patata caliente era la sentencia de muerte para la persona sentada junto a la que tenía la patata. Josephus amañó el juego para quedar el último y convenció a la penúltima víctima de que debían ren-

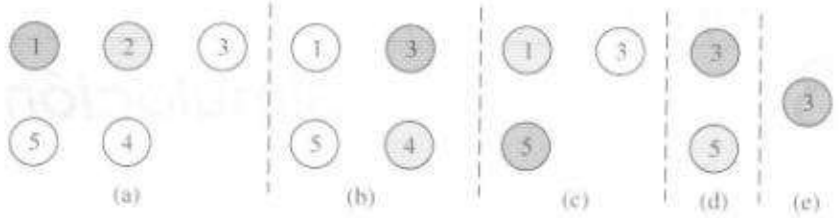


Figura 13.1 El problema Josephus. En cada paso el círculo más oscuro representa a la persona que tiene la patata inicialmente, y el ligeramente sombreado representa al jugador que recibe la patata caliente (y que por tanto es eliminado). Los pases se realizan en el sentido de las agujas del reloj.

dirse. Gracias a ello hemos podido saber de este juego; en efecto, Josephus hizo trampas[1].

Si $M = 0$, entonces se van eliminando los jugadores en orden y el último jugador siempre gana. Para otros valores de M, las cosas no son tan obvias. La Figura 13.1 muestra que si $N = 5$ y $M = 1$, entonces los jugadores son eliminados en el siguiente orden: 2, 4, 1, 5. En este caso gana el jugador 3. Los pasos son los siguientes:

1. Al principio la patata está en manos del jugador número 1. Después de pasarla una vez, la tiene el jugador número 2.
2. El jugador 2 es eliminado. El jugador 3 toma la patata y se la pasa al jugador 4.
3. El jugador 4 es eliminado. El jugador 5 toma la patata y se la pasa al jugador 1.
4. El jugador 1 es eliminado. El jugador 3 toma la patata y se la pasa al jugador 5.
5. El jugador 5 es eliminado, y por tanto el jugador 3 gana.

En primer lugar escribiremos un programa que simule, paso a paso, el juego para cualesquiera valores de N y M. El tiempo de ejecución de la simulación es $O(MN)$, lo cual es aceptable si el número de veces que se pasa la patata es pequeño. Cada paso lleva un tiempo $O(M)$ porque en él se llevan a cabo M pases. Mostraremos entonces cómo implementar cada paso en un tiempo $O(\log N)$, independientemente del número de pases que se realizan. El tiempo de la simulación es entonces $O(N \log N)$.

13.1.1 La solución simple

La fase de pasar la patata en el problema Josephus sugiere que representemos los jugadores mediante una lista enlazada. Creamos una lista enlazada en la que los elementos 1, 2, ..., N son insertados en orden. Inicializamos entonces un iterador con el primer elemento. Cada pase de la patata corresponde a una operación `avanzar` sobre el iterador. Cuando llegamos al último jugador de los que queden

[1] Teague el relato de esta historia. La versión que nosotros resolvemos difiere de la explicación histórica. le pide que resuelva la versión histórica.

en la lista, implementamos el pase reiniciando el iterador con el primer elemento. Esta acción imita un círculo. Cuando hemos terminado de pasar, eliminamos el elemento sobre el que ha aterrizado el iterador.

Por razones explicadas en el Capítulo 16, es más fácil eliminar el elemento posterior a la posición actual usando el método `eliminarSig`. Esto se refleja en el código de la Figura 13.2, donde inicializamos `actual` con el último jugador, en lugar del primero. En las líneas 7 y 8 se declaran la lista enlazada anónima y su iterador. Usando un bucle construimos la lista inicial en las líneas 12 y 13. La semántica de `insertar`, tal y como se especifica en la clase de la Figura 16.7, nos dice que las inserciones se realizan tras la posición del iterador y que el resultado de una inserción coloca el iterador en la posición recién insertada. De esta forma obtenemos la lista deseada.

```
1  /**
2   * Devuelve el ganador del problema Josephus.
3   * Implementación mediante una lista enlazada.
4   */
5  public static int josephus( int personas, int pases )
6  {
7      ListaEnlazadaIter p = new ListaEnlazadaIter (
8                                          new ListaEnlazada( ) );
9          // Construye la lista
10     try
11     {
12         for( int i = 1; i <= personas; i++ )
13            p.insertar( new Integer( i ) );
14     }
15     catch( ElementoNoEncontrado e ) { } // No puede ocurrir
16
17         // Juega al juego:
18         // Observe: p está siempre en el jugador anterior
19     while( personas-- != 1 )
20     {
21         for( int i = 0; i < pases; i++ )
22         {
23            p.avanzar( );          // Avanzar
24            if( !p.estaDentro( ) ) // Si nos pasamos del último
25               p.primero( );       // jugador volvemos al primero
26         }
27
28         if( !p.eliminarSig( ) )   // Elimina el siguiente jugador
29         {
30            // eliminarSig falla si p es el último elemento, luego
31            p.cero( );        // para el último elemento, asigna a p
32            p.eliminarSig( );// el elemento 0 para eliminar el 1
33         }
34     }
35         // Obtiene el primer y único jugador y devuelve el # de
           jugadores
36     p.primero( );
37     return ( (Integer)( p.recuperar( ) ) ).intValue( );
38 }
```

Figura 13.2 Implementación del problema Josephus usando listas enlazadas.

En la Figura 13.2, el código de las líneas 21 a 33 lleva a cabo un paso del algoritmo pasando la patata (líneas 21 a 26) y eliminando después un jugador (líneas 28 a 33). Esto se repite hasta que el test de la línea 19 nos dice que solamente queda un jugador. En ese momento, obtenemos el jugador restante llamando al método `primero` en la línea 36. (Recuerde que durante la mayor parte del algoritmo, el iterador se encuentra en el jugador anterior al actual). En la línea 37 accedemos al jugador y devolvemos su número.

Es fácil ver que el tiempo de ejecución de esta rutina es $O(MN)$ porque éste es exactamente el número de pases que se producen durante el algoritmo, lo cual es aceptable para M pequeño. Sin embargo, observe que en el caso en que $M = 0$, el tiempo de ejecución no es $O(0)$, sino $O(N)$, ya que cuando se interpreta una expresión O no basta con multiplicar.

13.1.2 Un algoritmo más eficiente

Se puede obtener un algoritmo más eficiente si usamos una estructura de datos que soporte el acceso al k-ésimo elemento más pequeño (en tiempo logarítmico). Esto nos permite implementar cada ronda de pases con una única operación. La Figura 13.1 muestra por qué. Supongamos que quedan N jugadores y que estamos actualmente en el jugador P empezando por el principio. Inicialmente N es el número total de jugadores y P es 1. Después de M pases, estaremos en el jugador $((M + P) \bmod N)$, excepto si tenemos que devolver el jugador 0, en cuyo caso vamos al jugador N. El cálculo es algo rebuscado, pero el concepto no lo es.

Aplicando este cálculo a la Figura 13.1, vemos que como M es 1, N es inicialmente 5, y P es inicialmente 1, el nuevo valor de P es 2. Tras la eliminación, N se reduce a 4, pero aún estamos en la posición 2 (como la parte (b) de la figura sugiere). El siguiente valor de P es 3 (como se muestra en la parte (b)), luego se elimina el tercer elemento de la lista y N se reduce a 3. El siguiente valor de P es 4 mod 3, es decir 1, con lo que volvemos al primer jugador de la lista (como se muestra en la parte (c)). Se elimina este jugador y N pasa a valer 2. Puesto que 2 mod 2 es 0, hacemos que P valga N, y por tanto el último jugador de la lista es el eliminado, lo cual concuerda con la parte (d). Tras la eliminación, N es 1 y hemos terminado.

Todo lo que necesitamos es una estructura que soporte de forma eficiente la operación `buscarKesimo` y un método para insertar secuencialmente los jugadores en la estructura de datos. Hay varias alternativas parecidas. Todas ellas usan el hecho de que un árbol binario de búsqueda soporta la operación `buscarKesimo` en tiempo medio logarítmico o si usamos un árbol binario de búsqueda sofisticado en un tiempo logarítmico en el caso peor. En consecuencia, si somos cuidadosos podemos esperar un algoritmo $O(N \log N)$.

El método más simple consiste en insertar los elementos secuencialmente en un árbol binario de búsqueda eficiente en el peor de los casos, como un árbol rojinegro, un AA-árbol o un árbol de ensanchamiento (todos estos árboles se estudian en capítulos posteriores). Entonces cuando sea oportuno podemos llamar a `buscarKesimo` y a `eliminar`. Resulta que los árboles de ensanchamiento son una elección excelente para esta aplicación, porque las operaciones `buscarKesimo` e `insertar` son inusualmente eficientes y `eliminar` no es demasiado difícil de codificar. No obstante aquí usaremos una estructura alternativa, porque la implementación de esta estructura de datos, que se proporciona en capítulos posteriores, deja como ejercicio la implementación de `buscarKesimo`.

Usamos la clase `ABBConRango` que soporta la operación `buscarKesimo` y que se implementa completamente en la Sección 18.2. Se basa en el árbol binario de búsqueda ordinario y por tanto no tiene eficiencia logarítmica en el caso peor sino solamente en media. En consecuencia, no podemos limitarnos a insertar los elementos secuencialmente; esto provocaría que el árbol de búsqueda exhibiera su peor comportamiento.

```
1  /**
2   * Construye recursivamente un árbol perfectamente equilibrado
3   * en tiempo O( N log N ) mediante repetidas inserciones.
4   * En la llamada inicial a debería estar vacío.
5   */
6  public static void construirArbol( ABBConRango a,
7                                     int inf, int sup )
8  {
9      int centro = ( inf + sup ) / 2;
10
11     if( inf <= sup )
12     {
13         try
14           { a.insertar( new MiEntero( centro ) ); }
15         catch( ElementoDuplicado e ) { } // No puede suceder
16
17         construirArbol( a, inf, centro - 1 );
18         construirArbol( a, centro + 1, sup );
19     }
20 }
21
22 /**
23  * Devuelve el ganador del problema Josephus.
24  * Implementación usando un árbol de búsqueda.
25  */
26 public static int josephus( int personas, int pases )
27 {
28     ABBConRango a = new ABBConRango( );
29     try
30     {
31         construirArbol( a, 1, personas );
32
33         int rango = 1;
34         while( personas > 1 )
35         {
36             rango = ( rango + pases ) % personas;
37             if( rango == 0 )
38                 rango = personas;
39
40             a.eliminar( a.buscarKesimo( rango ) );
41             personas--;
42         }
43
44         return ( (MiEntero) ( a.buscarKesimo( 1 ) ) ).intValue( );
45     }
46     catch( Exception e )
47       { return -1; } // No puede suceder
48 }
```

Figura 13.3 Solución $O(N \log N)$ para el problema Josephus.

Hay varias opciones, una de las cuales consiste en insertar una permutación aleatoria de 1, ..., N en el árbol de búsqueda. Otra, consiste en construir un árbol binario de búsqueda perfectamente equilibrado usando un método de la clase. Puesto que un método de la clase tendría acceso al manejo interno del árbol de búsqueda, se podría hacer en tiempo lineal. Se deja escribir esta rutina como el Ejercicio 18.19, una vez se hayan estudiado los árboles.

El método que utilizamos consiste en escribir una rutina que inserte elementos en orden equilibrado. Insertando el elemento central en la raíz y construyendo recursivamente los dos subárboles de la misma manera, obtenemos un árbol equilibrado. El coste de nuestra rutina es un aceptable $O(N \log N)$. Aunque no es tan eficiente como lo sería la rutina lineal de la clase, no afecta de forma adversa al tiempo de ejecución asintótico del algoritmo completo. Se garantiza que las operaciones `eliminar` son logarítmicas. Esta rutina se llama `construirArbol`, y su código aparece en la Figura 13.3, junto con el método `josephus`.

13.2 Simulación dirigida por eventos

Pasamos ahora al problema de la simulación de un banco descrito en la introducción. Tenemos un sistema en el que los clientes llegan y esperan en una cola hasta que una de las ventanillas está libre. La llegada de los clientes está gobernada por una función de distribución de probabilidades, al igual que el tiempo de servicio al cliente (cantidad de tiempo durante el que es atendido una vez que una ventanilla está libre). Estamos interesados en estadísticas como cuál es el tiempo medio que tiene que esperar un cliente en la cola y qué porcentaje del tiempo dedica cada empleado a atender peticiones. (Si hay demasiadas ventanillas, algunos no harán nada durante largos períodos.)

Para ciertas distribuciones de probabilidad y valores de k, se pueden calcular estos valores estadísticos de forma exacta. Pero a medida que k se hace más grande el análisis se hace considerablemente más difícil, por lo que resulta atractivo usar un computador para simular el funcionamiento del banco. De esta forma, los directores del banco pueden decidir cuántas ventanillas se necesitan para asegurar un servicio razonablemente desahogado. La mayor parte de las simulaciones implican conocimientos de probabilidad, estadística y teoría de colas.

13.2.1 Ideas básicas

Una simulación de eventos discretos consiste en el procesamiento de eventos. Aquí los dos eventos a considerar son (a) la llegada de un cliente y (b) la salida de un cliente, dejando libre una ventanilla.

Podemos usar una función de probabilidad para generar una secuencia de entrada formada por pares ordenados de tiempos de llegada y servicio de cada cliente, ordenados por tiempo de llegada[2]. No necesitamos usar la hora exacta del día, sino que usamos una unidad a la que nos referimos como *tic*.

[2] La probabilidad genera intervalos de tiempo entre llegadas, garantizando que éstas se producen en orden crono-

Inicializaremos el tiempo de la simulación en cero tics. Entonces avanzamos el reloj un tic cada vez, comprobando si se produce un evento. Si se produce, procesamos el evento (o eventos) y recalculamos los valores estadísticos. Cuando ya no quedan clientes en la secuencia de entrada y todas las ventanillas están libres, la simulación ha terminado. Ésta es una *simulación dirigida por tiempos discretos*.

El problema de esta estrategia de simulación es que su tiempo de ejecución no depende del número de clientes o de eventos (en este caso hay dos eventos por cliente), sino del número de tics, que en realidad no es parte de la entrada. Para ver por qué esto es importante, supongamos que cambiamos las unidades del reloj a microtics y que multiplicamos todos los tiempos de la entrada por 1 millón. El resultado sería que la simulación tardaría 1 millón de veces más en ejecutarse.

La clave para evitar este problema es ir avanzando el reloj en cada paso hasta el momento en el que se produce el siguiente evento. Luego tenemos una *simulación dirigida por eventos*. Esto es conceptualmente fácil de hacer. En cualquier punto, el siguiente evento que puede producirse es o bien la llegada del siguiente cliente, o bien la salida de uno de los clientes de una ventanilla. Puesto que los tiempos en los que se producen eventos están disponibles, solamente necesitamos encontrar el evento más cercano en el tiempo y procesar dicho evento (actualizando el reloj con el tiempo en el que se produce el evento).

Si el evento es una salida, el procesamiento incluye la actualización de los valores estadísticos con los datos del cliente que se marcha y mirar la cola para ver si hay otro cliente esperando. Si es así, añadimos ese cliente, recalculamos los valores estadísticos que proceda, calculamos la hora en que se marchará el cliente y añadimos su salida al conjunto de eventos a la espera de producirse.

Si el evento es una llegada, comprobamos si hay una ventanilla libre. Si no hay ninguna colocamos al cliente en la cola. En caso contrario, le asignamos al cliente una ventanilla, calculamos el tiempo de salida del cliente, y añadimos dicha salida al conjunto de eventos a la espera de producirse.

La cola de clientes se puede implementar como una cola. Puesto que necesitamos encontrar el evento más cercano en el tiempo, el conjunto de eventos debe organizarse como una cola de prioridad. El siguiente evento es o bien una llegada o una salida (lo que ocurra antes); ambos tipos de eventos están igualmente disponibles. Una simulación dirigida por eventos es apropiada si se espera que el número de tics entre eventos sea grande.

13.2.2 Ejemplo: simulación de un banco de módems

El punto crucial en la gestión de una simulación es la organización de los eventos en una cola de prioridad. Para centrarnos en este punto, consideraremos una simulación de un sistema muy simple: un *banco de módems* del centro de computación de una universidad.

Un banco de módems consiste en una gran colección de módems. Por ejemplo, la Universidad Internacional de Florida (FIU) tiene 96 módems disponibles para los estudiantes. A un módem se accede marcando un número de teléfono. Si alguno de los 96 módems está disponible, entonces el usuario se conecta a uno de

ellos, pero si se están usando todos, entonces el teléfono dará la señal de ocupado. Nuestra simulación modeliza el servicio proporcionado por el banco de módems. Las variables son las siguientes:

- El número de módems en el banco.
- La distribución de probabilidad que gobierna los intentos de marcado.
- La distribución de probabilidad que gobierna el tiempo de conexión.
- El tiempo (real) que cubrirá la simulación.

La simulación del banco de módems es una versión simplificada de la simulación de los cajeros del banco porque no hay cola de espera. Cada marcado es una llegada, y el tiempo de conexión es el tiempo de servicio. Eliminando la cola de espera, eliminamos la necesidad de mantener una estructura cola, por lo que solamente necesitamos una estructura de datos, la cola de prioridad. En el Ejercicio 13.17 se le pide incorporar una cola, de forma que si todos los módems están ocupados se irán colocando en la cola hasta L llamadas. Para simplificar las cosas, no calculamos valores estadísticos, sino que en su lugar listamos cada evento a medida que se producen. También suponemos que los intentos de conexión suceden a intervalos constantes, mientras que en una buena simulación sería necesario modelizar el tiempo entre llegadas mediante un proceso aleatorio. En la Figura 13.4 se muestra la salida de una simulación.

```
1  El usuario 0 marca en minuto 0 y se conecta durante 1 minutos
2  El usuario 0 cuelga en minuto 1
3  El usuario 1 marca en minuto 1 y se conecta durante 5 minutos
4  El usuario 2 marca en minuto 2 y se conecta durante 4 minutos
5  El usuario 3 marca en minuto 3 y se conecta durante 11 minutos
6  El usuario 4 marca en minuto 4 pero recibe señal de ocupado
7  El usuario 5 marca en minuto 5 pero recibe señal de ocupado
8  El usuario 6 marca en minuto 6 pero recibe señal de ocupado
9  El usuario 1 cuelga en minuto 6
10 El usuario 2 cuelga en minuto 6
11 El usuario 7 marca en minuto 7 y se conecta durante 8 minutos
12 El usuario 8 marca en minuto 8 y se conecta durante 6 minutos
13 El usuario 9 marca en minuto 9 pero recibe señal de ocupado
14 El usuario 10 marca en minuto 10 pero recibe señal de ocupado
15 El usuario 11 marca en minuto 11 pero recibe señal de ocupado
16 El usuario 12 marca en minuto 12 pero recibe señal de ocupado
17 El usuario 13 marca en minuto 13 pero recibe señal de ocupado
18 El usuario 3 cuelga en minuto 14
19 El usuario 14 marca en minuto 14 y se conecta durante 6 minutos
20 El usuario 8 cuelga en minuto 14
21 El usuario 15 marca en minuto 15 y se conecta durante 3 minutos
22 El usuario 7 cuelga en minuto 15
23 El usuario 16 marca en minuto 16 y se conecta durante 5 minutos
24 El usuario 17 marca en minuto 17 pero recibe señal de ocupado
25 El usuario 15 cuelga en minuto 18
26 El usuario 18 marca en minuto 18 y se conecta durante 7 minutos
27 El usuario 19 marca en minuto 19 pero recibe señal de ocupado
```

Figura 13.4 Ejemplo de salida para la simulación del banco de módems: tenemos 3 módems; se intenta un marcado cada minuto; el tiempo medio de conexión es de 5 minutos; la simulación se ejecuta durante 19 minutos.

La clase de la simulación requiere otra clase que representa los eventos, la cual recibe el nombre de clase `Evento` y se muestra en la Figura 13.5. Los atributos son el número de cliente, el instante en que sucederá el evento y una indicación del tipo del evento (`MARCADO` o `COLGADO`). Si esta simulación fuera más compleja, con varios tipos de eventos, con seguridad convertiríamos a `Evento` en una clase base y derivaríamos subclases de ella. No lo hacemos aquí para no complicar las cosas ni oscurecer el funcionamiento básico del algoritmo de simulación. La clase `Evento` contiene constructores e implementa la interfaz de la clase `Comparable`. Observe que tiene un constructor de cero parámetros de forma que se puede crear un centinela $-\infty$ para una cola de prioridad que almacena una colección de objetos de la clase `Evento`. La clase `Evento` usa atributos amistosos para las clases dentro del mismo paquete.

La clase para la simulación de los módems, `SimModem`, se muestra en la Figura 13.6. Consta de un conjunto de atributos, un constructor y dos métodos. Los atributos incluyen un número aleatorio `r` mostrado en la línea 25. En la línea 26, `conjEventos` se declara como una `ColaPrioridad` de objetos `Evento`. Hay tres atributos más. Uno de ellos es `modemsLibres`, que inicialmente es el número de módems en la simulación, pero que cambia a medida que los usuarios se conectan y cuelgan. Los otros dos son `tpoMedioLlamada` y `frecLlamadas`, que son parámetros de la simulación. Recuerde que se llevará a cabo un intento de marcado cada `frecLlamadas` tics. El constructor en las líneas 18 y 19, que se implementa en la Figura 13.7, inicializa estos atributos y coloca la primera llegada en la cola de prioridad `conjEventos`.

```
1  import Soporte.*; import Soporte.Comparable;
2  /**
3   * La clase de los eventos.
4   * Implementa el interfaz Comparable
5   * para ordenar los eventos por el instante en que se producen.
6   */
7  class Evento implements Comparable
8  {
9      static final int MARCADO = 1;
10     static final int COLGADO = 2;
11
12     public Evento( )
13       { this( 0, 0, MARCADO ); }
14
15     public Evento( int nombre, long tpo, int tipo )
16       { quien = nombre; tiempo = tpo; que = tipo; }
17
18     public boolean menorQue( Comparable lder )
19       { return tiempo < ( (Evento) lder ).tiempo; }
20
21     public int compara( Comparable lder )
22     {
23         return menorQue( lder ) ? -1 :
24                lder.menorQue( tnis ) ? 1 : 0;
25     }
26
27     int quien;     // Número del usuario
28     long tiempo;   // Cuándo sucederá el evento
29     int que;       // MARCADO o COLGADO
30 }
```

Figura 13.5 Clase `Evento` usada en la simulación del banco de módems.

```
1  import Excepciones.*;
2  import java.util.*;
3  import EstructurasDatos.*;
4  // Clase SimModem: lleva a cabo una simulación
5  //
6  // CONSTRUCTOR: con tres parámetros: número de módems,
7  //      tiempo medio de conexión y tiempo entre
8  //      llegadas
9  //
10 // ********************MÉTODOS PÚBLICOS*********************
11 // void ejecSim( )        --> Ejecuta una simulación
12
13 /**
14  * Clase SimModem.
15  */
16 public class SimModem
17 {
18     public SimModem( int modems, double tpoMedio,
19                      long intervalo )
20       { /* Figura 13.7 */ }
21
22     public void ejecSim( long tiempoParada )
23       { /* Figura 13.9 */ }
24
25     private Random r;                    // Número aleatorio
26     private ColaPrioridad conjEventos;// Eventos pendientes
27
28         // Parámetros básicos de la simulación
29     private int modemsLibres;            // Número de módems sin usar
30     private double tpoMedioLlamada;      // Longitud de una llamada
31     private long frecLlamadas;           // Intervalo entre llamadas
32
33     private void sigLlamada( long delta )
34       { /* Figura 13.8 */ }
35 }
```

Figura 13.6 Esqueleto de la clase SimModem.

```
1  /**
2   * Constructor.
3   * @param modems número de módems.
4   * @param tpoMedio duración media de una llamada.
5   * @param intervalo tiempo medio entre llamadas.
6   */
7  public SimModem( int modems, double tpoMedio,
8                   long intervalo )
9  {
10     conjEventos     = new MonticuloBinario( new Evento( ) );
11     modemsLibres    = modems;
12     tpoMedioLlamada = tpoMedio;
13     frecLlamadas    = intervalo;
14     r               = new Random( );
15     sigLlamada( frecLlamadas ); // Planifica la primera llamada
16 }
```

Figura 13.7 Constructor SimModem.

La clase para la simulación consta solamente de dos métodos. El primero, sigLlamada, mostrado en la Figura 13.8, añade una petición de MARCADO al conjunto de eventos. Mantiene dos variables de clase (globales): el número del siguiente usuario que intentará marcar y el tiempo en el que se producirá el evento. De nuevo, suponemos, para simplificar, que las llamadas se producen a intervalos regulares. En la práctica, usaríamos un generador de números aleatorios para modelizar la secuencia de marcados.

```
1      // Usado solamente por sigLlamada
2  private int numUsuario = 0;
3  private long tpoSigLlamada = 0;
4
5  /**
6   * Coloca un nuevo evento MARCADO en la cola de eventos.
7   * Luego avanza el reloj hasta que sucede el siguiente
8   * evento MARCADO. En la práctica, usaríamos un número
9   * aleatorio para determinar el tiempo.
10  */
11 private void sigLlamada( long delta )
12 {
13     conjEventos.insertar( new Evento( numUsuario++, tpoSigLlamada,
14                                       Evento.MARCADO ) );
15     tpoSigLlamada += delta;
16 }
```

Figura 13.8 Rutina sigLlamada, que coloca un nuevo evento MARCADO en la cola de eventos y avanza el reloj hasta el momento en el que se produce el siguiente evento MARCADO.

El otro método es ejecSim, mostrado en la Figura 13.9, al que se llama para ejecutar la simulación completa. Es la rutina que hace casi todo el trabajo. Se llama con un único parámetro que indica la duración de la simulación. Mientras el conjunto de eventos no sea vacío, procesamos eventos. Observe que ello no debería suceder nunca, porque cuando llegamos a la línea 13, hay exactamente una petición de marcado en la cola de prioridad, más una petición de colgado para cada módem conectado en ese momento. Siempre que eliminamos un evento en la línea 13 y se confirma que es un marcado, generamos un nuevo evento de marcado en la línea 42. Además, en la línea 37 se genera un evento de colgado cuando el marcado tiene éxito. Por tanto, la única forma de terminar la rutina es haciendo que en algún momento sigLlamada no genere un evento, o ejecutando a ejecutar la instrucción break de la línea 17 (lo cual es más probable).

A continuación se muestra un resumen de cómo se procesan los distintos eventos. Si el evento es de colgado, incrementamos modemsLibres en la línea 21 e imprimimos un mensaje en las líneas 22 y 23. Si el evento es de marcado, generamos una parte de una frase como salida que registra el intento, y entonces si hay módems disponibles, conectamos al usuario. Para ello, decrementamos modemsLibres en la línea 31 y generamos un tiempo de conexión (usando una distribución de Poisson en lugar de una uniforme) en la línea 32. Entonces imprimimos el resto de la línea de salida en las líneas 33 y 34 y añadimos un evento de colgado al conjunto de eventos (líneas 35 a 37). Si no hay módems libres, generamos un mensaje de señal de ocupado. En ambos casos, se concluye generando un nuevo evento de marcado. La Figura 13.10 muestra el estado de la cola de prioridad después de cada eliminarMin para las primeras fases de la salida del ejemplo que se muestra en la Figura 13.4. El momento en que se produce cada evento se muestra en negrita, y el número de módems libres (si hay alguno) se muestra a la derecha de la cola de prioridad. La secuencia de pasos es la siguiente:

```
/**
 * Ejecuta la simulación hasta llegar a tpoParada.
 * Produce una salida como la de la Figura 13.4.
 */
public void ejecSim( long tpoParada )
{
    Evento e = null;
    long duracion;

    while( !conjEventos.esVacia( ) )
    {
        try
          { e = (Evento) conjEventos.eliminarMin( ); }
        catch( DesbordamientoInferior ex ) { } // No puede suceder

        if( e.tiempo > tpoParada )
            break;

        if( e.que == Evento.COLGADO ) // COLGADO
        {
            modemsLibres++;
            System.out.println( "El usuario " + e.quien +
                        " cuelga en minuto "+ .tiempo );
        }
        else                          // MARCADO
        {
            System.out.print( "El usuario " + e.quien +
                    " marca en minuto " + e.tiempo + " " );
            if( modemsLibres > 0 )
            {
                modemsLibres--;
                duracion = r.poisson( tpoMedioLlamada );
                System.out.println( "y se conecta durante "
                                + duracion + " minutos" );
                e.tiempo += duracion;
                e.que = Evento.COLGADO;
                conjEventos.insertar( e );
            }
            else
                System.out.println( "pero recibe señal de ocupado" );

            sigLlamada( frecLlamadas );
        }
    }
}
```

Figura 13.9 Rutina básica de simulación.

1. Se inserta el primer MARCADO.
2. Después de eliminar MARCADO, se conecta al usuario, lo cual genera un COLGADO y una nueva petición de MARCADO.
3. Se procesa una petición de COLGADO.
4. Se procesa una petición de MARCADO que resulta en una conexión, por lo que se añaden tanto un evento MARCADO como un evento COLGADO (tres veces).
5. Una petición de MARCADO falla; en su lugar se genera una nueva petición de MARCADO (tres veces).

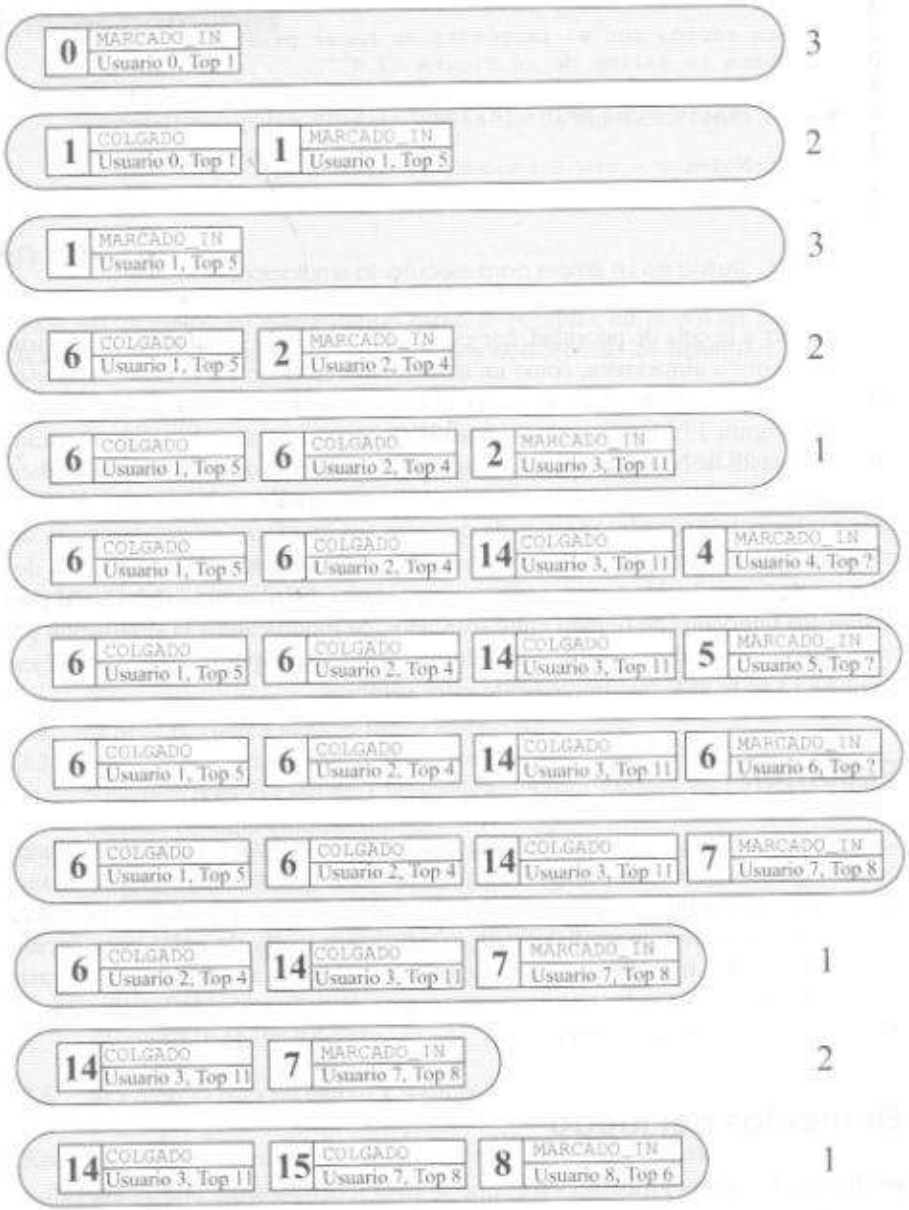


Figura 13.10 Estado de la cola de prioridad para la simulación del banco de módems tras cada paso de la simulación.

6. Se procesa una petición de COLGADO (dos veces).
7. Una petición de MARCADO tiene éxito, y se añaden un evento MARCADO y otro COLGADO.

De nuevo, si Evento fuera una clase base abstracta, sería de esperar que se definiera un procedimiento procesarEvento a través de la jerarquía Evento, con lo que no necesitaríamos largas cadenas de instrucciones if/else. Sin embargo,

```
/**
 * main rápido con el propósito de hacer pruebas.
 * Genera la salida de la Figura 13.4
 */
public static void main( String [ ] args )
{
    SimModem s = new SimModem( 3, 5.0, 1 );
    s.ejecSim( 20 );
}
```

Figura 13.11 Rutina `main` simple para ejecutar la simulación.

para acceder a la cola de prioridad, que está en la clase de la simulación, necesitaríamos que `Evento` almacenara, como un atributo, una referencia a la clase `SimModem` (en la que está contenido). Esto se haría en el momento de aplicación del constructor.

En la Figura 13.11 se muestra una rutina `main` que genera la salida mostrada en la Figura 13.4. No obstante, no es apropiado usar una distribución de Poisson para modelizar el tiempo de conexión. Una elección mejor sería usar una distribución exponencial (pero las razones de que esto sea en efecto así quedan fuera del alcance de este libro). Tampoco es adecuado asumir un tiempo fijo entre cada dos intentos de marcado. De nuevo, sería mejor usar una distribución exponencial para definir los intervalos de tiempo entre marcados. Si modificamos la simulación para incorporar estas distribuciones, el reloj debería ser de tipo `double`. En el Ejercicio 13.13 se le pide que implemente estos cambios.

Resumen

La simulación es un área importante de la computación. Hay muchas más dificultades implicadas en el tema que las que podemos discutir. La simulación será acertada en la medida en que el modelo probabilístico lo sea, luego se requiere una sólida formación en probabilidad, estadística y teoría de colas para decidir acertadamente los tipos de distribuciones de probabilidad a utilizar en cada caso. La simulación es, además, una importante área de aplicación de las técnicas de modelización orientadas a objetos.

Elementos del juego

problema Josephus Problema en el que se pasa repetidamente una patata caliente. Cuando se termina de pasar, el jugador que se queda con la patata queda eliminado. El juego continúa entonces, y el jugador que quede el último gana.

simulación Uso importante de los computadores en el que el computador emula el funcionamiento de un sistema real y genera medidas estadísticas de su comportamiento.

simulación dirigida por eventos Simulación en la que en cada paso el reloj se avanza hasta el siguiente evento.

simulación dirigida por tiempos discretos Simulación en la que cada unidad de tiempo se procesa consecutivamente. Resulta inapropiada si el intervalo entre eventos sucesivos es grande.

tic Unidad de tiempo en una simulación.

Errores comunes

1. El error más habitual en simulación consiste en basarnos en un modelo probabilístico pobre. Una simulación resulta tan buena como acertada sea la del correspondiente modelo.

En Internet

Todos los ejemplos de este capítulo están disponibles en la red en el directorio **Chapter13**. A continuación se enumeran los nombres de los ficheros.

Josephus.java	Contiene las dos implementaciones de `josephus` y un `main` para probarlos.
ModemSim.java	Contiene el código de la simulación del banco de módems.

Ejercicios

Cuestiones breves

13.1. Si $M = 0$, ¿quién gana el juego Josephus?

13.2. Muestre el funcionamiento del algoritmo Josephus con un árbol binario de búsqueda para el caso de 7 personas y 3 pases. Incluya un dibujo del árbol después de cada eliminación.

13.3. ¿Hay algún valor de M para el que el jugador 1 gana un juego Josephus de 30 personas?

13.4. Muestre el estado de la cola de prioridad después de cada una de las diez primeras líneas de la simulación mostrada en la Figura 13.4.

Problemas teóricos

13.5. Sea $N = 2^k$ para un entero k. Demuestre que si M es 1, entonces el jugador 1 siempre gana el juego Josephus.

13.6. Sea $J(N)$ el ganador de un juego Josephus con N jugadores y $M = 1$, demuestre lo siguiente:

a) Si N es par, $J(N) = 2J(N/2) - 1$.
b) Si N es impar y $J(\lceil N/2 \rceil) \neq 1$, entonces $J(N) = 2J(\lceil N/2 \rceil) - 3$.
c) Si N es impar y $J(\lceil N/2 \rceil) = 1$, entonces $J(N) = N$.

13.7. Use los resultados del Ejercicio 13.6 para escribir un algoritmo que devuelva el ganador de un juego Josephus con N jugadores y $M = 1$. ¿Cuál es el tiempo de ejecución de dicho algoritmo?

13.8. Proporcione una fórmula general para el ganador de un juego Josephus con N jugadores cuando $M = 2$.

13.9. Usando el algoritmo para $N = 20$, determine el orden de inserción en el `ABBConRango`.

13.10. Demuestre que después de aplicar `construirArbol` (de la Figura 13.3), cada hoja está en uno de los dos últimos niveles del árbol.

Problemas prácticos

13.11. Escriba un programa que resuelva la versión histórica del problema Josephus. Proporcione los algoritmos que hacen uso de la lista enlazada y del árbol de búsqueda.

13.12. Implemente el algoritmo Josephus usando una cola. Cada pase de la patata viene dado por un `quitarPrimero` seguido de un `insertar`.

13.13. Modifique la simulación de forma que el reloj se represente mediante una variable de tipo `double`, y tanto el tiempo entre dos intentos de marcado como el tiempo de conexión se modelicen con una distribución exponencial.

13.14. Modifique la simulación del banco de módems de forma que `Evento` sea una clase base abstracta, y `EventoMarcado` y `EventoColgado` sean clases derivadas de ella. La clase `Evento` debería almacenar una referencia `SimModem` como atributo adicional, que se inicializaría en el constructor. También debería proporcionarse un método abstracto implementado en las clases derivadas, llamado `procesarEvento`, al que se llamaría desde `ejecSim` para procesar el evento.

Prácticas de programación

13.15. Implemente el algoritmo Josephus usando árboles de ensanchamiento (*véase* el Capítulo 21) e inserción secuencial. (La clase de árboles de ensanchamiento está disponible en la red, pero necesitará añadir un método `buscarKesimo`). Compare su eficiencia con el que aparece en el libro y con un algoritmo que use un algoritmo lineal de construcción de un árbol equilibrado.

13.16. Reescriba el algoritmo Josephus de la Figura 13.3 para usar un *montículo de medianas* (Ejercicio 6.20). Use una implementación simple del montículo de medianas en la que los elementos se mantengan ordenados. Compare los tiempos de ejecución de este algoritmo y del que usa un árbol binario de búsqueda.

13.17. Suponga que la FIU ha instalado un sistema el cual, cuando los módems están ocupados, pone las llamadas de teléfono en una cola. Reescriba la rutina de simulación para permitir colas de distintos tamaños e incluso de tamaño infinito.

13.18. Reescriba la simulación para recoger mediciones estadísticas en lugar de producir como salida cada evento. Después compare la velocidad de la simulación, suponiendo que se dispone de varios cientos de módems y que la simulación es muy larga, con otras que utilicen otras versiones de colas con prioridad (algunas de las cuales están disponibles en la red), como por ejemplo las siguientes:

a) La cola de prioridad asintóticamente ineficiente que se describe en el Ejercicio 6.16.
b) La cola de prioridad asintóticamente ineficiente que se Ejercicio 6.17.
c) Árboles de ensanchamiento (*véase* el Capítulo 21).
d) Montículos sesgados[3] (*véase* el Capítulo 22).
e) Montículos de emparejamientos[4] (*véase* el Capítulo 22).

13.19. Implemente un applet que ilustre el funcionamiento del j usando tanto el algoritmo que utiliza una lista enlazada com un árbol de búsqueda.

[3] *N. del T.*: En inglés «skew heaps».
[4] *N. del T.*: En inglés «pairing heaps».

CAPÍTULO

14 Grafos y caminos

Este capítulo estudia los *grafos* y muestra cómo resolver un problema concreto de gran interés, denominado cálculo de caminos mínimos. Éste es un problema fundamental en computación ya que muchas aplicaciones interesantes pueden modelarse mediante un grafo. Ejemplos de cálculo del camino mínimo son: calcular la ruta más rápida en un transporte, o dirigir el correo electrónico a través de una red de computadores. Este capítulo estudia diversas variantes del problema que dependen de cómo interpretemos el significado de «mínimo» y la clase de propiedades que tengan los grafos. Los problemas de búsqueda del camino mínimo son interesantes porque, aunque los algoritmos son sencillos, también son lentos con grafos de gran tamaño a menos que se tenga cuidado con las estructuras de datos elegidas.

En este capítulo veremos:

- Definiciones formales de grafo y de sus componentes.
- Las estructuras de datos utilizadas para representar un grafo.
- Algoritmos para resolver diversas variantes del problema del camino mínimo, dando implementaciones completas en Java.

14.1 Definiciones

Un grafo $G = (V, E)$ está formado por un conjunto de vértices, V, y un conjunto de aristas, E. Cada arista es un par (v, w), donde $v, w \in V$. En ocasiones los vértices se denominan *nodos*, y las aristas *arcos*. Si los pares que definen las aristas están ordenados, se dice que el grafo es *dirigido*. Los grafos dirigidos se denominan en ocasiones *digrafos*. En un digrafo, el vértice w es *adyacente* al vértice v si y sólo si $(v, w) \in E$. Algunas veces las aristas tienen una tercera componente, denominada *peso* o *coste*. En este capítulo todos los grafos que consideramos serán dirigidos.

El grafo de la Figura 14.1 tiene los siguientes 7 vértices:

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$$

y 12 aristas:

$$E = \begin{Bmatrix} (V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10) \\ (V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2) \\ (V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1) \end{Bmatrix}.$$

Los vértices adyacentes a V_3 son: V_2, V_4, V_5, V_6. En este grafo $|V| = 7$ y $|E| = 12$, donde $|S|$ representa el cardinal de S.

Un *camino* en un grafo es una secuencia de vértices $w_1, w_2, \ldots, w_N$ tal que $(w_i, w_{i+1}) \in E$ para $1 \leq i < N$. La *longitud* de dicho camino es el número de aristas en el camino, es decir, $N - 1$. Esta longitud se denomina *longitud del camino sin pesos*. La *longitud del camino con pesos* es la suma de los costes de las aristas en el camino. Por ejemplo, V_0, V_3, V_5 es un camino desde el vértice V_0 al V_5. La longitud del camino es dos aristas, y la longitud del camino con pesos es 9 (éste es el camino más corto de V_0 a V_5). Sin embargo, si el coste es importante, el camino con pesos más corto entre estos vértices tiene coste 6 y es V_0, V_3, V_6, V_5. Consideramos en particular caminos de un vértice a él mismo. Cuando un camino tal no contiene aristas, su longitud es 0; ésta es la forma más conveniente de tratar este caso especial. Un *camino simple* es un camino en el que todos los vértices son distintos, excepto en lo que refiere al primero y al último, que sí pueden ser iguales.

Un *ciclo* en un grafo dirigido es un camino de longitud al menos 1 que cumple $w_1 = w_N$; este ciclo es simple si el camino lo es. Un grafo dirigido acíclico, a veces referenciado por su abreviatura, GDA, es un grafo dirigido sin ciclos.

Un ejemplo de la vida real que puede modelarse mediante un grafo es un sistema de aeropuertos. Cada aeropuerto es un vértice. Dos vértices están conectados por una arista si hay un vuelo sin paradas entre los correspondientes aeropuertos. La arista puede tener un coste, representando el tiempo, la distancia o el coste del vuelo. Generalmente, la existencia de una arista (v, w) implicará la existencia de la arista (w, v). Pero es razonable suponer que el coste de estas aristas podría ser diferente, ya que volar en sentido diferente podría llevar más tiempo (dependiendo de los vientos dominantes) o costar más (dependiendo de las tarifas locales). Un problema natural que nos gustaría resolver con cierta celeridad es el de determinar el mejor vuelo entre dos aeropuertos, donde «mejor» podría significar el camino con menor número de aristas, o podría entenderse con respecto a alguna de las medidas de peso (distancia, coste, etc.).

Un segundo ejemplo de la vida real que podemos modelar con un grafo es el envío de mensajes de correo electrónico a través de una red de computadores. Los vértices representarían computadores, las aristas enlaces entre pares de computadores y el coste de cada arista sería el coste de la comunicación (pasos de teléfono por megabyte), coste de retraso (segundos por megabyte), o combinaciones de éstos y otros factores.

En la mayoría de los grafos, hay como mucho una arista entre un vértice v y otro w (esto incluye el caso en el que hay una arista en cada dirección). En consecuencia, $|E| \leq |V|^2$. Cuando la mayoría de los vértices están presentes, tenemos $|E| = \Theta(|V|^2)$, y el grafo se denomina *denso* (pues tiene un gran número de aristas).

En la mayoría de las aplicaciones, los grafos son *dispersos* en vez de densos. Por ejemplo, en el modelo de los vuelos, no esperamos que haya vuelos directos entre cada par de aeropuertos. Por el contrario, se dará el caso en el que algunos pocos aeropuertos están muy bien conectados y el resto tiene relativamente pocos vuelos. De la misma forma, en un sistema complejo de transporte de multitudes que involucre trenes y autobuses, desde una estación se pueden alcanzar directamente, lo que estará representado por una arista, sólo unas pocas estaciones. También, en una red de computadores, la mayoría de los computadores están conectadas a unos pocos computadores locales. Por tanto, en la mayoría de los casos, el grafo es relativamente *disperso*, teniéndose $|E| = \Theta(|V|)$, o quizás algo más (no hay una definición estándar de disperso). Por tanto, es particularmente importante que los algoritmos que desarrollemos sean eficientes para grafos dispersos.

14.1.1 Representación

La primera cuestión a considerar es cómo se representa un grafo internamente. Supongamos que los vértices se numeran de forma secuencial empezando en 0, como sugiere el grafo de la Figura 14.1. Una forma sencilla de representar un grafo es utilizar una matriz bidimensional. Esta representación se denomina *matriz de adyacencia*. Para cada arista (v, w), `a[v][w]` representa el coste de la arista; las aristas inexistentes se pueden representar mediante un `INFINITO` lógico. La inicialización del grafo parece requerir que la matriz de adyacencia completa se inicialice a `INFINITO` y, después, para cada arista se actualice la entrada apropiada de la matriz. En este marco, la inicialización conllevaría un tiempo $O(|V|^2)$. Aunque es posible evitar el tiempo cuadrático de inicialización (*véase* Ejercicio 14.3), el coste en espacio será aún $O(|V|^2)$, que resulta razonable para grafos densos pero es totalmente inaceptable para grafos dispersos.

Para grafos dispersos, una representación mejor es la suministrada por las *listas de adyacencia*. Para cada vértice, se mantiene una lista enlazada de todos sus vértices adyacentes. La representación mediante listas de adyacencia del grafo de la Figura 14.1 se muestra en la Figura 14.2. Ya que cada arista aparece en un nodo de una lista, el número de nodos en las listas es exactamente igual al número de aristas. En consecuencia, se utiliza un espacio $O(|E|)$ para almacenar los nodos. Ya que tenemos $|V|$ listas, también se necesita un espacio adicional $O(|V|)$. Si asumimos que todo vértice está en alguna arista, el número de aristas es, al menos, $\lceil |V|/2 \rceil$, por lo que se puede descartar todo término $O(|V|)$ cuando está presente un término $O(|E|)$. Por consiguiente, decimos que el espacio necesario es $O(|E|)$, o lineal con respecto al tamaño del grafo.

Las listas de adyacencia pueden construirse en tiempo lineal a partir de la lista de aristas. Partimos de todas las listas vacías, y cuando nos encontramos con una arista $(v, w, c_{v,w})$, añadimos a la lista de adyacencia de v una entrada formada por w y el coste $c_{v,w}$. La inserción puede realizarse en cualquier punto de la lista, pero resultaría especialmente conveniente hacerlo al principio de la lista, para así garantizar tiempo constante. De esta forma cada arista puede insertarse en tiempo constante, por lo que la estructura de listas de adyacencia puede construirse en tiempo lineal. Observe que cuando insertamos una arista no comprobamos si ya está presente. Esto no puede hacerse en tiempo constante (si utilizamos una lista enlazada sin más), y la comprobación acabaría con la cota lineal en tiempo de la construcción.

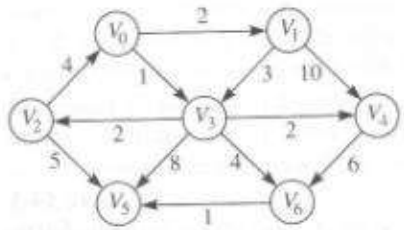


Figura 14.1 Grafo dirigido.

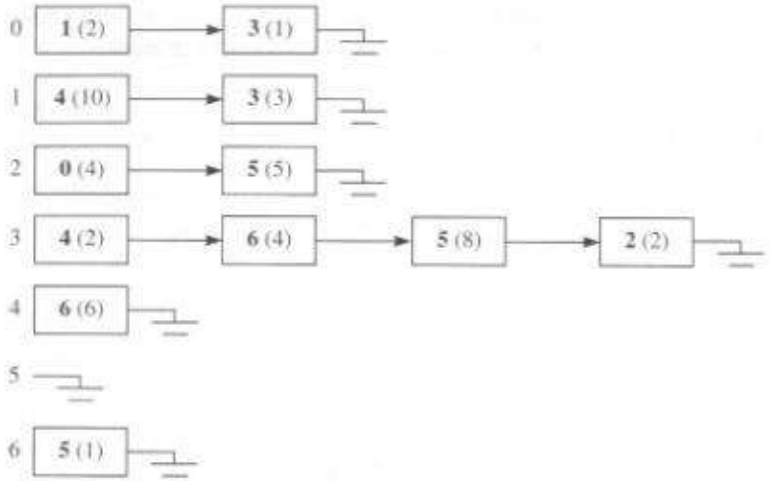


Figura 14.2 Representación con listas de adyacencia del grafo de la Figura 14.1: los nodos en la lista *i* representan los vértices adyacentes a *i* y el coste de la arista correspondiente.

En cualquier caso, esto no tiene gran importancia, pues la mayoría de los algoritmos que funcionan con grafos seguirán funcionando con *multigrafos* en los que hay dos o más aristas de diferentes costes conectando un par de vértices, aunque en algún caso habría que tomar ciertas precauciones al respecto.

En la mayoría de las aplicaciones de la vida real los vértices tienen nombres, en vez de estar denotados por números consecutivos. En tal caso necesitamos proporcionar una forma de transformar dichos nombres en números. La forma más fácil de hacer esto es proporcionar un *diccionario* en el que se asigna a cada nombre de vértice un número interno en el rango de 0 a $|V| - 1$ (el número de vértices se determina al ir ejecutando el programa). Los números internos se van asignando según se va leyendo el grafo. El primer número asignado es el cero. Cada vez que se introduce una arista, comprobamos si se le ha asignado número a cada uno de los dos vértices, viendo si están en el diccionario. Si es así, utilizamos el número interno. En otro caso, asignamos al vértice el siguiente número disponible e insertamos en el diccionario el nombre del vértice junto al número. Con esta transformación, todos los algoritmos sobre grafos utilizarán solamente los números internos.

Ocasionalmente, necesitaremos mostrar el nombre real del vértice y no el número interno, por lo que deberemos guardar, para cada número interno, el correspondiente nombre de vértice. Una forma de hacer esto es guardar una cadena de caracteres para cada vértice. Esta técnica se utilizará más adelante en el capítulo para implementar la clase `Grafo`. Esta clase y los algoritmos de búsqueda de camino mínimo requieren varias estructuras de datos: listas enlazadas, colas, tablas hash y colas de prioridad. Las colas y colas de prioridad se utilizan en varios cálculos de caminos mínimos. Las listas enlazadas y tablas hash se utilizan para representar el grafo. En particular, una tabla hash implementa el diccionario.

Antes de mostrar el esqueleto de la clase `Grafo`, examinemos la Figura 14.3, que muestra cómo será representado nuestro grafo. Como indica la tabla *Entrada*, el usuario proporciona una lista de aristas, una por línea. Al ir a ejecutar el

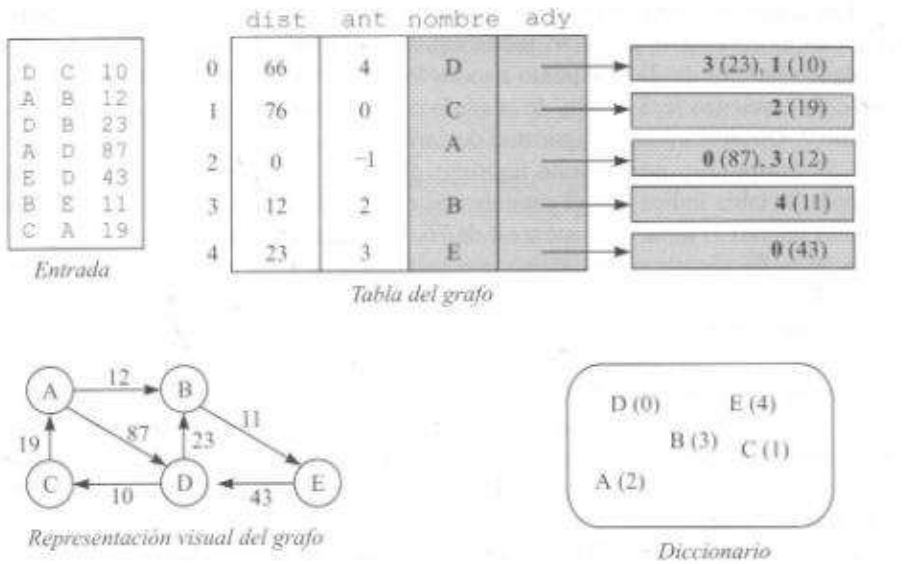


Figura 14.3 Estructuras de datos utilizadas en el cálculo de los caminos mínimos, con un grafo de entrada tomado de un fichero. El camino con pesos más corto desde `A` hasta `C` es (A, B, E, D, C), cuyo coste es 76.

algoritmo no conocemos los nombres de ningún vértice, ni cuántos vértices hay, ni cuántas aristas. Utilizamos dos estructuras de datos para representar el grafo: un diccionario y una tabla. Mantener un diccionario nos permitirá determinar el número interno de cada vértice. Por ejemplo, ya que `D` es el primer vértice en el fichero de entrada, le será asignado el número 0. `C` es el segundo vértice en el fichero de entrada, por lo que se le asignará el número 1. La otra estructura de datos es una gran tabla que almacena información sobre todos los vértices. Como es evidente a partir de la Figura 14.3, la *Tabla del grafo* mantiene cuatro informaciones por cada vértice:

- `dist`: la longitud del camino más corto (con peso o sin peso, dependiendo del algoritmo) desde el vértice de comienzo hasta este vértice. Este valor es calculado por el algoritmo de búsqueda del camino mínimo.
- `ant`: el vértice anterior en el camino más corto a este vértice.
- `nombre`: el nombre correspondiente a este vértice. Se fija cuando el vértice se coloca en el diccionario y no cambiará nunca. Ninguno de los algoritmos de búsqueda del camino mínimo lo utilizará. Sólo se utiliza al imprimir el camino final.
- `ady`: una lista de vértices adyacentes. Se crea al leer el grafo. Ninguno de los algoritmos de búsqueda del camino mínimo modificará dichas listas.

Para ser más específicos, en la Figura 14.3 los elementos sombreados no serán modificados por ninguno de los cálculos del algoritmo de camino mínimo. Estos elementos representan el grafo de entrada y no cambiarán a menos que cambie el grafo (quizás por la posterior incorporación o eliminación de aristas). Los elementos que no están sombreados serán computados por los algoritmos de búsqueda del camino mínimo. Antes del cálculo, podemos suponer que están sin inicializar.

Los algoritmos de búsqueda del camino mínimo que estudiaremos corresponden a un *único origen*. Es decir, todos suponen un punto de origen y calculan los caminos mínimos desde este punto a todos los demás vértices. En este ejemplo, el punto de comienzo será A, que, de acuerdo con el diccionario, tiene como número interno 2. Observe cómo el algoritmo declara que el camino más corto hasta A vale 0. El campo `ant` nos permite imprimir el camino, y no sólo su longitud. Por ejemplo, la tabla indica que el camino más corto desde el vértice origen a C (con número interno 1) tiene un coste total de 76. Obviamente, el último vértice del camino es C. El vértice en el camino que está antes de C es el vértice 0, o sea D. Antes de D está el vértice 4, que es el E. Antes del E está el vértice 3, es decir, B. Y antes de B está el vértice 2, es decir, A, que es el origen. Retrocediendo con el campo `ant` vamos construyendo el camino mínimo. Aunque esta traza nos da el camino en orden inverso, es sencillo invertirlo. El resto de esta sección describe cómo se construye la parte sombreada de la tabla del grafo y presenta el método que imprime un camino mínimo, asumiendo que ya se han calculado los campos `dist` y `ant`. Los algoritmos para rellenar los caminos mínimos se discutirán uno a uno más adelante.

La Figura 14.4 muestra el elemento básico que se coloca en las listas de adyacencia: un número interno de vértice y el coste de la correspondiente arista. Asumimos que la clase `Grafo` utiliza un valor de tipo `String` para el nombre del vértice y un valor de tipo `int` para el coste de la arista. La clase `Vertice` se muestra en la Figura 14.5. Se proporciona un campo adicional denominado `extra`, que tiene diferentes usos según el algoritmo. Todos los demás elementos siguen la descripción anterior. El único método es el constructor que inicializa la lista enlazada.

Ya estamos preparados para examinar el esqueleto de la clase `Grafo`. Su parte pública se muestra en la Figura 14.6 y su parte privada en la Figura 14.7. En las líneas de la 33 a la 35 se declaran algunas constantes. La línea 35 almacena el valor de `INFINITO` (dividimos por 3 para que `INFINITO + INFINITO` no esté fuera de rango). A continuación vienen los atributos.

`dicVertices` almacena el diccionario. Implementamos el diccionario utilizando una tabla hash que almacena un `ElementoHash` que contiene el nombre de un vértice y su número, pero en la que se busca basándonos en el nombre del vér-

```
1  /**
2   * Esta clase representa el elemento básico
3   * de las listas de adyacencia.
4   */
5  class Arista
6  {
7                          // El primer vértice está implícito
8      public int dest;    // Segundo vértice de la arista
9      public int coste;   // Coste de la arista
10
11     public Arista( int d, int c )
12     {
13         dest=d;
14         coste=c;
15     }
16 }
```

Figura 14.4 El elemento básico almacenado en una lista de adyacencia.

```
1  /**
2   * Esta clase representa el elemento básico
3   * almacenado para un vértice.
4   */
5  class Vertice
6  {
7      String nombre; // El nombre real
8      Lista ady;     // La lista de adyacencia
9
10     int dist;      // Coste (después de ejecutar el algoritmo)
11     int ant;       // Vértice anterior en el camino mínimo
12     int extra;     // Variable extra usada por los algoritmos
13
14     Vertice( String nom )
15     {
16         nombre=nom; // Nombre compartido en tabla hash
17         ady =new ListaEnlazada( ); // Nueva lista
18     }
19 }
```

Figura 14.5 La clase `Vertice` almacena información de cada vértice.

```
1  import EstructurasDatos.*;
2  import Soporte.*; import Soporte.Comparable;
3  import Excepciones.*;
4  import java.util.StringTokenizer;
5  import java.io.*;
6
7  // Clase Grafo: calcula caminos mínimos
8  //
9  // CONSTRUCCIÓN: sin inicializador
10 //
11 // ******************OPERACIONES PÚBLICAS**********************
12 // void insArista( String orig, String dest, int coste )
13 //                        --> Inserta arista adicional
14 // boolean procesarPeticion( BufferedReader in )
15 //                        --> Ejecuta varios alg. camino mínimo
16 // ************************ERRORES*******************************
17 // Se hace alguna comprobación de errores sobre si el grafo
18 // es correcto, los parámetros de procesarPeticion son
19 // vértices, y para estar seguros de que el grafo cumple las
20 // propiedades requeridas por cada algoritmo
21
22 /**
23  * Clase Grafo: calcula caminos mínimos.
24  */
25 public class Grafo
26 {
27     public Grafo( )
28       { /* Figura 14.8 */ }
29     public void insArista( String orig, String dest, int coste )
30       { /* Figura 14.12 */ }
31     public boolean procesarPeticion ( BufferedReader in )
32       { /* Figura 14.16 */ }
```

Figura 14.6 Esqueleto de la clase `Grafo` (parte 1: la parte pública).

```
33      private static final int TAMANYO_TABLA_INI = 50;
34      private static final int VERTICE_NULO      = -1;
35      private static final int INFINITO          = 2147483647 / 3;
36
37      private TablaHash dicVertices; // Almacena números internos
38      private Vertice [ ] tabla;     // El vector tabla
39      private int numVertices;       // Número actual de vértices
40
41      private void duplicarVectorTabla( )
42        { /* muy usual, no mostrado */ }
43      private int insNodo( String nombreVertice )
44        { /* Figura 14.10 */ }
45      private void insAristaInterna( int orig, int dest, int coste )
46        { /* Figura 14.11 */ }
47      private void limpiarDatos( )
48        { /* Figura 14.13 */ }
49
50          // Varios algoritmos de caminos mínimos que
51          // requieren un número interno para empezar
52      private void sinPesos( int nodoOrig )
53        { /* Figura 14.24 */ }
54      private boolean dijkstra( int nodoOrig )
55        { /* Figura 14.29 */ }
56      private boolean negativos( int nodoOrig )
57        { /* Figura 14.31 */ }
58      private boolean aciclico( int nodoOrig )
59        { /* Figura 14.34 */ }
60  }
```

Figura 14.7 Esqueleto de la clase `Grafo` (parte 2: la parte privada).

tice. Esto se discutirá brevemente al estudiar la implementación. La línea 38 es la tabla del grafo, `tabla`. El número de vértices en la tabla se almacena en `numVertices` (línea 39). El resto de la clase proporciona métodos para realizar la inicialización, añadir vértices, imprimir el camino más corto y realizar varios cálculos de caminos mínimos. Cada rutina se discute al estudiar su implementación.

Lo primero es el constructor. La Figura 14.8 muestra que el constructor inicializa el número de vértices a 0 y construye la `tabla`. Se crea un diccionario vacío.

Para implementar el diccionario, declaramos una clase denominada `ElementoHash` que almacenará el nombre del vértice, `nombre`, y su número interno, `rango`. La igualdad y la función hash se realizan solamente a partir del campo `nombre`. Las implementaciones se muestran en la Figura 14.9.

```
1  /**
2   * Constructor.
3   */
4  public Grafo( )
5  {
6      numVertices = 0;
7      tabla       = new Vertice[ TAMANYO_TABLA_INI ];
8      dicVertices = new TablaExploracionCuadratica( );
9  }
```

Figura 14.8 Constructor de la clase `Grafo`.

```
1  /**
2   * Esta clase representa la entrada básica
3   * en el diccionario de vértices.
4   * Implementa el interfaz Hashable proporcionando
5   * las funciones hash y equals.
6   */
7  class ElementoHash implements Hashable
8  {
9      public String nombre;        // El nombre real
10     public int rango;            // El número asignado
11
12     public ElementoHash( )
13       { this( null ); }
14
15     public ElementoHash( String nom )
16       { nombre = nom; }
17
18     public int hash( int tamanyoTabla )
19       { return TablaExploracionCuadratica.hash( nombre, tamanyoTabla ); }
20
21     public boolean equals( Object lder )
22       { return nombre.equals( ((ElementoHash) lder).nombre ); }
23 }
```

Figura 14.9 `ElementoHash` utilizado para implementar el diccionario.

```
1  /**
2   * Si nombreVertice ya es un vértice, devuelve su
3   * número interno. En otro caso, lo añade como nuevo vértice,
4   * y devuelve su nuevo número interno.
5   */
6  private int insNodo( String nombreVertice )
7  {
8      ElementoHash hashV = new ElementoHash( nombreVertice );
9      ElementoHash resultado;
10
11     try
12     {
13         resultado = (ElementoHash) dicVertices.buscar( hashV );
14         return resultado.rango;
15     }
16     catch( ElementoNoEncontrado e )
17     {
18         // Vértice nuevo
19         hashV.rango = numVertices;
20         hashV.nombre = new String( nombreVertice );
21         dicVertices.insertar( hashV );
22
23         if( numVertices == tabla.length )
24             duplicarVectorTabla( );
25         tabla[ numVertices ] = new Vertice( hashV.nombre );
26         return numVertices++;
27     }
28 }
```

Figura 14.10 Rutina `insNodo`, devuelve el número interno de `nombreVertice`.

insNodo, mostrado en la Figura 14.10, devuelve el número interno del vértice, correspondiente al parámetro nombreVertice. El nombre de la rutina refleja el hecho de que si nombreVertice no se ha encontrado todavía (lo que viene reflejado por su ausencia en el diccionario), se le asigna el próximo número interno disponible y se inserta en el diccionario, inicializándose también su entrada en el vector tabla.

El procedimiento empieza con una consulta a la tabla hash. Para hacer esto, debemos crear primero, en la línea 8, un objeto de la clase ElementoHash, hashV, inicializándolo con nombreVertice (utilizando un constructor apropiado).

Una vez construido hashV, podemos realizar una búsqueda en la tabla hash y almacenar el valor devuelto en resultado. Si la búsqueda tiene éxito, podemos devolver el campo rango de resultado, y acabamos. En otro caso, hemos encontrado un vértice nuevo.

En la línea 19, le asignamos a ese nuevo vértice como número interno numVertices, el cual representa el próximo índice disponible en la tabla (porque el índice empieza en 0).

En la línea 21, realizamos una inserción en la tabla hash. Observe cómo creamos un nuevo objeto de la clase String que será referenciado por la tabla hash y por el vector tabla. Se construye la correspondiente entrada Vertice, y se añade a tabla en la línea 25. Si la tabla del grafo tabla se llena, en la línea se duplica su tamaño. La línea 26 completa la rutina devolviendo numVertices (el número interno asignado a nombreVertice) e incrementando numVertices.

La rutina de la Figura 14.11 añade una arista cuyos vértices vienen dados por números internos. Ésta es una rutina sencilla porque lo único que precisa es crear un objeto de la clase Arista con el vértice destino y el coste, e insertarlo en la lista de adyacencias correspondiente al vértice de origen. La rutina de uso público se muestra en la Figura 14.12. insArista inserta una arista cuyos vértices vienen

```
1  /**
2   * Añade una arista dados los números internos de sus vértices.
3   */
4  private void insAristaInterna( int orig, int dest, int coste )
5  {
6      ListaIter p = new ListaEnlazadaIter( tabla[ orig ].ady );
7      try
8        { p.insertar( new Arista( dest, coste ) ); }
9      catch( ElementoNoEncontrado e ) { } // No puede ocurrir
10 }
```

Figura 14.11 Añade al grafo la arista (orig, dest, coste) insertándola dentro de la lista de adyacencia de orig; orig y dest son números internos.

```
1  /**
2   * Añade la arista ( orig, dest, coste ) al grafo.
3   */
4  public void insArista( String orig, String dest, int coste )
5  {
6      insAristaInterna( insNodo( orig ), insNodo( dest ), coste );
7  }
```

Figura 14.12 La misma rutina que insAristaInterna, pero aquí orig y dest son objetos de la clase String.

dados por valores de tipo String. Ésta es otra rutina sencilla pues se limita a llamar a insNodo para obtener los números internos correspondientes. Tras ello se llama a insAristaInterna.

```
1  /**
2   * Inicializa la tabla.
3   */
4  private void limpiarDatos( )
5  {
6      for( int i = 0; i < numVertices; i++ )
7      {
8          tabla[ i ].dist = INFINITO;
9          tabla[ i ].ant = VERTICE_NULO;
10         tabla[ i ].extra = 0;
11     }
12 }
```

Figura 14.13 Rutina para inicializar los campos de la tabla, que se usan en los algoritmos de búsqueda de caminos mínimos.

```
1  /**
2   * Imprime recursivamente el camino mínimo a nodoDest
3   * (especificado por su número interno).
4   * imprimirCamino es la rutina guía
5   */
6  private void imprimirCaminoRec( int nodoDest )
7  {
8      if( tabla[ nodoDest ].ant != VERTICE_NULO )
9      {
10         imprimirCaminoRec( tabla[ nodoDest ].ant );
11         System.out.print( " a " );
12     }
13     System.out.print( tabla[ nodoDest ].nombre );
14 }
```

Figura 14.14 Rutina recursiva para imprimir el camino mínimo.

```
1  /**
2   * Rutina guía para tratar casos inalcanzables e
3   * imprimir el coste total. Llama a la rutina recursiva
4   * para imprimir el camino mínimo a nodoDest.
5   */
6  private void imprimirCamino( int nodoDest )
7  {
8      if( tabla[ nodoDest ].dist == INFINITO )
9          System.out.println( tabla[ nodoDest ].nombre +
10                             " es inalcanzable" );
11     else
12     {
13         imprimirCaminoRec( nodoDest );
14         System.out.println( " (el coste es " +
15                             tabla[ nodoDest ].dist + ")" );
16     }
17     System.out.println( );
18 }
```

Figura 14.15 Rutina para imprimir el camino mínimo consultando la tabla.

Los atributos eventualmente calculados por algún algoritmo de búsqueda de caminos mínimos son inicializados por la rutina `limpiarDatos`, mostrada en la Figura 14.13. La siguiente es una rutina para imprimir el camino mínimo una vez su cálculo ha sido realizado. Recuerde que el campo `ant` puede utilizarse para recorrer el camino hacia atrás. Sin embargo, esto nos dará el camino en orden inverso. Esto no es un problema si utilizamos recursión; los vértices en el camino a `dest` son los mismos que aquéllos en el camino al vértice anterior a `dest` (en el camino), seguidos de `dest`. Esta estrategia se traduce directamente en la pequeña rutina recursiva mostrada en la Figura 14.14. La rutina supone que existe el camino en cuestión. `imprimirCamino`, mostrada en la Figura 14.15, realiza primero esta comprobación; si no existe camino, imprime un mensaje. En otro caso, llama a la rutina recursiva e imprime el coste del camino.

```
1  /**
2   * Procesa una petición; devuelve falso si se acaba el fichero.
3   */
4  public boolean procesarPeticion( BufferedReader in )
5  {
6      String nombreOrig, nombreDest;
7      ElementoHash orig = new ElementoHash( );
8      ElementoHash dest = new ElementoHash( );
9
10     try
11     {
12         System.out.println( "Introduzca nodo origen:" );
13         if( ( nombreOrig = in.readLine( ) ) == null )
14           return false;
15         System.out.println( "Introduzca nodo destino:" );
16         if( ( nombreDest = in.readLine( ) ) == null )
17           return false;
18     }
19     catch( IOException e )
20     {
21         System.out.println( "Error: " + e );
22         return false;
23     }
24
25     try
26     {
27         orig.nombre = nombreOrig;
28         orig = (ElementoHash) ( dicVertices.buscar( orig ) );
29         dest.nombre = nombreDest;
30         dest = (ElementoHash) ( dicVertices.buscar( dest ) );
31
32         if( dijkstra( orig.rango ) )
33           imprimirCamino( dest.rango );
34         else
35           System.out.println( "Dijkstra ha fallado" );
36     }
37     catch( ElementoNoEncontrado e )
38       { System.err.println( "Vértice no está en el grafo" ); }
39     return true;
40 }
```

Figura 14.16 Para probar, `procesarPeticion` llama al algoritmo de búsqueda del camino mínimo (con pesos).

La última rutina, a parte de las que calculan los caminos mínimos, es el método `procesarPeticion`. En la Figura 14.16 se muestra una implementación sencilla que pregunta el vértice origen y el vértice destino y ejecuta el algoritmo de búsqueda del camino con peso mínimo (el código en Internet ejecuta también otros algoritmos adicionales).

```
1  /**
2   * Rutina main que: pide el nombre de un fichero que
3   * contenga un grafo; construye el grafo a partir del fichero;
4   * repetidamente pide dos vértices y ejecuta los algoritmos
5   * de caminos mínimos. El fichero de datos es una secuencia
6   * de líneas: origen destino coste.
7   */
8  public static void main( String [ ] args )
9  {
10     System.out.println( "Introduzca fichero con el grafo:" );
11     BufferedReader in = new BufferedReader( new
12                         InputStreamReader( System.in ) );
13     FileReader fich;
14     String nombreFichero = "";
15
16     try
17     {
18         nombreFichero = in.readLine( );
19         fich = new FileReader( nombreFichero );
20     }
21     catch( Exception e )
22       { System.err.println( e ); return; }
23
24     BufferedReader fichGrafo = new BufferedReader( fich );
25     Grafo g = new Grafo( );
26
27         // Lee las aristas y las inserta
28     try
29     {
30         String linea;
31         while( ( linea = fichGrafo.readLine( ) ) != null )
32         {
33             StringTokenizer st = new StringTokenizer( linea );
34             try
35             {
36                 if( st.countTokens( ) != 3 )
37                     throw new Exception( );
38                 String orig = st.nextToken( );
39                 String dest = st.nextToken( );
40                 int coste = Integer.parseInt( st.nextToken( ) );
41                 g.insArista( orig, dest, coste );
42             }
43             catch( Exception e )
44               { System.err.println( "Error: " + linea ); }
45         }
46     }
47     catch( Exception e )
48       { System.err.println( "Error: " + e ); }
49     while( g.procesarPeticion( in ) )
50         ;
51 }
```

Figura 14.17 Ejemplo de rutina `main`.

Una rutina `main` sencilla, mostrada en la Figura 14.17, ilustra el comportamiento de la clase. Pide el nombre de un fichero que contenga un grafo. Repetidamente lee líneas de la entrada, asignando la línea a un objeto de la clase `StringTokenizer`, y analiza la línea. Esta técnica nos permite comprobar que cada línea contiene los tres campos de una arista. Cuando se encuentran los tres campos, se llama a `insArista`. Una vez leído el grafo, se ejecuta el algoritmo de búsqueda del camino con peso mínimo, llamando a `procesarPeticion`.

14.2 Problema del camino mínimo sin pesos

La *longitud del camino sin pesos* mide el número de aristas. Esta sección estudia el problema de encontrar el camino que tenga menor longitud sin pesos entre dos vértices especificados.

PROBLEMA DEL CAMINO SIN PESOS MÍNIMO CON UN ÚNICO ORIGEN
Encontrar el camino más corto (medido por el número de aristas) desde el vértice O a cualquier otro vértice.

El problema del camino mínimo sin pesos es un caso especial del problema del camino mínimo con pesos (en el que todos los pesos valen 1). Por tanto, debería tener una solución más eficiente que el problema con pesos. Esto en efecto es cierto, aunque los algoritmos para todos los problemas de caminos mínimos son muy similares.

14.2.1 Teoría

Para resolver el problema del camino mínimo sin pesos, utilizamos el grafo de la Figura 14.1, con V_2 como nodo origen O. Por ahora, nos ocuparemos de encontrar la longitud de los caminos mínimos. Más tarde, también llevaremos cuenta del correspondiente camino.

Inmediatamente, podemos decir que el camino más corto desde O a V_2 es un camino de longitud 0. Esta información produce el grafo de la Figura 14.18. Ahora podemos empezar a mirar los vértices que están a distancia 1 de O. Si hacemos esto, vemos que V_0 y V_5 están a una arista de distancia de O. Esto se muestra en la Figura 14.19.

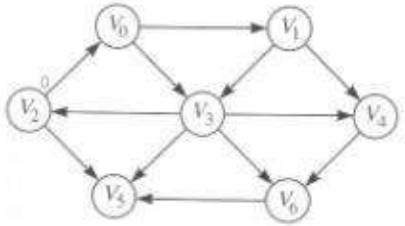


Figura 14.18 El grafo después de marcar el nodo origen como alcanzable con cero aristas.

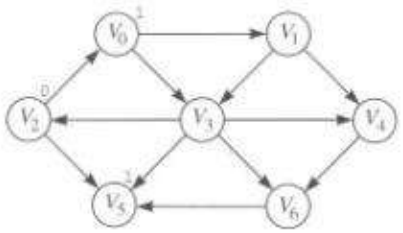


Figura 14.19 Grafo después de encontrar todos los vértices cuyo camino mínimo desde el origen tiene longitud 1.

El siguiente paso es buscar los vértices cuyo camino mínimo desde O sea exactamente 2. Hacemos esto buscando los nodos adyacentes a V_0 y V_5 (los vértices a distancia 1) cuyo camino mínimo no se conozca aún. Esta búsqueda nos dice que el camino más corto a V_1 y V_3 es 2. La Figura 14.20 muestra el progreso realizado hasta el momento.

Finalmente, examinando los vértices adyacentes a los últimamente alcanzados V_1 y V_3, encontramos que V_4 y V_6 tienen un camino mínimo de 3 aristas. Y ya hemos calculado el camino para todos los vértices. La Figura 14.21 muestra el resultado final del algoritmo.

Esta estrategia para buscar un camino se denomina *búsqueda en anchura*. Opera procesando los vértices por niveles: los vértices más cercanos al origen se procesan los primeros, y los más alejados los últimos.

La Figura 14.22 ilustra un principio fundamental. Si un camino al vértice v tiene coste D_v, y w es adyacente a v, entonces existe un camino a w de coste $D_w = D_v + 1$. Todos los algoritmos de búsqueda de caminos mínimos empiezan con $D_w = \infty$ y van reduciendo este valor cuando se explora un v apropiado. Para

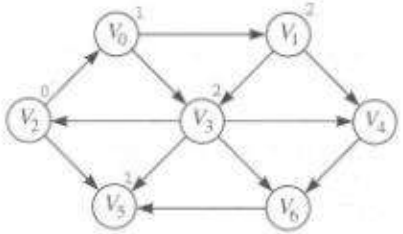


Figura 14.20 Grafo después de encontrar todos los vértices cuyo camino mínimo desde el origen tiene longitud 2.

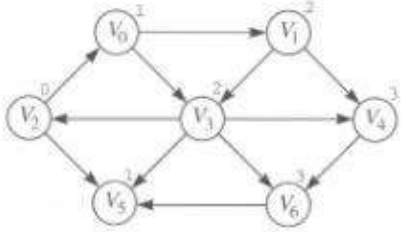


Figura 14.21 Caminos mínimos finales.

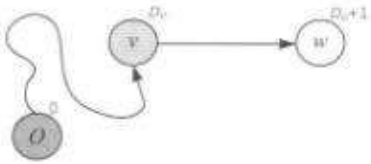


Figura 14.22 Si w es adyacente a v y hay un camino a v, también hay un camino a w.

hacer esto eficientemente, los vértices deben explorarse de forma sistemática. Cuando se explora un determinado v, se realizan las actualizaciones de los vértices w adyacentes recorriendo la lista de adyacencia de v.

De la discusión anterior, concluimos el siguiente algoritmo para la resolución del problema de los caminos mínimos sin pesos: sea D_i la longitud del camino más corto desde O hasta i. Sabemos que $D_O = 0$, e inicialmente $D_i = \infty$ para todo $i \neq O$. Mantenemos nuestra *atención* fija en un vértice que inicialmente es O y vamos saltando de vértice en vértice. Si v es el vértice sobre el que nos, entonces, para todo w adyacente a v, hacemos $D_w = D_v + 1$ si $D_w = \infty$. Esto refleja el hecho de que podemos llegar a w siguiendo un camino a v y extendiendo el camino con la arista (v, w). De nuevo esto se ilustra en la Figura 14.22. Ya que nuestra atención va procesando cada vértice en orden de distancias al origen, y cada arista suma exactamente uno a la longitud del camino a w, podemos estar seguros de que la primera vez que D_w deja de ser ∞, tomará el valor del camino mínimo a w. Por la misma razón tenemos que el penúltimo vértice en el camino a w es v, por lo que una línea adicional de código nos permitirá almacenar el camino en cuestión.

Después de haber procesado todos los vértices adyacentes a v, pasamos nuestra atención a otro vértice u (en el que todavía no nos hayamos fijado) tal que $D_u = D_v$. Si ello no es posible, pasamos a un u que satisfaga $D_u = D_v + 1$. Si tampoco es posible, es que hemos acabado. La Figura 14.23 muestra cómo vamos visitando vértices y actualizando las distancias. El nodo con sombra clara en cada estado representa nuestro punto actual de atención. En este dibujo y los que siguen, los estados se muestran consecutivamente de arriba abajo, y de izquierda a derecha.

El detalle que falta concretar es la estructura de datos subyacente. Hay dos acciones básicas a realizar. Debemos, primero, buscar repetidamente el vértice en el que depositaremos nuestra atención. En segundo lugar, durante el algoritmo necesitamos comprobar todos los vértices w adyacentes a v (el vértice actual). La segunda acción se implementa fácilmente iterando a través de la lista de adyacencia de v. De hecho, ya que cada arista se procesa exactamente una vez, el coste total de todas las iteraciones es $O(|E|)$. La primera acción es más complicada: no podemos limitarnos a explorar la tabla buscando el vértice apropiado, pues cada exploración requeriría un tiempo $O(|V|)$ y necesitamos hacerlo $O(|V|)$ veces. Entonces el coste total sería $O(|V|^2)$, lo cual es inaceptable para grafos dispersos. Afortunadamente, esto no es necesario.

Cuando a un vértice w se le rebaja su D_w desde ∞, se convierte en candidato para fijar nuestra atención en él en algún momento del futuro. Una vez hayamos fijado nuestra atención en todos los vértices que se encuentran a igual distancia del origen, D_v, que el actual, pasaremos a considerar los que se encuentran a distancia $D_v + 1$, entre los que se encuentra w. Por lo que w sólo le queda esperar pacientemente su turno. Claramente, no tiene por qué hacerlo delante de cualquier

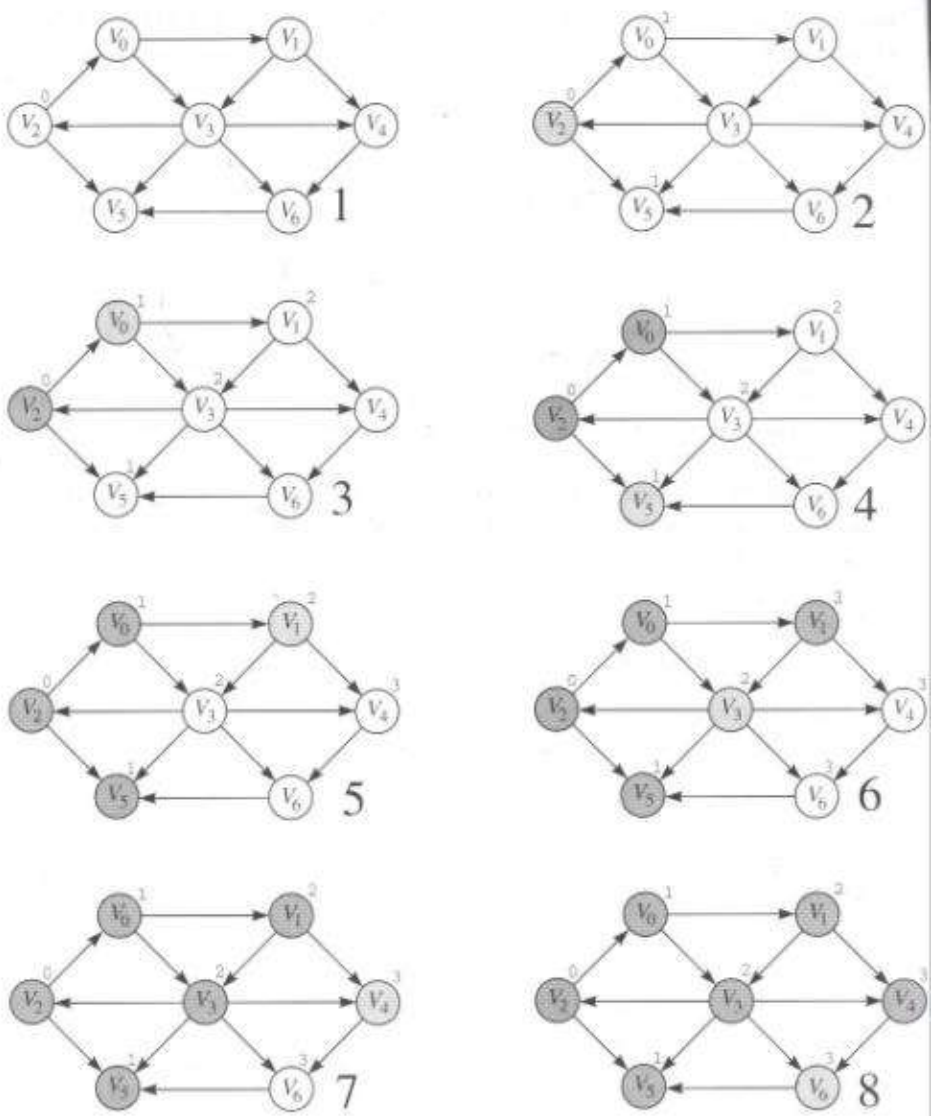


Figura 14.23 Cómo se recorre el grafo en el cálculo de los caminos mínimos sin pesos. Los vértices más oscuros han sido ya totalmente procesados, los vértices más claros todavía no han sido considerados como v, y aquel con sombreado intermedio es el vértice v actual. Los estados van de arriba abajo y de izquierda a derecha, según indica la numeración.

otro nodo cuya distancia se haya rebajado ya, por lo que puede colocarse al final de una cola de vértices en espera.

Para seleccionar el siguiente vértice v en el que fijemos nuestra atención, simplemente tomamos el primer vértice de la cola. Empezamos con una cola vacía, y para que el mecanismo empiece a funcionar colocamos en la cola el vértice origen O. Ya que cada vértice se introduce en la cola y se elimina de ella exactamente

una vez, y ya que las operaciones sobre las colas son constantes, el coste total de elegir un vértice *para el algoritmo completo* es $O(|V|)$. Por tanto, el coste de la búsqueda en anchura está dominado por los recorridos de las listas de adyacencias, y en consecuencia es $O(|E|)$, o sea lineal con respecto al tamaño del grafo.

14.2.2 Implementación en Java

La implementación del algoritmo de los caminos mínimos sin peso se realiza en el método `sinPeso`, mostrado en la Figura 14.24. El código es una traducción literal del algoritmo descrito anteriormente. La inicialización de las líneas de la 9 a la 11 inicializa todas las distancias a infinito, D_O a 0 y añade a la cola el vértice origen. Mientras que la cola no se quede vacía, hay vértices que visitar. Por tanto, en la línea 17, pasamos a fijar la atención en el vértice v que es el primero de la cola. La línea 19 itera sobre su lista de adyacencia y produce todos los vértices `w` adyacentes a `v`. El test $D_w = \infty$ se realiza en la línea 22. Si es `true`, la actualización $D_w = D_v + 1$ se realiza en la línea 24, junto con la actualización del campo `ant` y la inserción en la cola de `w`, en las líneas 25 y 26, respectivamente.

```
1  /**
2   * Calcula el camino mínimo sin pesos.
3   */
4  private void sinPesos( int nodoOrig )
5  {
6      int v, w;
7      Cola q=new ColaVec( );
8
9      limpiarDatos( );
10     tabla[ nodoOrig ].dist = 0;
11     q.insertar( new Integer( nodoOrig ) );
12
13     try
14     {
15         while( !q.esVacia( ) )
16         {
17             v=( (Integer) q.quitarPrimero( ) ).intValue( );
18             ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
19             for( ; p.estaDentro( ); p.avanzar( ) )
20             {
21                 w=( (Arista) p.recuperar( ) ).dest;
22                 if( tabla[ w ].dist == INFINITO )
23                 {
24                     tabla[ w ].dist = tabla[ v ].dist + 1;
25                     tabla[ w ].ant = v;
26                     q.insertar( new Integer( w ) );
27                 }
28             }
29         }
30     }
31     catch( DesbordamientoInferior e ) { } // No puede ocurrir
32 }
```

Figura 14.24 Algoritmo para el problema del camino mínimo sin pesos utilizando una búsqueda en anchura.

14.3 Problema de los caminos mínimos con pesos positivos

La *longitud de un camino con pesos* es la suma del coste de las aristas del camino. Esta sección considera el problema de encontrar el camino más corto con pesos. En el problema de los caminos mínimos con pesos positivos, las aristas tienen costes no negativos. Queremos calcular los caminos mínimos desde un vértice origen al resto de vértices. Como se verá enseguida, la hipótesis de que los costes de las aristas son no negativos es muy importante pues permite utilizar un algoritmo sencillo y relativamente eficiente. El método descrito se conoce como *algoritmo de Dijkstra*. En la siguiente sección estudiaremos un algoritmo más lento que funciona aun con aristas negativas.

PROBLEMA DE LOS CAMINOS MÍNIMOS CON COSTE POSITIVO Y ORIGEN ÚNICO

Encontrar el camino más corto (medido con su coste total) desde el vértice origen O al resto de vértices. El coste de cada arista es no negativo.

14.3.1 Teoría: algoritmo de Dijkstra

El problema de los caminos mínimos con pesos se resuelve de una manera similar al problema sin pesos. Sin embargo, debido al coste de las aristas, unas pocas cosas cambian.

Las siguientes cuestiones deben ser examinadas:

1. ¿Cómo se ajusta D_w?
2. ¿Cómo buscamos el siguiente vértice v en el que centrar nuestra atención?

Empezamos examinando como se altera D_w. Al resolver el problema sin pesos, si $D_w = \infty$, hacíamos $D_w = D_v + 1$ porque rebajábamos el valor de D_w si el vértice v ofrecía un camino más corto a w. La dinámica del algoritmo aseguraba que sólo necesitábamos alterar D_w una vez. Sumábamos 1 a D_w porque la longitud del camino a w es 1 más que la longitud del camino a v. Si aplicamos la misma lógica al caso con pesos, deberíamos hacer $D_w = D_v + c_{v,w}$, cuando este nuevo valor de D_w sea mejor que el original. Sin embargo, ahora no podemos garantizar que D_w se actualice una sola vez. En consecuencia, D_w debería modificarse si su valor actual es mayor que $D_v + c_{v,w}$ (en vez de simplemente compararlo con ∞). Simplemente, el algoritmo decide si de momento es una buena idea pasar por v para ir hasta w. El coste original D_w es el coste sin utilizar v; el coste $D_v + c_{v,w}$ es el camino más barato utilizando los vértices v considerados hasta la fecha.

La Figura 14.25 muestra una situación típica. Anteriormente durante la ejecución del algoritmo, se rebajó la distancia a w a 8 cuando fijamos nuestra atención en el vértice u. Sin embargo, cuando consideramos v, necesitamos volver a cambiar la distancia a w, rebajándola a 6, pues tenemos un nuevo camino mínimo provisional. Esto nunca ocurría en el algoritmo sin pesos porque todas las aristas sumaban 1 a la longitud del camino, por lo que $D_u \leq D_v$ implicaba $D_u + 1 \leq D_v + 1$, y por tanto $D_w \leq D_v + 1$. Aquí, aunque $D_u \leq D_v$, es todavía posible que el camino a w se mejore al considerar v.

La Figura 14.25 muestra otro punto importante. Cuando se rebaja la distancia a w, es siempre porque es adyacente a algún vértice en el que hemos fijado nuestra atención. Por ejemplo, después de visitar v y completado su procesamiento, el valor de D_w será 6 y el último vértice en el camino será un nodo ya visitado. De forma análoga, el vértice anterior a v debe haberse visitado, y así sucesivamente. Por tanto, en cualquier momento, el valor de D_w representa un camino desde O hasta w utilizando como nodos intermedios solamente vértices que han sido ya visitados. Este hecho crucial nos lleva al Teorema 14.1.

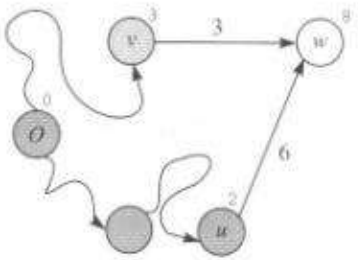


Figura 14.25 Tenemos fijada la atención en el vértice v; w es adyacente; D_w debería reducirse a 6.

Si vamos pasando nuestra atención a un vértice aún no visitado de entre los que minimicen el valor D_i, el algoritmo producirá correctamente los caminos mínimos siempre que no haya aristas con coste negativo.

Llamemos «paso» a cada visita a un nodo. Probaremos por inducción que, después de cada paso, los valores de D_i para vértices ya visitados corresponden al camino mínimo, y que los valores de D_i para los demás vértices corresponden al camino más corto hasta ellos que utiliza solamente como nodos intermedios vértices ya visitados. Ya que el primer nodo visitado es el origen, la afirmación es cierta para el primer paso. Supongamos que es correcta para los primeros k estados. Sea v el vértice visitado en el paso k+1. Supongamos, con la intención de llegar a una contradicción, que hay un camino de O a v de longitud menor que D_v. Este camino debe pasar por un vértice intermedio que no haya sido todavía visitado. Llamemos u al primer vértice intermedio en el camino que no haya sido visitado. La situación se muestra en la Figura 14.26. El camino hasta u utiliza sólo nodos intermedios visitados, por lo que, por inducción, D_u representa la distancia óptima a v. Es más, $D_u < D_v$, porque u está en el supuesto camino a v. Esto es una contradicción, porque si fuera así habríamos elegido u como siguiente vértice a visitar en lugar de v. La demostración se completa mostrando que el resto de valores D_i para nodos no visitados siguen siendo correctos; esto es evidente, dada la regla de actualización.

La Figura 14.27 muestra los pasos del algoritmo de Dijkstra. Lo que nos queda por hacer es elegir las estructuras de datos apropiadas. Para grafos densos, podemos recorrer la tabla buscando el vértice apropiado. Como en el caso sin pesos,

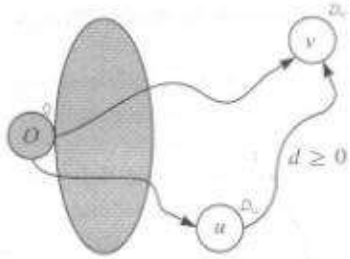


Figura 14.26 Si D_v es mínimo entre todos los vértices no visitados y si todas las aristas tienen coste no negativo, entonces D_v representa el camino mínimo hasta v.

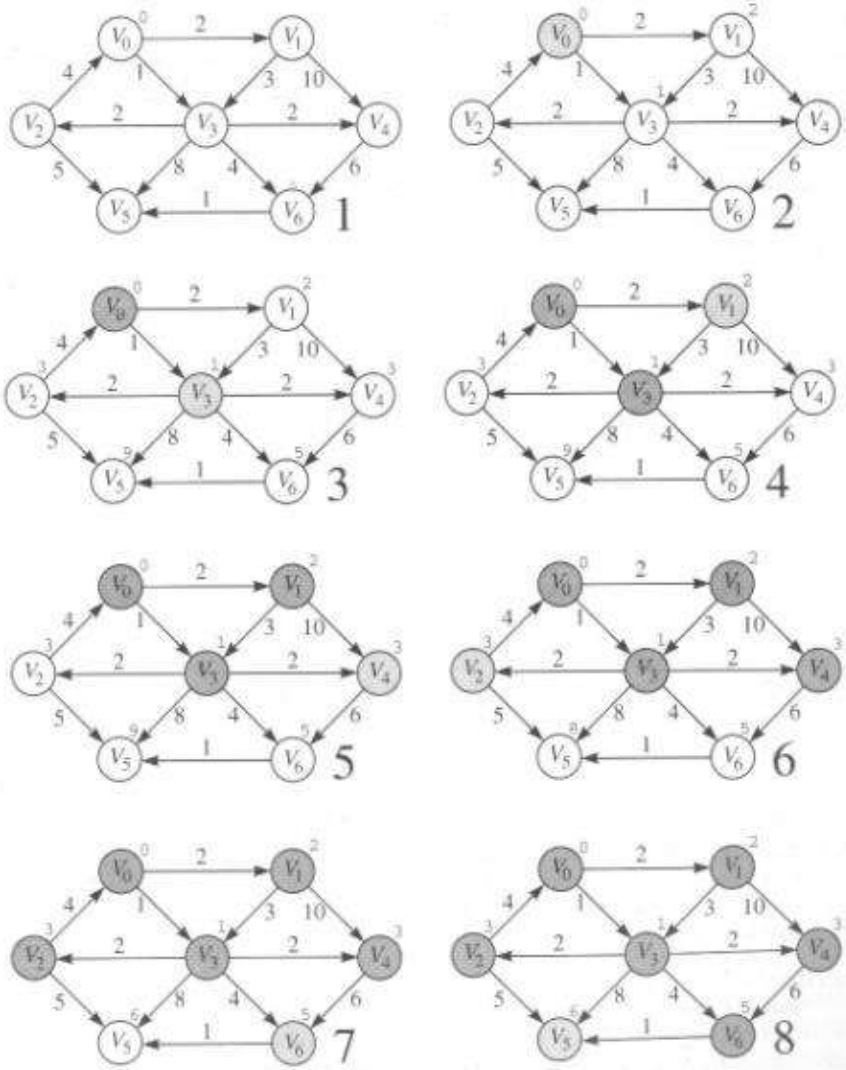


Figura 14.27 Pasos del algoritmo de Dijkstra. Sigue los mismos convenios que en la Figura 14.23.

esto nos llevaría a un tiempo $O(|V|^2)$, que es óptimo para un grafo denso. Pero para un grafo disperso, podemos hacerlo mejor.

Ciertamente, una cola no funcionaría. El hecho de que debamos buscar el vértice v con menor D_v sugiere la utilización de una cola de prioridad. Hay dos formas de utilizar esta cola de prioridad. Una forma es almacenar cada vértice en la cola de prioridad y utilizar la distancia (obtenida consultando la tabla del grafo) como función de ordenación. Cuando alteramos cualquier D_w, debemos actualizar la cola de prioridad restableciendo la propiedad de orden. Esto equivale a una operación `flotar`. Para hacer esto necesitamos ser capaces de encontrar la posición de w en la cola de prioridad. No todas las implementaciones de las colas de prioridad soportan esta operación. En otras palabras, `flotar` no es una operación de la interfaz `ColaPrioridad`. Una implementación que sí soporta esta operación es el montículo de emparejamientos; su utilización en esta aplicación se discute en el Capítulo 22.

En vez de utilizar una cola de prioridad compleja, utilizaremos un método que funciona con cualquier estructura de datos que implemente el interfaz `ColaPrioridad`. Nuestro método consistirá en insertar en la cola de prioridad un objeto formado por w y D_w cuando reducimos D_w. Para elegir el nuevo vértice v a visitar, eliminaremos repetidamente el menor elemento (utilizando las distancias) de la cola de prioridad hasta que salga un elemento no visitado. Ya que el tamaño de la cola de prioridad podría ser tan grande como $|E|$, y hay como mucho $|E|$ inserciones y eliminaciones, el tiempo de ejecución sería $O(|E| \log |E|)$. Ya que $|E| \leqslant |V|^2$ implica $\log |E| \leqslant 2 \log |V|$, obtenemos el mismo coste $O(|E| \log |V|)$ que tendríamos si utilizáramos el primer método (en el cual el tamaño de la cola de prioridad es como mucho $|V|$).

```
1  /**
2   * Objeto almacenado en la cola de prioridad
3   * en el algoritmo de Dijkstra
4   */
5  class Camino implements Comparable
6  {
7      int dest;     // w
8      int coste;    // D(w)
9
10     static Camino infNeg = new Camino( ); // Centinela
11
12     Camino( )
13       { this( 0 ); }
14
15     Camino( int d )
16       { this( d, 0 ); }
17
18     Camino( int d, int c )
19       { dest = d; coste = c; }
20
21     public boolean menorQue( Comparable lder )
22       { return coste < ( (Camino) lder ).coste; }
23
24     public int compara( Comparable lder )
25       { return coste < ( (Camino) lder ).coste ? -1 :
26                coste > ( (Camino) lder ).coste ? 1 : 0; }
27 }
```

Figura 14.28 Elemento básico almacenado en la cola de prioridad.

14.3.2 Implementación en Java

El objeto que se colocará en la cola de prioridad se muestra en la Figura 14.28. Está formado por w y D_w, y una función de comparación definida en base a D_w. La Figura 14.29 muestra la rutina `dijkstra` que calcula los caminos mínimos.

```
1  /**
2   * Algoritmo de Dijkstra utilizando un monticulo binario.
3   * Devuelve false si se detectan aristas negativas.
4   */
5  private boolean dijkstra( int nodoOrig )
6  {
7      int v, w;
8      ColaPrioridad cp=new MonticuloBinario( Camino.infNeg );
9      Camino vrec;
10
11     limpiarDatos( );
12     tabla[ nodoOrig ].dist=0;
13     cp.insertar( new Camino( nodoOrig, 0 ) );
14
15     try
16     {
17         for( int nodosVistos=0; nodosVistos < numVertices;
18                                                 nodosVistos++ )
19         {
20             do
21             {
22                 if( cp.esVacia( ) )
23                     return true;
24                 vrec=(Camino) cp.eliminarMin( );
25             } while( tabla[ vrec.dest ].extra != 0 );
26
27             v=vrec.dest;
28             tabla[ v ].extra=1;
29
30             ListaIter p=new ListaEnlazadaIter( tabla[ v ].ady );
31             for( ; p.estaDentro( ); p.avanzar( ) )
32             {
33                 w=( (Arista) p.recuperar( ) ).dest;
34                 int cvw=( (Arista) p.recuperar( ) ).coste;
35
36                 if( cvw < 0 )
37                     return false;
38
39                 if( tabla[ w ].dist > tabla[ v ].dist+cvw )
40                 {
41                     tabla[ w ].dist=tabla[ v ].dist+cvw;
42                     tabla[ w ].ant=v;
43                     cp.insertar( new Camino( w, tabla[ w ].dist ) );
44                 }
45             }
46         }
47     }
48     catch( DesbordamientoInferior e ) { } // No puede ocurrir
49
50     return true;
51 }
```

Figura 14.29 Algoritmo de búsqueda de caminos mínimos con pesos: algoritmo de Dijkstra.

La línea 8 declara la cola de prioridad `cp`. Recuerde que debemos proporcionar un centinela para que se garantice que sea menor o igual que cualquier objeto insertado. `vrec`, declarado en la línea 9, almacenará el resultado de cada `eliminarMin`. Como en el caso del algoritmo sin pesos, empezamos inicializando todas las distancias a infinito, haciendo $D_O = 0$ y colocando el vértice origen dentro de la estructura de datos.

Cada iteración del bucle `for` más externo empieza en la línea 17 fijando nuestra atención en el vértice v y procesándolo, examinando los vértices adyacentes w. v se elige eliminando repetidamente elementos de la cola de prioridad (en la línea 24) hasta encontrar un vértice aún no procesado. Utilizamos el atributo `extra` para almacenar esta información. Inicialmente `extra` vale 0. Por tanto, cuando el vértice no ha sido procesado, la comparación del bucle `while` fallará en la línea 25. Cuando el vértice se procesa, `extra` se actualiza a 1 (en la línea 28). La cola de prioridad podría quedarse vacía si, por ejemplo, alguno de los vértices es inalcanzable. En este caso, podemos acabar inmediatamente. El bucle de las líneas 31 a 45 es muy similar al caso del algoritmo sin pesos. La diferencia está en que en la línea 34 debemos extraer `cvw` de la lista de adyacencia, comprobar que el coste no es negativo (en otro caso, nuestro algoritmo podría producir respuestas incorrectas) y sumar `cvw`, en vez de 1, en las líneas 39 y 41.

14.4 Problema del camino mínimo con costes negativos

El algoritmo de Dijkstra necesita que los costes de las aristas sean no negativos. Esto es razonable para la mayoría de las aplicaciones, pero en ocasiones es demasiado restrictivo. Esta sección estudia brevemente el caso más general.

PROBLEMA DE LOS CAMINOS MÍNIMOS CON COSTES NEGATIVOS Y ÚNICO ORIGEN

Buscar el camino más corto (medido con su coste total) desde el vértice O al resto de vértices. Los costes de las aristas pueden ser negativos.

14.4.1 Teoría

La demostración de la corrección del algoritmo de Dijkstra necesita que los costes de las aristas, y por tanto los de los caminos, sean no negativos. De hecho, si el grafo tiene aristas de coste negativo, el algoritmo de Dijkstra no funciona correctamente. El problema se encuentra en que una vez que un vértice v ha sido procesado, puede existir un camino de vuelta a v desde otro vértice u no procesado, con coste bastante negativo. En tal caso, el camino que va desde O hasta v pasando por u es mejor que ir de O a v sin pasar por u. Si esto último ocurriera, estaríamos en dificultades: no sólo el camino a v sería incorrecto, sino que tendríamos que visitar de nuevo v, porque los vértices alcanzables desde v podrían también ser afectados.

Hay otro problema adicional por el que preocuparse. Consideremos el grafo de la Figura 14.30. El camino de V_3 a V_4 tiene coste 2. Sin embargo, existe un camino más corto siguiendo el ciclo V_3, V_4, V_1, V_3, V_4, que tiene coste -3. Y este camino no es aún el más corto, ya que podríamos dar un número arbitrario de vueltas al ciclo. Por lo que el camino más corto entre estos dos puntos está indefinido.

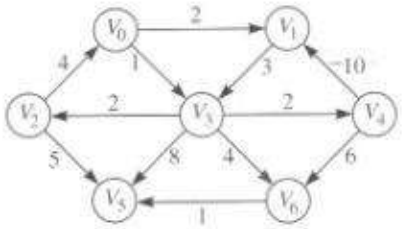


Figura 14.30 Grafo con ciclo de coste negativo.

Este problema no se restringe a nodos en el ciclo. El camino más corto de V_2 a V_5 también está indefinido porque hay una forma de entrar y salir del bucle. Este bucle se denomina *ciclo de coste negativo*; cuando existe uno en un grafo, algunos (probablemente muchos) caminos mínimos no están definidos. Las aristas de coste negativo no son malas por sí mismas; son los ciclos los que lo son. Nuestro algoritmo encontrará los caminos mínimos o hará notar la existencia de ciclos con coste negativo.

Una combinación de los algoritmos con pesos y sin pesos resolverá el problema, pero a costa de un drástico incremento potencial en el tiempo de ejecución. Como hemos sugerido anteriormente, cuando se modifica D_w, debemos (re)visitar este vértice en algún momento del futuro. En consecuencia, utilizamos una cola como hicimos en el algoritmo sin pesos, pero usando $D_v + c_{v,w}$ como medida de la distancia (como en el algoritmo de Dijkstra).

Cuando visitamos el vértice v por i-ésima vez, el valor de D_v es la longitud del camino más corto formado por, a lo sumo, i aristas. Una demostración de esto se deja como ejercicio (Ejercicio 14.9). En consecuencia, si no hay ciclos de coste negativo, un vértice puede salir de la cola como mucho $|V|$ veces y el algoritmo consume un tiempo $O(|E|\,|V|)$. Es más, si un vértice sale de la cola más de $|V|$ veces, habremos detectado un ciclo de coste negativo.

14.4.2 Implementación en Java

La implementación del algoritmo de búsqueda de caminos mínimos con pesos negativos se presenta en la Figura 14.31. Hemos hecho un pequeño cambio con respecto a la descripción del algoritmo: no añadimos un vértice a la cola cuando ya está en ella. Para hacer esto, utilizamos el campo `extra`. Cuando se añade un vértice a la cola, incrementamos `extra` (en la línea 40). Cuando se quita de la cola, lo incrementamos de nuevo (en la línea 26). Por tanto, `extra` es impar si el vértice está en la cola, y `extra/2` nos dice cuántas veces ha salido de la cola (y esto explica la pregunta de la línea 26). Cuando se cambia la distancia a algún w, pero ya está en la cola (porque `extra` es impar), no lo añadimos de nuevo. Sin embargo, le sumamos 2 para indicar que podría haber salido y entrado de la cola (esto puede acelerar algo el algoritmo en presencia de ciclos negativos). Esto se hace en las líneas 40 y 43. El resto del algoritmo utiliza código que ya se ha visto en el algoritmo sin pesos (Figura 14.24) y en el algoritmo de Dijkstra (Figura 14.29).

```
/**
 * Ejecuta el algoritmo de caminos mínimos.
 * Se permiten aristas con coste negativo.
 * Devuelve false si se detectan ciclos negativos.
 */
private boolean negativos( int nodoOrig )
{
    int v, w;
    Cola q = new ColaVec( );
    int cvw;

    limpiarDatos( );
    tabla[ nodoOrig ].dist = 0;
    q.insertar( new Integer( nodoOrig ) );
    tabla[ nodoOrig ].extra++;

    // Incrementa extra cuando el vértice v entra o sale
    // de la cola. Si el vértice esta a punto de entrar, pero ya
    // está en la cola, cuenta como si entrara y saliera
    try
    {
        while( !q.esVacia( ) )
        {
            v = ( (Integer) q.quitarPrimero( ) ).intValue( );

            if( tabla[ v ].extra++ > 2 * numVertices )
                return false; // existe un ciclo

            ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
            for( ; p.estaDentro( ); p.avanzar( ) )
            {
                w = ( (Arista) p.recuperar( ) ).dest;
                cvw = ( (Arista) p.recuperar( ) ).coste;
                if( tabla[ w ].dist > tabla[ v ].dist + cvw )
                {
                    tabla[ w ].dist = tabla[ v ].dist + cvw;
                    tabla[ w ].ant = v;

                    // Insertar sólo si no está en la cola
                    if( tabla[ w ].extra++ % 2 == 0 )
                        q.insertar( new Integer( w ) );
                    else // cuenta como una eliminación fantasma
                        tabla[ w ].extra++;
                }
            }
        }
    }
    catch( DesbordamientoInferior e ) { } // No puede ocurrir
    return true;
}
```

Figura 14.31 Algoritmo de caminos mínimos con costes negativos: se permiten aristas negativas.

14.5 Problemas de caminos en grafos acíclicos

Una clase importante de grafos son aquellos que no contienen ciclos. El problema del camino mínimo es más sencillo si el grafo no contiene ciclos. Por ejemplo, no debemos preocuparnos de la posible existencia de ciclos negativos, ya que no hay ciclos en absoluto. En consecuencia, estudiaremos el siguiente problema:

> ***PROBLEMA DE LOS CAMINOS MÍNIMOS CON PESOS DESDE UN ÚNICO NODO SOBRE GRAFOS ACÍCLICOS***
> *Encontrar el camino más corto (medido con su coste total) en un grafo acíclico desde el vértice O hasta cualquier otro vértice. No hay restricciones sobre el coste de las aristas.*

Pero antes de abordar el problema de los caminos mínimos en este caso general, estudiaremos un problema relacionado: la ordenación topológica.

14.5.1 Ordenación topológica

Un *orden topológico* ordena los vértices de un grafo dirigido acíclico de tal forma que si hay un camino de u a v, entonces v aparece después de u en la ordenación. Por ejemplo, se puede utilizar un grafo para representar los prerrequisitos entre cursos de una universidad. Una arista (v, w) indica que el curso v debe ser completado antes de que pueda afrontarse el curso w. Un orden topológico de los cursos es cualquier secuencia que no viole los prerrequisitos. Una ordenación *topológica* encuentra todos los ordenes topológicos posibles sobre un grafo acíclico dado.

Es evidente que no existirá ningún orden topológico si el grafo contiene ciclos, ya que para cualquier par de vértices v y w en el ciclo, hay un camino de v a w y de w a v. Por tanto, cualquier ordenación de v y w estaría en contradicción con alguno de los caminos. Un grafo puede tener varios ordenes topológicos, y en la mayoría de los casos, cuando busquemos uno de ellos cualquiera nos valdrá.

Un algoritmo sencillo para realizar la ordenación topológica consiste en encontrar primero un vértice v que no tenga aristas de entrada. Se imprime este vértice y se borra (lógicamente) del grafo junto con sus aristas. Tras ello, se sigue aplicando la misma estrategia al resto del grafo. Para formalizar esto, definimos como *grado de entrada* de un vértice v el número de aristas (u, v).

Calculamos los grados de entrada de todos los vértices del grafo. En la práctica, «eliminar lógicamente» significa que reducimos en 1 el grado de entrada de cada vértice adyacente a v. La Figura 14.32 muestra la aplicación del algoritmo a un grafo acíclico. El grado de entrada se calcula para cada vértice. V_2 tiene grado 0; por lo que es el primero en la ordenación. Si hubiera varios vértices con grado 0, podríamos elegir cualquiera de ellos. Cuando se eliminan del grafo V_2 y sus aristas, los grados de V_0, V_3 y V_5 se decrementan en 1. Ahora V_0 tiene grado 0, por lo que es el siguiente en la ordenación topológica, y se reducen los grados de V_1 y V_3. El algoritmo continúa, y el resto de vértices se examinan en el orden V_1, V_3, V_4, V_6 y V_5. Al reiterar, no borramos físicamente las aristas del grafo; la eliminación (virtual) de aristas sólo facilitará la visualización de cómo decrecen los grados de entrada.

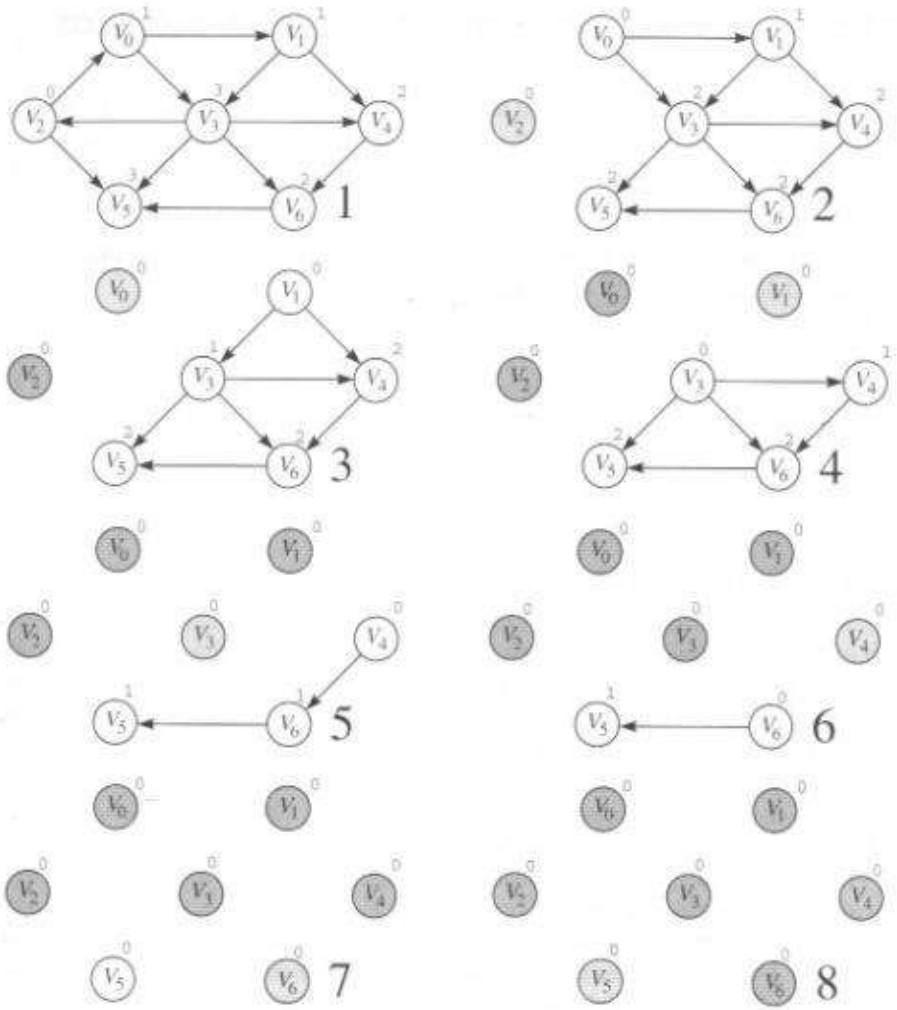


Figura 14.32 Ordenación topológica. Sigue el mismo convenio que la Figura 14.23.

Dos cuestiones importantes a considerar son la corrección y la eficiencia. Claramente, cualquier orden producido por el algoritmo es un orden topológico. La cuestión es si todo grafo acíclico tiene una ordenación topológica, y si es así, si nuestro algoritmo garantiza que encuentra una. La respuesta es afirmativa en ambos casos.

Si en cualquier momento hay vértices aún no visitados, pero ninguno tiene grado de entrada cero, es porque existe un ciclo. Para ver esto, tomemos cualquier vértice A_0. Ya que A_0 tiene una arista de entrada, podemos tomar como A_1 cualquier vértice conectado a A_0. Pero como A_1 tiene también una arista de entrada, podemos tomar como A_2 un vértice conectado a A_1. Repetimos el proceso N veces, donde N es el número de vértices sin visitar en el grafo. Entre $A_0, A_1, \ldots, A_N$ debe haber dos vértices idénticos (ya que hay N vértices, pero $N + 1$ elementos). Un recorrido hacia atrás entre estos A_i y A_j idénticos, exhibe un ciclo.

El algoritmo puede implementarse en tiempo lineal manteniendo en una cola todos los vértices no procesados de grado de entrada cero. Inicialmente, todos los vértices de grado de entrada cero se colocan en la cola. Para encontrar el siguiente vértice en el orden topológico, simplemente tomamos el primero de la cola, eliminándolo de ella. Cuando se reduce a cero el grado de un vértice, se introduce en la cola. Si la cola se queda vacía antes de haber procesado todos los vértices, es porque el grafo contiene un ciclo. El tiempo de ejecución es lineal, por la misma razón argüida en el algoritmo de caminos mínimos sin pesos.

14.5.2 Teoría del algoritmo de caminos mínimos con un grafo acíclico

Una aplicación importante de la ordenación topológica es su uso para resolver el problema de los caminos mínimos sobre grafos acíclicos. La idea es la siguiente: ir visitando los vértices según un orden topológico.

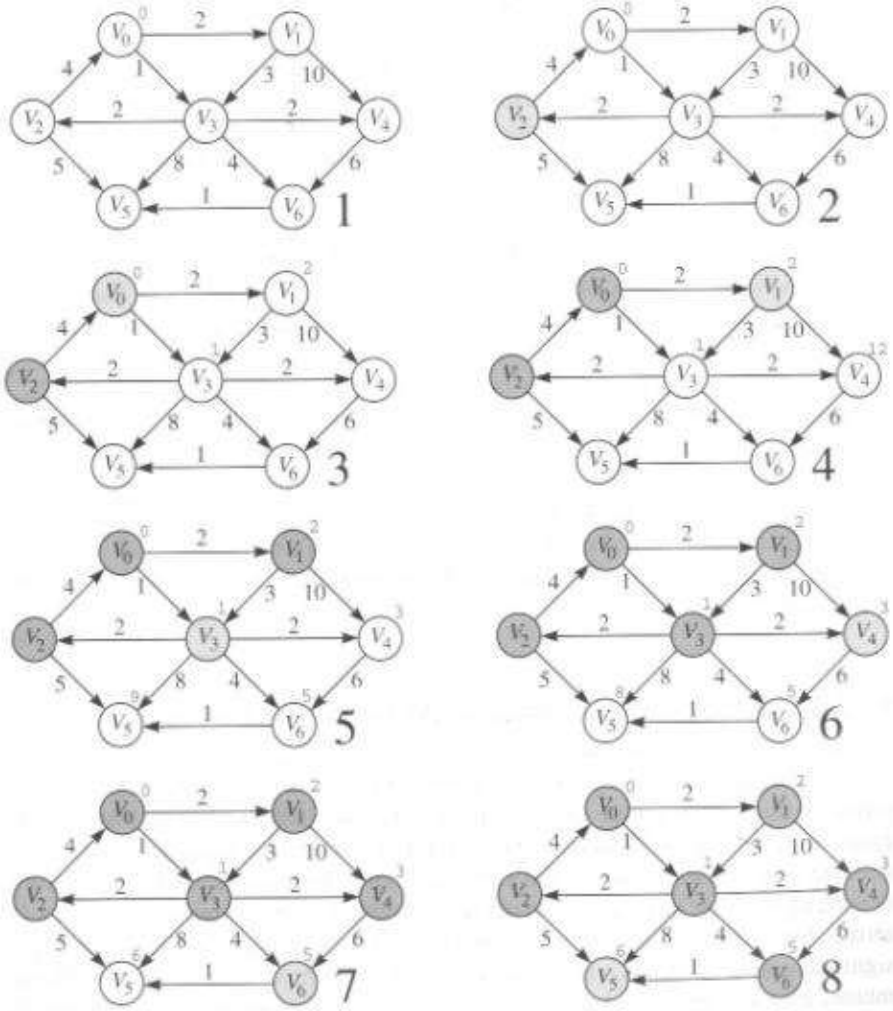


Figura 14.33 Pasos del algoritmo de caminos mínimos sobre un grafo acíclico. Sigue el mismo convenio que la Figura 14.23.

Esta idea funciona porque cuando visitamos el vértice v, estamos seguros de que D_v no será reducido nunca más, ya que por definición de orden topológico, no hay aristas que lleguen a v desde nodos no visitados. La Figura 14.33 muestra los estados del algoritmo de caminos mínimos, utilizando una ordenación topológica para guiar la visita de los vértices. Observe que la secuencia de vértices no es la misma que en el algoritmo de Dijkstra. Observe también que los vértices visitados antes de llegar al vértice origen, son inalcanzables desde el origen, por lo que no tienen influencia en las distancias hasta ningún vértice.

Ya que no necesitamos una cola de prioridad, y sólo necesitamos incorporar la ordenación topológica al cálculo del camino mínimo, tenemos que el algoritmo se ejecutará en tiempo lineal y funcionará correctamente en presencia de aristas negativas.

14.5.3 Implementación en Java

La implementación del algoritmo de caminos mínimos para grafos acíclicos se muestra en la Figura 14.34. Utilizamos una cola para realizar una ordenación topológica, y mantenemos la información sobre el grado de entrada en el campo `extra`. Las líneas de la 13 a la 19 calculan los grados de entrada, y en las líneas de la 21 a la 23, colocamos en la cola cada vértice con grado 0.

Entonces, en la línea 27 se retira repetidamente un vértice de la cola. Observe que si la cola se vacía, el bucle `for` terminará por la comprobación de la línea 25. Si el bucle termina por la existencia de un ciclo, será reflejado en la línea 50. En otro caso, el bucle en la línea 30 recorre la lista de adyacencia, obteniéndose el valor de w en la línea 32. Inmediatamente se decrementa el grado de entrada de w en la línea 33, y, si ha llegado a cero, se coloca este vértice en la cola (en la línea 34).

Recuerde que si el vértice actual v aparece antes de O en la ordenación topológica, entonces v es inalcanzable desde O. En consecuencia, seguirá cumpliéndose $D_v = \infty$ y por tanto, no puede aparecer en ningún camino a los vértices w alcanzables. Realizamos la preceptiva comprobación en la línea 36, y si se da el caso, no se hace ningún calculo de distancias. En otro caso, en las líneas de la 39 a la 44, utilizamos el mismo cálculo que en el algoritmo de Dijkstra para actualizar D_w cuando ello es oportuno.

14.5.4 Una aplicación: análisis de caminos críticos

Un uso importante de los grafos acíclicos es el *análisis de caminos críticos*, una forma de análisis utilizada para planificar las tareas asociadas a un proyecto. El grafo de la Figura 14.35 sirve como ejemplo. Cada vértice representa una actividad que debe realizarse, junto con el tiempo que se tarda en realizarla. El grafo se denomina grafo de actividades, en el cual los vértices representan actividades y las aristas representan relaciones de precedencia. La existencia de una arista (v, w) significa que la actividad v debe realizarse antes de empezar la tarea w. Evidentemente, esto implica que el grafo debe ser acíclico. Asumiremos que las actividades que no dependen (directa o indirectamente) entre sí pueden ser realizadas en paralelo por diferentes servidores.

```
1  // Algoritmo lineal, funciona sólo con grafos acíclicos
2  private boolean aciclico( int nodoOrig )
3  {
4      int v, w, iteraciones=0;
5      Cola q = new ColaVec( );
6
7      limpiarDatos( );
8      tabla[ nodoOrig ].dist = 0;
9
10     try
11     {
12         // Calcula los grados de entrada
13         for( v = 0; v < numVertices; v++ )
14         {
15             ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
16             for( ; p.estaDentro( ); p.avanzar( ) )
17                 tabla[ ( (Arista) p.recuperar( ) ).dest ].
18                                                     extra++;
19         }
20         // Insertar en la cola vértices de grado cero
21         for( v = 0; v < numVertices; v++ )
22             if( tabla[ v ].extra == 0 )
23                 q.insertar( new Integer( v ) );
24
25         for( iteraciones = 0; !q.esVacia( ); iteraciones++ )
26         {
27             v=( (Integer) q.quitarPrimero( ) ).intValue( );
28
29             ListaIter p = new ListaEnlazadaIter( tabla[ v ].ady );
30             for( ; p.estaDentro( ); p.avanzar( ) )
31             {
32                 w = ( (Arista) p.recuperar( ) ).dest;
33                 if( --tabla[ w ].extra == 0 )
34                     q.insertar( new Integer( w ) );
35
36                 if( tabla[ v ].dist == INFINITO )
37                     continue;
38
39                 int cvw = ( (Arista) p.recuperar( ) ).coste;
40                 if( tabla[ w ].dist > tabla[ v ].dist + cvw )
41                 {
42                     tabla[ w ].dist = tabla[ v ].dist + cvw;
43                     tabla[ w ].ant = v;
44                 }
45             }
46         }
47     }
48     catch( DesbordamientoInferior e ) { } // No puede ocurrir
49
50     return iteraciones == numVertices;
51 }
```

Figura 14.34 Algoritmo de caminos mínimos para grafos acíclicos.

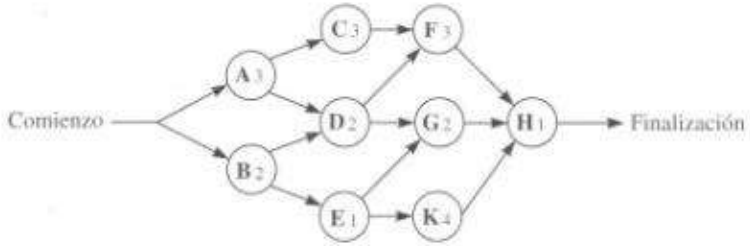


Figura 14.35 Grafo de actividades.

Este tipo de grafo puede utilizarse para modelar el desarrollo de proyectos, en el cual son de interés varias cuestiones importantes. En primer lugar, ¿cuál es el menor tiempo de terminación del proyecto? El grafo muestra que se necesitan diez unidades a través del camino *A*, *C*, *F*, *H*. Otra cuestión importante es ¿qué actividades se pueden retrasar, y por cuánto tiempo, sin afectar al tiempo mínimo de terminación? Por ejemplo, retrasar cualquiera de las tareas *A*, *C*, *F* o *H* haría que el tiempo de terminación rebasara las diez unidades. Por otro lado, la actividad *B* es menos crítica y puede retrasarse hasta dos unidades de tiempo sin afectar al tiempo de terminación final.

Para realizar estos cálculos, convertimos el grafo de actividades en un *grafo de eventos*, en el cual cada evento corresponde a la terminación de una actividad y de todas sus actividades dependientes. Las actividades correspondientes a los eventos alcanzables desde un nodo v en el grafo de eventos no pueden comenzar a ejecutarse hasta después de haber acabado la actividad correspondiente al evento v. Este grafo puede construirse automáticamente o a mano (a partir del grafo de actividades). Quizás sea necesario insertar aristas o vértices falsos para evitar introducir dependencias falsas (o faltas erróneas de dependencias). El grafo de eventos correspondiente al grafo de actividades de la Figura 14.35 se muestra en la Figura 14.36. Obsérvese la presencia de tales aristas y vértices falsos.

Para encontrar el tiempo de terminación mínimo del proyecto, sólo necesitamos calcular la longitud del camino *más largo* desde el primer evento al último evento. Para grafos generales, el problema del camino máximo generalmente no tiene sentido, por la posibilidad de *ciclos de coste positivo*, que son los equivalentes a los ciclos de costes negativos en los problemas de los caminos mínimos. Si hay ciclos de coste positivo, podemos preguntar por el camino simple más largo. Sin embargo, no se conoce ninguna solución satisfactoria para este problema. Afortunadamente, el grafo de eventos es acíclico; por lo que no tenemos que preocuparnos de la presencia de ciclos. Es fácil adaptar el algoritmo de caminos mínimos para

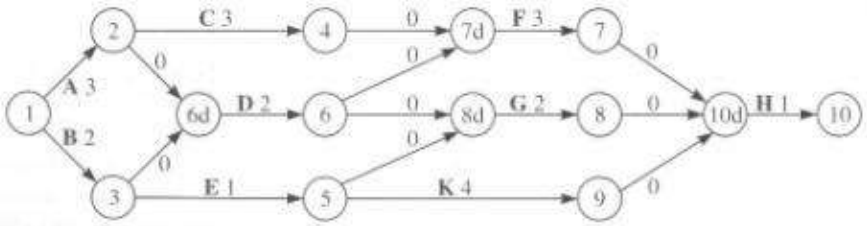


Figura 14.36 Grafo de eventos.

calcular el tiempo de terminación menor para todos los nodos del grafo. Si denotamos el tiempo de terminación mínimo para el nodo i por EC_i, entonces las reglas aplicables son

$$EC_1 = 0 \quad \text{y} \quad EC_w = \text{Max}_{(v,w) \in E}(EC_v + c_{v,w}).$$

La Figura 14.37 muestra el tiempo de terminación mínimo para cada vértice en nuestro ejemplo de grafo de eventos. También podemos calcular el tiempo máximo, LC_i, en el que cada evento puede terminar sin afectar el tiempo de terminación final. Las fórmulas en este caso son

$$LC_N = EC_N \quad \text{y} \quad LC_v = \text{Min}_{(v,w) \in E}(LC_w - c_{v,w}).$$

Estos valores pueden calcularse en tiempo lineal manteniendo, para cada vértice, una lista de los vértices adyacentes y precedentes. Los tiempos de terminación mínimos se calculan siguiendo un orden topológico, y el tiempo de terminación máximo se calcula siguiendo el orden topológico inverso. Los tiempos de terminación máximos se muestran en la Figura 14.38.

El *tiempo de espera* para cada arista en el grafo de eventos es la cantidad de tiempo que puede retrasarse la terminación de la actividad correspondiente sin retrasar el tiempo de terminación total. Es fácil ver que

$$Espera_{(v,w)} = LC_w - EC_v - c_{v,w}.$$

La Figura 14.39 muestra la espera para cada actividad en el grafo de eventos. Para cada nodo, el número de encima es el mínimo tiempo de terminación y el número de abajo es el tiempo de terminación máximo.

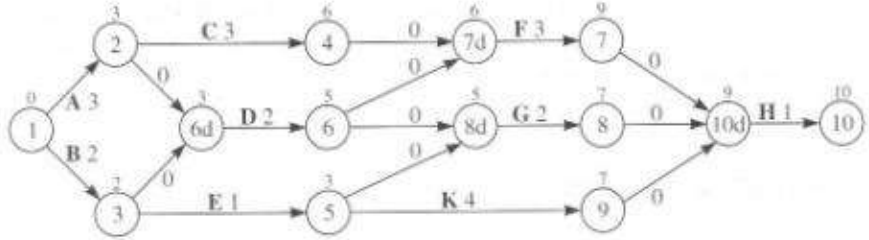


Figura 14.37 Tiempos de terminación mínimos.

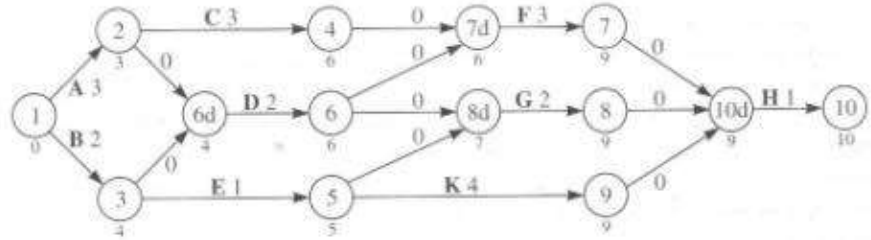


Figura 14.38 Tiempos de terminación máximos.

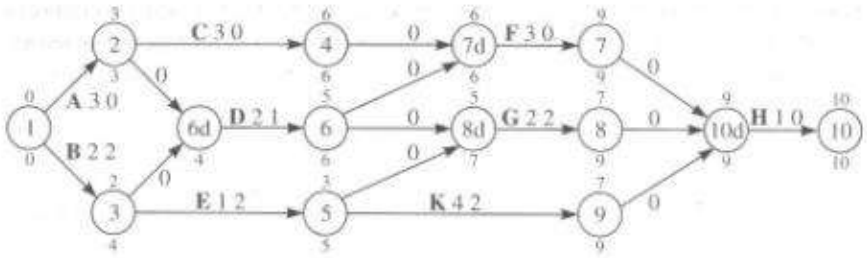


Figura 14.39 Tiempos de terminación mínimos, máximos y de espera (información adicional en cada arista).

Algunas actividades tienen tiempo de espera cero. Éstas son actividades críticas que deben necesariamente acabar a su hora. Hay al menos un camino formado completamente con aristas de tiempo de espera cero; tal camino se denomina *camino crítico*.

Resumen

Este capítulo ha mostrado cómo se pueden utilizar grafos para modelar muchos problemas de la vida real y en particular cómo calcular el camino mínimo bajo gran variedad de circunstancias. Muchos de los grafos que aparecen en la práctica son típicamente muy dispersos, por lo que es importante elegir estructuras de datos apropiadas para implementarlos.

Para grafos sin pesos, los caminos mínimos pueden calcularse en tiempo lineal utilizando una búsqueda en anchura. Para grafos con pesos positivos, se necesita algo más de tiempo, utilizando el algoritmo de Dijkstra y una cola de prioridad eficiente. Para grafos con pesos negativos, el problema se hace aún más complicado. Finalmente, para grafos acíclicos, el tiempo de ejecución vuelve a ser lineal, con la ayuda de una ordenación topológica.

Elementos del juego

algoritmo de Dijkstra Un algoritmo que resuelve el problema de los caminos mínimos con peso.

análisis de caminos críticos Una forma de análisis utilizada en la planificación de tareas asociada a un proyecto.

búsqueda en anchura Un procedimiento de búsqueda que procesa los vértices por niveles. Aquéllos más cercanos al origen se evalúan primero. Nos movemos de vértice a vértice actualizando las distancias de los vértices adyacentes.

camino Secuencia de vértices conectados por aristas.

camino simple Un camino cuyos vértices son todos distintos, excepto el primero y el último que pueden ser iguales.

ciclo En un grafo dirigido, un camino que empieza y termina en el mismo vértice y que contiene al menos una arista.

ciclo de coste negativo Un ciclo cuyo coste es menor que cero. Hace que la mayoría de los caminos mínimos, si no todos, estén indefinidos porque podemos dar vueltas arbitrariamente por el ciclo y obtener caminos tan pequeños como queramos.

ciclo de coste positivo En el problema del camino más largo, el equivalente a los ciclos de coste negativo para el problema del camino más corto.

coste (peso) de una arista La tercera componente de una arista, que mide el coste de atravesarla.

grado de entrada Número de aristas entrantes a un vértice.

grafo Está formado por un conjunto de vértices y un conjunto de aristas que conectan vértices.

grafo de actividades Un grafo con actividades como vértices y relaciones de precedencia como aristas.

grafo de eventos Un grafo que tiene como vértices eventos que corresponden a la terminación de una actividad y de todas sus actividades dependientes. Las aristas muestran qué actividad debe completarse para avanzar de un vértice al siguiente. El tiempo de terminación mínimo es la longitud del camino más largo en dicho grafo.

grafo dirigido Un grafo cuyas aristas son pares ordenados de vértices.

grafo dirigido acíclico (GDA) Un tipo de grafo dirigido que no contiene ciclos.

grafos densos y dispersos Un grafo denso tiene un gran número de aristas (generalmente cuadrático). Los grafos típicos no son densos, sino dispersos.

lista de adyacencia Un vector de listas utilizado para representar un grafo. Utiliza un espacio lineal.

longitud de un camino El número de aristas de un camino.

longitud de un camino con peso La suma del coste de las aristas del camino.

longitud de un camino sin peso El número de aristas de un camino.

matriz de adyacencia Una representación de un grafo mediante una matriz, que utiliza un espacio cuadrático.

origen único Un algoritmo que calcula los caminos mínimos desde un punto de origen a todos los vértices del grafo.

tiempo de espera Cantidad de tiempo que puede retrasarse una actividad, sin afectar al tiempo total de terminación.

vértice adyacente w es adyacente a v si hay una arista de v a w.

Errores comunes

1. Un error común es no comprobar si el grafo de entrada satisface las condiciones requeridas por el algoritmo utilizado (por ejemplo tener todas sus aristas positivas o ser acíclico).
2. En la clase `Camino`, la función de comparación compara solamente el campo `coste`. Si el campo `dest` se utiliza para la comparación, el algoritmo puede parecer que funciona para grafos pequeños, pero para grafos grandes es sin duda incorrecto y dará respuestas subóptimas. Sin embargo, nunca producirá un camino que no exista, por lo que este error es difícil de detectar.
3. El algoritmo del camino mínimo sobre grafos con costes negativos debe contener una comprobación para detectar ciclos negativos; en otro caso, ciclará indefinidamente.

En Internet

Todos los algoritmos de este capítulo se encuentran disponibles en la red, en un único fichero, en el directorio **Chapter14**. El nombre del fichero es el siguiente:

Graph.java Traducido por `Grafo.java`, contiene la clase `Grafo` con la rutina `main`.

Ejercicios

Cuestiones breves

14.1. Encuentre el camino más corto sin pesos desde V_3 al resto de vértices en el grafo de la Figura 14.1.

14.2. Encuentre el camino más corto con pesos desde V_2 al resto de vértices en el grafo de la Figura 14.1.

Problemas teóricos

14.3. Muestre cómo evitar el coste cuadrático de la inicialización de las matrices de adyacencia, manteniendo un tiempo de acceso constante a cada arista.

14.4. Explique cómo modificar el algoritmo de caminos mínimos sin pesos, de tal forma que cuando haya más de un camino mínimo (en términos del número de aristas) el empate se deshaga en favor del de menor peso total.

14.5. Explique cómo modificar el algoritmo de Dijkstra para que cuente el número de caminos mínimos diferentes desde v hasta w.

14.6. Explique cómo modificar el algoritmo de Dijkstra de tal forma que cuando haya más de un camino mínimo desde v hasta w, se elija el camino con menor número de aristas.

14.7. Dé un ejemplo de grafo con aristas negativas, pero sin ciclos negativos, en el que el algoritmo de Dijkstra dé una respuesta errónea.

14.8. Considere el siguiente algoritmo para resolver el problema del camino mínimo con aristas negativas: sume una constante c a cada coste de arista, eliminando los costes negativos; calcule los caminos mínimos en este grafo; traslade el resultado al grafo original. ¿Qué es incorrecto en este algoritmo?

14.9. Demuestre la corrección del algoritmo del camino mínimo con aristas negativas. Para hacerlo, muestre que cuando visitamos el vértice v por i-ésima vez, el valor de D_v es la longitud del camino más corto formado, a lo sumo, por i aristas.

14.10. Dé un algoritmo de tiempo lineal para buscar el camino de mayor peso en un grafo acíclico. ¿Se puede extender su algoritmo a grafos con ciclos?

14.11. Muestre que si los costes de las aristas son 0 o 1 exclusivamente, el algoritmo de Dijkstra puede implementarse en tiempo lineal utilizando una cola doble (Sección 15.4).

Problemas prácticos

14.12. Este capítulo reivindica que en la implementación de los algoritmos sobre grafos que trabajan con grandes datos de entrada, la elección de estructuras de datos adecuadas es crucial para asegurar un rendimiento razonable. Para cada uno de los ejemplos siguientes, en los que se utiliza una estructura de datos pobre o un algoritmo malo, proporcione un análisis en notación O del resultado y compárelo con el rendimiento de los algoritmos y estructuras de datos utilizados en el texto. Incorpore un sólo cambio cada vez. Debería ejecutar sus tests con grafos razonablemente grandes y aleatoriamente dispersos. Haga lo siguiente:

a) Cuando se lee una arista, comprobe si ya está en el grafo.
b) Implemente el diccionario utilizando un recorrido secuencial de la tabla de vértices.
c) Implemente la cola utilizando el algoritmo del Ejercicio 6.13 (esto debería afectar al algoritmo de caminos mínimos sin pesos).
d) En el algoritmo de caminos mínimos sin pesos, implemente la búsqueda del vértice de menor coste con un recorrido secuencial de la tabla de vértices.
e) Implemente la cola de prioridad utilizando el algoritmo del Ejercicio 6.16 (esto debería afectar al algoritmo de caminos mínimos con pesos).
f) Implemente la cola de prioridad utilizando el algoritmo del Ejercicio 6.17 (esto debería afectar al algoritmo de caminos mínimos con pesos).
g) En el algoritmo de caminos mínimos con pesos, implemente la búsqueda del vértice de menor coste como un recorrido secuencial de la tabla de vértices.
h) En el algoritmo de caminos mínimos sobre grafos acíclicos, implemente la búsqueda del vértice con grado de entrada cero con un recorrido secuencial de la tabla de vértices.
i) Implemente cada uno de los algoritmos utilizando una matriz de adyacencia en vez de listas de adyacencia.

Prácticas de programación

14.13. Un grafo dirigido es fuertemente conexo si hay un camino desde cada vértice a cualquier otro. Haga lo siguiente:

a) Elija cualquier vértice O. Muestre que si el grafo es fuertemente conexo, un algoritmo de caminos mínimos declarará que todos los vértices son alcanzables desde O.
b) Muestre que si un grafo es fuertemente conexo, si la dirección de todas sus aristas se invierte y se ejecuta un algoritmo de caminos mínimos desde O, todos los vértices resultarán alcanzables.
c) Muestre que las comprobaciones en (a) y (b) son suficientes para decidir si un grafo es fuertemente conexo (es decir, todo grafo que pase ambos tests será fuertemente conexo).

d) Escriba un programa que compruebe si un grafo es fuertemente conexo. ¿Cuál es su tiempo de ejecución?

Explique cómo puede resolverse cada uno de los siguientes problemas aplicando un algoritmo de caminos mínimos. Diseñe un mecanismo para representar la entrada y escriba un programa que resuelva el problema.

14.14. La entrada es una lista de puntuaciones obtenidas en una liga de un deporte (donde no hay empates). Si todos los equipos han ganado al menos una vez y han perdido al menos una vez, podemos generalmente «probar», por un argumento transitivo tonto, que cualquier equipo es mejor que cualquier otro. Por ejemplo, suponga que en una liga de seis equipos donde todo el mundo juega 3 partidos, tenemos los siguientes resultados: *A* vence a *B* y a *C*; *B* vence a *C* y a *F*; *C* vence a *D*; *D* vence a *E*; *E* vence a *A*; *F* vence a *D* y a *E*. Entonces podemos probar que *A* es mejor que *F* porque *A* venció a *B* que a su vez venció a *F*. De igual forma, podemos probar que *F* es mejor que *A* porque *F* venció a *E* y *E* venció a *A*. Dada una lista de resultados y dos equipos *X* e *Y*, busque una demostración (si existe) de que *X* es mejor que *Y* o indique que tal demostración no existe.

14.15. Una palabra puede transformarse en otra sustituyendo un carácter. Suponga que tenemos un diccionario con palabras de 4 letras. Dé un algoritmo que determine si una palabra *A* puede transformarse en otra palabra *B*, mediante una serie de sustituciones de un carácter, y si es así, que muestre la correspondiente serie de palabras. Por ejemplo, `mesa` se convierte en `poto` con la secuencia `mesa`, `meta`, `seta`, `seto`, `peto`, `poto`.

14.16. La entrada es una colección de divisas y sus tasas de cambio. ¿Hay una secuencia de intercambios que produzca dinero mágicamente? Por ejemplo, si las divisas son *X*, *Y* y *Z*, y las tasas de cambio son 1 *X* es igual a 2 *Y*, 1 *Y* es igual a 2 *Z* y 1 *X* es igual a 3 *Z*, entonces 300 *Z* comprarán 100 *X*, que a su vez comprarán 200 *Y*, que a su vez comprarán 400 *Z*. Por lo que se obtendría un beneficio mágico del 33 por ciento.

14.17. Un estudiante necesita aprobar cierto número de cursos para graduarse, y estos cursos tienen prerrequisitos que deben cumplirse. Supongamos que todos los cursos se ofrecen todos los semestres y que un estudiante puede matricularse en un número indeterminado de cursos. Dada una lista de cursos y sus prerrequisitos, calcule una planificación que requiera el mínimo número de semestres.

14.18. El objetivo del *juego de Kevin Bacon* consiste en conectar al actor de una película con Kevin Bacon por papeles en una misma película. El mínimo número de conexiones es el *número de Bacon* de ese actor. Por ejemplo, Tom Hanks tiene un número de Bacon 1, porque trabajó en *Apolo* 13 con Kevin Bacon. Sally Field tiene un número de Bacon 2, porque trabajó en *Forrest Gump* con Tom Hanks, que trabajó en *Apolo* 13 con Kevin Bacon. Casi todo actor conocido tiene un número de Bacon 1 o 2. Supongamos que tiene una lista extensa de actores y películas, con papeles, y haga lo siguiente:

a) Explique cómo encontrar el número de Bacon de un actor.
b) Explique cómo encontrar el actor con mayor número de Bacon.
c) Explique cómo buscar el menor número de conexiones entre dos actores arbitrarios.

Bibliografía

Las listas de adyacencia para representar grafos fueron utilizadas por primera vez en [3]. El algoritmo de Dijkstra para el problema de los caminos mínimos fue descrito originalmente en [2]. El algoritmo para aristas con coste negativo se ha tomado de [1]. Un test más eficiente para la terminación se describe en [6], donde también se muestra cómo las estructuras de datos juegan un papel importante en una amplia variedad de algoritmos sobre grafos. La ordenación topológica está tomada de [4]. Una gran cantidad de aplicaciones en la vida real de algoritmos sobre grafos puede encontrarse en [5], junto con referencias a otros textos.

1. R. E. Bellman, «On a Routing Problem», *Quarterly of Applied Mathematics* **16** (1958), 87-90.
2. E. W. Dijkstra, «A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik* **1** (1959), 269-271.
3. J. E. Hopcroft y R. E. Tarjan, «Algorithm 447: Efficient Algorithms for Graph Manipulation», *Communications of the ACM* **16** (1973), 372-378.
4. A. B. Kahn, «Topological Sorting of Large Networks», *Communications of the ACM* **5** (1962), 558-562.
5. D. E. Knuth, *The Stanford GraphBase*, Addison-Wesley, Reading, Mass. (1993).
6. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Penn. (1985).

Parte IV

Implementaciones

CAPÍTULO

15 Pilas y colas

Este capítulo discute las implementaciones de las estructuras de datos de pilas y colas. Recuerde del Capítulo 6, que sus operaciones básicas deberían realizarse en tiempo constante. Para las pilas y las colas existen dos técnicas básicas de conseguir operaciones con coste constante. La primera consiste en almacenar los elementos de forma contigua empleando un vector, mientras que la segunda consiste en almacenarlos de forma no contigua en una lista enlazada. En este capítulo se presentan las implementaciones de las dos estructuras de datos empleando ambas técnicas. El código implementa las interfaces presentadas en el Capítulo 6.

En este capítulo veremos:

- Una implementación de las pilas basada en un vector.
- Una implementación de las colas basada en un vector.
- Una implementación de las pilas empleando listas enlazadas.
- Una implementación de las colas empleando listas enlazadas.
- La utilización de la herencia para obtener una nueva estructura de datos, llamada *cola doble*.

15.1 Implementación dinámica de vectores

En esta sección implementaremos las pilas y colas empleando un vector. Los algoritmos resultantes son extremadamente eficientes y sencillos de codificar.

15.1.1 Pilas

Como muestra la Figura 15.1, una pila puede implementarse usando un vector y un entero. El entero `cdp` (*cima de pila*) es el índice del vector correspondiente al elemento situado en la cima de la pila. Así, cuando `cdp` es -1, la pila está vacía. Para `apilar`, se incrementa `cdp` y se inserta el nuevo elemento en la posición `cdp` del vector. El acceso a la cima de la pila resulta trivial, y la operación de `desapilar` se realiza decrementando `cdp`. En la Figura 15.1 mostramos la pila resultante tras las tres operaciones siguientes: `apilar(a)`, `apilar(b)` y `desapilar`, partiendo de una pila vacía.

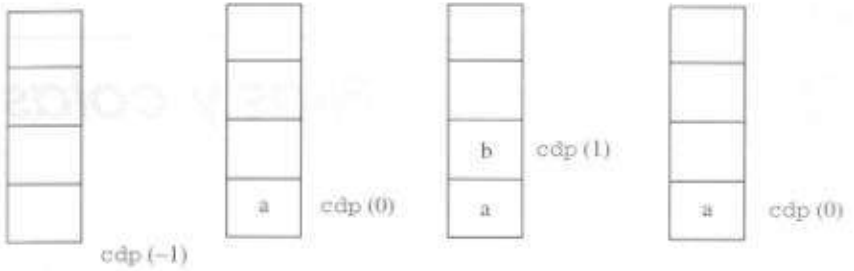


Figura 15.1 Comportamiento de las rutinas de las pilas: pila vacía; `apilar(a)`; `apilar(b)`; `desapilar`.

La Figura 15.2 muestra el esqueleto de la clase `PilaVec` basada en un vector. En ella se especifican dos atributos. `elVector`, que se expande siempre que es necesario, almacena los elementos de la pila. `cimaDePila` es el índice de la cima actual de la pila. En una pila vacía dicho índice es -1. El constructor correspondiente se muestra en la Figura 15.3.

Los métodos públicos se enumeran en las líneas 27 a 38 del esqueleto. La implementación de muchas de estas rutinas es simple. El método `apilar` se muestra en la Figura 15.4. Cuando no es necesario doblar el tamaño del vector, lo que es un ejercicio estándar ya estudiado, su código se reduce al que aparece en la línea 9. Recuérdese, de la Sección 1.4.3, que el uso prefijo del operador `++` significa que `cimaDePila` se incrementa primero, y su nuevo valor se emplea para indexar `elVector`. Las rutinas restantes son igualmente breves, tal y como se muestra en las Figuras 15.5 y 15.6. El operador postfijo `--` empleado en la Figura 15.6 significa que, aunque `cimaDePila` se decrementa, es su valor original el que se emplea para indexar `elVector`.

Cuando el vector no se duplica, cada operación se realiza en tiempo constante. Por su parte, la complejidad de una operación `apilar` que incluya la duplicación del vector es $O(N)$. Si fuese necesaria la duplicación con mucha frecuencia, deberíamos preocuparnos; sin embargo, esto claramente no es así, ya que la duplicación de un vector de N elementos debe estar precedida por, al menos, $N/2$ operaciones `apilar` que no necesitan de la duplicación del vector. Como consecuencia, podemos repartir el coste $O(N)$ de la duplicación entre las operaciones de `apilar`, aumentando su coste sólo en una pequeña constante. Esta técnica recibe el nombre de *amortización*.

Un ejemplo de amortización en la vida real es el pago de impuestos. En lugar de pagar la cuantía total el 15 de abril, el gobierno recibe el dinero a través de las retenciones. La cuantía total es la misma, la única diferencia es *cuándo* se pagan los impuestos. Lo mismo es cierto para el tiempo empleado en las operaciones `apilar`. Podemos considerar el coste de la duplicación en el momento en el que se produce, o suponerlo repartido entre las operaciones `apilar` previas, igualando su coste. Una cota amortizada indica que valoramos cada operación de una secuencia compartiendo de forma justa el coste total. En nuestro ejemplo, usando este concepto obtenemos que el coste de la duplicación del vector no es en absoluto excesivo, conservándose de hecho el coste (amortizado) constante.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase PilaVec
6  //
7  // CONSTRUCCIÓN: sin ninguna inicialización
8  //
9  // ********************OPERACIONES PÚBLICAS*********************
10 // void apilar( x )         --> Inserta x
11 // void desapilar( )         --> Elimina el último elemento insertado
12 // Object cima( )            --> Devuelve el último elemento insertado
13 // Object cimaYDesapilar( )--> Devuelve, y elimina, el elemento más r
14 // boolean esVacia( )        --> Devuelve true si pila vacía; si no, fal
15 // void vaciar( )            --> Elimina todos los elementos
16 // ********************************ERRORES*********************************
17 // desapilar, cima y cimaYDesapilar sobre una pila vacía
18
19 /**
20  * Implementación de las pilas basada en un vector.
21  */
22 public class PilaVec implements Pila
23 {
24     public PilaVec( )
25       { /* Figura 15.3 */ }
26
27     public boolean esVacia( )
28       { return cimaDePila == -1; }
29     public void vaciar( )
30       { cimaDePila = -1; }
31     public void apilar( Object x )
32       { /* Figura 15.4 */ }
33     public Object cima( ) throws DesbordamientoInferior
34       { /* Figura 15.5 */ }
35     public void desapilar( ) throws DesbordamientoInferior
36       { /* Figura 15.5 */ }
37     public Object cimaYDesapilar( ) throwa DesbordamientoInferior
38       { /* Figura 15.6 */ }
39
40     private Object [ ] elVector;
41     private int cimaDePila;
42
43     static final int CAPACIDAD_POR_DEFECTO = 10;
44
45     private void duplicarVector( )
46       { /* Implementación usual, no se muestra */ }
47 }
```

Figura 15.2 Esqueleto de la clase `PilaVec`.

```
1 /**
2  * Construye la pila.
3  */
4 public PilaVec( )
5 {
6     elVector = new Object[ CAPACIDAD_POR_DEFECTO ];
7     cimaDePila = -1;
8 }
```

Figura 15.3 Constructor sin parámetros de las `PilaVec` basadas en un vector.

```
1  /**
2   * Inserta un nuevo elemento en la pila.
3   * @param x el elemento a insertar.
4   */
5  public void apilar( Object x )
6  {
7      if( cimaDePila + 1 == elVector.length )
8          duplicarVector( );
9      elVector[ ++cimaDePila ] = x;
10 }
```

Figura 15.4 Método `apilar` de las `Pilas` basadas en un vector.

```
1  /**
2   * Devuelve el último elemento insertado en la pila.
3   * @return el último elemento insertado en la pila.
4   * @exception DesbordamientoInferior cuando la pila esté vacía.
5   */
6  public Object cima( ) throws DesbordamientoInferior
7  {
8      if( esVacia( ) )
9          throw new DesbordamientoInferior( "Cima" );
10     return elVector[ cimaDePila ];
11 }
12
13 /**
14  * Elimina el último elemento insertado en la pila.
15  * @exception DesbordamientoInferior cuando la pila esté vacía.
16  */
17 public void desapilar( ) throws DesbordamientoInferior
18 {
19     if( esVacia( ) )
20         throw new DesbordamientoInferior( "Desapilar" );
21     cimaDePila--;
22 }
```

Figura 15.5 Métodos `cima` y `desapilar` de las `Pilas` basadas en un vector.

```
1  /**
2   * Devuelve y elimina el último elemento insertado
3   * en la pila.
4   * @return el último elemento insertado en la pila
5   * @exception DesbordamientoInferior si la pila está vacía.
6   */
7  public Object cimaYDesapilar( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "CimaYDesapilar" );
11     return elVector[ cimaDePila-- ];
12 }
```

Figura 15.6 Método `cimaYDesapilar` de las `Pilas` basadas en un vector.

15.1.2 Colas

La forma más sencilla de implementar una cola consiste en almacenar sus elementos en un vector, colocando el elemento en cabeza en la primera posición del mismo (es decir, en el índice 0). Si `fin` representa la posición del último elemento de la cola, entonces para insertar un elemento no tenemos más que incrementar `fin`, insertando el elemento en esa posición. El problema es que la operación `quitarPrimero` es muy costosa. Ello es debido a la exigencia de posicionar los elementos de la cola desde el principio del vector, con lo que forzamos a la rutina `quitarPrimero` a desplazar una posición todos los elementos del vector, una vez eliminado el primero.

En la Figura 15.7 se muestra cómo se resuelve este problema, bastaría con incrementar `cabeza` cuando se realiza una operación `quitarPrimero`, en lugar de desplazar todos los elementos. Entonces, cuando la cola tiene un único elemento, `fin` y `cabeza` representan la posición en el vector de dicho elemento. Consistentemente, en una cola vacía, `fin` debe inicializarse a `cabeza-1`.

Esta implementación hace que tanto `insertar` como `quitarPrimero` se ejecuten en tiempo constante. El principal problema que presenta esta codificación se aprecia en la primera línea de la Figura 15.8. Después de tres ejecuciones más de `quitarPrimero`, no podríamos añadir más elementos, aunque la cola (más exactamente, el vector que la sustenta) no esté realmente llena. La duplicación del vec-

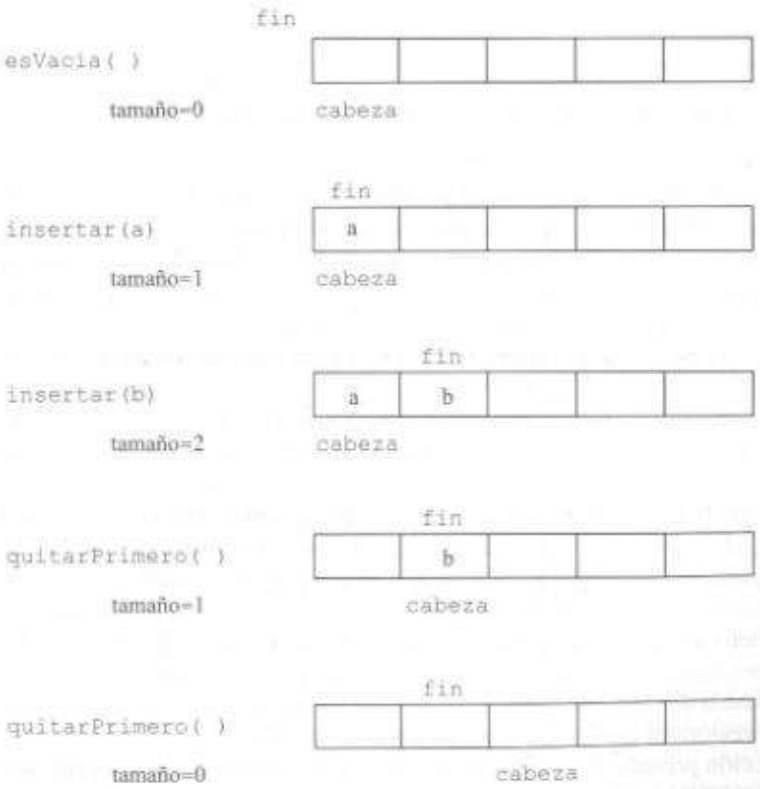


Figura 15.7 Implementación de las colas basada en un vector.

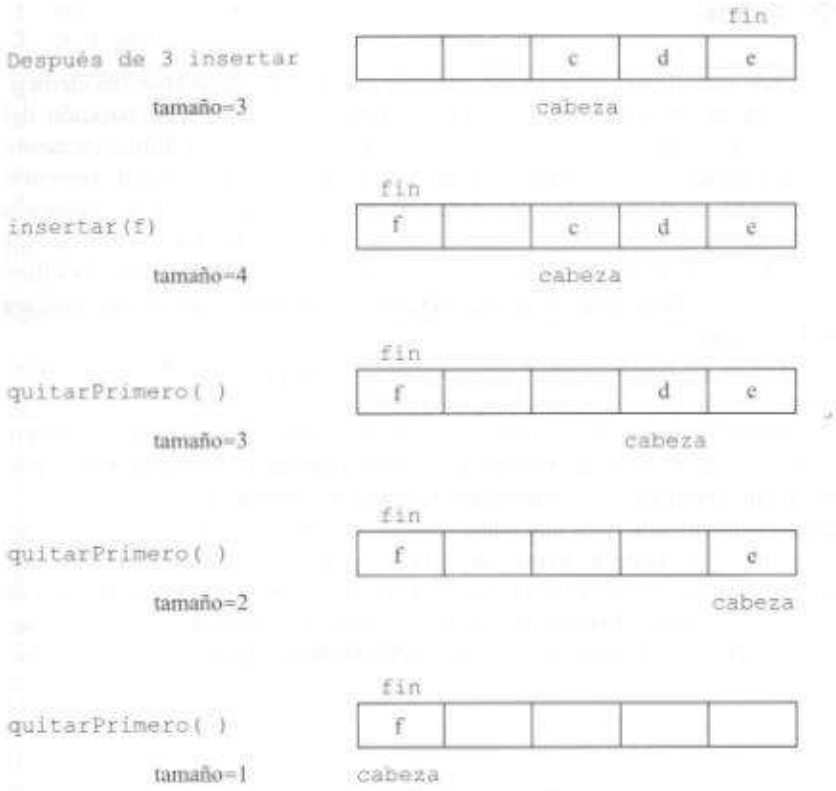


Figura 15.8 Implementación de las colas empleando la implementación circular.

tor sólo resolvería momentáneamente el problema. Esto es debido a que, incluso si el tamaño del vector fuera 1.000, después de 1.000 operaciones `insertar`, no queda espacio libre en la cola, independientemente de su tamaño actual. Incluso si se han realizado 1.000 acciones de `quitarPrimero`, vaciando con ello a nivel abstracto la cola, no podemos añadir ningún elemento más.

Sin embargo, tal y como muestra la Figura 15.8, existe un montón de espacio libre; todas las posiciones anteriores a `cabeza` están vacías, por lo que podrían reciclarse. Con tal objetivo introducimos la *circularidad*: cuando `fin` o `cabeza` rebasan la última posición del vector, se reposicionan en el principio del mismo. Esta implementación recibe el nombre de *implementación circular*. Ahora sólo necesitamos duplicar el vector cuando el número de elementos en la cola sea igual al número de elementos insertados en el vector. Para realizar `insertar(f)` situamos `fin` al principio del vector y colocamos `f` en esa posición. Ahora, tras las tres acciones `quitarPrimero`, también situamos `cabeza` al principio del vector.

El esqueleto de la clase genérica `ColaVec` se muestra en la Figura 15.9. La clase `ColaVec` tiene cuatro atributos: un vector que puede expandirse dinámicamente, el número de elementos que están en cada momento en la cola, y las posiciones en el vector del primer y último elemento de la cola.

En la sección privada de la clase se declaran dos métodos. Estos métodos son empleados internamente por los métodos de `ColaVec`, pero no están disponibles para el usuario de la clase. El primero de ellos es la rutina `incrementar`, que

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase ColaVec
6  //
7  // CONSTRUCCIÓN: sin ninguna inicialización
8  //
9  // ***********************OPERACIONES PÚBLICAS*********************
10 // void insertar( x )      --> Inserta x
11 // Object primero( )       --> Devuelve el elemento más antiguo ins
12 // Object quitarPrimero( ) --> Devuelve y elimina el elemento más a
13 // boolean esVacia( )      --> Devuelve true si cola vacía; si no,
14 // void vaciar( )          --> Elimina todos los elementos
15 // ***************************ERRORES****************************
16 // primero o quitarPrimero de una cola vacía
17
18 /**
19  * Implementación de las colas basada en un vector.
20  */
21 public class ColaVec implements Cola
22 {
23     public ColaVec( )
24       { /* Figura 15.11 */ }
25
26     public boolean esVacia( )
27       { return tamanyoActual == 0; }
28     public void insertar( Object x )
29       { /* Figura 15.12 */ }
30     public void vaciar( )
31       { /* Figura 15.15 */ }
32     public Object quitarPrimero( ) throws DesbordamientoInferior
33       { /* Figura 15.14 */ }
34     public Object primero( ) throws DesbordamientoInferior
35       { /* Figura 15.14 */ }
36
37     private Object [ ] elVector;
38     private int tamanyoActual;
39     private int cabeza;
40     private int fin;
41
42     static final int CAPACIDAD_POR_DEFECTO = 10;
43
44     private int incrementar( int x )
45       { /* Figura 15.10 */ }
46     private void duplicarCola( )
47     { /* Figura 15.13 */ }
48 }
```

Figura 15.9 Esqueleto de la clase `ColaVec`.

añade 1 a su parámetro y devuelve el nuevo valor. Como servirá para implementar la circularidad, si su resultado es igual al tamaño del vector, se ajusta a cero. Esto se muestra en la Figura 15.10. La otra rutina es `duplicarCola`, que es invocada cuando `insertar` precisa de la duplicación del vector. Es ligeramente más complicada que `duplicarVector`, ya que ahora los elementos de la cola no están ne-

```
1  /**
2   * Método interno para el incremento con circularidad.
3   * @param x cualquier índice en el rango de elVector.
4   * @return x+1, o 0 si x está al final de elVector.
5   */
6  private int incrementar( int x )
7  {
8      if( ++x == elVector.length )
9          x = 0;
10     return x;
11 }
```

Figura 15.10 Rutina de generación de la circularidad.

cesariamente almacenados empezando en la posición cero del vector. Como consecuencia, los elementos deben ser copiados con cuidado. `duplicarCola` se discute junto a `insertar`.

La mayoría de los métodos públicos se asemejan a sus equivalentes en las pilas, incluyendo el constructor mostrado en la Figura 15.11. Este constructor no es particularmente singular, salvado el hecho de que debemos asegurarnos que `cabeza` y `fin` reciben los valores adecuados.

En la Figura 15.12 se muestra la rutina `insertar`. La estrategia básica es muy sencilla, tal y como se muestra en las líneas 9 a 11. `duplicarCola`, mostrado en la Figura 15.13, es similar a la rutina de duplicación de las pilas, salvo en lo que se refiere a la copia de los vectores, en la que el recorrido del vector viejo no comienza en la primera posición. En este caso, en las líneas 11 a 13 el método salta a lo largo del vector antiguo y copia cada elemento en el nuevo. Los resultados se copian en el nuevo vector, recorriendo éste desde la primera posición.

```
1  /**
2   * Construye la cola.
3   */
4  public ColaVec( )
5  {
6      elVector = new Object[ CAPACIDAD_POR_DEFECTO ];
7      vaciar( );
8  }
```

Figura 15.11 Constructor sin parámetros de las `Colas` basadas en un vector.

```
1  /**
2   * Inserta un nuevo elemento en la cola.
3   * @param x el elemento a insertar.
4   */
5  public void insertar( Object x )
6  {
7      if( tamanyoActual == elVector.length )
8          duplicarCola( );
9      fin = incrementar( fin );
10     elVector[ fin ] = x;
11     tamanyoActual++;
12 }
```

Figura 15.12 Método `insertar` de las `Colas` basadas en un vector.

```
1  /**
2   * Método interno para expandir elVector.
3   */
4  private void duplicarCola( )
5  {
6      Object [ ] nuevoVector;
7
8      nuevoVector = new Object[ elVector.length * 2 ];
9
10             // Copia de los elementos de la cola
11     for( int i = 0; i < tamanyoActual;
12                         i++, cabeza = incrementar( cabeza ) )
13         nuevoVector[ i ] = elVector[ cabeza ];
14
15     elVector = nuevoVector;
16     cabeza = 0;
17     fin = tamanyoActual - 1;
18 }
```

Figura 15.13 Expansión dinámica de la clase de las `Colas` basadas en un vector.

Observe los nuevos valores de `cabeza` y `fin`, en las líneas 16 y 17. Las rutinas `quitarPrimero` y `primero` se muestran en la Figura 15.14; ambas son cortas y sencillas. Finalmente, `vaciar` se muestra en la Figura 15.15. De nuevo es claro que las rutinas de las colas se ejecutan en tiempo constante. Del mismo modo que en el caso de las pilas, el coste de la duplicación de vectores puede amortizarse sobre la secuencia de operaciones `insertar`.

```
1  /**
2   * Devuelve y elimina el elemento más antiguo insertado
3   * en la cola.
4   * @return el elemento más antiguo insertado en la cola.
5   * @exception DesbordamientoInferior si la cola está vacía.
6   */
7  public Object quitarPrimero( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "QuitarPrimero" );
11     tamanyoActual--;
12
13     Object valorDevuelto = elVector[ cabeza ];
14     cabeza = incrementar( cabeza );
15     return valorDevuelto;
16 }
17
18 /**
19  * Devuelve el elemento más antiguo insertado en la cola.
20  * @return el elemento más antiguo insertado en la cola.
21  * @exception DesbordamientoInferior si la cola está vacía.
22  */
23 public Object primero( ) throws DesbordamientoInferior
24 {
25     if( esVacia( ) )
26         throw new DesbordamientoInferior( "Primero" );
27     return elVector[ cabeza ];
28 }
```

Figura 15.14 Métodos `quitarPrimero` y `primero` de las `Colas` basadas en un vector.

```
/**
 * Vaciado lógico de la cola.
 */
public void vaciar( )
{
    tamanyoActual = 0;
    cabeza = 0; fin = -1;
}
```

Figura 15.15 Rutina `vaciar` para las `Colas` basadas en un vector.

15.2 Implementaciones con listas enlazadas

Una alternativa a la implementación contigua empleando vectores, consiste en utilizar una lista enlazada. Recuerde de la Sección 6.4, que en una lista enlazada almacenamos cada elemento en un objeto distinto, que contiene además una referencia al siguiente objeto de la lista.

La ventaja de las listas enlazadas es que el exceso de gasto de memoria en que se incurre se reduce a una referencia por elemento, mientras que el almacenamiento contiguo en vectores exige un espacio eventualmente mayor, igual al número actual de posiciones vacías del vector (además de algo de memoria adicional durante el proceso de duplicación). Es cierto que en Java esta ventaja es poco relevante ya que las posiciones vacías del vector representan referencias nulas, que consumen muy poco espacio. Sin embargo, en otros lenguajes de programación, esta ventaja sería importante ya que las posiciones vacías suelen guardar instancias de objetos no inicializadas, que consumen un notable espacio. Además de por este motivo, las implementaciones con listas enlazadas se discuten por las siguientes razones:

1. Es importante comprender implementaciones alternativas que pueden ser útiles en otros lenguajes de programación.
2. Las implementaciones que emplean listas enlazadas son más reducidas que las correspondientes versiones con vectores, especialmente en el caso de las colas.
3. Dichas implementaciones ilustran algunos principios que se esconden tras las operaciones generales con listas enlazadas presentadas en el Capítulo 16.

Para que la implementación sea competitiva con las implementaciones contiguas con un vector, debemos ser capaces de ejecutar las operaciones básicas de listas enlazadas en tiempo constante. Esto es sencillo, ya que los cambios en la lista enlazada se limitan a los elementos de los dos extremos (`cabeza` y `fin`) de la lista.

15.2.1 Pilas

El interfaz `Pila` puede implementarse empleando una lista enlazada, en la que la cima de la pila está representada por el primer elemento de la lista. Esto se ilustra en la Figura 15.16. Para implementar la operación `apilar`, creamos un nuevo nodo y lo incluimos en la lista como primer elemento. Dicho nodo se genera por la

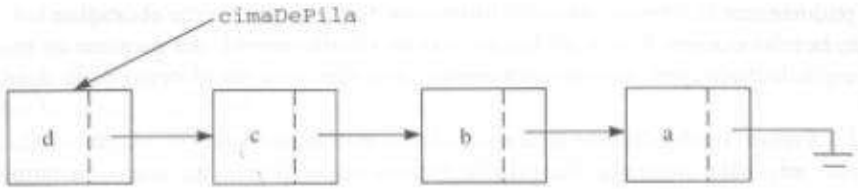


Figura 15.16 Implementación de una pila mediante una lista enlazada.

llamada a `new`. Para implementar el método `desapilar`, basta con desplazar la cima de la pila al segundo elemento de la lista (si es que existe). La pila vacía se representa mediante la lista vacía.

Claramente, cada operación es constante, ya que al restringir las operaciones al primer elemento de la lista, conseguimos que todos los cálculos sean independientes del tamaño de la misma. Todo esto se conserva en las implementaciones en Java.

La Figura 15.17 muestra la declaración de los nodos de la lista. Un `NodoLista` está compuesto por dos atributos: `dato` guarda un elemento y `siguiente` almacena una referencia que apunta al siguiente nodo de la lista. Se dispone de dos constructores para `NodoLista`. El primero de ellos se puede emplear en la forma siguiente

```
NodoLista nodo1 = new NodoLista( x );
```

donde `x` es un elemento. El segundo puede usarse para inicializar la referencia siguiente por medio de una referencia `NodoLista` (o `null`):

```
NodoLista nodo2 = new NodoLista( x, nodo1 );
```

Obsérvese que el acceso a la nueva clase `NodoLista` es amistoso. Esto significa que este tipo no pertenece al ámbito global, pero es visible para el resto de clases

```
package EstructurasDatos;

// Nodo básico almacenado en una lista enlazada.
// Nótese que esta clase no es accesible fuera
// del paquete EstructurasDatos.

class NodoLista
{
        // Constructores
    NodoLista( Object elElemento )
      { this( elElemento, null ); }

    NodoLista( Object elElemento, NodoLista n )
      { dato = elElemento; siguiente = n; }

        // Atributos amistosos; accesibles desde otras rutinas del pac
    Object dato;
    NodoLista siguiente;
}
```

Figura 15.17 Clase `NodoLista`.

del paquete `EstructurasDatos`. Esto es positivo, ya que fuerza el ocultamiento de la información. Los `NodoLista` son un detalle interno del paquete de estructuras de datos, pero son completamente invisibles para los usuarios de la clase `Pila`.

La Figura 15.18 muestra la clase `PilaLi` que implementa el interfaz `Pila` usando una lista enlazada. La pila se representa mediante un único atributo: `cimaDePila` es una referencia que apunta al primer `NodoLista` de la lista enlazada. El constructor de la línea 24 indica que se crea una pila vacía inicializando `cimaDePila` a `null`.

```
kage EstructurasDatos;

ort Excepciones.*;

Clase PilaLi

CONSTRUCCIÓN: sin ninguna inicialización

************************OPERACIONES PÚBLICAS************************
void apilar( x )          --> Inserta x
void desapilar( )         --> Elimina el último elemento insertado
Object cima( )            --> Devuelve el último elemento insertado
Object cimaYDesapilar( ) --> Devuelve y elimina el elemento más reciente
boolean esVacia( )        --> Devuelve true si pila vacía; si no, false
void vaciar( )            --> Elimina todos los elementos
*************************ERRORES**********************************
Desapilar, cima y cimaYDesapilar sobre una pila vacía


Implementación de las pilas basada en una lista enlazada.

lic class PilaLi implements Pila

 public PilaLi( )
   { cimaDePila = null; }

 public boolean esVacia( )
   { return cimaDePila == null; }
 public void vaciar( )
   { cimaDePila = null; }
 public void apilar( Object x )
   { cimaDePila = new NodoLista( x, cimaDePila ); }

 public void desapilar( ) throws DesbordamientoInferior
   { /* Figura 15.19 */ }
 public Object cima( ) throws DesbordamientoInferior
   { /* Figura 15.19 */ }
 public Object cimaYDesapilar( ) throws DesbordamientoInferior
   { /* Figura 15.19 */ }

 private NodoLista cimaDePila;
```

.18 Esqueleto de la clase `Pila` basada en una lista enlazada (`PilaLi`).

La operación `apilar` requiere una única línea de código: la creación de un nuevo `NodoLista`. El atributo contiene el elemento `x` a insertar, y la referencia `siguiente` para el nuevo nodo es la `cimaDePila` inicial. El nodo creado pasa a ser la nueva cima. Todo esto se realiza en la línea 32

Las rutinas restantes se muestran el la Figura 15.19. La operación `desapilar` también es muy sencilla. Tras el test obligatorio para detectar la pila vacía, se actualiza `cimaDePila` de modo que apunte al segundo elemento de la lista. Análogamente, `cima` y `cimaYDesapilar` son rutinas muy breves.

```
 1 /**
 2  * Devuelve el último elemento insertado en la pila.
 3  * No modifica la pila.
 4  * @return el último elemento insertado en la lista.
 5  * @exception DesbordamientoInferior si la pila está vacía.
 6  */
 7 public Object cima( ) throws DesbordamientoInferior
 8 {
 9     if( esVacia( ) )
10         throw new DesbordamientoInferior( "Cima" );
11     return cimaDePila.dato;
12 }
13
14 /**
15  * Devuelve el último elemento insertado en la pila.
16  * @exception DesbordamientoInferior si la pila está vacía.
17  */
18 public void desapilar( ) throws DesbordamientoInferior
19 {
20     if( esVacia( ) )
21         throw new DesbordamientoInferior( "Desapilar" );
22     cimaDePila = cimaDePila.siguiente;
23 }
24
25 /**
26  * Devuelve y elimina el último elemento insertado
27  * en la pila.
28  * @return el último elemento insertado en la pila.
29  * @exception DesbordamientoInferior si la pila está vacía.
30  */
31 public Object cimaYDesapilar( ) throws DesbordamientoInferior
32 {
33     if( esVacia( ) )
34         throw new DesbordamientoInferior( "cimaYDesapilar" );
35
36     Object datoCima = cimaDePila.dato;
37     cimaDePila = cimaDePila.siguiente;
38     return datoCima;
39 }
```

Figura 15.19 Componentes sencillas de la clase `Pila` basada en una lista enlazada (`PilaLi`).

15.2.2 Colas

Las colas pueden implementarse empleando una lista enlazada, siempre que se mantengan referencias que apunten al principio y al final de la lista. La Figura 15.20 ilustra la idea general.

Los métodos de las colas son prácticamente idénticos a los de las pilas. En la Figura 15.21 se muestra el esqueleto de la clase `ColaLi`. En ella no hay nada nuevo, fuera del hecho de que mantenemos dos referencias en lugar de una.

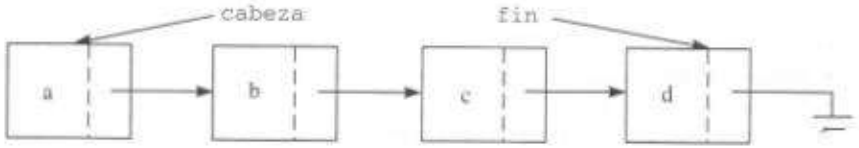


Figura 15.20 Implementación de una cola basada en una lista enlazada.

```
ckage EstructurasDatos;

port Excepciones.*;

 Clase ColaLi

 CONSTRUCCIÓN: sin ninguna inicialización

 ********************OPERACIONES PÚBLICAS**************************
void insertar( x )        --> Inserta x
Object primero( )         --> Devuelve el elemento más antiguo insertado
Object quitarPrimero( )   --> Devuelve y elimina el elemento más reciente
boolean esVacia( )        --> Devuelve true si cola vacía; si no, false
void vaciar( )            --> Elimina todos los elementos
 ************************ERRORES*********************************
 primero o quitarPrimero sobre una cola vacía

*
 Implementación de las colas basada en una lista enlazada.
/
blic class ColaLi implements Cola

  public ColaLi( )
    { vaciar( ); }

  public boolean esVacia( )
    { return cabeza == null; }
  public void vaciar( )
    { cabeza = fin = null; }
  public Object quitarPrimero( ) throws DesbordamientoInferior
    { /* Figura 15.22 */ }
  public Object primero( ) throws DesbordamientoInferior
    { /* Figura 15.23 */ }
  public void insertar( Object x )
    { /* Figura 15.25 */

  private NodoLista cabeza;
  private NodoLista fin;
```

5.21 Esqueleto de la clase `Cola` basada en una lista enlazada (`ColaLi`).

En la Figura 15.22 se implementa `quitarPrimero`. Dicha rutina es lógicamente equivalente a la operación de las pilas `desapilar`. `primero` es muy sencillo y se muestra en la Figura 15.23.

El método `insertar` distingue dos casos. Si la cola está vacía, creamos una cola con un único elemento, invocando a `new`, y actualizando `cabeza` y `fin` de modo que ambos referencien al único nodo existente. En caso contrario, creamos un nuevo nodo con el dato `x`, lo añadimos al final de la lista y actualizamos el final de la lista a ese nuevo nodo. Todo esto se ilustra en la Figura 15.24. Nótese que la inserción del primer elemento es un caso especial, ya que no existe ninguna referencia `siguiente` a la que pueda apuntar el nuevo nodo. Todo esto se hace en la Figura 15.25.

```
1  /**
2   * Devuelve y elimina el elemento más antiguo insertado
3   * en la cola.
4   * @return el elemento más antiguo insertado en la cola.
5   * @exception DesbordamientoInferior si la cola está vacía.
6   */
7  public Object quitarPrimero( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "QuitarPrimero" );
11
12     Object valorDevuelto = cabeza.dato;
13     cabeza = cabeza.siguiente;
14     return valorDevuelto;
15 }
```

Figura 15.22 Método `quitarPrimero` para la clase `Cola` basada en una lista enlazada (`ColaLi`).

```
1  /**
2   * Devuelve el último elemento insertado en la cola.
3   * No modifica la cola.
4   * @return el último elemento insertado en la cola.
5   * @exception DesbordamientoInferior si la cola está vacía.
6   */
7  public Object primero( ) throws DesbordamientoInferior
8  {
9      if( esVacia( ) )
10         throw new DesbordamientoInferior( "Primero" );
11     return cabeza.dato;
12 }
```

Figura 15.23 Método `primero` para la clase `Cola` basada en una lista enlazada (`ColaLi`).

15.3 Comparación de los dos métodos

Las operaciones de las dos versiones presentadas, basadas en vectores y listas enlazadas, se ejecutan en tiempo constante. Por tanto, son tan rápidas como para no suponer un cuello de botella para ningún algoritmo. Como consecuencia, raramente importa qué versión se esté empleando.

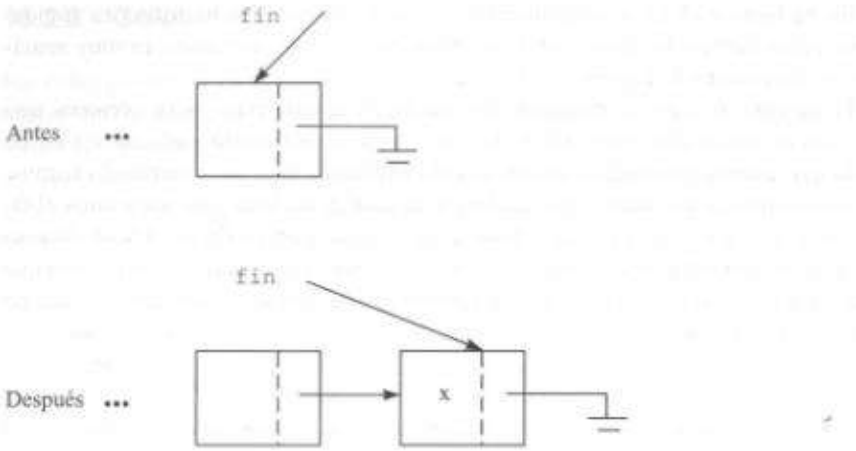


Figura 15.24 Operación `insertar` para la implementación basada en una lista enlazada.

```
1  /**
2   * Inserta un nuevo elemento en la cola.
3   * @param x el elemento a insertar.
4   */
5  public void insertar( Object x )
6  {
7      if( esVacia( ) ) // Genera una cola de un solo dato
8          fin = cabeza = new NodoLista( x );
9      else // Caso regular
10         fin = fin.siguiente = new NodoLista( x );
11 }
```

Figura 15.25 Método `insertar` para la clase `Cola` basada en una lista enlazada (`ColaLi`).

Las versiones basadas en vectores parecen ser más rápidas que las basadas en listas enlazadas, especialmente cuando se dispone a priori de una estimación ajustada de la capacidad. Si se cuenta con un constructor adicional que permita especificar la capacidad inicial (*véase* el Ejercicio 15.3) y la estimación es correcta, no será necesario realizar ninguna duplicación. Además, el acceso secuencial permitido por el vector, es típicamente más rápido que el acceso potencialmente no secuencial de la memoria dinámica.

Sin embargo, la implementación basada en vectores presenta dos inconvenientes. En primer lugar, en el caso de las colas, la implementación mediante vectores es más compleja que la implementación mediante listas enlazadas, debido al código necesario para la circularidad y sus implicaciones en la duplicación del vector. Nuestra implementación de la duplicación de vectores no es tan eficiente como podría hacerse (*véase* el Ejercicio 15.8). Una implementación más rápida necesitaría algunas líneas adicionales de código. También la implementación de las pilas basada en un vector es algo más larga que la correspondiente basada en una lista enlazada.

El segundo inconveniente aparece en otros lenguajes de programación, pero no en Java. Cuando duplicamos un vector, necesitamos temporalmente tres veces más espacio de lo que la cantidad de datos podría sugerir. Esto es debido a que en la duplicación se necesita memoria para almacenar tanto el vector viejo como el nuevo (de tamaño doble). Más aún, si la cola alcanza un gran tamaño, el índice de ocupación del vector estará entre el 50 y el 100 por ciento: en media, esto supone que está lleno en un 75 por ciento, es decir, que por cada tres elementos del vector, una posición está vacía. Como consecuencia, el espacio perdido en media es el 33 por ciento del utilizado, y cuando la tabla está medio llena llega a ser del 100 por ciento. En Java esto no supone un gran problema, ya que cada elemento del vector es sólo una referencia. En otros lenguajes, como C++, los objetos se almacenan directamente, en lugar de ser referenciados. En estos lenguajes, el espacio perdido puede ser significativo, cuando lo comparamos con las versiones basadas en listas enlazadas, que requieren únicamente una referencia extra por cada elemento.

15.4 Colas dobles

Este capítulo concluye con una discusión acerca del uso de la herencia para derivar una nueva estructura de datos. Una *cola doble* es similar a una cola, excepto por el hecho de que está permitido el acceso por ambos extremos. El Ejercicio 14.11 describe una aplicación de las colas dobles. Los términos empleados en lugar de `insertar` y `quitarPrimero`, son `anyadirPorDelante`, `anyadirPorDetras`, `eliminarPorDelante` y `eliminarPorDetras`. La Figura 15.26 muestra la clase derivada `ColaDoble`. La derivamos a partir de `ColaVec` ya que la eliminación de nodos al final de una lista no está soportada de forma eficiente.

El constructor por defecto de esta clase vendrá dado por una llamada a `super`, lo que resulta adecuado. `insertar` y `quitarPrimero` pueden emplearse, puesto que al extender clases no podemos eliminar métodos. El método `primero` de la clase de las colas no se cambia, se hereda sin modificación alguna. `anyadirPorDetras` y `eliminarPorDelante` invocan a las rutinas `insertar` y `quitarPrimero`. Las únicas rutinas que han de ser completamente implementadas son `anyadirPorDelante`, `eliminarPorDetras` y `verFinal`. Para hacerlo, necesitamos que los atributos de la clase original sean protegidos, tras lo cual podemos escribir los nuevos métodos. Esto se deja como ejercicio al lector en el Ejercicio 15.5.

Resumen

En este capítulo se describen las implementaciones de las clases de las pilas y de las colas. Ambos tipos de estructuras pueden representarse empleando un vector o una lista enlazada. En ambos casos, las operaciones se ejecutan en tiempo constante, por lo que todas ellas son muy rápidas.

En otros lenguajes de programación, la implementación basada en vectores emplea más memoria pero menos tiempo que la correspondiente implementación basada en listas enlazadas, conduciéndonos a la típica disquisición de tiempo versus memoria. En Java, esta discusión no es muy relevante, debido al tratamiento de los objetos con referencias.

```
istructurasDatos;
de las colas dobles

UCCIÓN: sin ninguna inicialización

**********OPERACIONES PÚBLICAS**********
nyadirPorDelante( Object x )--> Inserta x por delante
nyadirPorDetras( Object x ) --> Inserta x por detrás
: eliminarPorDelante( )      --> Elimina el elemento de la cabeza
: eliminarPorDetras( )       --> Elimina el elemento del final
: primero( )                 --> Devuelve el elemento de la cabeza
: verFinal( )                --> Devuelve el elemento del final
n esVacia( )                 --> Devuelve true si vacía; si no, false
aciar( )                     --> Elimina todos los elementos
ar y quitarPrimero están disponibles, pero no deben emplearse
**********ERRORES**********
iza una excepción si se elimina o se consulta una cola doble vacía

aDoble extends ColaVec

c void anyadirPorDelante( Object x )
* Ejercicio 15.5 */ }
c void anyadirPorDetras( Object x )
insertar( x ); }
c Object eliminarPorDelante( ) throws DesbordamientoInferior
eturn quitarPrimero( ); }
c Object eliminarPorDetras( ) throws DesbordamientoInferior
* Ejercicio 15.5 */ }
c Object verFinal( ) throws DesbordamientoInferior
* Ejercicio 15.5 */ }
sVacia, vaciar y primero son métodos heredados
```

Clase de las colas dobles `ColaDoble` derivada a partir de `ColaVec`.

Elementos del juego

circularidad Se produce cuando `cabeza` o `fin` regresan al principio del vector al haber rebasado la última posición del mismo.

cola doble Cola en la que está permitido el acceso por ambos extremos. La mayor parte de su funcionalidad se deriva de la clase de las colas.

implementación circular de vectores Implementación en la que se emplea la circularidad para representar una cola.

Errores comunes

1. La implementación circular de vectores requerida en el caso de las colas puede hacerse fácilmente de forma errónea cuando se intenta reducir el código necesario. Por ejemplo, no debe evitarse el uso de `tamanyoActual`, intentando calcular el tamaño del vector a partir de `fin` y `cabeza`.
2. Utilizar una implementación de estas estructuras en la que no pueda garantizarse el acceso a la estructura de datos en tiempo constante es un grave error. No existe ninguna justificación para una ineficiencia tal.

En Internet

Están disponibles los ficheros que se citan a continuación, encontrándose en el directorio **DataStructures**. Algunos programas de prueba se vieron en el Capítulo 6, mientras que otros se emplean en las aplicaciones de la Parte III del libro.

StackAr.java	Implementa una pila empleando un vector. Es la versión inglesa de la clase `PilaVec`.
StackLi.java	Implementa una pila empleando una lista enlazada. Es la versión inglesa de la clase `PilaLi`.
QueueAr.java	Implementa una cola empleando un vector. Es la versión inglesa de la clase `ColaVec`.
QueueLi.java	Implementa una cola empleando una lista enlazada. Es la versión inglesa de la clase `ColaLi`.

Ejercicios

Cuestiones breves

15.1. Dibuje el valor de las estructuras que soportan la implementación de las pilas y de las colas (para las implementaciones con vectores y con listas enlazadas) tras cada paso de la siguiente secuencia de operaciones: *añadir*(1), *añadir*(2), *eliminar*, *añadir*(3), *añadir*(4), *eliminar*, *eliminar*, *añadir*(5). En la implementación con vectores, suponer que el tamaño inicial de la estructura es tres.

Problemas prácticos

15.3. Compare los tiempos de ejecución de las dos versiones de la clase `Pila`, la basada en vectores y la basada en listas enlazadas. Emplee objetos de tipo `Integer`.

15.3. Añada constructores a las clases de las pilas y de las colas en sus versiones basadas en vectores. Dichos constructores deben permitir especificar la capacidad inicial del vector.

15.4. Escriba un método `main` que cree y manipule simultáneamente dos pilas, una de objetos de tipo `Integer` y otra de objetos de tipo `Double`.

15.5. Complete la implementación de la clase `DobleCola`.

15.6. Implemente la clase de las pilas basadas en un vector empleando la clase predefinida `Vector`. ¿Cuáles son las ventajas y desventajas de esta aproximación al problema?

15.7. Implemente la clase de las pilas basadas en un vector extendiendo la clase predefinida `Vector`. ¿Cuáles son las ventajas y desventajas de esta aproximación al problema?

15.8. El código de la duplicación de vectores de la clase `ColaVec` es excesivamente costoso debido a las repetidas llamadas a `incrementar`. Escriba una versión más eficiente, evitando todas las llamadas a `incrementar`. Para hacerlo debe controlar cuando se emplea la circularidad. De ser así, emplee dos bucles separados para realizar la copia. En cualquier otro caso, use un solo bucle.

15.9. Usar un `Vector` para implementar una cola basada en un vector es algo problemático, ya que el método `setSize` no es lo que se necesita. Muestre cómo redistribuir los elementos de la cola tras la operación `setSize`, de modo que sólo cambien, a lo sumo, la mitad de las referencias del `Vector`. (No obstante, esto sigue empleando más movimientos que la implementación que no usa la clase `Vector`.)

Prácticas de programación

15.10. Una cola doble restringida permite insertar elementos en ambos extremos de la estructura, pero sólo permite accesos y eliminaciones en la cabeza. Se pide hacer lo siguiente:

a) Emplee herencia para derivar esta nueva clase a partir de `Cola`.
b) Emplee herencia para derivar esta nueva clase a partir de `ColaDoble`.

15.11. Supongamos que deseamos añadir la operación `encontrarMinimo` (pero no `eliminarMinimo`) en el repertorio de las pilas. Se pide hacer lo siguiente:

a) Emplee herencia para derivar la nueva clase, e implemente `encontrarMinimo` de modo que se recorra secuencialmente la pila de elementos.
b) En lugar de emplear herencia, implemente la nueva clase empleando dos pilas, tal y como se propuso en el Ejercicio 6.5.
c) Emplee herencia, junto con el algoritmo descrito en el Ejercicio 6.5, para la derivación de la nueva clase.

15.12. Supongamos que queremos añadir la operación `encontrarMinimo` (pero no `eliminarMinimo`) en el repertorio de las colas dobles. Se pide hacer lo siguiente:

a) Emplee herencia para derivar la nueva clase, e implemente `encontrarMinimo` de modo que se recorra secuencialmente la pila de elementos. Debe hacer las elecciones adecuadas para decidir que atributos serán protegidos.
b) En lugar de emplear herencia, implemente la nueva clase empleando cuatro colas. Si una eliminación vacía una cola, deben reorganizarse los elementos restantes.

15.13. Implemente un applet que muestre cómo varía el estado de una pila cuando se aplican sobre ella las operaciones básicas. Incluya una GUI que permita al usuario especificar las operaciones. Además, debe permitirse la elección (mediante una componente de elección) entre la implementación basada en vectores y la implementación basada en listas enlazadas.

15.14. Repita el Ejercicio 15.13 para el caso de una cola.

CAPÍTULO

16 Listas enlazadas

En el Capítulo 15 se mostró que se pueden usar las listas enlazadas para almacenar elementos de forma no contigua. Las listas enlazadas usadas en ese capítulo eran simples debido a que todos los accesos se realizaban en uno de los dos extremos. En este capítulo veremos:

- Cómo permitir el acceso a cualquier elemento usando una lista enlazada general.
- Los algoritmos generales usados para las operaciones sobre listas enlazadas.
- Cómo la *clase iteradora* proporciona un mecanismo seguro para recorrer y acceder a las listas enlazadas.
- Variaciones de las listas, como las *listas doblemente enlazadas* y las *listas enlazadas circulares*.
- Cómo podemos usar la herencia para derivar una clase de *listas enlazadas ordenadas*.

16.1 Ideas básicas

Este capítulo implementa las listas enlazadas y permite acceso general (operaciones de inserción, eliminación y consulta arbitrarias de un elemento) a la lista. La lista enlazada básica consta de una colección de nodos situados en la memoria dinámica conectados entre sí. En una lista simplemente enlazada, cada nodo se compone de un dato y una referencia al siguiente nodo de la lista. El último nodo de la lista contiene una referencia siguiente `null`. Supondremos que el nodo viene dado por la declaración `NodoLista` del Capítulo 15:

```
class NodoLista
{
    Object    dato;
    NodoLista siguiente;
        // Constructores
}
```

Se puede acceder al primer nodo de la lista a través de un puntero, lo cual se muestra en la Figura 16.1. Podemos imprimir la lista enlazada o buscar en ella comenzando en el primer elemento y siguiendo la cadena de punteros `siguiente`.

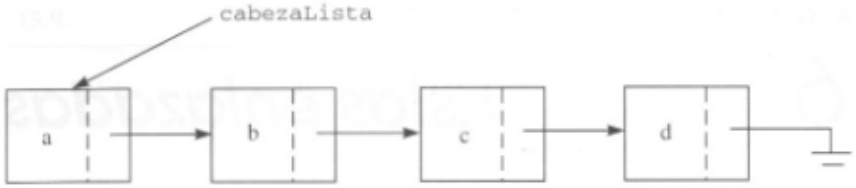


Figura 16.1 Lista enlazada básica.

Las dos operaciones básicas que se pueden realizar son la inserción y eliminación de un elemento arbitrario `x`.

En el caso de la inserción, debemos determinar dónde va a tener lugar la inserción. Si partimos de una referencia a algún nodo de la lista, el lugar donde más fácilmente se puede insertar es el inmediatamente posterior a ese elemento. Como ejemplo, en la Figura 16.2 se muestra cómo insertamos `x` después del elemento `a` en una lista enlazada. Debemos llevar a cabo los siguientes pasos:

```
tmp = new NodoLista( );              // Crear un nodo nuevo
tmp.dato = x;                        // Colocar x en el campo dato
tmp.siguiente = actual.siguiente;    // El siguiente al nodo x es b
actual.siguiente = tmp;              // El siguiente al nodo a es x
```

Como resultado de estas instrucciones, la lista anterior `...a,b,...` ahora es `...a,x,b,...`. Podemos simplificar el código, puesto que `NodoLista` tiene un constructor que inicializa los atributos directamente. Así, obtenemos

```
tmp = new NodoLista( x, actual.siguiente ); // Crear un nuevo nodo
actual.siguiente = tmp;     // El siguiente al nodo a es x
```

En este punto, queda claro que ya no se necesita `tmp`, por lo que podemos limitarnos a la siguiente línea:

```
actual.siguiente = new NodoLista( x, actual.siguiente );
```

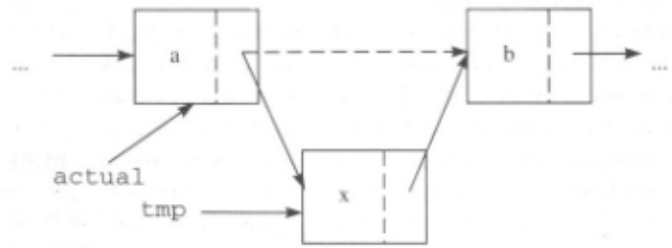


Figura 16.2 Inserción en una lista enlazada: se crea un nodo nuevo (`tmp`), se copia `x` en él y se determinan los valores de la referencia `siguiente` de `tmp` y de `actual`.

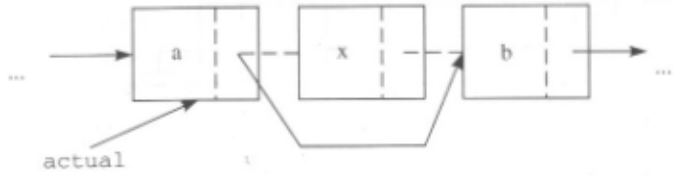


Figura 16.3 Eliminación en una lista enlazada.

La eliminación se puede ejecutar con un simple cambio de una referencia. La Figura 16.3 muestra que para eliminar el elemento `x` de la lista enlazada, hacemos que `actual` sea el nodo anterior a `x`, tras lo que hacemos que la referencia `siguiente` de `actual` salte por encima de `x`. Esto se expresa mediante la siguiente instrucción

```
actual.siguiente = actual.siguiente.siguiente;
```

La lista `...a,x,b,...` es ahora `...a,b,...`.

Esto resume el comportamiento básico de insertar y eliminar en posiciones arbitrarias de una lista enlazada. La propiedad fundamental de las listas enlazadas es que se pueden hacer cambios en una lista realizando una cantidad constante de movimientos de datos, lo cual es una gran mejora respecto a la implementación usando vectores, ya que la contigüidad en un vector implica que siempre que se añade o elimina un elemento, se deben desplazar todos los elementos que le siguen.

16.1.1 Nodos cabecera

Un problema de la descripción básica es que asume que siempre que se elimina un elemento, hay algún elemento anterior a él. En consecuencia, la eliminación del primer elemento de una lista enlazada se convierte en un caso especial. De forma análoga, la rutina de inserción no nos permite insertar un elemento en la primera posición de la lista, ya que las inserciones están restringidas a aquellas posiciones posteriores a alguna otra. Así, aunque el algoritmo básico funciona bien, hay algunos casos especiales molestos que debemos tratar.

Los casos especiales son siempre problemáticos en el diseño de algoritmos y conducen con frecuencia a errores en el código. Por ello es preferible escribir código que evite dichos casos especiales. Una forma de lograr esto en este caso consiste en introducir un *nodo cabecera*.

Un nodo cabecera es un nodo extra de la lista enlazada que no guarda ningún dato pero que sirve para satisfacer el requerimiento de que cada nodo que contenga un elemento tenga un nodo anterior. En la Figura 16.4 se muestra el nodo cabecera para la lista `a,b,c`. Observe que `a` ya no es un caso especial. Se puede borrar como cualquier otro nodo, colocando `actual` en el nodo anterior a él. También podemos añadir fácilmente un nuevo elemento al principio de la lista, colocando `actual` en el nodo cabecera y llamando a la rutina de inserción. Usando el nodo cabecera, simplificamos el código a cambio de una penalización despreciable en espacio. En aplicaciones más complejas, la cabecera no solamente simplifica el código sino que también aumenta la velocidad, ya que, a la postre, un menor número de tests requiere un menor tiempo.

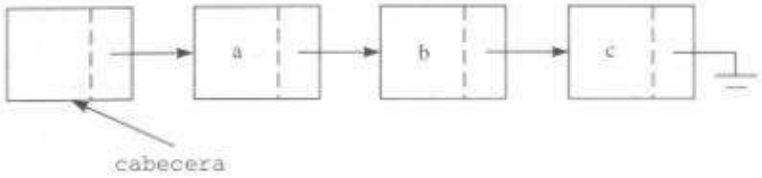


Figura 16.4 Uso de un nodo cabecera en la lista enlazada.

El uso de nodos cabecera es controvertido. Algunos argumentan que el hecho de evitar casos especiales no es justificación suficiente para añadir celdas ficticias, considerando el uso del nodo cabecera como poco un truco sucio al viejo estilo de los primeros programadores. Aun así, nosotros los usaremos pues nos permiten concentrarnos en las manipulaciones básicas de los punteros sin oscurecer el código con casos especiales. Si se deben o no usar nodos cabecera es cuestión de preferencias personales. Además, en una implementación de la clase su uso sería completamente transparente al usuario. Pero debemos ser cuidadosos; la rutina de impresión no debe imprimir el contenido de la cabecera y las rutinas de búsqueda deben saltarse ese nodo. Colocarnos al principio de la lista ahora significa colocar la posición actual en `cabecera.siguiente`. Como se muestra en la Figura 16.5, una lista con nodo cabecera es vacía si `cabecera.siguiente` es `null`.

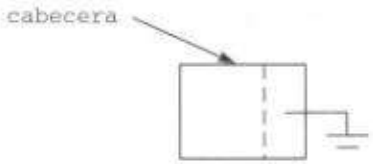


Figura 16.5 Lista vacía cuando se usa un nodo cabecera.

16.1.2 Clases iteradoras

La estrategia primitiva habitual identifica una lista enlazada con un puntero al nodo cabecera. Entonces se puede acceder a cada elemento individual de la lista proporcionando un puntero al nodo que lo contiene. El problema con esta estrategia es que es difícil comprobar si se producen errores. Un usuario podría cambiar un puntero de forma que apuntase a algo que está en otra lista. Una forma de garantizar que esto no puede pasar es almacenar la posición actual como parte de la clase `Lista`. Para ello, añadimos un segundo atributo, `actual`. Entonces, puesto que todos los accesos a la lista se realizan a través de los métodos de la clase, podemos estar seguros de que `actual` siempre representa un puntero a un nodo de la lista, un puntero al nodo cabecera o bien es igual a `null`.

Este esquema tiene un problema: puesto que solamente se dispone de una posición, no se soporta el caso en que dos iteradores necesiten acceder independientemente a la lista. Una forma de evitar este problema consiste en definir una *clase iteradora* separada. La clase `Lista` no mantendría entonces ninguna noción de posición actual y solamente contendría los métodos que tratan a la lista como una unidad, como `esVacia` y `vaciar`. Las rutinas que dependen del conocimiento de la posición de la lista en la que nos encontramos se encontrarían en la clase iteradora. Ésta mantiene una noción de la posición actual y el nodo cabecera de la lista que representa (el puntero al nodo cabecera puede ser inicializado por el constructor). El posibilitar el acceso a la lista se consigue haciendo que la clase iteradora pertenezca al mismo paquete.

Podemos ver cada instancia de una clase iteradora como un marco en el que solamente se permiten operaciones legales sobre las listas como, por ejemplo, avanzar en la lista. En este capítulo implementaremos la interfaz `Lista` mediante la clase `ListaEnlazada` y la interfaz `ListaIter` mediante la clase `ListaEnlazadaIter`.

Para ver cómo funciona todo esto, estudiemos el método estático mostrado en la Figura 16.6, que se puede usar en cualquier punto, el cual devuelve la longitud de una lista enlazada. Se declara `itr` como un iterador que puede acceder a la lista enlazada `laLista`. Esto requiere el uso de un constructor para `ListEnlazadaIter` que tome una `ListaEnlazada` como parámetro.

La variable `itr` se inicializa con el primer elemento de `laLista` (saltando por encima del nodo cabecera), pero de cualquier modo llamamos al método `primero`. La llamada al método `estaDentro` refleja el test `p!=null`, que se haría si `p` fuera un puntero normal a un `NodoLista`. Finalmente, la llamada al método `avanzar` refleja la asignación `p=p.siguiente`.

Por tanto, puesto que la clase iteradora define unos pocos métodos simples, podemos iterar sobre las listas de una forma muy natural. La siguiente sección muestra cómo se implementa todo esto en Java. Las rutinas son increíblemente simples.

```
1  // Devuelve el número de elementos que contiene laLista
2  static public int longLista( ListaEnlazada laLista )
3  {
4      int longitud = 0;
5      ListaEnlazadaIter itr = new ListaEnlazadaIter( laLista );
6
7      for( itr.primero( ); itr.estaDentro( ); itr.avanzar( ) )
8          longitud++;
9      return longitud;
10 }
```

Figura 16.6 Método que devuelve la longitud de una lista.

16.2 Implementación en Java

Comenzamos la implementación de las listas enlazadas mostrando el esqueleto de su clase. Implementamos la interfaz descrita en la Sección 6.4. La clase `ListaEnlazada` se muestra en la Figura 16.7. En la línea 28, declaramos el único atributo de la clase, esto es, el puntero al nodo cabecera.

El constructor de la línea 20 asigna memoria al nodo cabecera. Observe que el atributo `dato` del nodo cabecera almacena la referencia `null`. Se declaran dos métodos, `esVacia` y `vaciar`, ambos consistentes en una sola línea. La clase `ListaEnlazada` es simple porque la mayor parte del trabajo se traslada a la clase iteradora.

```
1  package EstructurasDatos;
2
3  // Clase ListaEnlazada
4  //
5  // CONSTRUCCIÓN: sin inicializador
6  // El acceso se realiza a través de la clase ListaEnlazadaIter
7  //
8  // ******************OPERACIONES PÚBLICAS*******************
9  // boolean esVacia( ) --> Devuelve true si es vacía
10 // void vaciar( )     --> Elimina todos los elementos
11 // ************************ERRORES************************
12 // No hay errores especiales
13
14 /**
15  * Implementación de las listas usando listas enlazadas con nodo
16  * cabecera. El acceso a la lista es vía ListaEnlazadaIter.
17  */
18 public class ListaEnlazada implements Lista
19 {
20    public ListaEnlazada( )
21      { cabecera = new NodoLista( null ); }
22    public boolean esVacia( )
23      { return cabecera.siguiente == null; }
24    public void vaciar( )
25      { cabecera.siguiente = null; }
26
27      // Atributos amistosos, luego ListaEnlazadaIter puede acceder
28    NodoLista cabecera;    // Referencia al nodo cabecera
29 }
```

Figura 16.7 Clase `ListaEnlazada`.

Las Figuras 16.8 y 16.9 muestran el esqueleto de `ListaEnlazadaIter`. Ésta es una clase muy interesante que requiere una atención especial a los detalles. Los atributos de la clase son `laLista`, que es una referencia a la lista, y `actual`, que es una referencia al nodo actual. En el constructor se inicializa `laLista`, que después ya no debe cambiar. Luego un iterador está irrevocablemente unido a una sola lista[1].

En las líneas 11 a 16 se declara un constructor. El parámetro es un objeto `ListaEnlazada`, y como resultado de la construcción, `laLista` pasa a referenciar a ese parámetro y `actual` apunta al primer nodo de la lista (saltándose la cabecera, por supuesto). Si la lista es vacía, `actual` apuntará al nodo cabecera, ya que esperamos que la siguiente operación que se produzca sea `insertar`. No hay constructor de cero parámetros, por lo que si intentáramos declarar un objeto `ListaEnlazadaIter` sin un inicializador de lista, se produciría un error en tiempo de compilación. Por conveniencia, en las líneas 24 a 26 se proporciona otro constructor que acepta una `Lista`.

Nos quedan por ver los métodos de la clase. Tenemos que ser muy cuidadosos al explicar lo que hace cada método. Por ejemplo, `insertar` añade el elemento `x` en la posición inmediatamente posterior a `actual` en la lista. ¿Qué sucede si `actual` es `null`? Por supuesto, se produce un error. ¿Cuál es el nuevo valor de `actual`?

[1] Se puede forzar haciendo que `laLista` sea un atributo `final`, ya que Java 1.1 permite la inicialización de un constructores. Sin embargo, en este libro no hemos hecho esta especialización.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase ListaEnlazadaIter: mantiene la "posición actual"
6  //
7  // CONSTRUCCIÓN: con la ListaEnlazada a la que ListaIter
8  //     está permanentemente ligada
9  //
10 // ******************OPERACIONES PÚBLICAS*******************
11 // void insertar( x )    --> Inserta x tras la posición actual
12 // void eliminar( x )    --> Elimina x
13 // boolean buscar( x )   --> Coloca actual para poder ver x
14 // void cero( )          --> Coloca actual antes de la posición 1
15 // void primero( )       --> Coloca actual en la posición 1
16 // void avanzar( )       --> Avanza
17 // boolean estaDentro( ) --> True si es una posición valida
18 // Object recuperar      --> Elemento en la posición actual
19 // ************************ERRORES************************
20 // Excepciones por accesos, inserciones o eliminaciones ilegales.
21
22 /**
23  * Implementación mediante listas enlazadas con nodo cabecera
24  *     del iterador de listas.
25  */
```

Figura 16.8 Sección de comentarios de `ListaEnlazadaIter`.

En nuestro caso, si la inserción ha tenido éxito, `actual` apuntará al nodo recientemente insertado, pero si no, no se modificará. Usamos en cada método comentarios *javadoc* para documentar el comportamiento, incluyendo la especificación de los errores que pueden aparecer y los cambios en el estado del iterador.

Discutamos el resto de los métodos. El método `buscar` busca `x`. Si `x` está en la lista, se mueve `actual` para apuntar al nodo que lo contiene y `buscar` devuelve `true`. En caso contrario, `buscar` devuelve `false` y `actual` no se ve modificado.

El método `eliminar` busca el elemento `x`. Si está en la lista lo elimina y `actual` pasa a apuntar al nodo cabecera. En caso contrario lanza una excepción y `actual` no se ve modificado. Es importante cambiar `actual`, ya que no deseamos en absoluto que apunte a un nodo que ya ha sido eliminado. Una alternativa perfectamente razonable sería que apuntara al nodo anterior al recientemente eliminado. Ninguno de estas dos alternativas es infalible, ya que si hubiera dos iteradores recorriendo la lista y uno de ellos eliminara un nodo, el otro recorrido podría verse afectado negativamente.

Una rutina que en ocasiones (*véase* la Sección 13.1) resulta muy útil, es `eliminarSig`, cuya implementación se deja como Ejercicio 16.9.

El método `estaDentro` se usa para comprobar si la posición actual está en la lista. No altera `actual`. El método `recuperar` devuelve el elemento situado en la posición actual (o bien `null` si la posición actual no es un nodo perteneciente a la lista). Tampoco modifica `actual`. El método `cero` coloca `actual` en el nodo cabecera, permitiendo así las inserciones al principio de la lista. El método `primero` coloca `actual` en el primer nodo de la lista, aunque si ésta es vacía le asignará `null`. Finalmente, `avanzar` es el método que lleva a cabo la iteración. Asigna

```
1  public class ListaEnlazadaIter implements ListaIter
2  {
3      /**
4       * Construye la lista.
5       * Como resultado, la posición actual es el primer
6       * elemento, a menos que la lista sea vacía, en cuyo caso
7       * la posición actual es el elemento 0-ésimo.
8       * @param unaLista un objeto ListaEnlazada a la que este
9       *     iterador está permanentemente ligado.
10      */
11     public ListaEnlazadaIter( ListaEnlazada unaLista )
12     {
13         laLista = unaLista;
14         actual = laLista.esVacia( ) ?
15                  unaLista.cabecera : unaLista.cabecera.siguiente;
16     }
17
18     /**
19      * Este constructor se proporciona por conveniencia. Si
20      * unaLista no es un objeto ListaEnlazada, se lanzará una
21      * excepción ClassCastException. En caso contrario, tiene
22      * el mismo comportamiento que el constructor de arriba.
23      */
24     public ListaEnlazadaIter( Lista unaLista ) throws
25                                                ClassCastException
26       { this( ( ListaEnlazada ) unaLista );}
27
28     /**
29      * Coloca la posición actual en el nodo cabecera.
30      */
31     public void cero( )
32       { actual = laLista.cabecera; }
33
34     public void insertar( Object x ) throws ElementoNoEncontrado
35       { /* Figura 16.12 */ }
36     public boolean buscar( Object x )
37       { /* Figura 16.13 */ }
38     public void eliminar( Object x ) throws ElementoNoEncontrado
39       { /* Figura 16.14 */ }
40     public Object recuperar( )
41       { /* Figura 16.15 */ }
42     public void primero( )
43       { /* Figura 16.15 */ }
44     public void avanzar( )
45       { /* Figura 16.15 */ }
46     public boolean estaDentro( )
47       { /* Figura 16.15 */ }
48
49     protected ListaEnlazada laLista;    // Lista
50     protected NodoLista actual;         // Posición actual
51 }
```

Figura 16.9 Esqueleto para la clase `ListaEnlazadaIter`.

`actual.siguiente` a `actual`, siempre que `actual` no sea `null`, en cuyo caso no sucede nada.

Los métodos `primero`, `avanzar`, `estaDentro` y `eliminar` no lanzan excepciones porque el uso habitual de métodos en el seno de bucles `for` requeriría un bloque `try/catch` que en este caso no está justificado.

Podríamos haber añadido más métodos, pero este conjunto básico es bastante potente. Algunos métodos, como `retroceder`, no están soportados de forma eficiente con esta versión de las listas enlazadas. Más adelante en este capítulo, se estudian otras versiones de la lista enlazada que permiten la implementación en tiempo constante de éste y otros métodos. Por ahora veamos cómo se puede usar la clase `ListaEnlazadaIter` para implementar la copia y la impresión de las listas enlazadas.

En la Figura 16.10 se muestra la implementación de un método de copia de listas. En la línea 5, hacemos una comprobación de alias y terminamos inmediatamente si dicha comprobación devuelve `true`. En caso contrario, llamamos a `vaciar` para vaciar la lista destino. Después declaramos un iterador de listas (`itrIzq`) para recorrer la lista destino `lizq` y un segundo iterador (`itrDer`) para recorrer `lder`. Al principio, `itrIzq` apunta al nodo cabecera, pues la lista está vacía, e `itrDer` apunta al primer nodo de la lista `lder`. En las líneas 14 y 15 se ejecuta un bucle con el que llamamos repetidamente a `insertar` para añadir los elementos de `lder` en `lizq`. Puesto que la posición actual de la lista destino es siempre el último nodo insertado, `lizq` se construye como una copia exacta de `lder`. De una forma prácticamente idéntica, en la Figura 16.11 se usa un bucle para imprimir el contenido de la lista enlazada `laLista`.

```
1  // Método que copia lder en lizq;
2  // Solamente llama a métodos públicos
3  public static void copiar(ListaEnlazada lizq,ListaEnlazada lder )
4  {
5      if( lizq == lder ) // Comprobación de alias
6          return;
7
8      lizq.vaciar( );
9      ListaEnlazadaIter itrIzq = new ListaEnlazadaIter( lizq );
10     ListaEnlazadaIter itrDer = new ListaEnlazadaIter( lder );
11
12     try
13     {
14         for( ; itrDer.estaDentro( ); itrDer.avanzar( ) )
15             itrIzq.insertar( itrDer.recuperar( ) );
16     }
17     catch( ElementoNoEncontrado e ) { } // No puede ocurrir
18 }
```

Figura 16.10 Método de copia para las listas enlazadas, usando `ListaEnlazadaIter` para recorrerlas.

```
1  // Método simple de impresión
2  public static void imprimir( ListaEnlazada laLista )
3  {
4      if( laLista.esVacia( ) )
5          System.out.print( "Lista vacía" );
6      else
7      {
8          ListaEnlazadaIter itr = new ListaEnlazadaIter( laLista );
9          for( ; itr.estaDentro( ); itr.avanzar( ) )
10             System.out.print( itr.recuperar( ) + " " );
11     }
12     System.out.println( );
13 }
```

Figura 16.11 Método de impresión de `ListaEnlazada`.

Ahora que hemos visto qué aspecto tiene `ListaEnlazadaIter` y cómo se usa, podemos implementar sus métodos. La Figura 16.12 muestra el cuerpo de `insertar`. Como hemos dicho anteriormente en el capítulo, si no hiciéramos las comprobaciones de errores la rutina estaría formada por una sola línea de código. La comprobación de errores añade una línea; además, para una mayor claridad, hemos decidido añadir una variable intermedia. Aun así se sigue tratando de un fragmento de código muy reducido.

En la Figura 16.13 se muestra la rutina `buscar`, en la que recorremos la lista elemento a elemento hasta que o bien llegamos al puntero `null` del final de la lista, o bien encontramos el elemento buscado. Observe cuidadosamente que el orden en que se llevan a cabo los tests de la línea 10 es importante porque de esa forma aprovechamos la evaluación no estricta. Como se estudió en la Sección 1.5.2, si la primera mitad del `&&` se evalúa a `false`, entonces el resultado es `false`, y no es necesario ejecutar la segunda mitad.

Análogamente, el método `eliminar` de la Figura 16.14 busca el elemento `x`. Al final del bucle, `itr` o bien apunta al nodo anterior al que contiene `x`, o bien al último nodo, si no se ha encontrado `x`. Entonces se lleva a cabo la eliminación usando el algoritmo descrito previamente. El resto de las rutinas se muestran en la Figura 16.15. Son todas ellas rutinas muy sencillas.

```
1  /**
2   * Inserta tras la posición actual.
3   * actual apunta al nodo insertado con éxito.
4   * @param x el elemento a insertar.
5   * @exception ElementoNoEncontrado si la posición actual es null.
6   */
7  public void insertar( Object x ) throws ElementoNoEncontrado
8  {
9      if( actual == null )
10         throw new ElementoNoEncontrado( "Error de inserción" );
11     NodoLista nuevoNodo = new NodoLista( x, actual.siguiente );
12     actual = actual.siguiente = nuevoNodo;
13 }
```

Figura 16.12 Rutina insertar para la clase `ListaEnlazadaIter`.

```
1  /**
2   * Coloca la posición actual en el primer nodo que contiene
3   * un elemento. Si no encontramos x, actual no se ve modificado.
4   * @param x el elemento a buscar.
5   * @return true si encontramos el elemento x; si no, false .
6   */
7  public boolean buscar( Object x )
8  {
9      NodoLista itr = laLista.cabecera.siguiente;
10     while( itr != null && !itr.dato.equals( x ) )
11         itr = itr.siguiente;
12
13     if( itr == null )
14         return false;
15
16     actual = itr;
17     return true;
18 }
```

Figura 16.13 Rutina buscar para `ListaEnlazadaIter`.

```
1  /**
2   * Elimina la primera aparición de un elemento.
3   * Si aparece en la lista se coloca actual al principio;
4   * pero si no aparece en la lista, no se modifica.
5   * @param x elemento a eliminar.
6   * @exception ElementoNoEncontrado si no se encuentra.
7   */
8  public void eliminar( Object x ) throws ElementoNoEncontrado
9  {
10     NodoLista itr = laLista.cabecera;
11     while( itr.siguiente != null &&
12            !itr.siguiente.dato.equals( x ) )
13         itr = itr.siguiente;
14
15     if( itr.siguiente == null )
16         throw new ElementoNoEncontrado( "Fallo en eliminación" );
17
18     itr.siguiente = itr.siguiente.siguiente;  // Eliminación
19     actual = laLista.cabecera;                // Inicializa actual
20 }
```

Figura 16.14 Rutina eliminar para la clase `ListaEnlazadaIter`.

```
1  /**
2   * Devuelve el elemento almacenado en la posición actual.
3   * @return el elemento almacenado o null si la posición
4   *     actual no está en la lista.
5   */
6  public Object recuperar( )
7  {
8      return estaDentro( ) ? actual.dato : null;
9  }
10
11 /**
12  * Coloca la posición actual en el primer nodo.
13  * Esta operación es válida para listas vacías.
14  */
15 public void primero( )
16 {
17     actual = laLista.cabecera.siguiente;
18 }
19
20 /**
21  * Avanzar la posición actual al nodo siguiente de la lista.
22  * Si la posición actual es null, no hace nada.
23  * En esta rutina no se lanzan excepciones porque su uso más
24  * habitual (dentro de un bucle for) requeriría
25  * que el programador añadiera un bloque try/catch.
26  */
27 public void avanzar( )
28 {
29     if( actual != null )
30         actual = actual.siguiente;
31 }
32
33 /**
34  * Comprueba si la posición actual apunta a un elemento válido.
35  * @return true si la posición actual no es null y no está
36  *     apuntando al nodo cabecera.
37  */
38 public boolean estaDentro( )
39 {
40     return actual != null && actual != laLista.cabecera;
41 }
```

Figura 16.15 Rutinas de iteración para la clase `ListaEnlazadaIter`.

16.3 Listas doblemente enlazadas y listas enlazadas circulares

Como se mencionó en la Sección 16.2, las listas enlazadas simples no soportan de forma eficiente algunas operaciones importantes. Por ejemplo, aunque es fácil ir al principio de la lista, ir al final consume bastante tiempo. Aunque podemos avanzar fácilmente usando `avanzar`, la implementación de `retroceder` no se puede hacer de forma eficiente con una sola referencia `siguiente`. En algunas aplicaciones esto puede ser crítico. Por ejemplo, cuando diseñamos un editor de textos, podemos mantener la imagen interna del fichero como una lista enlazada de líneas. En este caso, desearíamos movernos fácilmente por el fichero tanto hacia abajo como hacia arriba, insertar antes y después de una línea, en lugar de solamente detrás, y poder saltar rápidamente a la última línea. Un momento de reflexión nos sugiere que para implementar estas operaciones de forma eficiente, deberíamos tener dos referencias en cada nodo: una al siguiente nodo de la lista y otro al anterior. Para terminar de hacer simétrica la representación, añadiríamos además un nodo final además del nodo cabecera. Estas listas reciben el nombre de *doblemente enlazadas*. La Figura 16.16 muestra la lista doblemente enlazada que contiene los elementos `a`, `b`. Cada nodo contiene ahora dos referencias (`siguiente` y `anterior`), con lo que se pueden realizar de forma sencilla búsquedas y recorridos en ambos sentidos. Obviamente, son necesarios algunos cambios pequeños pero importantes.

En primer lugar, una lista vacía consiste ahora en un nodo `cabecera` y uno `final` conectados entre sí, como se muestra en la Figura 16.17. Observe que no son necesarios en los algoritmos `cabecera.anterior` y `final.siguiente`, por lo que ni siquiera se inicializan. La comprobación de lista vacía es ahora

```
cabecera.siguiente == final
```

o bien

```
final.anterior == cabecera
```

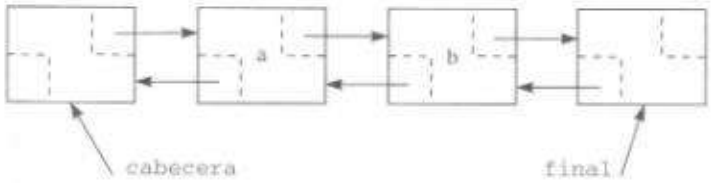


Figura 16.16 Lista doblemente enlazada.

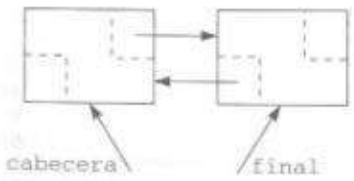


Figura 16.17 Lista doblemente enlazada vacía.

Ya no volvemos a usar `null` para decidir si un avance nos ha llevado hasta el final de la lista, sino que lo sabemos cuando llegamos a `cabecera` o a `final` (recuerde que podemos ir en cualquier sentido). El método `retroceder` se puede implementar mediante

```
actual = actual.anterior;
```

Antes de pasar a algunas de las operaciones adicionales que están disponibles, veamos cómo cambian las operaciones de inserción y eliminación. Naturalmente ahora podemos hacer `insertarAntes` e `insertarDespues`. En `insertarDespues` se hallan implicados el doble de cambios de punteros en las listas doblemente enlazadas que en las listas enlazadas simples. Si escribimos cada instrucción de forma explícita, obtenemos

```
nNodo = new NodoListaEnlazadaDoble( );
nNodo.anterior = actual;                  // Asigna valor a x.anterior
nNodo.siguiente = actual.siguiente;       // Asigna valor a x.siguiente
nNodo.anterior.siguiente = nNodo;         // Asigna valor a a.siguiente
nNodo.siguiente.anterior = nNodo;         // Asigna valor a b.anterior
actual = nNodo;
```

Como se ha mostrado antes en el capítulo, los dos primeros cambios de referencias se podrían juntar en el constructor. En la Figura 16.18 se ilustran los cambios de `anterior` y `siguiente` (siguiendo el orden *1, 2, 3, 4*).

La Figura 16.18 también se puede usar para guiarnos en el algoritmo de eliminación. Al contrario que en las listas enlazadas simples, ahora podemos eliminar el nodo actual porque tenemos acceso inmediato al nodo anterior. Por tanto, para `eliminar x` en la Figura 16.18, tenemos que cambiar la referencia `siguiente` de `a` y la `anterior` de `b`. Los cambios básicos son

```
actual.anterior.siguiente = actual.siguiente;
                                          // Actualiza a.siguiente
actual.siguiente.anterior = actual.anterior;
                                          // Actualiza b.anterior
actual = cabecera;
```

Para realizar una implementación completa de una lista doblemente enlazada, necesitamos decidir qué operaciones se van a soportar. Es razonable esperar que haya el doble de operaciones que en las listas enlazadas simples. Cada procedi-

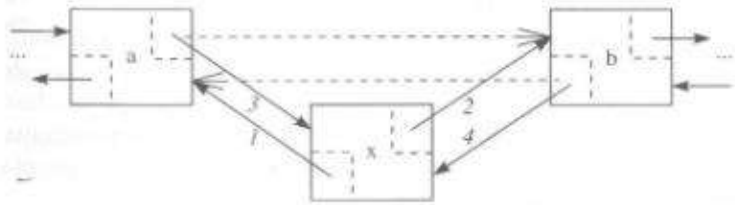


Figura 16.18 Inserción en una lista doblemente enlazada mediante la construcción de un nuevo nodo y la modificación posterior de las referencias, en el orden indicado.

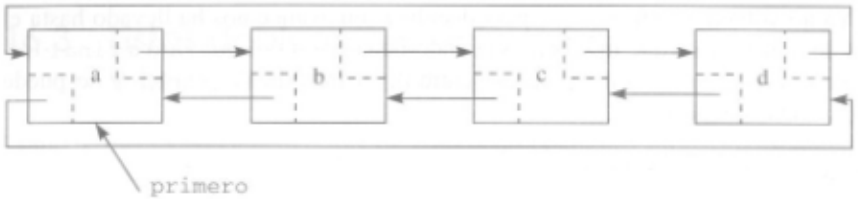


Figura 16.19 Lista circular doblemente enlazada.

miento individual será muy similar a las rutinas de las listas enlazadas simples, sólo las operaciones dinámicas implican cambios adicionales en las referencias. Para la mayoría de las rutinas, el código está dominado por las comprobaciones de error. Aunque algunas de ellas cambian, por ejemplo ya no comparamos con la referencia `null`, éstas no se hacen más complejas. Dejamos la implementación de la clase como práctica de programación en el Ejercicio 16.14.

Una práctica habitual es crear una lista enlazada circular, en la que el último nodo apunta al primero. Esto se puede hacer con o sin cabecera, aunque normalmente se hace sin ella, puesto que el propósito principal de la cabecera es asegurar que cada nodo tiene uno anterior, lo cual ya es cierto para una lista enlazada circular no vacía. Sin cabecera solamente tendremos que tratar como un caso especial la lista vacía. Se mantiene una referencia al primer nodo, pero éste ya no es el nodo cabecera. Podemos usar simultáneamente listas enlazadas circulares y listas doblemente enlazadas, como se muestra en la Figura 16.19. La lista circular es útil en situaciones en las que queremos que las búsquedas permitan circularidad, como por ejemplo en algunos editores de texto en los que una vez que se llega al final del fichero, se sigue buscando desde el principio hasta el punto en el que nos encontrábamos. En el Ejercicio 16.16 se le pide implementar una lista circular doblemente enlazada.

16.4 Listas enlazadas ordenadas

A veces desearíamos guardar los elementos de forma ordenada en la lista enlazada. La diferencia fundamental entre una lista enlazada ordenada y una no ordenada es la rutina de inserción. De hecho, podemos obtener una clase de listas ordenadas limitándonos a modificar la rutina de inserción que ya tenemos escrita. El hecho de que la rutina `insertar` sea parte de la clase `ListaEnlazadaIter` nos lleva a pensar que deberíamos tener una nueva clase `ListaIterOrd` basada en ella, lo cual se muestra en la Figura 16.20.

El constructor, mostrado en las líneas 18 y 19, simplemente llama al constructor de la superclase. El nuevo método `insertar` requiere que `x` sea de la clase `Comparable`, pero no sobrescribe el método `insertar` de la clase `ListaEnlazadaIter` porque tiene un valor de la clase `Object` como parámetro. En las líneas 21 a 28 proporcionamos una definición que sí lo sobrescribe e intenta llamar al método `insertar` en orden.

Como ejemplo, la Figura 16.21 ilustra llamadas a varios métodos `insertar`. Los métodos `insertar` que tienen un manejador de excepciones son los que esperan un valor de la clase `Object` como parámetro. La llamada de la línea 7, a la

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4  import Soporte.*; import Soporte.Comparable;
5
6  // Clase ListaIterOrd; mantiene la "posición actual"
7  //
8  // CONSTRUCCIÓN: con una ListaEnlazada a la que la
9  //     ListaIterOrd está permanentemente ligada
10 // ******************OPERACIONES PÚBLICAS*********************
11 // void insertar( x )     --> Inserta x en orden
12 // El resto de operaciones idénticas a las de ListaEnlazadaIter
13 // ***************************ERRORES*************************
14 // Excepciones lanzadas por acceso o eliminación ilegal.
15
16 public class ListaIterOrd extends ListaEnlazadaIter
17 {
18     public ListaIterOrd ( Lista unaLista )
19       { super( unaLista ); }
20
21     public void insertar( Object x ) throws ElementoNoEncontrado
22     {
23         if( x instanceof Comparable )
24             insertar( (Comparable) x );
25         else
26             throw new ElementoNoEncontrado( "ListaIterOrd " +
27             " insertar requiere un objeto de tipo Comparable" );
28     }
29
30         // Inserta de forma ordenada.
31     public void insertar( Comparable x )
32     {
33         ListaEnlazadaIter ant = new ListaEnlazadaIter( laLista );
34         ListaEnlazadaIter act = new ListaEnlazadaIter( laLista );
35
36         ant.cero( );
37         act.primero( );
38         while( act.estaDentro( ) &&
39              ( (Comparable)( act.recuperar( ) )).menorQue( x ) )
40         {
41             act.avanzar( );
42             ant.avanzar( );
43         }
44
45         try
46           { ant.insertar( x ); }
47         catch( ElementoNoEncontrado e ) { } // No puede ocurrir
48         actual = ant.actual;
49     }
50 }
```

Figura 16.20 Clase iteradora para las listas enlazadas ordenadas en la que las inserciones deben respetar el orden.

que se le pasa como parámetro una referencia `Comparable` estáticamente tipada, es la única llamada directa al método que acepta un valor de la clase `Comparable`. La llamada de la línea 9 lanzará una excepción, ya que `obj` apunta a un valor de

```
1  public static void main( String [ ] args )
2  {
3      MiEntero m = new MiEntero( 5 );
4      Object obj = new Integer( 5 );
5      ListaIterOrd itr = new ListaIterOrd( new ListaEnlazada( ) );
6
7      itr.insertar( m ); // Llama a insertar( Comparable )
8
9      try { itr.insertar( obj ); }
10     catch( ElementoNoEncontrado e )
11       { System.out.println( "Elemento1 no Comparable" ); }
12
13     obj = m;
14     try { itr.insertar( obj ); }
15     catch( ElementoNoEncontrado e )
16       { System.out.println( "Elemento2 no Comparable" ); }
17 }
```

Figura 16.21 Programa que ilustra qué método `insertar` se está usando.

tipo `Integer`, el cual no implementa la interfaz `Comparable`. La llamada de la línea 14 invoca al método `insertar` que acepta un valor de la clase `Object`, ya que aunque en tiempo de ejecución es cierto que `obj` apunta a un objeto de la clase `Comparable`, es el tipo estático el usado para decidir a qué método se está llamando. Aun así, ese método `insertar` llamará al método `insertar` que procesa valores de la clase `Comparable` (línea 24 de la Figura 16.20). Ésta es la razón por la que sobrescribimos el método `equals` con un `Object` como parámetro.

El código del nuevo método `insertar`, mostrado en las líneas 31 a 49 de la Figura 16.20, usa dos objetos `ListaIterOrd` para recorrer la lista correspondiente hasta que se encuentra el punto correcto de inserción. En ese punto, podemos aplicar la rutina `insertar` de `ListaEnlazadaIter`. Para usar objetos de la clase `ListaIterOrd`, debemos saber a qué objeto de la clase `Lista` se refieren. Se trata de `laLista`, que es un atributo protegido de la clase `ListaEnlazadaIter` y por tanto es accesible.

¿Necesitamos realmente usar `ListaEnlazadaIter` en la nueva rutina `insertar`? La respuesta es no, puesto que tanto `actual` como `laLista` son accesibles. El Ejercicio 16.17 le pide rescribir el código para evitar usar `ListaEnlazadaIter`.

Una deficiencia de nuestro esquema es que se puede acceder a cualquier lista usando métodos tanto de `ListaEnlazadaIter` como de `ListaIterOrd`, por lo que no tenemos garantías de que los métodos de `ListaIterOrd` estén trabajando sobre una lista ordenada. Podemos arreglar la situación añadiendo un atributo extra a la clase `ListaEnlazada` que especifique si la lista está ordenada o no. Este atributo podría inicializarse en la primera llamada a `insertar`, y las siguientes llamadas verificarían que no estamos mezclando los dos tipos

Resumen

En este capítulo se han implementado las listas enlazadas usando una clase iteradora. Nuestra clase iteradora `ListaIter` trabaja con la clase `Lista`. Cualquier operación que dependa de alguna manera del posicionamiento en la lista se define en la clase `ListaIter`. La clase `Lista` define solamente aquellas operaciones que ven a las listas como una entidad (por ejemplo, `esVacia`). La clase `ListaIter` mantiene una noción de posición actual que se actualiza al avanzar, buscar, insertar e incluso eliminar en la lista. El capítulo ha examinado también variaciones de las listas enlazadas, incluyendo las listas doblemente enlazadas, que permiten el recorrido de la lista en ambos sentidos. Finalmente, hemos visto que es relativamente fácil derivar una clase de listas enlazadas ordenadas a partir de la clase base de listas enlazadas.

Elementos del juego

clase iteradora Clase que mantiene la posición actual y contiene todas las rutinas que dependen del conocimiento de la posición de la lista en que nos encontramos.

lista doblemente enlazada Lista enlazada que permite recorridos en ambos sentidos mediante el almacenamiento de dos referencias por nodo.

lista enlazada circular Lista enlazada en la que la referencia siguiente del último nodo apunta al primero. Esto es útil en tareas que implican circularidad.

lista enlazada ordenada Lista enlazada en la que los elementos están ordenados. Se deriva una clase `ListaIterOrd` a partir de `ListaEnlazadaIter`.

nodo cabecera Nodo extra en la lista enlazada que no almacena ningún dato pero que sirve para satisfacer el requerimiento de que cada nodo tenga uno anterior. Este nodo nos permite evitar los casos especiales, como la inserción de un nuevo primer elemento y la eliminación del primer elemento de la lista.

Errores comunes

1. El error más común en las listas enlazadas consiste en conectar incorrectamente los nodos al llevar a cabo una inserción. La conexión es especialmente delicada cuando se trabaja con listas doblemente enlazadas.
2. Los métodos de una clase no deberían permitir desreferenciar un puntero `null`.
3. Se pueden producir problemas cuando hay varios iteradores accediendo a una misma lista simultáneamente. Por ejemplo, ¿qué sucedería si un iterador elimina un nodo al que otro iterador va a acceder? Resolver este tipo de problemas requiere un trabajo adicional.
4. Tenga cuidado al sobrescribir un método con otro cuyos parámetros sean un subtipo de los originales (como es el caso de `equals`).

En Internet

La clase de listas enlazadas, incluyendo las listas enlazadas ordenadas, está disponible en el directorio **DataStructures**. En el directorio **Chapter16** se proporciona un programa de prueba que también incluye los métodos de la Figura 16.6 y de la Figura 16.11, y el programa de la Figura 16.21.

LinkedList.java	Contiene la implementación mediante listas enlazadas de las listas (traducida por `ListaEnlazada`).
LinkedListItr.java	Contiene la implementación mediante listas enlazadas de un iterador para las listas (traducido por `ListaEnlazadaIter`).
SortListItr.java	Contiene la implementación de un iterador para listas ordenadas (traducido por `ListaIterOrd`).
TestList.java	Contiene un programa de prueba para las listas.
WhichInsert.java	Contiene la rutina de la Figura 16.21.

Ejercicios

Cuestiones breves

16.1. Dibuje una lista enlazada vacía con nodo cabecera.
16.2. Dibuje una lista doblemente enlazada vacía con nodo cabecera y final.

Problemas teóricos

16.3. Escriba un algoritmo para imprimir de forma inversa una lista enlazada simple, usando solamente un espacio extra constante, lo cual implica que no puede usar recursión.
16.4. Discuta sobre si sería mejor convertir `primero` en ilegal sobre las listas vacías.

Problemas prácticos

16.5. Modifique la rutina `buscar` de la clase `ListaEnlazadaIter` para devolver la última aparición del elemento `x`.
16.6. Mirar el contenido del siguiente objeto en una `ListaEnlazadaIter` requiere la aplicación de `avanzar`. En algunos casos, puede ser preferible mirar el elemento siguiente de la lista, sin avanzar. Escriba el método con la siguiente declaración que realiza esto en el caso general. El método `vistazo` devuelve una `ListaEnlazadaIter` que se corresponde con las `k` posiciones siguientes a `actual`.

```
ListaEnlazadaIter vistazo( int k );
```

16.7. Explique lo que le pasará al método añadido en el Ejercicio 16.6 cuando se derive la clase `ListaIterOrd` a partir de `ListaEnlazadaIter`.
16.8. Modifique el método `eliminar` de la clase `ListaEnlazadaIter` de forma que se eliminen todas las apariciones de `x`.
16.9. Añada la rutina `eliminarSig` a la clase `ListaEnlazadaIter`. Él mismo eliminará el elemento situado tras la posición actual. ¿Cómo se tratan los posibles errores?
16.10. Implemente una clase `Pila` eficiente usando una `ListaEnlazada` como atributo. Necesitará un iterador `ListaEnlazadaIter`, que puede ser o bien un atributo o una variable local en aquellas rutinas que lo necesiten.
16.11. Implemente una clase `Cola` eficiente usando (como en el Ejercicio 16.10):

a) Una lista enlazada e iteradores apropiados. ¿Cuántos de estos iteradores deben ser atributos para conseguir una implementación eficiente?
b) Una lista doblemente enlazada e iteradores apropiados.

16.12. Implemente `retroceder` para las listas enlazadas simples. Observe que tardará un tiempo lineal.
16.13. Implemente las listas enlazadas sin nodo cabecera.

Prácticas de programación

16.14. Implemente una clase de listas doblemente enlazadas, incluyendo:

a) Métodos `irAPrincipio` e `irAFinal`.
b) Métodos `primero` y `ultimo`.
c) Métodos `insertarAntes` e `insertarDespues`.
d) Métodos `buscarPrimero` y `buscarUltimo` (la búsqueda comienza desde el principio y el final, respectivamente).
e) Métodos `eliminarPrimero` y `eliminarUltimo`

16.15. Escriba un editor de líneas. La sintaxis de los comandos es similar al editor de líneas *ed* de Unix. Se mantiene una copia interna del fichero como una lista enlazada de líneas. Para poder moverse hacia arriba y hacia abajo en el fichero, necesitará una lista doblemente enlazada. La mayor parte de los comandos, los cuales se muestran en la Figura 16.22, están representados mediante una cadena de un solo carácter, aunque algunos están formados por dos caracteres y requieren un argumento (o más).
16.16. Implemente una lista circular doblemente enlazada.
16.17. Diseñe de nuevo `ListaEnlazada`, `ListaEnlazadaIter` y `ListaIterOrd` para no permitir el intercalamiento de comandos `insertar` procedentes de `ListaEnlazadaIter` y `ListaIterOrd`.
16.18. Si el orden en que se almacenan los elementos en una lista no es importante, entonces podemos acelerar las búsquedas usando la siguiente heurística, conocida como *mover-al-frente*: siempre que acceda a un elemento, muévalo al principio de la lista. La razón por la que esto constituye una mejora, es que los elementos a los que se accede frecuentemente tienden a migrar hacia el principio de la lista, mientras que los menos frecuentemente accedidos migran hacia el final de la lista. En consecuencia, los elementos más frecuentemente accedidos requieren una búsqueda más breve. Implemente la heurística mover-al-frente para las listas enlazadas.
16.19. Escriba rutinas `union` e `interseccion` que devuelvan la unión e intersección de dos listas enlazadas. Asuma que las listas de entrada están ordenadas.

Comando	Función
1	Ir al principio.
a	Añadir texto tras la línea actual hasta encontrar un punto(.).
d	Borrar línea actual.
dr num num	Borrar varias líneas.
f name	Cambiar el nombre del fichero actual (para la siguiente escritura).
g num	Ir a una línea.
h	Obtener ayuda.
i	Como concatenar, pero añade las líneas antes de la línea actual.
m num	Mover la línea actual después de otra línea.
mr num num num	Mover varias líneas como una unidad después de otra línea.
n	Activar y desactivar la numeración de líneas.
p	Imprimir la línea actual.
pr num num	Imprimir varias líneas.
q!	Salir sin guardar.
r name	Leer y pegar otro fichero en el actual.
s text text	Sustituir un texto por otro.
t num	Copiar la línea actual después de otra línea.
tr num num num	Copiar varias líneas después de otra línea.
w	Guardar fichero en disco.
x!	Salir guardando.
$	Ir a la última línea.
-	Subir una línea.
+	Bajar una línea.
=	Imprimir el número actual de línea.
/ text	Buscar hacia delante un patrón.
? text	Buscar hacia atrás un patrón.
#	Imprimir el número de líneas y caracteres del fichero.

Figura 16.22 Comandos para el editor del Ejercicio 16.15.

CAPÍTULO

17 Árboles

El árbol es una estructura fundamental en la computación. Casi todos los sistemas operativos almacenan los ficheros en árboles o en estructuras de aspecto arbóreo. Los árboles se usan también en diseño de compiladores, procesamiento de textos y algoritmos de búsqueda. En el Capítulo 18 se discute esta última aplicación.

En este capítulo veremos:

- Una definición de árbol general y una discusión sobre cómo se usa en un sistema de ficheros.
- Un estudio de los *árboles binarios*.
- Cómo se implementan las operaciones sobre los árboles usando recursión.
- Cómo se recorre un árbol de forma no recursiva.

17.1 Árboles generales

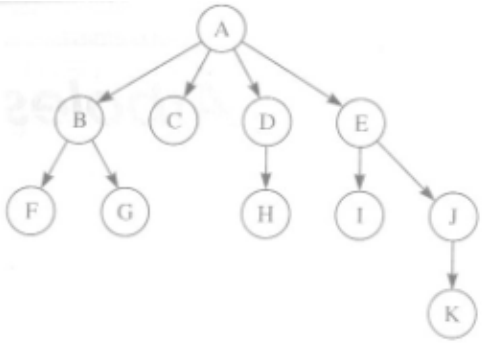
Los árboles se pueden definir de dos formas: recursiva y no recursivamente. La definición no recursiva es la más directa, por lo que comenzamos por ella. La formulación recursiva nos permite escribir algoritmos simples para manipular árboles.

17.1.1 Definiciones

Recuerde, del Capítulo 6, que un *árbol* consiste en un conjunto de nodos y otro de aristas orientadas que conectan pares de nodos. A lo largo de este libro, solamente consideramos árboles con raíz. Un árbol con raíz tiene las siguientes propiedades:

- Se distingue un nodo como raíz.
- A cada nodo c, exceptuando la raíz, le llega una arista desde exactamente otro nodo p, al cual se le llama *padre* de c. Decimos que c es uno de los *hijos* de p.
- Hay un único camino desde la raíz hasta cada nodo. El número de aristas del mismo recibe el nombre de *longitud del camino*.

La Figura 17.1 muestra un árbol. El nodo raíz es A, y los hijos de A son B, C, D y E. Como A es la raíz, no tiene padre. Todos los demás nodos tienen padre; por



Nodo	Altura	Profundidad
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

Figura 17.1 Un árbol, con información de altura y profundidad.

ejemplo, el padre de B es A. Los nodos que no tienen hijos reciben el nombre de *hojas*. Las hojas de este árbol son C, F, G, H, I y K. La longitud del camino de A a K es tres (aristas), y la del camino de A a A es de cero aristas.

Un árbol con N nodos debe tener $N - 1$ aristas porque a cada nodo, excepto a la raíz, le llega una arista. La *profundidad* de un nodo en un árbol es la longitud del camino que va desde la raíz hasta ese nodo. En consecuencia, la profundidad de la raíz es siempre 0, y la de cualquier nodo es la de su padre más uno. La *altura* de un nodo es la longitud del camino que va desde el nodo hasta la hoja más profunda bajo él. Luego la altura de E es 2. La altura de cualquier nodo es 1 más que la mayor altura de un hijo suyo. La altura de un árbol se define como la altura de su raíz.

Los nodos que tienen el mismo padre son *hermanos*, luego B, C, D y E son todos hermanos. Si hay un camino del nodo u al nodo v, entonces decimos que u es un *ascendiente* de v y que v es un *descendiente* de u. Si $u \neq v$, entonces u es un *ascendiente propio* de v y v es un *descendiente propio* de u. El *tamaño* de un nodo es igual al número de descendientes que tiene (incluyendo dicho nodo). El tamaño de B es 3, y el de C es 1. El tamaño de un árbol se define como el tamaño de su raíz, luego el tamaño del árbol de la Figura 17.1 es igual al tamaño de A, que es 11.

Se puede dar una definición alternativa de los árboles en forma recursiva: un árbol es o bien vacío o consiste en una raíz y cero o más subárboles no vacíos T_1, T_2, ..., T_k, cada una de cuyas raíces está conectada por medio de una arista con la

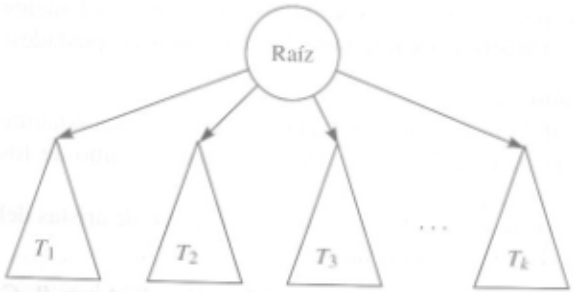


Figura 17.2 Un árbol visto de forma recursiva.

raíz. Una consecuencia de esta definición es que cada nodo de un árbol define un subárbol, que es aquel que le tiene por raíz. Se tiene así una identificación entre nodo y subárbol, que hemos utilizado más arriba al definir el tamaño de un nodo. En algunas clases de árboles, como los *árboles binarios* que se estudian más adelante en el capítulo, se puede permitir que algunos de los subárboles sean vacíos. Esta visión de los árboles se ilustra en la Figura 17.2.

17.1.2 Implementación

Una forma de implementar un árbol consistiría en tener en cada nodo una referencia a cada uno de sus hijos además de a los datos que contiene. Pero debido a que el número de hijos puede variar mucho y dicho número no se conoce a priori, puede que el hecho de conectar a los hijos directamente con el padre en la estructura de datos sea impracticable por la gran cantidad de espacio necesario. La solución es simple: mantener los hijos de cada nodo en una lista enlazada de nodos. Esto hace que cada nodo guarde dos referencias: una a su hijo más a la izquierda (siempre que no sea ya una hoja) y otra a su hermano de la derecha (siempre que no sea ya el situado más a la derecha). A este tipo de implementación se le llama *método primer hijo/siguiente hermano* y se ilustra en la Figura 17.3. Las flechas que apuntan hacia abajo son referencias al `primerHijo`, y las que van de izquierda a derecha son referencias al `siguienteHermano`. Las referencias `null` no se muestran en el dibujo pues hay demasiadas, y pueden deducirse del mismo. En este árbol, el nodo B tiene una referencia a su hermano C y a su primer hijo F, mientras que otros nodos poseen solamente una de estas referencias y algunos de ellos ninguna de las dos. Dada esta representación, es un ejercicio sencillo escribir una clase de árboles que implemente la interfaz de la Figura 6.15.

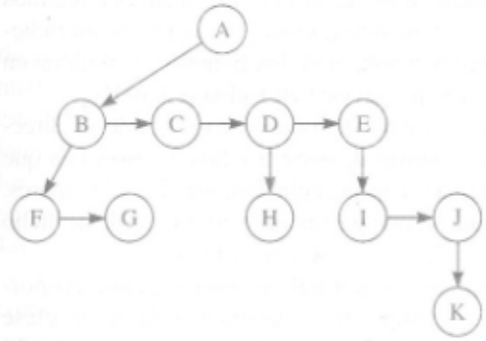


Figura 17.3 Representación primer hijo/siguiente hermano del árbol de la Figura 17.1.

17.1.3 Una aplicación: sistemas de ficheros

Hay muchas aplicaciones de los árboles. Una de las más conocidas es la estructura de directorios en muchos de los sistemas operativos más utilizados, como Unix, VAX/VMS y DOS. La Figura 17.4 muestra un directorio típico en el sis-

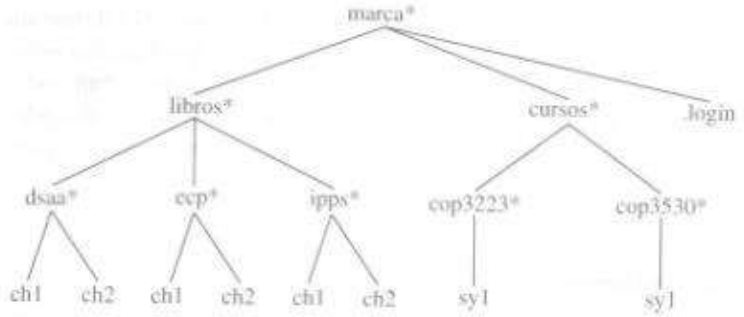


Figura 17.4 Directorio Unix.

junto al nombre indica que `marca` es un directorio.) El directorio `marca` tiene tres hijos —`libros`, `cursos` y `.login`— dos de los cuales son a su vez directorios. Luego `marca` contiene dos subdirectorios y un fichero. El nombre de fichero `marca/libros/dsaa/ch1` se obtiene tomando tres veces el hijo más a la izquierda. Cada `/` indica una arista, produciendo así un nombre de camino. Si el camino comienza en la raíz del sistema completo de ficheros, en lugar de en un directorio arbitrario dentro del sistema, entonces tenemos un nombre de camino completo; en caso contrario tenemos un nombre de camino relativo al directorio actual.

Este sistema de ficheros jerárquico es muy popular porque permite a los usuarios organizar sus datos de una forma estructurada y sencilla. Además, se pueden almacenar dos ficheros con el mismo nombre en distintos directorios, ya que por accederse a ellos a través de caminos distintos desde la raíz, sus nombres de camino completo son distintos. Un directorio en el sistema de ficheros Unix es simplemente un fichero con una lista de todos sus hijos[1], por lo que se pueden recorrer los directorios usando un esquema de iteración que implemente la clase abstracta de la Figura 6.15, es decir, podemos iterar sobre cada hijo. De hecho, en algunos sistemas, si a un directorio se le aplica el comando normal para imprimir un fichero, entonces se pueden ver en la salida los nombres de los ficheros contenidos en ese directorio (junto con otra información que no está en formato ASCII).

Supongamos que queremos listar los nombres de todos los ficheros de un directorio (incluyendo sus subdirectorios). El formato de nuestra salida consistirá en que los nombres de los ficheros de profundidad d aparecerán sangrados d tabuladores. En la Figura 17.5 se muestra un algoritmo simple para hacer esto. La salida de dicho algoritmo para el directorio de la Figura 17.4 se muestra en la Figura 17.6.

Por ahora usaremos pseudocódigo, pero al final de esta sección proporcionamos una implementación en Java. Suponemos la existencia de la clase `SistemaFichero` y de dos métodos, `imprimirNombre` y `esDirectorio`. El método `imprimirNombre` produce como salida el objeto `SistemaFichero` actual con una sangría de d tabuladores, y `esDirectorio` comprueba si el objeto `SistemaFichero` actual es un directorio, en cuyo caso devuelve `true`. Usando estos dos métodos se puede escribir fácilmente la rutina recursiva `listar`, a la cual tenemos que pasarle el parámetro `prof`, que indica el nivel actual del directorio relativo a la raíz, es decir, la profundidad a la que nos encontramos. En la llamada inicial a `listar`, el parámetro `prof` toma el valor 0, lo que significa ausencia de sangría al mostrar la raíz. Este parámetro de profundidad es una variable interna y difícilmente una rutina que llamara a `listar` tendría que conocer su valor, esto es, considerarlo como un parámetro. Por ello el pseudocódigo especifica para ella un valor por defecto de 0 (no es legal especificar un valor por defecto en Java).

[1] el sistema de ficheros de Unix también tiene una entrada (`.`) que hace referencia a sí mismo y otra (`..`) que del directorio. Esto introduce un ciclo, por lo que técnicamente, el sistema de ficheros de Unix no es un árbol estructura semejante a un árbol.

```
1 void listar( int prof = 0 ) // La profundidad es inicialmente 0
2 {
3     imprimirNombre( prof ); // Imprime el nombre del objeto
4     if ( esDirectorio( ) )
5         for cada fichero c en este directorio (para cada hijo)
6             c.listar( prof + 1 );
7 }
```

Figura 17.5 Pseudocódigo para listar el contenido de un directorio y sus subdirectorios en un sistema jerárquico de ficheros.

```
marca
    libros
        dsaa
            ch1
            ch2
        ecp
            ch1
            ch2
        ipps
            ch1
            ch2
    cursos
        cop3223
            sy1
        cop3530
            sy1
        .login
```

Figura 17.6 Listado del contenido del directorio representado por el árbol de la Figura 17.4.

Es bastante sencillo seguir la lógica del algoritmo: se imprime el objeto actual con el sangrado adecuado y si la entrada es un directorio, procesamos recursivamente todos sus hijos, uno a uno. Estos hijos son un nivel más profundos y por tanto se debe añadir al sangrado un tabulador adicional. Al efecto hacemos la llamada recursiva con `prof + 1`. Es difícil imaginar un fragmento de código más simple que éste para llevar a cabo lo que a priori parece una tarea bastante compleja.

Esta técnica algorítmica se conoce como *recorrido en preorden*, en el cual se lleva a cabo el trabajo sobre el nodo antes de procesar los hijos. Además de ser un algoritmo compacto, se trata de un algoritmo con coste en tiempo lineal, lo cual tiene su mérito. Por qué esto es en efecto así, se discutirá más adelante en el capítulo.

Otro método habitual para recorrer un árbol es el *recorrido en postorden*, en el cual se lleva a cabo el trabajo sobre un nodo después de haber procesado sus hijos. Como ejemplo, la Figura 17.7 representa la misma estructura de directorio que la de la Figura 17.4. Los números entre paréntesis representan el número de bloques de disco que ocupa cada fichero. Puesto que los directorios son ficheros, también usan bloques de disco (para almacenar los nombres de sus hijos e información sobre ellos).

Supongamos que queremos calcular el número total de bloques que ocupan todos los ficheros en el árbol. La forma más natural de hacer esto es buscando el número total de bloques usados por todos sus hijos (los cuales pueden ser directorios que deben evaluarse recursivamente): `libros` (41), `cursos` (8) y `.login` (2). El número total de bloques es entonces el total en todos los hijos más los bloques usados en la raíz (1), es decir 52. La rutina `tamanyo` de la Figura 17.8 implementa esta estrategia. Si el objeto `SistemaFichero` actual no es un directorio, `tamanyo` simplemente devuelve el número de bloques que utiliza. En caso contrario, se añade el número de bloques del directorio actual al número de bloques encontrados (recursivamente) en todos los hijos. Para ilustrar la diferencia entre los recorridos en preorden y en postorden, en la Figura 17.9 se muestra cómo el algoritmo va produciendo el tamaño de cada directorio (o fichero). En ella vemos que se ha usado un recorrido en postorden clásico porque el tamaño total de un directorio no se conoce hasta que se conocen los de sus hijos. De nuevo el tiempo de ejecución es lineal. En la Sección 17.4 se puede encontrar mucha más información sobre el recorrido de árboles.

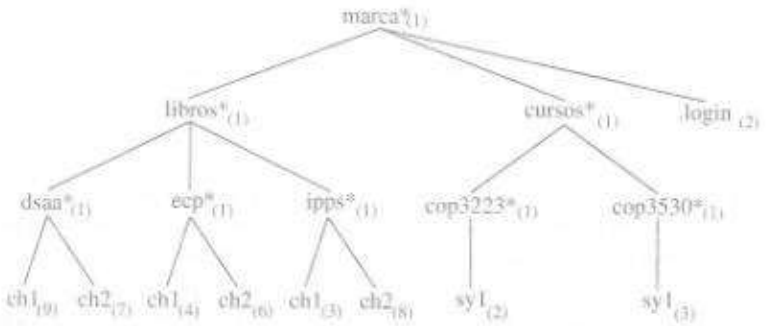


Figura 17.7 Directorio Unix con los tamaños de los ficheros.

```
1  int tamanyo( )
2  {
3      int tamtotal = sizeOfThisFile( );
4
5      if ( isDirectory( ) )
6          for cada fichero en este directorio (para cada hijo)
7              tamtotal += c.tamanyo( );
8
9      return tamtotal;
10 }
```

Figura 17.8 Pseudocódigo para calcular el tamaño total de todos los ficheros de un directorio.

```
                ch1                 9
                ch2                 7
            dsaa                   17
                ch1                 4
                ch2                 6
            ecp                    11
                ch1                 3
                ch2                 8
            ipps                   12
        libros                     41
                sy1                 2
            cop3223                 3
                sy1                 3
            cop3530                 4
        cursos                      8
        .login                      2
marca                              52
```

Figura 17.9 Traza del método `tamanyo`.

Implementación en Java

Java proporciona una clase `File` contenida en el paquete `java.io` que se puede usar para recorrer jerarquías de directorios. Podemos usarla para implementar el pseudocódigo de la Figura 17.5, así como el método `tamanyo`, lo cual se hace en el código disponible en la red. La clase `File` proporciona varios métodos útiles.

Se puede construir un objeto `File` proporcionando un nombre. El método `getName` proporciona el nombre de un objeto `File`. No incluye la parte del camino correspondiente a los directorios, la cual se obtiene usando `getPath`. El método `isDirectory` devuelve `true` si el objeto `File` es un directorio, y `length` devuelve su tamaño en bytes. Si el fichero es un directorio, el método `list` devuelve un vector de valores de tipo `String` que representan los nombres de los ficheros en el directorio (sin incluir `.` ni `..`).

Para implementar el objeto `SistemaFichero` descrito en el pseudocódigo, simplemente extendemos la clase `File` y proporcionamos un constructor, `imprimirNombre` y `listar`, ya que el método `esDirectorio` que necesitábamos ya nos lo proporciona `File` mediante `isDirectory`. Esto se muestra en la Figura 17.10. La única parte difícil son las líneas 36 y 37, donde debemos construir el objeto `SistemaFichero hijo`. El nombre completo del fichero consiste en el nombre del directorio seguido por un separador (`/` en Unix; `\` en DOS) y seguido por el nombre del fichero. Se proporciona también una rutina `main` simple.

17.2 Árboles binarios

Un *árbol binario* es un árbol en el que ningún nodo puede tener más de dos hijos. Puesto que solamente hay dos hijos, podemos llamarles una vez distinguidos uno del otro `izquierdo` y `derecho`. La definición recursiva nos dice en esta ocasión que un árbol binario es o bien vacío o consta de una raíz, un árbol (hijo) izquierdo y otro derecho. Los hijos izquierdo y derecho pueden ser a su vez vacíos, de modo que un nodo con un solo hijo podría tener un hijo izquierdo o un hijo derecho.

```
1  import java.io.*;
2
3  public class SistemaFichero extends File
4  {
5      // Constructor
6      public SistemaFichero( String nombre )
7      {
8          super( nombre );
9      }
10
11     // Imprime el nombre del fichero con sangrado
12     public void imprimirNombre( int prof )
13     {
14         for( int i = 0; i < prof; i++ )
15             System.out.print( "\t" );
16         System.out.println( getName( ) );
17     }
18
19     // Método guía público para listar los ficheros
20     public void listar( )
21     {
22         listar( 0 );
23     }
24
25     // Método recursivo para listar los ficheros de un directorio
26     private void listar( int prof )
27     {
28         imprimirNombre( prof );
29
30         if( isDirectory( ) )
31         {
32             String [ ] hijos = list( );
33
34             for( int i = 0; i < hijos.length; i++ )
35             {
36                 SistemaFichero hijo = new SistemaFichero( getPath( )
37                                       + separatorChar + hijos[ i ] );
38                 hijo.listar( prof + 1 );
39             }
40         }
41     }
42
43     // Rutina main para listar los ficheros del directorio actual
44     public static void main( String [ ] args )
45     {
46         SistemaFichero f = new SistemaFichero( "." );
47         f.listar( );
48     }
49 }
```

Figura 17.10 Implementación en Java del listado de un directorio.

Usaremos repetidamente la definición recursiva en el diseño de algoritmos sobre árboles binarios. Éstos tienen un gran número de usos importantes, dos de los cuales se ilustran en la Figura 17.11.

Figura 17.11 Usos de los árboles binarios: a la izquierda un árbol de expresión y a la derecha un árbol de codificación de Huffman.

Un uso de los árboles binarios lo constituyen los *árboles de expresión*, que son la estructura de datos fundamental en el diseño de compiladores. Las hojas de un árbol de expresión son operandos, como constantes o nombres de variables; el resto de los nodos contienen operadores. El árbol será binario en tanto y cuando todos los operadores implicados lo sean. Aunque éste es el caso más simple, es posible que los nodos tengan más de dos hijos (y en el caso de operadores unarios, solamente uno). Podemos evaluar un árbol de expresión T aplicando el operador de la raíz a los valores obtenidos a partir de la evaluación recursiva de sus árboles izquierdo y derecho. En nuestro caso, esto produce el valor de `(a+((b-c)*d))`. En la Sección 11.2 estudiamos la construcción y evaluación de los árboles de expresión.

Un segundo uso de los árboles binarios es el *árbol de codificación de Huffman*, que se usa para implementar un algoritmo simple pero relativamente eficiente de compresión de datos. Cada símbolo del alfabeto se guarda en una hoja. Su código se obtiene siguiendo el camino desde la raíz hasta esa hoja, teniendo en cuenta que bajar por una arista izquierda se corresponde con un 0 y bajar por una derecha se corresponde con un 1. Luego `b` se codifica como 100. En la Sección 12.1 discutimos la construcción del árbol óptimo, que genera la mejor codificación posible.

Los árboles binarios también se utilizan como soporte en los árboles binarios de búsqueda (*véase* Capítulo 18), que permiten inserciones y acceso a los elementos en tiempo logarítmico, y en las colas con prioridad, que soportan acceso y eliminación del mínimo de una colección de elementos. Varias implementaciones eficientes de las colas con prioridad usan árboles, como se discute en los Capítulos 20 a 22.

La Figura 17.12 proporciona el esqueleto de la clase `NodoBinario`. Las líneas 40 a 42 nos dicen que cada nodo consta de un dato junto con dos punteros a los hijos. Se proporcionan tres constructores. El de la línea 18 tiene cero parámetros. El de la línea 20 se usa para construir un `NodoBinario` dado un dato como parámetro. En estos dos constructores se asigna `null` a las referencias que apuntan a los hijos izquierdo y derecho. El tercer constructor, en la línea 22, inicializa todos los atributos del `NodoBinario`. Observe que estos constructores requieren inicialización o bien de los dos hijos o de ninguno de ellos, con lo que evitamos objetos parcialmente inicializados.

El acceso a los atributos se permite desde otras clases del mismo paquete, por lo que no proporcionamos métodos modificadores ni de acceso. El método `duplicar`, declarado en la línea 36, se usa para generar una copia del árbol cuya raíz es el nodo actual. Las rutinas `tamanyo` y `altura`, declaradas en las líneas 26

```
1  package EstructurasDatos;
2  // Clase NodoBinario: almacena un nodo en un árbol
3  // CONSTRUCCIÓN: (a) sin parámetros, (b) con un Object,
4  //      o (c) un Object, un puntero izquierdo y otro derecho.
5  //
6  // ******************OPERACIONES PÚBLICAS*******************
7  // int tamanyo( )           --> Devuelve el tamaño de un árbol
8  // int altura( )            --> Devuelve la altura de un árbol
9  // void imprimirPostOrden( )--> Imprime recorrido en postorden
10 // void imprimirOrdenSim( ) --> Imprime recorrido orden simétrico
11 // void imprimirPreOrden( ) --> Imprime el recorrido en preorden
12 // NodoBinario duplicar( )  --> Devuelve una copia del árbol
13 // ********************************ERRORES*********************************
14 // Ninguno
15
16 final class NodoBinario
17 {
18     NodoBinario( )
19       { this( null ); }
20     NodoBinario( Object elDato )
21       { this( elDato, null, null ); }
22     NodoBinario( Object elDato, NodoBinario menor,
23                                 NodoBinario mayor )
24       { dato = elDato; izquierdo = menor; derecho = mayor; }
25
26     static int tamanyo( NodoBinario a )
27       { /* Figura 17.19 */ }
28     static int altura( NodoBinario a )
29       { /* Figura 17.21 */ }
30     void imprimirPostOrden( )
31       { /* Figura 17.22 */ }
32     void imprimirOrdenSim( )
33       { /* Figura 17.22 */ }
34     void imprimirPreOrden( )
35       { /* Figura 17.22 */ }
36     NodoBinario duplicar( )
37       { /* Figura 17.17 */ }
38
39     // Atributos amistosos; accesibles por otras rutinas del paquete
40     Object      dato;
41     NodoBinario izquierdo;
42     NodoBinario derecho;
43 }
```

Figura 17.12 Esqueleto de la clase `NodoBinario`.

a 28, calculan las propiedades por ellas nombradas para el nodo referenciado por el parámetro `a`. Estas rutinas se implementarán en la Sección 17.3. También proporcionamos, en las líneas 30 a 34, rutinas que imprimen el contenido del subárbol que tiene como raíz el nodo actual, usando varias estrategias recursivas para recorrerlo. Las formas de recorrer un árbol se estudian en la Sección 17.4. ¿Por qué pasamos un parámetro a las rutinas `tamanyo` y `altura`, pero usamos el objeto actual para los recorridos y el método `duplicar`? En realidad, no hay una razón concreta para esto, es sólo una cuestión de estilo. Aquí presentamos ambos estilos. Las implementaciones mostrarán que las diferencias entre ellos surgen cuando se quiere comprobar si un árbol es `null`.

En esta sección describiremos la implementación de la clase `ArbolBinario`. Se proporciona una clase `NodoBinario` separada para simplificar la implementación de algunas de las rutinas recursivas. En la Figura 17.13 se muestra el esqueleto de la clase `ArbolBinario`. La mayoría de las rutinas son sencillas de implementar pues se limitan a llamar a métodos de la clase `NodoBinario`. En la línea 44 se declara el único atributo, una referencia al nodo raíz.

```
1  package EstructurasDatos;
2  // Clase ArbolBinario: almacena un árbol binario
3  // CONSTRUCCIÓN: (a) sin parámetros o (b) con un objeto a
4  //      colocar en la raiz de un árbol de un único elemento.
5  //
6  // ******************OPERACIONES PÚBLICAS*******************
7  // void imprimirPostOrden( ) --> Imprime recorrido en postorden
8  // void imprimirOrdenSim( )--> Imprime recorrido orden simétrico
9  // void imprimirPreOrden( )--> Imprime el recorrido en preorden
10 // boolean esVacio( )       --> true si el árbol es vacío
11 // void vaciar( )           --> Construye un árbol vacío
12 // void unir( Object raiz, ArbolBinario a1, ArbolBinario a2 )
13 //                          --> Construye un nuevo árbol
14 // ********************************ERRORES*********************************
15 // Impresión de mensajes de error por uniones ilegales
16
17 public class ArbolBinario
18 {
19     public ArbolBinario( )
20       { raiz = null; }
21     public ArbolBinario( Object elemRaiz )
22       { raiz = new NodoBinario( elemRaiz ); }
23
24     public void imprimirPreOrden( )
25       { if( raiz != null ) raiz.imprimirPreOrden( ); }
26     public void imprimirOrdenSim( )
27       { if( raiz != null ) raiz.imprimirOrdenSim( ); }
28     public void imprimirPostOrden( )
29       { if( raiz != null ) raiz.imprimirPostOrden( ); }
30     public boolean esVacio( )
31       { return raiz == null; }
32     public void vaciar( )
33       { raiz = null; }
34     public void unir( Object elemRaiz, ArbolBinario a1,
35                                        ArbolBinario a2 )
36       { /* Figura 17.16 */ }
37     public int tamanyo( )
38       { return NodoBinario.tamanyo( raiz ); }
39     public int altura( )
40       { return NodoBinario.altura( raiz ); }
41     public void duplicar( ArbolBinario lder )
42       { if( this != null ) raiz = lder.raiz.duplicar( ); }
43
44     protected NodoBinario raiz;
45 }
```

Figura 17.13 Esqueleto de la clase `ArbolBinario`.

Se proporcionan dos constructores. El de la línea 19 crea un árbol vacío, mientras que el de la línea 21 crea un árbol con un único nodo. En las líneas 24 a 29 se declaran rutinas para recorrer el árbol, las cuales aplican métodos de la clase `NodoBinario` a la `raiz` después de verificar que el árbol no es vacío. Una estrategia alternativa de recorrido de árboles que se puede implementar es el *recorrido por niveles*. En la Sección 17.4 se estudian todas las rutinas de recorrido. En las líneas 32 y 30 se proporcionan rutinas para construir un árbol vacío y para comprobar si un árbol es vacío respectivamente.

Hay dos rutinas más en la clase. En las líneas 41 y 42 definimos el método `duplicar`. Después de comprobar el aliasing, llamamos al método `duplicar` de la clase `NodoBinario` para obtener una copia del árbol `ider`. Entonces asignamos el resultado a la raíz del árbol. Observe que antes de poder aplicar el método de la clase `NodoBinario` al nodo referenciado por `raiz`, debemos verificar que `raiz` no es `null`.

El último método de la clase es la rutina `unir`, la cual usa dos árboles —`a1` y `a2`— y un elemento para crear un nuevo árbol que tiene al elemento como raíz y los dos árboles como hijos izquierdo y derecho. En principio, se trata de una rutina de una sola línea:

```
raiz = new NodoBinario( elemRaiz, a1.raiz, a2.raiz );
```

Si las cosas fueran siempre tan sencillas, los programadores no encontrarían trabajo. Afortunadamente para nuestras carreras profesionales, hay complicaciones. La Figura 17.14 muestra el resultado del sencillo método `unir` de una sola línea. Se hace patente un problema: los nodos de los árboles `a1` y `a2` se encuentran ahora en dos árboles, los originales y el resultado de la unión. Esta compartición podría ser un problema si quisiéramos eliminar o alterar de alguna otra forma los árboles originales o bien sus apariciones como subárboles, pues cualquier modificación en uno u otro repercutiría inmediatamente en el otro, pues se trata en realidad del mismo objeto. Este en ocasiones podría no ser el efecto pretendido.

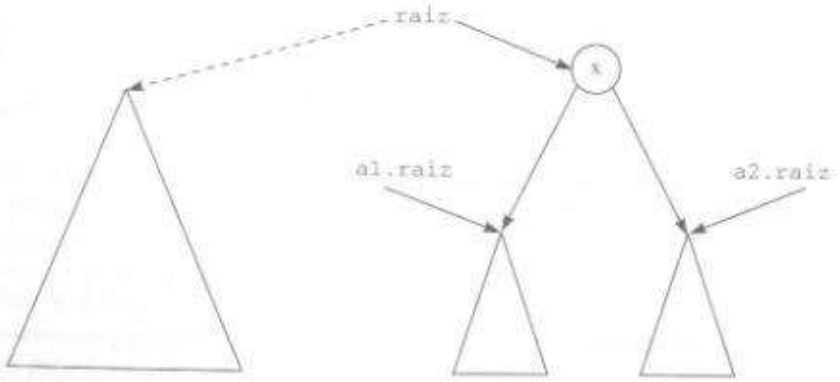


Figura 17.14 Resultado de un método `unir` ingenuo: hay compartición de subárboles.

La solución es en principio sencilla. Podemos asegurar que los nodos no aparecen en dos árboles asignando `null` a `a1.raiz` y `a2.raiz` después de `unir`. Surgen complicaciones cuando consideramos algunas llamadas que contienen aliasing:

```
a1.unir( x, a1, a2 );
a2.unir( x, a1, a2 );
a1.unir( x, a3, a3 );
```

Los dos primeros casos son similares, por lo que consideraremos sólo el primero. En la Figura 17.15 se muestra un dibujo de la situación. Puesto que `a1` es un alias del objeto actual, `a1.raiz` y `raiz` son alias. En consecuencia, tras la llamada a `new`, si ejecutamos `a1.raiz = null`, también estamos cambiando `raiz` a `null`. Por tanto, en estos casos hemos de tener mucho cuidado con los alias.

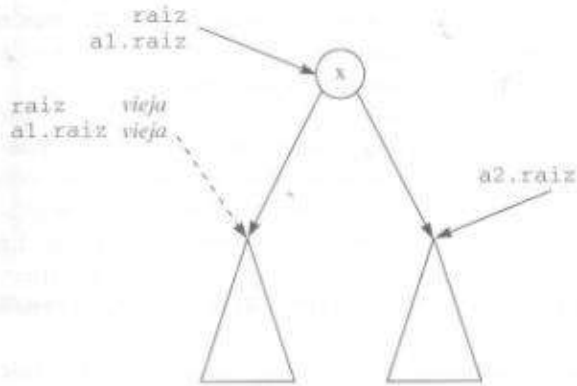


Figura 17.15 Problemas con el aliasing en la operación `unir`; `a1` es también el objeto actual.

```
1  /**
2   * Rutina unir para la clase ArbolBinario
3   * Forma un árbol nuevo a partir de elemRaiz, a1 y a2.
4   * Trata correctamente varias situaciones de aliasing.
5   */
6  public void unir( Object elemRaiz, ArbolBinario a1,
7                                     ArbolBinario a2 )
8  {
9      if( a1.raiz == a2.raiz && a1.raiz != null )
10     {
11         System.err.println( "arbolIzquierdo==arbolDerecho; " +
12                             "unir abortado" );
13         return;
14     }
15
16         // Creación de un nuevo nodo
17     raiz = new NodoBinario( elemRaiz, a1.raiz, a2.raiz );
18
19         // Nos aseguramos de que cada nodo está sólo en un árbol
20     if( this != a1 )
21         a1.raiz = null;
22     if( this != a2 )
23         a2.raiz = null;
24 }
```

Figura 17.16 Rutina unir para la clase `ArbolBinario`.

El tercer caso no debería permitirse porque colocaría los nodos de `a3` en dos sitios distintos de `a1`. Ahora bien, si `a3` representara un árbol vacío, el tercer caso sí estaría permitido. Al final resultó bastante peor de lo que esperábamos. El código resultante se muestra en la Figura 17.16. Lo que en un principio era una rutina de una sola línea se ha convertido en una bastante más larga.

17.3 Árboles y recursión

Ya que los árboles se pueden definir de forma recursiva, no sorprende el hecho de que la mayoría de las rutinas que implican a los árboles se implementen de forma más sencilla usando recursión. Aquí proporcionamos la implementación recursiva de la mayoría de los métodos restantes de las clases `NodoBinario` y `ArbolBinario`. Las rutinas resultantes son increíblemente compactas.

Comenzamos con el método `duplicar` de la clase `NodoBinario`. Puesto que es un método de la clase `NodoBinario`, sabemos que el árbol que estamos duplicando no es vacío. El algoritmo recursivo es sencillo. Primero creamos un nodo nuevo con el mismo dato que el nodo raíz actual. Después llamamos recursivamente dos veces a `duplicar` para generar el subárbol izquierdo y el derecho. En ambos casos, hacemos las llamadas recursivas después de comprobar que el correspondiente árbol es no vacío. Esta descripción verbal aparece codificada en la Figura 17.17.

El siguiente método que presentamos es la rutina `tamanyo` de la clase `NodoBinario`, la cual devuelve el tamaño del árbol cuya raíz es el nodo referenciado por `a`, que se pasa como parámetro. Si dibujamos el árbol recursivamente, como se muestra en la Figura 17.18, vemos que el tamaño de un árbol es igual al tamaño del hijo izquierdo más el del hijo derecho más 1, ya que la raíz aporta un nodo. Como en toda rutina recursiva necesitamos un caso base que pueda resolverse directamente. El árbol más pequeño que la rutina `tamanyo` trata es el árbol vacío (correspondiente a `a` `null`), cuyo tamaño es claramente 0. El resultado está implementado en la Figura 17.19. Podemos comprobar fácilmente que la recursión produce la respuesta correcta para los árboles de tamaño 1.

```
1  /**
2   * Devuelve un puntero al nodo raiz del duplicado del
3   * árbol binario cuya raíz es el nodo actual.
4   */
5  NodoBinario duplicar( )
6  {
7      NodoBinario raiz = new NodoBinario( dato );
8      if( izquierdo != null ) // Si hay un árbol izquierdo
9          raiz.izquierdo = izquierdo.duplicar( ); // Duplicarlo
10     if( derecho != null )   // Si hay un árbol derecho
11         raiz.derecho = derecho.duplicar( );     // Duplicarlo
12     return raiz;            // Devolver el árbol resultante
13 }
```

Figura 17.17 Rutina para devolver una copia del árbol con raíz en el nodo actual.

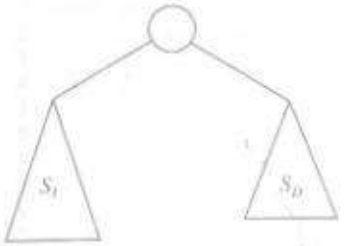


Figura 17.18 Visión recursiva utilizada para calcular el tamaño de un árbol: $S_A = S_I + S_D + 1$.

```
1  /**
2   * Devuelve el tamaño del árbol binario con raíz en a.
3   */
4  static int tamanyo( NodoBinario a )
5  {
6      if( a == null )
7          return 0;
8      else
9          return 1 + tamanyo( a.izquierdo ) + tamanyo( a.derecho );
10 }
```

Figura 17.19 Rutina para calcular el tamaño de un nodo.

La última rutina recursiva de esta sección calcula la altura de un nodo. Es difícil de escribir en forma no recursiva, pero se implementa trivialmente de forma recursiva, una vez nos fijamos en la descripción gráfica de un árbol. La Figura 17.20 muestra un árbol visto de forma recursiva. Supongamos que el hijo izquierdo tiene altura H_I y que el hijo derecho tiene altura H_D. Cualquier nodo que esté d niveles por debajo de la raíz del hijo izquierdo está $d + 1$ niveles por debajo de la raíz del árbol completo. Lo mismo sucede para el hijo derecho. Luego la longitud del camino que lleva al nodo más profundo en el árbol original es 1 más de la longitud de su camino desde la raíz del subárbol correspondiente. En consecuencia, si calculamos dicho valor para ambos subárboles, la respuesta que buscamos es el máximo de dichos valores más 1.

¿Qué sucede con el caso base? De nuevo, corresponde al árbol vacío. La afirmación de que la altura de un árbol vacío es 0, es errónea, porque un árbol de un solo nodo tiene altura 0 (una hoja siempre tiene altura 0). Para que la fórmula recursiva funcione correctamente con árboles de un solo nodo, definimos la altura

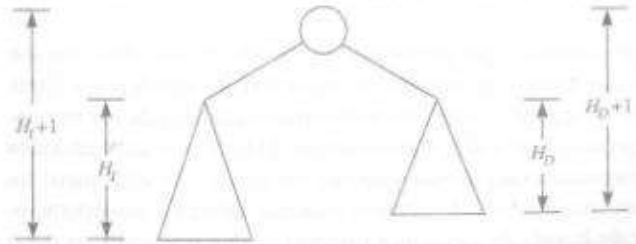


Figura 17.20 Visión recursiva del cálculo de la altura: $H_A = \text{Max}(H_I + 1, H_D + 1)$.

```
/**
 * Devuelve la altura del árbol binario con raiz en a.
 */
static int altura( NodoBinario a )
{
    if( a == null )
        return -1;
    else
        return 1 + Math.max( altura( a.izquierdo ),
                             altura( a.derecho ) );
}
```

Figura 17.21 Rutina para calcular la altura de un nodo.

del árbol vacío como -1. Ahora, como una hoja tiene dos subárboles de altura -1, su altura queda correctamente definida como 0. La rutina resultante se muestra en la Figura 17.21.

17.4 Recorrido de árboles: clases iteradoras

En este capítulo hemos visto cómo se puede usar la recursión para implementar los métodos de los árboles binarios. Cuando se aplica la recursión, calculamos información no sólo sobre un nodo, sino también sobre todos sus descendientes. Decimos entonces que estamos *recorriendo el árbol*. Dos recorridos conocidos que ya hemos visto anteriormente son el recorrido en preorden y el recorrido en postorden.

En un recorrido en preorden, primero se procesa el nodo y después se procesan recursivamente sus hijos. La rutina `duplicar` es un ejemplo de recorrido en preorden, pues lo primero que hacemos es crear la raíz. Después se copian de forma recursiva los hijos izquierdo y derecho.

En un recorrido en postorden, el nodo dado se procesa después de haber procesado recursivamente sus hijos. Dos ejemplos de ello son los métodos `tamanyo` y `altura`. En ambos casos, sólo podemos calcular el valor correspondiente de un nodo dado (por ejemplo, su tamaño o su altura) después de haber calculado el correspondiente valor para sus hijos.

Un tercer recorrido recursivo habitual es el *recorrido en orden simétrico*, en el cual primero se procesa recursivamente el hijo izquierdo, luego se procesa el nodo actual, y finalmente se procesa recursivamente el hijo derecho. Este mecanismo se usa para generar la expresión algebraica correspondiente a un árbol de expresión. Por ejemplo, en la Figura 17.11, el recorrido en orden simétrico genera la expresión `(a+((b-c)*d))`.

La Figura 17.22 ilustra rutinas que imprimen los nodos de un árbol binario usando cada uno de los algoritmos recursivos de recorrido de árboles. La Figura 17.23 muestra el orden en que se visitan los nodos para cada una de las tres estrategias. El tiempo de ejecución de todas ellas es lineal. Esto es así, pues en todos los casos cada nodo se imprime exactamente una vez, de modo que el número total de llamadas realizadas (incluyendo el trabajo constante debido a las operaciones de apilar y desapilar de la pila de ejecución interna) es una por nodo, es decir $O(N)$. Luego el tiempo total de ejecución es $O(N)$.

```
void imprimirPreOrden( )
{
    System.out.println( dato );            // Nodo
    if( izquierdo != null )
        izquierdo.imprimirPreOrden( );     // Izquierdo
    if( derecho != null )
        derecho.imprimirPreOrden( );       // Derecho
}

void imprimirPostOrden( )
{
    if( izquierdo != null )
        izquierdo. imprimirPostOrden( );   // Izquierdo
    if( derecho != null )
        derecho. imprimirPostOrden( );     // Derecho
    System.out.println( dato );            // Nodo
}

void imprimirOrdenSim ( )
{
    if( izquierdo != null )
        izquierdo. imprimirOrdenSim( );    // Izquierdo
    System.out.println( dato );            // Nodo
    if( derecho != null )
        derecho. imprimirOrdenSim( );      // Derecho
}
```

Figura 17.22 Rutinas para imprimir los nodos en preorden, postorden y orden simétrico.

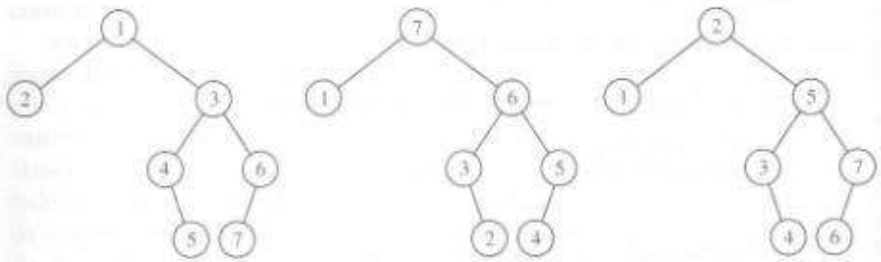


Figura 17.23 Rutas de acceso para los recorridos en preorden, postorden y orden simétrico.

¿Debemos necesariamente usar recursión para implementar los recorridos? Claramente la respuesta es no. Dado que, como ya se discutió en la Sección 7.3, la recursión se implementa internamente usando una pila. De modo que podríamos usar nuestra propia pila. El resultado es generalmente un programa algo más rápido, dado que, solamente tenemos que guardar en la pila lo que necesitamos en cada caso, en lugar de hacer que el compilador coloque un registro de activación completo. La diferencia en velocidad entre un algoritmo recursivo y otro no recursivo depende mucho de la plataforma. En la mayoría de los casos, la mejora en la velocidad no justifica el esfuerzo implicado en la eliminación de la recursión. Aun así, vale la pena saber cómo hacerlo por si acaso nuestra plataforma es de las que puede beneficiarse de la eliminación de la recursión, y también porque en ocasiones ver cómo se implementa un programa de forma no recursiva puede hacer que la recursión se entienda más claramente.

Presentaremos tres clases iteradoras, con el mismo espíritu que en las listas enlazadas. Cada una de ellas nos permitirá ir al primer nodo, avanzar al siguiente nodo, comprobar si hemos llegado al último nodo y acceder al nodo actual. El orden en que se accede a los nodos está determinado por el tipo de recorrido. Implementaremos también un *recorrido por niveles*, en el que los nodos se visitan de arriba abajo y de izquierda a derecha. Este recorrido es inherentemente no recursivo y de hecho usa una cola en lugar de una pila. Hecha esta salvedad, resulta ser bastante parecido al recorrido en preorden.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase ArbolIter; mantiene la "posición actual"
6  //
7  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
8  //
9  // ******************OPERACIONES PÚBLICAS*******************
10 //     primero y avanzar son abstractos; los demás son finales
11 // boolean esValido( ) --> True si estamos en una posición válida
12 // Object recuperar( ) --> Devuelve elemento en posición actual
13 // void primero( )     --> Coloca posición actual en el primero
14 // void avanzar( )     --> Avanza (prefijo)
15 // ********************************ERRORES*********************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 /**
19  * Clase iteradora de árboles.
20  */
21 abstract public class ArbolIter
22 {
23     public ArbolIter( ArbolBinario elArbol )
24     {
25         a = elArbol;
26         actual = null;
27     }
28
29     abstract public void primero( );
30
31     final public boolean esValido( )
32     {
33         return actual != null;
34     }
35
36     final public Object recuperar( ) throws ElementoNoEncontrado
37     {
38         if( actual == null )
39             throw new ElementoNoEncontrado( "recuperar de ArbolIter" );
40         return actual.dato;
41     }
42
43     abstract public void avanzar( ) throws ElementoNoEncontrado;
44
45     protected ArbolBinario a;          // El árbol
46     protected NodoBinario actual;      // Posición actual
47 }
```

Figura 17.24 Clase iteradora abstracta para los árboles.

La Figura 17.24 proporciona una clase abstracta para la iteración sobre los árboles. Cada iterador almacenará una referencia al árbol y un puntero al nodo actual, los cuales se declaran respectivamente en las líneas 45 y 46, inicializándose en el constructor. Los dos son declarados como `protected` para permitir acceder a ellos a las clases derivadas. En las líneas 29 a 43 se declaran varios métodos. Los métodos `esValido` y `recuperar` son invariantes en la jerarquía por lo que se proporciona una implementación y se declaran como `final`. Por su parte, cada tipo de iterador debe proporcionar los métodos abstractos `primero` y `avanzar`. Observe que el método `avanzar` difiere del de las listas enlazadas porque lanza una excepción. Se ha hecho así para ilustrar distintas estrategias.

17.4.1 Recorrido en postorden

El recorrido en postorden se implementa usando una pila para almacenar el estado actual. La cima de la pila representará el nodo que estamos visitando en un cierto instante en el recorrido en postorden. Pero podemos encontrarnos en tres situaciones distintas:

1. A punto de hacer una llamada recursiva al hijo izquierdo.
2. A punto de hacer una llamada recursiva al hijo derecho.
3. A punto de procesar el nodo actual.

En consecuencia, en el curso del recorrido cada nodo se colocará en la pila tres veces. Cuando se desapila un nodo de la pila por tercera vez, podemos marcarlo como el nodo actual a ser visitado.

En caso contrario, el nodo se está desapilando por primera o segunda vez, de modo que aún no está listo para ser visitado, por lo que volvemos a apilarlo en la pila y simulamos una llamada recursiva. Si el nodo está siendo desapilado por primera vez, necesitamos apilar el hijo izquierdo en la pila, si es no vacío. Si está siendo desapilado por segunda vez apilamos el hijo derecho, si es no vacío. En todo caso, desapilamos después, pasando a aplicar el mismo test. Observe que cada vez que desapilamos, estamos simulando la llamada recursiva al hijo apropiado. Si el hijo era vacío y por tanto nunca fue apilado en la pila, entonces cuando desapilemos, estaremos desapilando de nuevo el nodo original.

Eventualmente, o bien el proceso desapila un nodo por tercera vez, o bien la pila queda vacía. En este último caso, hemos recorrido ya todo el árbol. Inicializamos el algoritmo apilando una referencia a la raíz. Un ejemplo de cómo se manipula la pila se muestra en la Figura 17.25.

Un resumen rápido: la pila contiene los nodos que ya hemos recorrido pero que aún no hemos tratado completamente. Cuando un nodo se apila en la pila, el contador es respectivamente 0, 1 o 2 en cada una de las siguientes situaciones:

1. Si estamos a punto de procesar el hijo izquierdo.
2. Si estamos a punto de procesar el hijo derecho.
3. Si estamos a punto de procesar el nodo.

Sigamos la traza de lo que sucede en el recorrido en postorden. Inicializamos el recorrido apilando la raíz `a` en la pila, e inmediatamente se desapila `a` por primera vez, por lo que vuelve a colocarse en la pila y apilamos su hijo izquierdo `b`.

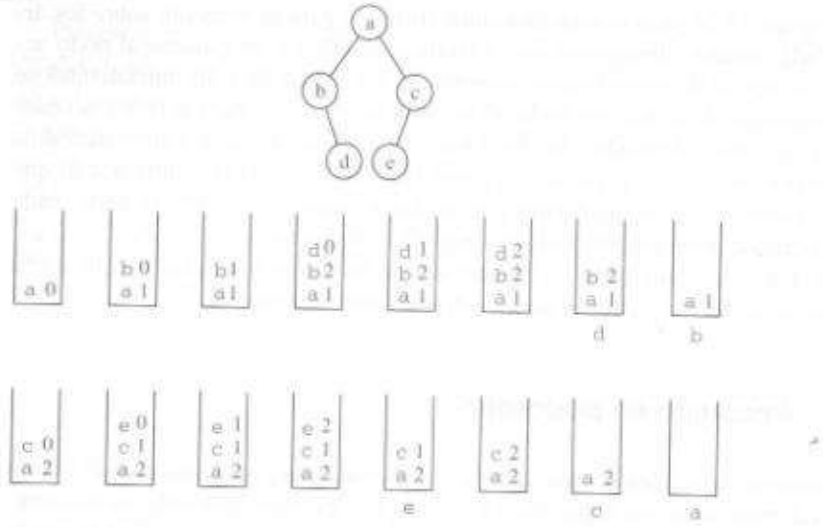


Figura 17.25 Estados de la pila durante el recorrido en postorden.

A continuación se desapila `b` por primera vez, por lo que se coloca de vuelta en la pila. Si existiera el hijo izquierdo de `b` se apilaría, pero como no tiene, se desapila `b` por segunda vez, se vuelve a colocar en la pila y se apila su hijo derecho `d`. Ahora se desapila `d` por primera vez, por lo que se coloca de nuevo en la pila. No se produce una operación de apilar, porque `d` no tiene hijo izquierdo, con lo que `d` se desapila por segunda vez y vuelve a la pila de nuevo. Como tampoco tiene hijo derecho, no se apila nada, sino que `d` se desapila por tercera vez, siendo marcado como nodo visitado. A continuación se desapila `b` por tercera vez, marcándose también como visitado.

Después se desapila `a` por segunda vez, colocándose de nuevo en la pila junto con su hijo derecho `c`. Se desapila `c` por primera vez, por lo que vuelve de nuevo a la pila, apilándose su hijo izquierdo `e`. A continuación `e` es desapilado, apilado, desapilado de nuevo y vuelto a apilar, y finalmente se desapila por tercera vez (esto sucede para todas las hojas). Luego se marca a `e` como nodo visitado. Después se desapila `c` por segunda vez, vuelve a la pila y como no tiene hijo derecho, es inmediatamente desapilado por tercera vez y marcado como visitado. Finalmente se desapila `a` por tercera vez y se marca como visitado. En este momento la pila está vacía, concluyendo el recorrido.

La clase `PostOrden`, que se muestra en la Figura 17.26, se implementa directamente usando el algoritmo descrito previamente. La clase `NodoPl` representa los objetos que se colocan en la pila. Sus objetos contienen una referencia a un nodo junto con un entero que almacena el número de veces que se ha desapilado el nodo en cuestión. Los objetos `NodoPl` se inicializan siempre reflejando el hecho de que aún no han sido desapilados.

La clase `PostOrden` se deriva de la clase `ArbolIter`. Añade una pila interna a los atributos heredados. La clase `PostOrden` se inicializa inicializando primero los atributos de `ArbolIter` para después apilar la raíz en la pila. Esto se ilustra en el constructor, en las líneas 23 a 28. El método `primero` se implementa limpiando la pila, apilando la raíz y llamando a `avanzar`.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase PostOrden; mantiene una "posición actual"
6  //      de acuerdo con un recorrido en postorden
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ***********************OPERACIONES PÚBLICAS*********************
11 // boolean esValido( )    --> True en una posición válida del árbol
12 // Object recuperar( )    --> Devuelve elemento en posición actual
13 // void primero( )        --> Coloca posición actual en primer nodo
14 // void avanzar( )        --> Avanza a la siguiente posición
15 // ****************************ERRORES**************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 /**
19  * Clase iteradora PostOrden.
20  */
21 public class PostOrden extends ArbolIter
22 {
23     public PostOrden( ArbolBinario elArbol )
24     {
25         super( elArbol );
26         p = new PilaVec( );
27         p.apilar( new NodoPl( a.raiz ) );
28     }
29
30     public void primero( )
31     {
32         p.vaciar( );
33         if( a.raiz != null )
34             p.apilar( new NodoPl( a.raiz ) );
35         try
36           { avanzar( ); }
37         catch( ElementoNoEncontrado e ) { } // Arbol vacio
38     }
39
40     protected Pila p;       // Pila de objetos NodoPl
41 }
42
43 class NodoPl
44 {
45     NodoBinario nodo;
46     int vecesDesap;
47
48     NodoPl( NodoBinario n )
49       { nodo = n; vecesDesap = 0; }
50 }
```

Figura 17.26 Clase `PostOrden` (clase completa exceptuando `avanzar`).

La Figura 17.27 implementa `avanzar`. Sigue la descripción anterior casi al pie de la letra. La línea 8 comprueba si la pila está vacía. Si lo está, ya hemos completado la iteración y podemos asignar `null` a `actual`, terminando después. Si no, llevamos a cabo repetidos apilados y desapilados hasta que se desapila un

```
1  /**
2   * Avanza la posición actual al siguiente nodo en el árbol,
3   *    de acuerdo con el esquema de recorrido en postorden.
4   * @exception ElementoNoEncontrado si la posición actual es null.
5   */
6  public void avanzar( ) throws ElementoNoEncontrado
7  {
8      if( p.esVacia( ) )
9      {
10         if( actual == null )
11           throw new ElementoNoEncontrado( "Avanzar de PostOrden" );
12         actual = null;
13         return;
14     }
15
16     NodoPl cnodo;
17
18     for( ; ; )
19     {
20         try
21           { cnodo = ( NodoPl ) p.cimaYDesapilar( ); }
22         catch( DesbordamientoInferior e )
23           { return; } // No puede ocurrir
24
25         if( ++cnodo.vecesDesap == 3 )
26         {
27             actual = cnodo.nodo;
28             return;
29         }
30
31         p.apilar( cnodo );
32         if( cnodo.vecesDesap == 1 )
33         {
34             if( cnodo.nodo.izquierdo != null )
35                 p.apilar( new NodoPl( cnodo.nodo.izquierdo ) );
36         }
37         else // cnodo.vecesDesap == 2
38         {
39             if( cnodo.nodo.derecho != null )
40                 p.apilar( new NodoPl( cnodo.nodo.derecho ) );
41         }
42     }
43 }
```

Figura 17.27 Método `avanzar` para la clase iteradora `PostOrden`.

elemento por tercera vez. Cuando esto sucede, el test de la línea 25 tiene éxito y podemos terminar. En caso contrario, en la línea 31 volvemos a apilar el elemento en la pila (observe que la componente `vecesDesap` ya ha sido incrementada en la línea 25). Entonces simulamos la llamada recursiva. Si el nodo ha sido desapilado por primera vez y tiene un hijo izquierdo, entonces se apila su hijo izquierdo. Si había sido desapilado por segunda vez y tiene un hijo derecho, se apila su hijo derecho. Observe que en todos los casos la construcción del objeto `NodoPl` implica que el nuevo nodo apilado se ha desapilado 0 veces.

Eventualmente el bucle `for` terminará al haberse desapilado algún nodo por tercera vez. Observe que a lo largo de la secuencia completa de iteraciones, puede haber a lo sumo $3N$ operaciones de apilar y desapilar. Ésta es una forma alternativa de establecer la linealidad de los recorridos en postorden.

17.4.2 Recorrido en orden simétrico

El recorrido en orden simétrico es análogo al recorrido en postorden excepto por el hecho de que un nodo se declara visitado cuando se desapila por segunda vez. Antes de concluir con el nodo, el iterador apila su hijo derecho (si existe) en la pila, de

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase OrdenSim: mantiene una "posición actual"
6  //     de acuerdo con el recorrido en orden simétrico
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ********************OPERACIONES PÚBLICAS*********************
11 // Las mismas que ArbolIter
12 // ***************************ERRORES*****************************
13 // Excepciones lanzadas por accesos o avances ilegales
14
15 public class OrdenSim extends PostOrden
16 {
17     public OrdenSim( ArbolBinario elArbol )
18       { super( elArbol ); }
19
20     public void avanzar( ) throws ElementoNoEncontrado
21     {
22         if( p.esVacia( ) )
23         {
24             if( actual == null )
25                 throw new ElementoNoEncontrado( "avanzar de OrdenSim" );
26             actual = null;
27             return;
28         }
29
30         NodoPl cnodo;
31         for( ; ; )
32         {
33             try
34               { cnodo = ( NodoPl ) p.cimaYDesapilar( ); }
35             catch( DesbordamientoInferior e )
36               { return; }   // No puede suceder
37
38             if( ++cnodo.vecesDesap == 2 )
39             {
40                 actual = cnodo.nodo;
41                 if( cnodo.nodo.derecho != null )
42                     p.apilar( new NodoPl( cnodo.nodo.derecho ) );
43                 return;
44             }
45                 // Se procesa primero
46             p.apilar( cnodo );
47             if( cnodo.nodo.izquierdo != null )
48                 p.apilar( new NodoPl( cnodo.nodo.izquierdo ) );
49         }
50     }
51 }
```

Figura 17.28 Clase iteradora `OrdenSim` completa.

forma que la siguiente llamada a avanzar pueda continuar recorriéndolo. Como es muy similar al recorrido en postorden, derivamos la clase OrdenSim a partir de la clase PostOrden (aun a pesar de que no exista una relación *ES-UN(A)*). El único cambio es una pequeña alteración en avanzar. La nueva clase se muestra en la Figura 17.28.

17.4.3 Recorrido en preorden

El recorrido en preorden es análogo al recorrido en postorden, excepto por el hecho de que un nodo se declara visitado la primera vez que se desapila. Al hacerlo, el iterador apila el hijo derecho y luego el hijo izquierdo. Es importante que nos fijemos en el orden de ambas operaciones: queremos procesar primero el hijo izquierdo y después el derecho, por lo que apilamos primero el derecho y luego el izquierdo.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase Preorden; mantiene una "posición actual"
6  //     de acuerdo con el recorrido en preorden
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ******************OPERACIONES PÚBLICAS*******************
11 // boolean esValido( ) --> True en una posición válida del árbol
12 // Object recuperar( ) --> Devuelve elemento en posición actual
13 // void primero( )     --> Coloca posición actual en primer nodo
14 // void avanzar( )     --> Avanza a la siguiente posición
15 // ***************************ERRORES************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 public class PreOrden extends ArbolIter
19 {
20     public PreOrden( ArbolBinario elArbol )
21     {
22         super( elArbol );
23         p = new PilaVec( );
24         p.apilar( a.raiz );
25     }
26
27     public void primero( )
28     {
29         p.vaciar( );
30         if( a.raiz != null )
31             p.apilar( a.raiz );
32         try
33           { avanzar( ); }
34         catch( ElementoNoEncontrado e ) { } // Árbol vacío
35     }
36
37     public void avanzar( ) throws ElementoNoEncontrado
38       { /* Figura 17.30 */ }
39
40     private Pila p; // Pila de objetos NodoBinario
41 }
```

Figura 17.29 Clase PreOrden con todos los métodos excepto avanzar

```
1  /**
2   * Avanza la posición actual al siguiente nodo en el árbol,
3   * de acuerdo con el esquema de recorrido en preorden.
4   * @exception ElementoNoEncontrado si la posición actual es null.
5   */
6  public void avanzar( ) throws ElementoNoEncontrado
7  {
8      if( p.esVacia( ) )
9      {
10         if( actual == null )
11             throw new ElementoNoEncontrado( "Avanzar de PreOrden" );
12         actual = null;
13         return;
14     }
15
16     try
17       { actual = ( NodoBinario ) p.cimaYDesapilar( ); }
18     catch( DesbordamientoInferior e )
19       { return; } // No puede suceder
20
21     if( actual.derecho != null )
22         p.apilar( actual.derecho );
23     if( actual.izquierdo != null )
24         p.apilar( actual.izquierdo );
25 }
```

Figura 17.30 Método avanzar de la clase iteradora PreOrden.

Podríamos derivar la clase PreOrden a partir de la clase OrdenSim o de la clase PostOrden, pero esto supondría un coste innecesario pues en esta ocasión ya no es necesario mantener un contador del número de veces que se ha desapilado cada elemento. En consecuencia, la clase PreOrden se deriva directamente de la clase ArbolIter. En la Figura 17.29 se muestra el esqueleto resultante con el constructor y el método primero.

En la línea 40 se añade a la colección de atributos de la clase ArbolIter una pila de referencias a nodos de árbol. El constructor y el método primero son similares a los que ya hemos visto. Como se ilustra en la Figura 17.30, el método avanzar es más simple: ya no necesitamos más un bucle for, pues tan pronto como en la línea 17 se desapila un nodo, éste se convierte en el nodo actual. En este momento apilamos el hijo derecho y despues el izquierdo, cuando uno u otro exista.

17.4.4 Recorrido por niveles

Concluimos implementando el *recorrido por niveles*. Este recorrido procesa los nodos comenzando en la raíz y avanzando de forma descendente y de izquierda a derecha. El nombre se deriva del hecho de que primero visitamos los nodos del nivel 0 (la raíz), después los del nivel 1 (los hijos de la raíz), los del nivel 2 (los nietos de la raíz), y así sucesivamente. Un recorrido por niveles se implementa usando una cola en lugar de una pila. La cola almacena los nodos que van a ser visitados. Cuando se visita un nodo, se colocan sus hijos al final de la cola, donde serán visitados después de los nodos que ya están en la cola. Es fácil ver que esto garantiza que los nodos se visitan por niveles. La clase PorNiveles se muestra

en las Figuras 17.31 y 17.32 y se asemeja mucho a la clase `PreOrden`. Las únicas diferencias estriban en que usamos una cola en lugar de una pila y que añadimos a la cola el hijo izquierdo y luego el derecho, en lugar de hacerlo al revés. Observe que la cola puede ser muy larga. En el peor de los casos, todos los nodos del último nivel (hasta $N/2$) podrían estar simultáneamente en la cola.

El recorrido por niveles implementa una técnica más general denominada *búsqueda en anchura*. En la Sección 14.2 se presenta un ejemplo de utilización de esta técnica en un marco más general.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase PorNiveles; mantiene una "posición actual"
6  //     de acuerdo con un recorrido por niveles
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ******************OPERACIONES PÚBLICAS*******************
11 // boolean esValido( ) --> True en una posición válida del árbol
12 // Object recuperar( ) --> Devuelve elemento en posición actual
13 // void primero( )     --> Coloca posición actual en primer nodo
14 // void avanzar( )     --> Avanza a la siguiente posición
15 // ******************ERRORES*********************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 /**
19  * Clase iteradora PorNiveles.
20  */
21 public class PorNiveles extends ArbolIter
22 {
23     public PorNiveles( ArbolBinario elArbol )
24     {
25         super( elArbol );
26         c = new ColaVec( );
27         c.insertar( a.raiz );
28     }
29
30     public void primero( )
31     {
32         c.vaciar( );
33         if( a.raiz != null )
34             c.insertar( a.raiz );
35         try
36           { avanzar( ); }
37         catch( ElementoNoEncontrado e ) { } // Árbol vacío
38     }
39
40     public void avanzar( ) throws ElementoNoEncontrado
41       { /* Figura 17.32 */ }
42
43     private Cola c; // Cola de objetos NodoBinario
44 }
```

Figura 17.31 Clase iteradora `PorNiveles` con la mayoría de sus métodos.

```
1  /**
2   * Avanza la posición actual al siguiente nodo en el árbol,
3   * de acuerdo con el esquema de recorrido por niveles.
4   * @exception ElementoNoEncontrado si la posición actual es null.
5   */
6  public void avanzar( ) throws ElementoNoEncontrado
7  {
8      if( c.esVacia( ) )
9      {
10         if( actual == null )
11             throw new ElementoNoEncontrado( "avanzar de PorNiveles" );
12         actual = null;
13         return;
14     }
15
16     try
17       { actual = ( NodoBinario ) c.quitarPrimero( ); }
18     catch( DesbordamientoInferior E )
19       { return; } // No puede suceder
20
21     if( actual.izquierdo != null )
22         c.insertar( actual.izquierdo );
23     if( actual.derecho != null )
24         c.insertar( actual.derecho );
25 }
```

Figura 17.32 Método `avanzar` para la clase iteradora `PorNiveles`.

Resumen

En este capítulo hemos estudiado los *árboles* y, en particular, los *árboles binarios*. Hemos visto cómo se pueden usar los árboles para implementar los sistemas de ficheros de muchos computadores y también algunas otras aplicaciones, como los árboles de expresión y de codificación, que se estudian con más detalle en la Parte III. Los algoritmos que trabajan sobre árboles hacen un uso extensivo de la recursión. Hemos examinado tres algoritmos recursivos de recorrido de árboles —reorden, postorden y orden simétrico— y hemos visto cómo se pueden implementar de forma no recursiva. También hemos examinado el recorrido por niveles, que forma la base de una importante técnica de búsqueda llamada búsqueda en anchura. En el siguiente capítulo estudiaremos otro tipo fundamental de árbol: el *árbol binario de búsqueda*.

Elementos del juego

altura Longitud del camino que va desde un nodo a la hoja más profunda por debajo de él.

árbol Definido no recursivamente, un árbol es un conjunto de nodos y otro de aristas orientadas que los conectan entre sí. Los árboles se definen de una forma natural de manera recursiva diciendo que un árbol es o bien vacío, o bien consta de una raíz y cero o más subárboles.

árbol binario Árbol en el que ningún nodo puede tener más de dos hijos. Se puede definir recursivamente de manera muy conveniente.

ascendiente y **descendiente** Si hay un camino de un nodo u a otro v, entonces u es ascendiente de v y v es descendiente de u.

ascendiente propio y **descendiente propio** En un camino de un nodo u a otro v, si $u \neq v$, entonces u es un ascendiente propio de v y v es un descendiente propio de u.

hermanos Nodos con el mismo padre.

hoja Nodo de un árbol que no tiene hijos.

método primer hijo/siguiente hermano Implementación de un árbol general en la que cada nodo guarda dos referencias por elemento: una al hijo más a la izquierda (si no es una hoja) y otra al hermano situado a su derecha.

padre e **hijo** Padres e hijos se definen de forma natural. Una arista orientada conecta al padre con cada hijo.

profundidad Longitud del camino de la raíz hasta el nodo.

recorrido en orden simétrico Tipo de recorrido en el que se procesa el nodo actual entre las llamadas recursivas para sus hijos.

recorrido en postorden Tipo de recorrido en el que se visita el nodo después de haber procesado los hijos. Tarda un tiempo constante por nodo.

recorrido en preorden Tipo de recorrido en el que se visita cada nodo antes de procesar sus hijos. El recorrido requiere un tiempo constante por nodo.

recorrido por niveles Tipo de recorrido en el que los nodos se vistan de forma descendiente y de izquierda a derecha. Se implementa usando una cola. Corresponde a una búsqueda en anchura.

tamaño de un nodo Número de descendientes que tiene un nodo (incluyéndole a él mismo).

Errores comunes

1. Permitir que un nodo pertenezca simultáneamente a dos árboles es generalmente una idea poco afortunada pues al realizar cambios en un subárbol se podrían provocar, sin desearlo, cambios simultáneos en varios subárboles.
2. Olvidar la comprobación de que un árbol es vacío es un error común. Si esto sucede en el marco de un algoritmo recursivo, probablemente el programa abortará.
3. Un error común cuando se trabaja con árboles consiste en pensar iterativamente en lugar de hacerlo recursivamente. Diseñe primero los algoritmos de forma recursiva, y sólo después conviértalos a iterativos, de resultar conveniente.

En Internet

Muchos de los ejemplos discutidos en este capítulo se utilizan en el Capítulo 18, en el que se estudian los árboles binarios de búsqueda. El código para las clases iteradoras es parte del paquete `EstructurasDatos` y debe guardarse en el directorio **EstructurasDatos**. Las versiones originales se encuentran en el directorio **DataStructures**. Estos iteradores usan clases `NodoBinario` y `ArbolBinario` ligeramente diferentes de las utilizadas en este capítulo. Esas clases también deben guardarse en el directorio **EstructurasDatos**. Las versiones originales se encuentran en el directorio **DataStructures**.

Las versiones originales de las clases `NodoBinario`, `ArbolBinario` y `SistemaFichero` de este capítulo se encuentran en el directorio **Chapter17**.

BinaryNode.java	Contiene la clase traducida por `NodoBinario` en la Figura 17.12.
BinaryTree.java	Contiene la clase traducida por `ArbolBinario` en la Figura 17.13.
FileSystem.java	Implementa el recorrido de directorios, traducido en la Figura 17.10 por `SistemaFichero`.
InOrder.java	Contiene la clase traducida por `OrdenSim`.
LevelOrder.java	Contiene la clase traducida por `PorNiveles`.
PostOrder.java	Contiene la clase traducida por `PostOrden`.
PreOrder.java	Contiene la clase traducida por `PreOrden`.
TreeIterator.java	Contiene la clase iteradora traducida por `ArbolIter`.

Ejercicios

Cuestiones breves

17.1. Para el árbol de la Figura 17.33, determine lo siguiente:

a) ¿Cúal es el nodo raíz?
b) ¿Qué nodos son las hojas?
c) ¿Cuál es la profundidad del árbol?
d) El resultado de los recorridos en preorden, postorden, orden simétrico y por niveles.

17.2. Para cada nodo del árbol de la Figura 17.33, haga lo siguiente:

a) Nombre el nodo padre.
b) Liste los hijos.
c) Liste los hermanos.
d) Calcule la altura.
e) Calcule la profundidad.
f) Calcule el tamaño.

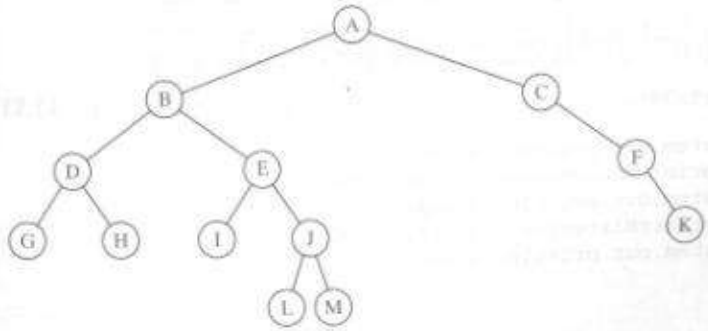


Figura 17.33 Árbol para los ejercicios 17.1 y 17.2.

17.3. ¿Cuál es la salida del método de la Figura 17.34 para el árbol de la Figura 17.25?

17.4. Muestre las operaciones realizadas sobre la pila al recorrer en orden simétrico y en preorden el árbol de la Figura 17.25.

Problemas teóricos

17.5. Muestre que el número máximo de nodos en un árbol binario de altura H es $2^{H+1} - 1$.

17.6. Un nodo completo es un nodo con dos hijos. Demuestre que en un árbol binario, el número de hojas es igual al número de nodos completos más 1.

17.7. ¿Cuántos hijos iguales a `null` hay en un árbol binario de N nodos? ¿Cuántos hay en un árbol M-ario de N nodos?

17.8. Consideremos un árbol binario con hojas $l_1, l_2, \ldots, l_M$ a profundidades $d_1, d_2, \ldots, d_M$, respectivamente. Demuestre que $\Sigma_{i=1}^{M} 2^{-d_i} \leq 1$ (llamada *desigualdad de Kraft*) y determine cuando se da la igualdad.

Problemas prácticos

17.9. Escriba métodos eficientes (proporcione las medidas O) que dada una referencia a un árbol binario T calculen lo siguiente:

a) El número de hojas de T.
b) El número de nodos de T que contienen exactamente un hijo distinto de `null`.
c) El número de nodos de T que contienen exactamente dos hijos distintos de `null`.

17.10. Implemente algunas de las rutinas recursivas incluyendo comprobaciones que garanticen que no se realiza ninguna llamada recursiva sobre un subárbol `null`. Compare el tiempo de ejecución con rutinas idénticas que retrasen el test hasta la primera línea del método recursivo.

17.11. Reescriba la clase iteradora de modo que se lance una excepción cuando se aplica `primero` a un árbol vacío. ¿Por qué puede ser esto una idea poco afortunada?

```
1  public void imprimirMisterioso( NodoBinario a )
2  {
3      if( a != null )
4      {
5          System.out.println( a.dato );
6          imprimirMisterioso( a.izquierdo );
7          System.out.println( a.dato );
8          imprimirMisterioso( a.derecho );
9          System.out.println( a.dato );
10     }
11 }
```

Figura 17.34 Programa misterioso para el Ejercicio 17.3.

Prácticas de programación

17.12. Implemente un comando que liste todos los ficheros de un directorio dado, incluyendo todos los subdirectorios, ordenados por tamaño decreciente. Incluya los tamaños y cualquier otra información que pueda obtener de los ficheros.

17.13. Mediante un programa se puede generar automáticamente un árbol binario para autoedición. Esto se podría hacer asignando una coordenada (x, y) a cada nodo del árbol, dibujando un círculo alrededor de dicha coordenada y conectando cada nodo distinto de la raíz con su padre. Supongamos que disponemos de un árbol binario almacenado en memoria y que cada nodo tiene dos atributos extra en los que almacenar sus coordenadas. Supongamos que (0, 0) se corresponde con la esquina superior izquierda. Se pide hacer lo siguiente:

a) La coordenada x se puede calcular usando el número que le corresponde en el recorrido en orden simétrico. Escriba una rutina que haga esto para cada nodo del árbol.
b) La coordenada y se puede calcular usando la profundidad del nodo. Escriba una rutina que haga esto para cada nodo del árbol.
c) Determine, en términos de una unidad imaginaria, cuáles serán las dimensiones del dibujo. También determine cómo ajustar las unidades de forma que la altura del árbol siempre sea aproximadamente dos tercios de su anchura.
d) Demuestre que cuando se usa este sistema, las aristas no se cruzan y que para cada nodo X, todos los elementos del subárbol izquierdo de X aparecen a su izquierda y todos los elementos del subárbol derecho de X aparecen a su derecha.
e) Determine si ambas coordenadas se pueden calcular por medio de un único método recursivo.
f) Escriba un programa de dibujo de árboles de propósito general que convierta un árbol en una secuencia de instrucciones de generación de gráficos del siguiente estilo (los círculos están numerados en el orden en que se pintan):

```
circulo( x, y ); // Dibuja un círculo con centro en (x, y)
dibujaLinea( i, j ); // Conecta el circulo i con el j
```

g) Escriba un programa que lea las instrucciones de generación de gráficos y genere código Java para dibujar sobre un lienzo (observe que tiene que escalar las coordenadas almacenadas en pixels).

17.14. Diseñe un applet que ilustre las distintas estrategias de recorrido de árboles.

CAPÍTULO

18 Árboles binarios de búsqueda

Para entradas de gran tamaño, el acceso en tiempo lineal de las listas enlazadas es prohibitivo. Este capítulo examina una opción alternativa a la lista enlazada: el *árbol binario de búsqueda*. El árbol binario de búsqueda es una estructura de datos sencilla que puede considerarse como una extensión del algoritmo de búsqueda binaria que permite tanto inserciones como eliminaciones. El tiempo de ejecución de muchas de sus operaciones es, en media, $O(\log N)$. Desgraciadamente, su eficiencia en el caso peor es $O(N)$.

En este capítulo veremos:

- En qué consiste el árbol binario de búsqueda básico.
- Cómo incluir búsquedas por posición en el orden (esto es, la operación `encontrarKesimo`).
- Tres modos diferentes de eliminar el caso peor $O(N)$ (los árboles AVL, los árboles rojinegros y los AA-árboles).
- Cómo puede mejorarse la eficiencia de las consultas en una base de datos de gran tamaño, empleando B-árboles.

18.1 Ideas básicas

Por lo general, un elemento se busca por su clave. Por ejemplo, un estudiante puede buscarse en la base de datos utilizando su DNI. En este caso, el número de DNI se considera la clave del elemento.

El *árbol binario de búsqueda* es un árbol binario que satisface la propiedad de la búsqueda ordenada. Esto significa que para cada nodo X del árbol, los valores de todas las claves de su subárbol izquierdo son menores que la clave de X y los valores de todas las claves de su subárbol derecho son mayores que la clave de X. En la Figura 18.1, el árbol de la izquierda es un árbol binario de búsqueda, mientras que el árbol de la derecha no lo es (la clave 8 no debería pertenecer al subárbol izquierdo de la clave 7). La propiedad que define los árboles binarios de búsqueda implica que todos los elementos del árbol pueden ordenarse de forma consistente (el recorrido en orden simétrico de la estructura muestra los elementos ordenados en forma ascendente). Dicha propiedad no permite en principio la exis-

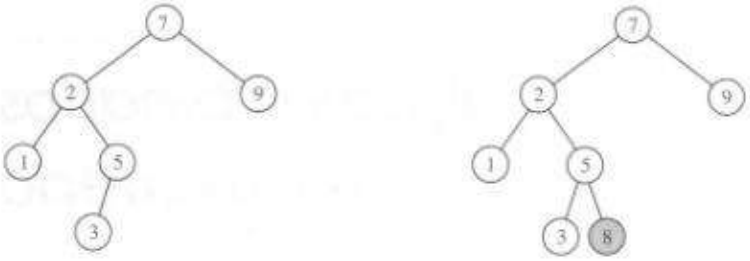


Figura 18.1 Dos árboles binarios (sólo el de la izquierda es de búsqueda).

tencia de elementos duplicados, pero es sencillo generalizarla de modo que se permita la repetición de claves. Generalmente, es mejor guardar los elementos con la misma clave en una estructura secundaria. Si los elementos son duplicados exactos, la mejor opción suele ser tener uno solo y mantener un contador con el número de repeticiones.

18.1.1 Las operaciones

La mayoría de las operaciones de los árboles binarios de búsqueda son sencillas de comprender. Podemos realizar una operación `buscar` comenzando en la raíz y desplazándonos repetidamente por las ramas izquierda o derecha, dependiendo del resultado de las comparaciones. Por ejemplo, para encontrar el elemento 5 en el árbol binario de búsqueda de la Figura 18.1, empezamos en el 7 y vamos por la rama izquierda. Esto nos conduce al 2, y desde aquí vamos a la derecha. Así, llegamos al 5. Para encontrar el 6, hubiéramos seguido el mismo camino. Cuando estuviésemos en el 5, iríamos por la derecha, encontrando una referencia `null`, por lo que el 6 no está en el árbol. La Figura 18.2 muestra que el 6 puede insertarse en el lugar donde ha finalizado su búsqueda fallida.

El árbol binario de búsqueda soporta de modo eficiente las operaciones `buscarMin` y `buscarMax`. Para ejecutar `buscarMin`, partimos de la raíz y recorremos repetidamente todos los nodos izquierdos hasta llegar a un nodo que no tenga hijo izquierdo. Dicha hoja es el elemento mínimo del árbol. `buscarMax` es análoga, salvo por el hecho de que el recorrido es ahora por la derecha. Nótese que el coste de ambas operaciones es proporcional al número de nodos del camino de búsqueda. El coste en media es logarítmico, pero puede ser lineal en el peor de los casos. Esto se explica posteriormente en este capítulo.

Figura 18.2 Los árboles binarios de búsqueda antes y después de insertar el 6.

La operación más complicada es `eliminar`. Una vez que hemos encontrado el nodo que debe ser borrado, es preciso considerar varias alternativas. El problema es que la eliminación de un nodo puede desconectar varias partes del árbol. Debemos ser cuidadosos en su manipulación de modo que se mantenga siempre la propiedad de la búsqueda ordenada. Además, es necesario evitar que el árbol tenga demasiada profundidad, puesto que, como ya hemos comentado anteriormente, la profundidad del árbol influye en el tiempo de ejecución de los algoritmos.

Si el nodo a borrar es una hoja, su eliminación no desconectará el árbol, por lo que puede ser eliminado sin más. Si el nodo tiene un solo hijo, podemos eliminarlo después de que su padre haya ajustado sus referencias hacia los hijos, saltando al nodo eliminado. Esto se muestra en la Figura 18.3, en la que eliminamos el 5. Observe que esto implica que `eliminarMin` y `eliminarMax` no son excesivamente complejos, ya que los nodos afectados son hojas o tienen un único hijo. Nótese también que la raíz es un caso especial porque no tiene padre. Sin embargo, cuando implementemos el método `eliminar`, éste se tratará en principio de forma uniforme, sin ser necesario un tratamiento especial.

El caso complicado llega cuando debemos eliminar un nodo con dos hijos. La estrategia general es reemplazar el elemento de ese nodo con el menor elemento de su subárbol derecho (que, como ya hemos comentado, se encuentra fácilmente) y eliminar después el nodo del menor elemento (que se considera vacío desde el punto de vista lógico). La segunda operación `eliminar` es sencilla, ya que el nodo mínimo de un árbol no tiene hijo izquierdo. La Figura 18.4 muestra un árbol

Figura 18.3 Los árboles binarios de búsqueda antes y después de eliminar el 5 (que tenía un único hijo).

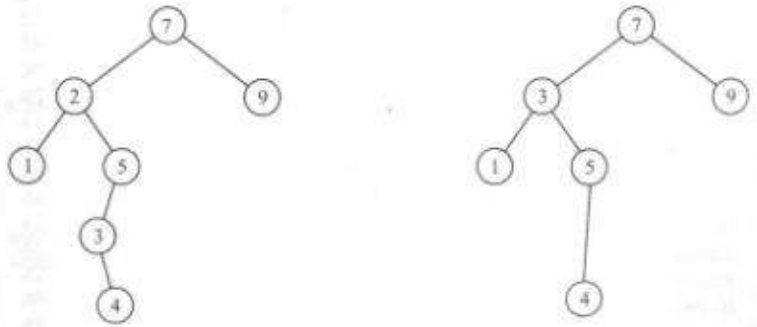


Figura 18.4 Los árboles binarios de búsqueda antes y después de eliminar el 2 (que tenía dos hijos).

inicial y el resultado tras eliminar el nodo 2. Sustituimos dicho nodo por el nodo de menor valor (3) de su subárbol derecho. Nótese que en cualquiera de los casos, la eliminación de un nodo no incrementa la profundidad del árbol.

18.1.2 Implementación en Java

En principio, los árboles binarios de búsqueda son sencillos de implementar. Unas pequeñas simplificaciones evitan que el código se complique en exceso. Para empezar, la Figura 18.5 muestra la clase `NodoBinario`. Como es usual, el acceso a los atributos es amistoso, pero la clase sólo es accesible desde dentro de su paquete. En las líneas 31 a 33 hemos incluido algunos atributos adicionales, ya que deseamos emplear las mismas declaraciones en árboles binarios de búsqueda más complejos, que serán discutidos más tarde en este capítulo. Por tanto, dichos atributos adicionales no se emplean en las implementaciones de esta sección. Una opción sería omitirlos de momento y emplear posteriormente la herencia, pero esto oscurece demasiado los conceptos básicos. En consecuencia, la clase `NodoBinario`

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4
5  // Nodo básico almacenado en todos los árboles binarios de búsqueda.
6  // Incluye los campos para todas las variaciones.
7  // Esta clase no es accesible fuera del
8  // paquete EstructurasDatos.
9
10 class NodoBinario
11 {
12         // Constructores
13     NodoBinario( Comparable elDato )
14     {
15         this( elDato, null, null );
16     }
17
18     NodoBinario( Comparable elDato, NodoBinario li, NodoBinario ld )
19     {
20         dato      = elDato;
21         izquierdo = li;
22         derecho   = ld;
23     }
24
25       // Atributos amistosos; accesibles por otras rutinas del paquete
26     Comparable dato;           // La información del nodo
27     NodoBinario izquierdo;     // Hijo izquierdo
28     NodoBinario derecho;       // Hijo derecho
29
30       // Información de equilibrio; se emplea una en cada tipo de árbol
31     int tamanyo = 1;  // Para árboles binarios de búsqueda con rango
32     int color   = 1;  // Para árboles rojinegros
33     int nivel   = 1;  // Para AA-árboles
34 }
```

Figura 18.5 La clase de los nodos de los árboles binarios de búsqueda

contiene la lista usual de atributos (el dato y dos referencias), más los atributos adicionales empleados más tarde. Se permiten varias formas para la construcción de un nodo.

En las Figuras 18.6 y 18.7 se muestra el esqueleto de la clase `ArbolBinarioBusqueda`. El principal atributo es una referencia a la raíz del árbol, llamada `raiz`.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase ArbolBinarioBusqueda
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ************************OPERACIONES PÚBLICAS************************
11 // void insertar( x )      --> Inserta x
12 // void eliminar( x )      --> Elimina x
13 // void eliminarMin( )     --> Elimina el menor elemento
14 // Comparable buscar( x )  --> Devuelve el elemento que ajusta con x
15 // Comparable buscarMin( ) --> Devuelve el menor elemento
16 // Comparable buscarMax( ) --> Devuelve el mayor elemento
17 // boolean esVacio( )      --> Devuelve true si vacío; si no, false
18 // void vaciar( )          --> Elimina todos los elementos
19 // void imprimirArbol( )   --> Imprime el árbol ordenadamente
20 // ************************************ERRORES************************
21 // Muchas rutinas lanzan ElementoNoEncontrado en condiciones degenerad
22 // insertar lanza ElementoDuplicado si el elemento está en el árbol
23
24 /**
25  * Implementa un árbol binario de búsqueda sin equilibrar.
26  * Nótese que todas las comparaciones se basan en el método compara.
27  */
28 public class ArbolBinarioBusqueda implements ArbolBusqueda
29 {
30     public ArbolBinarioBusqueda( )
31       { raiz = null; }
32
33     public void insertar( Comparable x ) throws ElementoDuplicado
34       { raiz = insertar( x, raiz ); }
35     public void eliminar( Comparable x ) throws ElementoNoEncontrado
36       { raiz = eliminar( x, raiz ); }
37     public void eliminarMin( ) throws ElementoNoEncontrado
38       { raiz = eliminarMin( raiz ); }
39     public Comparable buscarMin( ) throws ElementoNoEncontrado
40       { return buscarMin( raiz ).dato; }
41     public Comparable buscarMax( ) throws ElementoNoEncontrado
42       { return buscarMax( raiz ).dato; }
43     public Comparable buscar( Comparable x ) throws ElementoNoEncontrad
44       { return buscar( x, raiz ).dato; }
45     public boolean esVacio( )
46       { return raiz == null; }
47     public void vaciar( )
48       { raiz = null; }
49     public void imprimirArbol( )
50       { imprimirArbol( raiz ); }
```

Figura 18.6 Esqueleto de la clase `ArbolBinarioBusqueda` (parte 1)

```
51      protected NodoBinario
52      buscar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
53          { /* Figura 18.8 */ }
54
55      protected NodoBinario
56      buscarMin( NodoBinario t ) throws ElementoNoEncontrado
57        { /* Figura 18.9 */ }
58
59      protected NodoBinario
60      buscarMax( NodoBinario t ) throws ElementoNoEncontrado
61        { /* Figura 18.9 */ }
62
63      protected NodoBinario
64      insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
65        { /* Figura 18.10 */ }
66
67      protected NodoBinario
68      eliminarMin( NodoBinario t ) throws ElementoNoEncontrado
69        { /* Figura 18.11 */ }
70
71      protected NodoBinario
72      eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
73        { /* Figura 18.12 */ }
74
75      protected void imprimirArbol( NodoBinario t )
76        { /* No se muestra; ver el código en Internet como printTree */ }
77
78      protected NodoBinario raiz;
79  }
```

Figura 18.7 Esqueleto de la clase `ArbolBinarioBusqueda` (parte 2).

En el esqueleto aparecen siete métodos que actúan sobre el nodo que se pasa como parámetro. Ya se ha utilizado anteriormente esta técnica general en el Capítulo 17. La idea es que las clases públicas llamen a estas rutinas ocultas, pasándoles la `raiz` como parámetro. Son las rutinas ocultas las que realizan todo el trabajo. Hay dos detalles que debemos tener en cuenta. El primero de ellos es que los métodos que desde un punto de vista lógico son privados, tienen como atributo de visibilidad `protected`, en lugar de `private`. Esto es debido a que, más tarde en este capítulo, derivaremos otra clase a partir de `ArbolBinarioBusqueda`. El segundo detalle es que los métodos privados no son estáticos, aunque resulte tentador hacerlos estáticos. Pero es que si así se hiciera no podría emplearse posteriormente el tipado dinámico en la extensión de `ArbolBinarioBusqueda`.

El resto del esqueleto de `ArbolBinarioBusqueda` es una sencilla enumeración de los métodos cuyas implementaciones invocan a los métodos ocultos. El constructor, declarado en la línea 30, sólo inicializa la referencia `raiz` a `null`. De la línea 33 a la 50 se enumeran los métodos públicos junto con sus implementaciones.

`insertar` añade el elemento `x` en el árbol llamando a la rutina oculta `insertar` y pasándole `raiz` como parámetro. Se producirá un error si `x` está ya en el árbol, en cuyo caso, se lanzará una excepción `ElementoDuplicado`. `buscarMin`, `buscarMax` y `buscar` devuelven el elemento menor, mayor o justo el indicado, respectivamente. Si el elemento no está en el árbol, bien porque el árbol sea vacío o porque no hayamos insertado el elemento previamente, se lanza una excepción del tipo `ElementoNoEncontrado`.

`eliminarMin` elimina el menor elemento del árbol, y lanza una excepción si éste está vacío. `eliminar` elimina del árbol el elemento indicado `x`; también lanza, eventualmente, una excepción `ElementoNoEncontrado`. `vaciar` y `esVacio` son las rutinas usuales.

Al igual que en la mayoría de las estructuras de datos, la operación `buscar` es mucho más sencilla que `insertar`, e `insertar` es más simple que `eliminar`. La Figura 18.8 muestra la rutina `buscar`. En ella, mientras no se alcance una referencia `null`, o bien se ha encontrado el elemento buscado o debemos recorrer las ramas izquierda o derecha para hacerlo. El código implementa este algoritmo de forma bien sucinta. Observe el orden en el que se realizan las comprobaciones. Es crucial que se realice primero el test para la referencia `null`; de otro modo, el acceso `t.dato` sería incorrecto. Los tests restantes se ordenan colocando en último lugar el menos probable.

A primera vista parece que las instrucciones como `t = t.izquierdo` modifican la raíz del árbol, pero esto no es así. Nótese que en la llamada inicial, `t` es simplemente *otra referencia* al nodo al que apunta `raiz`. Aunque `t` cambie, pasando a referenciar a otros nodos, `raiz` no lo hace. Las llamadas a `buscarMin` y `buscarMax` son incluso más sencillas, ya que el recorrido se hace siempre en la misma rama. Estas rutinas se muestran en la Figura 18.9. Observe cómo se maneja cuidadosamente el caso del árbol vacío.

La rutina `insertar` se muestra en la Figura 18.10. Para simplificar el código se emplea recursión. Es posible también una implementación no recursiva; esta técnica se empleará más tarde en este mismo capítulo en los árboles rojinegros. El algoritmo básico es muy sencillo. Si el árbol está vacío, creamos un nuevo árbol con un único nodo. La comprobación se realiza en la línea 12, y el nuevo nodo se inserta en la línea 13. Al igual que antes, los cambios locales de `t` se pierden. Así, devolvemos la nueva raíz `t` en la línea 20. En la rutina pública `insertar`, se modifica `raiz` asignándole el valor que devuelve la rutina `insertar` oculta.

```
1   /**
2    * Método interno para buscar un elemento en un subárbol.
3    * @param x el elemento a buscar.
4    * @param t la raíz del subárbol.
5    * @return el nodo que contiene el elemento buscado.
6    * @exception ElementoNoEncontrado si el elemento
7    *     no está en el subárbol.
8    */
9   protected NodoBinario
10  buscar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
11  {
12      while( t != null )
13          if( x.compara( t.dato ) < 0 )
14              t = t.izquierdo;
15          else if( x.compara( t.dato ) > 0 )
16              t = t.derecho;
17          else
18              return t; // Encaja
19
20      throw new ElementoNoEncontrado( "buscar de ArbolBusqueda" );
21  }
```

Figura 18.8 Operación `buscar` de los árboles binarios de búsqueda.

```
/**
 * Método interno para buscar el menor elemento de un subárbol.
 * @param t la raíz del subárbol.
 * @return el nodo que contiene el menor elemento.
 * @exception ElementoNoEncontrado si el subárbol está vacío.
 */
protected NodoBinario
buscarMin( NodoBinario t ) throws ElementoNoEncontrado
{
    if( t == null )
        throw new ElementoNoEncontrado( "buscarMin de ArbolBusqueda" );

    while( t.izquierdo != null )
        t = t.izquierdo;
    return t;
}

/**
 * Método interno para buscar el mayor elemento de un subárbol.
 * @param t la raíz del subárbol.
 * @return el nodo que contiene el mayor elemento.
 * @exception ElementoNoEncontrado si el subárbol está vacío.
 */
protected NodoBinario
buscarMax( NodoBinario t ) throws ElementoNoEncontrado
{
    if( t == null )
        throw new ElementoNoEncontrado( "buscarMax de ArbolBusqueda" );

    while( t.derecho != null )
        t = t.derecho;
    return t;
}
```

ura 18.9 buscarMin y buscarMax de los árboles binarios de búsqueda.

```
/**
 * Método interno para insertar en un subárbol.
 * @param x el elemento a insertar.
 * @param t la raíz del subárbol.
 * @return la nueva raíz.
 * @exception ElementoDuplicado si el elemento que ajusta
 *            con x ya está en el subárbol de raíz t.
 */
protected NodoBinario
insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
{
    if( t == null )
        t = new NodoBinario( x, null, null );
    else if( x.compara( t.dato ) < 0 )
        t.izquierdo = insertar( x, t.izquierdo );
    else if( x.compara( t.dato ) > 0 )
        t.derecho = insertar( x, t.derecho );
    else
        throw new ElementoDuplicado( "insertar de ArbolBusqueda" );
    return t;
}
```

ura 18.10 Versión recursiva del método insertar de los árboles binarios de búsqueda.

Si el árbol no es vacío tenemos tres posibilidades. Si el elemento a insertar es menor que el elemento del nodo t, llamamos recursivamente a insertar sobre el subárbol derecho. Si es mayor, llamamos recursivamente a insertar sobre el subárbol izquierdo. Todo esto se codifica en las líneas 14 a 17. Nótese que devolvemos el resultado de la llamada recursiva a insertar. El tercer caso es que el elemento a insertar coincida con el elemento del nodo t; en este caso lanzamos una excepción.

Las restantes rutinas corresponden a las operaciones de eliminación. Como se ha indicado anteriormente en este capítulo, la operación eliminarMin es sencilla, ya que el nodo mínimo no tiene hijo izquierdo. Sólo debemos saltar sobre el nodo eliminado. Puede parecer que en cada paso se necesita conocer el padre del nodo actual, pero una vez más, empleando la recursión podemos evitar la referencia explícita al padre. El código se muestra en la Figura 18.11.

Si el árbol t es vacío, el método eliminarMin falla. En caso contrario, si t tiene hijo izquierdo, eliminamos de forma recursiva el menor elemento del subárbol izquierdo, por medio de la llamada recursiva de la línea 13. Si alcanzamos la línea 14, sabemos que en ese momento estamos situados sobre el menor nodo del árbol. Esto significa, en particular, que t es la raíz de un subárbol que no tiene hijo izquierdo. Si hacemos que t sea igual a t.derecho entonces t es ahora la raíz de un subárbol que ha perdido su menor elemento. Esto es lo que hacemos en la línea 15. Como antes, debemos devolver la raíz del árbol resultante. Pero, ¿al hacerlo no desconectamos el árbol? De nuevo, la respuesta es no. Si t era raiz, se devuelve el nuevo t, que se asigna a raiz en el método público, y el proceso es seguro. Si t no era raiz entonces es p.izquierdo, donde p es el padre de t en el momento de la llamada recursiva. El método que tiene a p como su parámetro (en otras palabras, el método que hizo la llamada al método actual) cambia p.izquierdo al nuevo t. Así, el hijo izquierdo del padre apunta a t, y el árbol queda conectado. Lo que hemos hecho es mantener el padre en la pila de la recursión en lugar de controlarlo explícitamente en un bucle iterativo.

Una vez empleado este truco en el caso más sencillo, podemos adaptarlo para la rutina general eliminar. Esto se muestra en la Figura 18.12. Si el árbol es vacío, el método eliminar falla, lanzándose una excepción en la línea 13. En caso

```
1  /**
2   * Método interno para eliminar el menor elemento de un subárbol.
3   * @param t la raíz del subárbol.
4   * @return la nueva raíz.
5   * @exception ElementoNoEncontrado si el subárbol está vacío.
6   */
7  protected NodoBinario
8  eliminarMin( NodoBinario t ) throws ElementoNoEncontrado
9  {
10     if( t == null )
11         throw new ElementoNoEncontrado( "eliminarMin de ArbolBusqueda
12     if( t.izquierdo != null )
13         t.izquierdo = eliminarMin( t.izquierdo );
14     else
15         t = t.derecho;
16     return t;
17 }
```

Figura 18.11 Método eliminarMin de la clase ArbolBinarioBusqueda.

```
1  /**
2   * Método interno para eliminar en un subárbol.
3   * @param x el elemento a eliminar.
4   * @param t la raíz del subárbol.
5   * @return la nueva raíz.
6   * @exception ElementoNoEncontrado si el elemento que ajusta
7   *        con x está en el subárbol de raíz t.
8   */
9  protected NodoBinario
10 eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
11 {
12     if( t == null )
13         throw new ElementoNoEncontrado( "eliminar de ArbolBusqueda" );
14     if( x.compara( t.dato ) < 0 )
15         t.izquierdo = eliminar( x, t.izquierdo );
16     else if( x.compara( t.dato ) > 0 )
17         t.derecho = eliminar( x, t.derecho );
18     else if( t.izquierdo != null && t.derecho != null ) // Dos hijos
19     {
20         t.dato = buscarMin( t.derecho ).dato;
21         t.derecho = eliminarMin( t.derecho );
22     }
23     else // Cambio de raíz
24         t = ( t.izquierdo != null ) ? t.izquierdo : t.derecho;
25     return t;
26 }
```

Figura 18.12 Método `eliminar` de la clase `ArbolBusquedaBinario`.

contrario, si el elemento a eliminar no se ajusta al nodo actual, llamamos recursivamente a `eliminar` modificando su parámetro, haciendo que éste sea el hijo izquierdo o el derecho, según corresponda. En otro caso alcanzamos la línea 18, habiendo encontrado el nodo a eliminar.

Recordemos que cuando el nodo a eliminar tiene dos hijos, lo reemplazamos con el menor elemento del subárbol derecho y después eliminamos dicho elemento mínimo. Esto se codifica en las líneas 20 y 21. En caso contrario, tenemos uno o ningún hijo. Si existe hijo izquierdo, entonces se iguala `t` a su hijo izquierdo, tal y como se haría en `eliminarMax`. Si no, sabemos que no hay hijo izquierdo y podemos igualar `t` a su hijo derecho. Todo esto se codifica de forma breve y sencilla en la línea 24, en la que también se cubre el caso de las hojas. Tras cualquiera de estas alternativas, el método `eliminar` devuelve la raíz del subárbol en la línea 25.

Existen dos detalles en esta implementación que debemos comentar. En primer lugar, durante las operaciones básicas de `insertar`, `buscar`, o `eliminar`, se hacen dos comparaciones por nodo, con el propósito de distinguir entre los casos <, = y >. Pero podríamos conseguir lo mismo con una sola comparación por nodo. La estrategia es muy similar a la seguida en el algoritmo de búsqueda binaria de la Sección 5.6. La adecuación de esta técnica para los árboles binarios de búsqueda se discute en la Sección 18.6.2, donde se trata el algoritmo de eliminación de los AA-árboles.

El segundo punto es que no tenemos por qué emplear la recursión para la inserción de elementos. De hecho, una implementación recursiva es, probablemente, más lenta que una no recursiva. En la Sección 18.5.3 se discute una implementación iterativa de `insertar`, en el contexto de los árboles rojinegros.

18.2 Búsqueda por posición en el orden

El árbol binario de búsqueda nos permite encontrar los elementos mínimo o máximo en tiempo equivalente al de una operación arbitraria `buscar`. En algunas ocasiones es importante poder acceder al K-ésimo menor elemento, para un valor arbitrario K. Esto se consigue fácilmente si mantenemos información sobre el tamaño de cada nodo del árbol.

Recuerde, de la Sección 17.1, que el tamaño de un nodo es su número de descendientes (incluyéndole a él mismo). Supongamos que se desea encontrar el K-ésimo menor elemento y que K es mayor o igual que 1 y menor o igual que la cantidad de nodos del árbol. La Figura 18.13 muestra que existen tres casos posibles, dependiendo de la relación entre K y el tamaño del subárbol izquierdo, denotado por S_L. Si K es igual a $S_L + 1$ entonces el K-ésimo menor elemento es la raíz, y hemos terminado. Si K es menor o igual que S_L entonces el K-ésimo menor elemento debe estar en el subárbol izquierdo, y podemos encontrarlo de forma recursiva. Una vez más, la recursión no es obligatoria; se emplea para simplificar la descripción del algoritmo. En el caso restante, el K-ésimo menor elemento es el $(K - S_L - 1)$-ésimo elemento más pequeño del subárbol derecho, y también puede encontrarse de forma recursiva.

El mayor esfuerzo se concentra en mantener el tamaño de los nodos durante los cambios. Estas modificaciones se producen en las operaciones `insertar`, `eliminar` y `eliminarMin`. En principio, el mantenimiento de la información es bastante simple. Durante una inserción, cada nodo del camino hasta el punto de inserción gana un nodo en su subárbol. Como consecuencia, el tamaño de cada uno de ellos se incrementa en una unidad, y el nuevo nodo insertado tiene tamaño 1. En `eliminarMin`, cada nodo del camino hasta el mínimo pierde un nodo en su subárbol, y así, su tamaño decrece en una unidad. Durante una operación `eliminar`, todos los nodos del camino hasta el que va a ser eliminado también pierden un nodo en sus subárboles. Como consecuencia de todo ello, podemos mantener fácilmente el tamaño de los nodos con una pequeña cantidad adicional de esfuerzo.

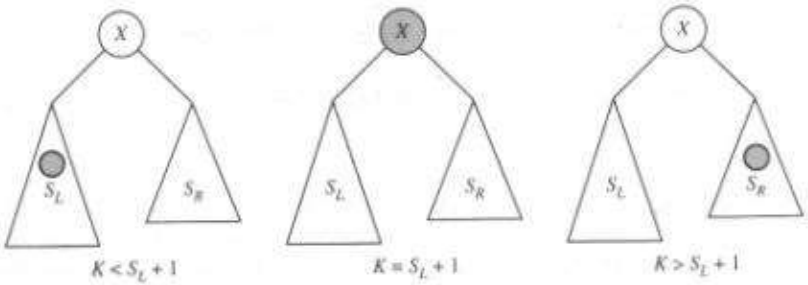


Figura 18.13 Empleo del atributo `tamanyo` para implementar el método `buscarKesimo`.

18.2.1 Implementación en Java

Desde el punto de vista lógico, los únicos cambios necesarios son añadir la operación `buscarKesimo` y mantener la información del atributo `tamanyo` en las rutinas `insertar`, `eliminar` y `eliminarMin`. Derivamos una nueva clase a partir de `ArbolBinarioBusqueda`, cuyo esqueleto se muestra en la Figura 18.14.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase ABBConRango
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ******************OPERACIONES PÚBLICAS*******************
11 // void insertar( x )      --> Inserta x
12 // void eliminar( x )      --> Elimina x
13 // void eliminarMin( )     --> Elimina el menor elemento
14 // Comparable  buscar( x ) --> Devuelve el elemento que ajusta con x
15 // Comparable buscarMin( ) --> Devuelve el menor elemento
16 // Comparable buscarMax( ) --> Devuelve el mayor elemento
17 // Comparable buscarKesimo( int k )
18 //                         --> Buscar K-ésimo menor elemento
19 // boolean esVacio( )      --> Devuelve true si vacío; si no, false
20 // void vaciar( )          --> Elimina todos los elementos
21 // void imprimirArbol( )   --> Imprime el árbol ordenadamente
22 // ********************ERRORES*****************************
23 // Muchas rutinas lanzan ElementoNoEncontrado en condiciones
     degeneradas
24 // insertar lanza ElementoDuplicado si el elemento está en el árbol
25
26 /**
27  * Implementa un árbol binario de búsqueda sin equilibrar.
28  * Nótese que todas las comparaciones se basan en el método compara.
29  */
30 public class ABBConRango extends ArbolBinarioBusqueda
31 {
32   public Comparable buscarKesimo( int k ) throws ElementoNoEncontrado
33     { return buscarKesimo( k, raiz ).dato; }
34
35   // Los métodos internos insertar, eliminar y eliminarMin se
36   // sobrescriben. Se añade el método interno para buscarKesimo
37 }
```

Figura 18.14 Esqueleto de la clase de los árboles binarios de búsqueda con búsqueda por posición.

Veamos los nuevos métodos públicos. Como el constructor no se define explícitamente, se emplea el constructor generado por defecto; los atributos heredados son inicializados por el constructor de `ArbolBinarioBusqueda`. El nuevo método público `buscarKesimo` se declara en la línea 32 y llama al método privado correspondiente. Los métodos públicos, como `insertar`, no necesitan programarse de nuevo, ya que sus cuerpos son idénticos a las versiones en `ArbolBinarioBusqueda`.

Los que cambian son los métodos de ayuda privados. Necesitamos que el `insertar` público invoque al correspondiente método de ayuda privado. Observe, sin embargo, que los métodos privados no deben tener el atributo `static`, ya que la decisión debe hacerse empleando ligado dinámico, en lugar de ligado estático.

La operación `buscarKesimo` de la Figura 18.15 se ha escrito de forma recursiva, aunque claramente no necesita serlo. Esta rutina sigue la descripción del algoritmo, línea a línea. La comprobación de la referencia `null` en la línea 11 es

```
1  /**
2   * Método interno para buscar el K-ésimo menor elemento de un subárbol.
3   * @param k el orden del elemento deseado (1 es el menor elemento).
4   * @return el nodo que contiene el K-ésimo menor elemento del subárbol.
5   * @exception ElementoNoEncontrado si k es menor que
6   *     1 o mayor que el tamaño del subárbol.
7   */
8  protected NodoBinario
9  buscarKesimo( int k, NodoBinario t ) throws ElementoNoEncontrado
10 {
11   if( t == null )
12     throw new ElementoNoEncontrado( "buscarKesimo de ABBR" );
13   int tamanyoIzquierdo = ( t.izquierdo != null ) ? t.izquierdo.tamanyo : 0;
14
15   if( k <= tamanyoIzquierdo )
16     return buscarKesimo( k, t.izquierdo );
17   if( k == tamanyoIzquierdo + 1 )
18     return t;
19   else
20     return buscarKesimo( k - tamanyoIzquierdo - 1, t.derecho );
21 }
```

Figura 18.15 Operación `buscarKesimo` de los árboles binarios de búsqueda con búsqueda por posición.

necesaria, ya que `k` puede no ser válido. En la línea 13 se calcula el tamaño del subárbol izquierdo. Si éste existe, el acceso a su atributo `tamanyo` nos da la respuesta pedida. Si no existe, podemos considerar que su tamaño es cero. Nótese que este test se realiza después de asegurar que la referencia `t` no es `null`.

`insertar` se muestra en la Figura 18.16. La parte que potencialmente tiene truco es que cuando la inserción tiene éxito, queremos en efecto incrementar el

```
1  /**
2   * Método interno para insertar en un subárbol, ajustando
3   *     el tamaño según corresponda.
4   * @param x el elemento a insertar.
5   * @param t la raiz del subárbol.
6   * @return la nueva raiz.
7   * @exception ElementoDuplicado si el elemento que ajusta con
8   *     x ya está en el subárbol de raiz t.
9   */
10 protected NodoBinario
11 insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
12 {
13     if( t == null )
14         return new NodoBinario( x, null, null );
15     else if( x.compara( t.dato ) < 0 )
16         t.izquierdo = insertar( x, t.izquierdo );
17     else if( x.compara( t.dato ) > 0 )
18         t.derecho = insertar( x, t.derecho );
19     else
20         throw new ElementoDuplicado( "insertar de ABBR" );
21
22     t.tamanyo++;
23     return t;
24 }
```

Figura 18.16 Operación `insertar` de los árboles binarios de búsqueda con búsqueda por posición.

```
/**
 * Método interno para eliminar el menor elemento de un subárbol,
 *     ajustando el tamaño de los nodos cuando corresponda.
 * @param t la raíz del subárbol.
 * @return la nueva raíz.
 * @exception ElementoNoEncontrado si el subárbol está vacío.
 */
protected NodoBinario
eliminarMin( NodoBinario t ) throws ElementoNoEncontrado
{
    if( t == null )
        throw new ElementoNoEncontrado( "eliminarMin de ABBR" );
    if( t.izquierdo == null )
        return t.derecho;
    t.izquierdo = eliminarMin( t.izquierdo );

    t.tamanyo--;
    return t;
}
```

Figura 18.17 Operación `eliminarMin` de los árboles binarios de búsqueda con búsqueda por posición.

```
/**
 * Método interno para eliminar de un subárbol, ajustando
 *     el tamaño de los nodos cuando corresponda.
 * @param x el elemento a eliminar.
 * @param t la raíz del subárbol.
 * @return la nueva raíz.
 * @exception ElementoNoEncontrado si no hay ningún elemento
 *     que ajuste con x en el subárbol de raíz t.
 */
protected NodoBinario
eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
{
    if( t == null )
        throw new ElementoNoEncontrado( "eliminar de ABBR" );
    if( x.compara( t.dato ) < 0 )
        t.izquierdo = eliminar( x, t.izquierdo );
    else if( x.compara( t.dato ) > 0 )
        t.derecho = eliminar( x, t.derecho );
    else if( t.izquierdo != null && t.derecho != null ) // Dos hijos
    {
        t.dato = buscarMin( t.derecho ).dato;
        t.derecho = eliminarMin( t.derecho );
    }
    else
        return ( t.izquierdo != null ) ? t.izquierdo : t.derecho;
    t.tamanyo--;
    return t;
}
```

Figura 18.18 Operación `eliminar` de los árboles binarios de búsqueda con búsqueda por posición.

atributo `tamanyo` de `t` y devolver la nueva raíz del subárbol. Pero si la llamada recursiva falla, dicho atributo no debe modificarse, debiéndose lanzar una excepción. Ahora bien, ¿es posible que en una inserción fallida los tamaños de algunos nodos cambien? La respuesta es no: `tamanyo` sólo se actualiza si la llamada recursiva termina sin lanzar ninguna excepción. Nótese que cuando se genera un nuevo nodo mediante la llamada `new`, el atributo `tamanyo` se inicializa a 1 en el constructor de `NodoBinario`.

La Figura 18.17 muestra que el mismo truco puede utilizarse en `eliminarMin`. Si la llamada recursiva finaliza con éxito, el atributo `tamanyo` se decrementa; si la llamada recursiva falla, `tamanyo` permanece invariante. `eliminar` es parecido, y se muestra en la Figura 18.18.

18.3 Análisis de las operaciones de los árboles binarios de búsqueda

Es fácil comprobar que el coste de cada operación de los árboles binarios de búsqueda (`insertar`, `buscar` y `eliminar`) es proporcional al número de nodos consultados durante la operación. Así, podemos considerar que el coste del acceso a cada nodo es 1 más su profundidad (recordamos que la profundidad mide el número de arcos en lugar del número de nodos). Todo esto nos da el coste de una búsqueda con éxito.

La Figura 18.19 muestra dos árboles. El de la izquierda es un árbol bien equilibrado con 15 nodos. El coste de acceder a cualquier nodo es de a lo sumo, 4 unidades, aunque algunos nodos requieren menos accesos. Esto es prácticamente análogo a la situación que se produce en el algoritmo de búsqueda binaria. En consecuencia, si el árbol está bien equilibrado, el coste de los accesos es logarítmico.

Desafortunadamente, no tenemos la garantía de que todo el árbol esté bien equilibrado. El segundo árbol de la Figura 18.19 es el ejemplo clásico de árbol no equilibrado. Aquí, los N nodos están en el camino a recorrer hasta llegar al nodo de mayor profundidad, de forma que el coste de la búsqueda en el peor de los casos es $O(N)$. Debido a que el árbol de búsqueda ha degenerado en una lista

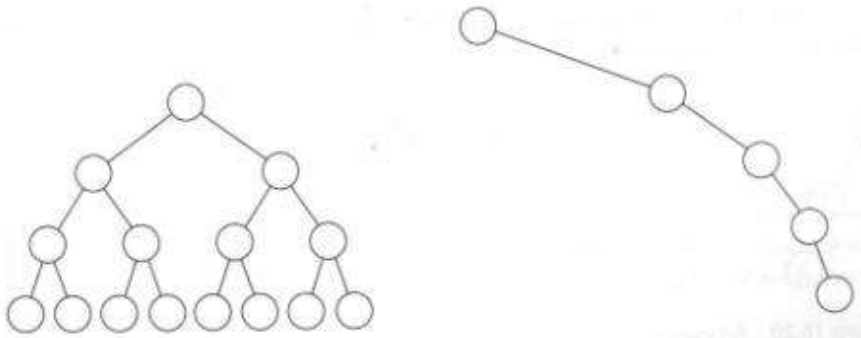


Figura 18.19 El árbol bien equilibrado de la izquierda tiene una profundidad de $\lfloor \log N \rfloor$; el árbol no equilibrado de la derecha tiene una profundidad de $N - 1$.

enlazada, el tiempo medio de la búsqueda en *esta instancia particular* es la mitad del coste en el caso peor, que también es $O(N)$. Así, se tienen dos extremos: en el caso mejor, el coste de los accesos es logarítmico, y en el caso peor es lineal.

Entonces, ¿cuál es la media? ¿Tienden la mayoría de los árboles binarios de búsqueda a inclinarse hacia los casos bien o mal equilibrados, o existe algún punto medio, como $\sqrt{N}$? La respuesta es idéntica a la obtenida en el caso de quicksort: el caso medio es un 38 por ciento peor que el caso mejor.

En esta sección demostramos que la profundidad media de los nodos de un árbol binario de búsqueda es logarítmica, asumiendo que cada árbol se genera como resultado de secuencias aleatorias de inserción (sin operaciones de tipo `eliminar`). Para comprender qué significa esto, considere el resultado de insertar tres elementos en el árbol vacío. Como sólo es importante su orden relativo, podemos suponer, sin pérdida de generalidad, que los tres elementos son 1, 2 y 3. Entonces, existen seis posibles órdenes de inserción: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) y (3, 2, 1). En nuestra demostración asumiremos que cada uno de estos casos es equiprobable. Los árboles binarios de búsqueda que resultan de estas inserciones se muestran en la Figura 18.20. Nótese que el árbol de raíz 2 se genera también por la secuencia (2, 1, 3). Así que algunos árboles son más probables que otros y, tal y como se muestra, los árboles bien equilibrados son más probables que los mal equilibrados (aunque esto no es evidente a partir del ejemplo propuesto por ser éste demasiado pequeño).

Comenzamos con la siguiente definición:

DEFINICIÓN: La *longitud del camino interno* de un árbol binario es igual a la suma de las profundidades de sus nodos.

Cuando en un árbol dividimos la longitud de su camino interno por el número de nodos del árbol, obtenemos la profundidad media de un nodo. Incrementando esta media en una unidad, obtenemos el coste medio de una búsqueda con éxito en el árbol. Por este motivo deseamos calcular la longitud media del camino interno de un árbol binario de búsqueda, donde la media se calcula considerando equiprobables todas las permutaciones de entrada. Esto se hace fácilmente considerando el árbol desde el punto de vista recursivo y empleando las técnicas mostradas en el análisis del quicksort en la Sección 8.6. La longitud media del camino interno se establece en al Teorema 18.1.

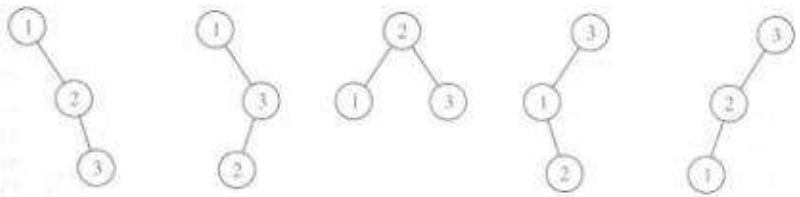


Figura 18.20 Árboles binarios de búsqueda obtenidos de la inserción de la permutación de 1, 2 y 3; la probabilidad de obtener el árbol bien equilibrado del centro es el doble que la de obtener cada uno de los restantes resultados.

La longitud del camino interno de un árbol binario de búsqueda es, aproximadamente en media, $1{,}38N\log N$, asumiendo que todas las permutaciones son equiprobables.

Sea $D(N)$ la longitud media del camino interno de los árboles de N nodos. $D(1) = 0$. Un árbol T con N nodos está formado por un subárbol izquierdo de i nodos, un subárbol derecho de $(N - i - 1)$-nodos, además de una raíz a profundidad 0 con $0 \leqslant i < N$. Por hipótesis, cada valor de i es igualmente probable. Para un i dado, $D(i)$ es la longitud media del camino interno del subárbol izquierdo respecto a su raíz. En T, todos estos nodos están un nivel más abajo. Así, la contribución media de los nodos del subárbol izquierdo a la longitud media del camino interno de T es $(1/N)\sum_{i=0}^{N-1} D(i)$, más 1 para cada nodo del subárbol izquierdo. Lo mismo se verifica para el subárbol derecho. Obtenemos así la recurrencia $D(N) = (2/N)(\sum_{i=0}^{N-1} D(i)) + N - 1$, que es idéntica a la recurrencia obtenida para quicksort, en la Sección 8.6, y que allí resolvimos. Como consecuencia, obtenemos que la longitud media del camino interno es $O(N\log N)$.

El algoritmo de inserción implica que el coste de una inserción es igual al coste de una búsqueda sin éxito, el cual se mide usando la *longitud del camino externo*. En algunas ocasiones, durante una inserción o en una búsqueda sin éxito, se alcanza el test `t==null`. Recordemos que en un árbol de N nodos existen $N + 1$ referencias `null`. La longitud del camino externo es el número total de nodos a los que se accede, incluyendo el nodo `null` para cada una de esas $N + 1$ referencias `null`. En ocasiones el nodo `null` recibe el nombre de *nodo externo*, lo que explica el término *longitud del camino externo*. Como se muestra posteriormente en este mismo capítulo, a veces es conveniente emplear un sustituto para el nodo `null`.

DEFINICIÓN: La *longitud del camino externo* de un árbol binario de búsqueda es la suma de los costes de acceso a todas las referencias `null`. Para estos propósitos, la referencia `null` de las hojas se considera como un nodo.

Dividiendo la longitud media del camino externo entre $N + 1$, obtenemos el coste medio de una inserción o de una búsqueda sin éxito. Al igual que en el algoritmo de búsqueda binaria, el coste medio de una búsqueda sin éxito es ligeramente superior al coste de una búsqueda exitosa. Ésta es la conclusión del Teorema 18.2

Para cualquier árbol T, si $IPL(T)$ es la longitud de su camino interno y $EPL(T)$ la longitud de su camino externo, entonces, si T tiene N nodos, se tiene $EPL(T) = IPL(T) + 2N$.

El teorema se demuestra por inducción y se deja como ejercicio al lector en el Ejercicio 18.8.

Es tentador concluir sin más que de estos resultados se sigue que el tiempo medio de ejecución de todas las operaciones es $O(\log N)$. Esto parece ser cierto en la práctica, pero no ha sido establecido analíticamente debido a que en la demostración de los resultados anteriores no se ha tenido en cuenta el efecto de las operaciones de eliminación. De hecho, un examen más detallado sugiere que nuestro algoritmo de eliminación probablemente planteará problemas, ya que `eliminar` siempre reemplaza un nodo borrado con dos hijos con un nodo del subárbol derecho. Podría parecer que el efecto de este reemplazamiento es el desequilibrio del árbol, inclinándolo hacia la izquierda. De hecho se ha demostrado que si se construye un árbol binario de búsqueda de forma aleatoria y se hacen sobre él N^2 pares de combinaciones aleatorias `insertar`/`eliminar` (en las que el orden de `insertar` y `eliminar` sea también aleatorio), entonces los árboles de búsqueda binarios resultantes tendrán una profundidad de $O(\sqrt{N})$. Sin embargo, no se ha podido demostrar que si nos limitamos a una cantidad razonable de operaciones `insertar` y `eliminar` aleatorias el árbol se desequilibre de forma apreciable. De hecho, y de forma un tanto sorprendente, para árboles de búsqueda pequeños, el algoritmo `eliminar` parece equilibrar el árbol. Como consecuencia, parece razonable asumir que para datos de entrada aleatorios, todas las operaciones tienen, en media, coste logarítmico, aunque esto no haya sido probado formalmente. En el Ejercicio 18.26 se describen algunas estrategias alternativas de eliminación.

El problema más importante no es el posible desequilibrio causado por el algoritmo `eliminar`, sino el hecho de que la secuencia de entrada esté ordenada, es entonces cuando nos encontramos en el caso peor. Cuando esto ocurre, tenemos un grave problema: el coste de cada operación es lineal (para secuencias de N operaciones) en lugar de logarítmico. La situación se corresponde a la diferencia entre el algoritmo quicksort y el de la ordenación por inserción. El tiempo de ejecución resultante es completamente inaceptable.

Más aún, no son sólo problemáticas las entradas ordenadas, sino cualquier entrada que contenga secuencias largas no aleatorias. Una solución a este problema es insistir en una condición estructural adicional llamada *equilibrio*: no se permite que ningún nodo esté a demasiada profundidad.

Existen numerosos métodos para implementar *árboles binarios de búsqueda bien equilibrados*. Muchos de ellos son bastante más complicados que los árboles binarios de búsqueda estándar, y en media, consumen más tiempo en las inserciones y las eliminaciones. Sin embargo, evitan los problemas de los embarazosos casos simples. Además, al generar árboles equilibrados, se consigue que los accesos sean más eficientes. Normalmente, las longitudes de sus caminos internos están bastante más cerca del caso óptimo $N \log N$ que lo que supondría el $1{,}38N \log N$ que se alcanzaba en el caso ordinario, de modo que el tiempo de búsqueda medio viene a ser un 25 por ciento menor que en dicho caso.

18.4 Árboles AVL

El primer tipo de árbol binario de búsqueda bien equilibrado fue el *árbol AVL* (debe su nombre a sus autores Adelson-Velskii y Landis). Los árboles AVL ilustran las ideas básicas de una amplia clase de árboles binarios de búsqueda bien equilibrados. Se tratan de árboles binarios de búsqueda con una condición adicional de equilibrio. Dicha condición debe ser fácil de mantener, asegurando que la profundidad del árbol sea siempre $O(\log N)$. La idea más sencilla es exigir que los subárboles izquierdo y derecho tengan la misma profundidad. La recursión nos indica que dicha idea se aplica a todos los nodos del árbol, ya que cada uno de ellos es a su vez la raíz de algún subárbol. Esta condición asegura que la profundidad del árbol es logarítmica. Sin embargo, es demasiado restrictiva ya que es complicado mantener la condición de equilibrio al insertar nuevos elementos. Así, los árboles AVL emplean una noción de equilibrio que es algo más débil, pero lo suficientemente fuerte para garantizar la profundidad logarítmica.

18.4.1 Propiedades

DEFINICIÓN: Un *árbol AVL* es un árbol binario de búsqueda con una propiedad adicional de equilibrio, según la cual, las alturas de los hijos derecho e izquierdo sólo pueden diferir, a lo sumo, en una unidad. Como es usual, tomamos -1 como la altura del árbol vacío.

La Figura 18.21 muestra dos árboles binarios de búsqueda. El árbol de la izquierda satisface la condición de los árboles AVL y por tanto es un árbol AVL. El árbol de la derecha, que se obtiene al insertar 1 empleando el algoritmo ordinario, no es un árbol AVL, ya que los hijos izquierdos de los nodos más oscuros son dos unidades más profundos que los correspondientes hijos derechos. Si se inserta el elemento 13 empleando el algoritmo ordinario de inserción en un árbol binario de búsqueda, el nodo 16 tampoco cumpliría la condición. Esto es debido a que la altura del subárbol izquierdo sería 1, mientras que la del subárbol derecho sería -1.

La condición de equilibrio AVL implica que el árbol tiene siempre una profundidad logarítmica. Para demostrarlo, basta mostrar que un árbol de altura H debe tener, por lo menos, C^H nodos, donde C es una constante mayor que 1. En otras palabras, el número mínimo de nodos de un árbol crece exponencialmente con la altura. Así, la profundidad máxima de un árbol con N elementos es $\log_C N$. El Teorema 18.3 muestra que todo árbol AVL de altura H tiene *muchos* nodos.

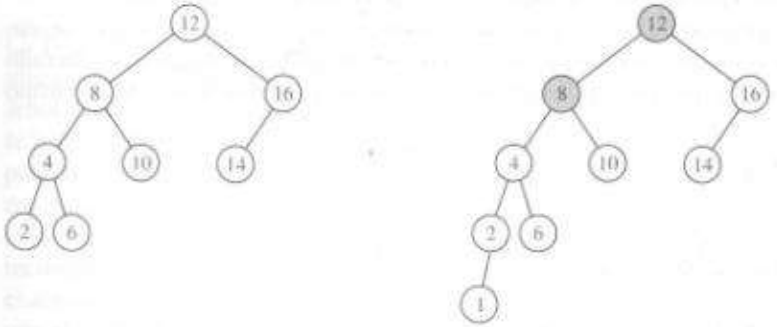


Figura 18.21 Dos árboles binarios de búsqueda: el árbol de la izquierda es un árbol AVL, mientras que el de la derecha no lo es (los nodos no equilibrados aparecen en un tono más oscuro).

Un árbol AVL de altura H tiene por lo menos $F_{H+3} - 1$ nodos, donde F_i es el i-ésimo número de Fibonacci (véase la Sección 7.3.4).

Sea S_H el tamaño del menor árbol AVL de altura H. Claramente $S_0 = 1$ y $S_1 = 2$. La Figura 18.22 muestra que el menor árbol AVL de altura H debe tener subárboles de tamaño $H - 1$ y $H - 2$. Esto es debido a que al menos un subárbol tiene altura $H - 1$ y la condición de equilibrio implica que las alturas de los subárboles pueden diferir a lo sumo en una unidad. Dichos subárboles deben tener el menor número de nodos posible para su altura, luego $S_H = S_{H-1} + S_{H-2} + 1$. Es sencillo completar la demostración empleando inducción.

Del Ejercicio 7.8, tenemos $F_i \approx \phi^i/\sqrt{5}$ donde $\phi = (1 + \sqrt{5})/2 \approx 1{,}618$. Como consecuencia, un árbol AVL de altura H tiene, aproximadamente, $\phi^{H+3}/\sqrt{5}$ nodos. Así, su profundidad es, a lo sumo, logarítmica. Más exactamente, la altura de un árbol AVL satisface

$$H < 1{,}44 \log (N + 2) - 1{,}328 \qquad \textbf{(18.1)}$$

luego la altura en el caso peor es a lo sumo un 44 por ciento mayor que la mínima posible en los árboles binarios.

La profundidad media de un nodo en un árbol AVL construido aleatoriamente, tiende a acercarse a log N. El valor exacto aún no ha sido establecido analíticamente. Todavía no se sabe si es de la forma $\log N + C$ o de la forma $(1 + \varepsilon) \log N + C$, para algún ε, que podría ser aproximadamente 0,01. Las simulaciones no han sido capaces de demostrar de modo convincente que una forma sea más plausible que la otra.

Una consecuencia de estas argumentaciones es que todas las operaciones de búsqueda en un árbol AVL tienen cota logarítmica en el caso peor. La dificultad estriba en que las operaciones que modifican el árbol, como `insertar` y `eliminar`, no son tan simples como antes. Esto es debido a que una inserción (o una eliminación) puede destruir el equilibrio de varios nodos del árbol, tal y como se muestra en la Figura 18.21. Se debe recuperar el equilibrio del árbol antes de considerar finalizada la operación. El algoritmo de inserción se describe aquí, dejándose como ejercicio en el Ejercicio 18.10 el algoritmo de eliminación.

Una observación clave es que tras una inserción, sólo los nodos que se encuentran en el camino desde el punto de inserción hasta la raíz pueden ver alterado su equilibrio, ya que sólo se modifican subárboles de dichos nodos. Esto mismo

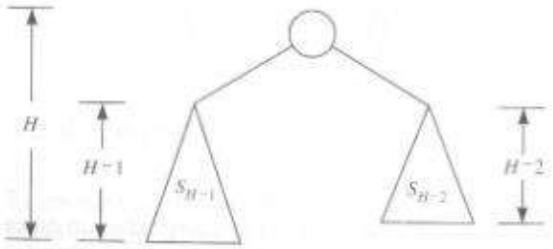


Figura 18.22 Árbol mínimo e altura H.

sucede en muchos de los algoritmos de los árboles de búsqueda equilibrados. Al ir recorriendo el camino hacia la raíz y actualizando la información del equilibrio, podemos encontrar un nodo cuyo nuevo equilibrio incumpla la condición AVL. Esta sección muestra cómo recuperar el equilibrio del árbol desde el primer nodo (es decir, el más profundo) que incumple la condición AVL, demostrando que el mecanismo propuesto garantiza que el árbol completo obtenido satisface la propiedad AVL.

Supongamos que el nodo cuyo equilibrio debemos ajustar es X. Como cualquier nodo tiene a lo sumo dos hijos, y la diferencia de las profundidades de los subárboles de X es 2, el incumplimiento de la propiedad puede darse en uno de los cuatro casos siguientes:

1. Una inserción en el subárbol izquierdo del hijo izquierdo de X.
2. Una inserción en el subárbol derecho del hijo izquierdo de X.
3. Una inserción en el subárbol izquierdo del hijo derecho de X.
4. Una inserción en el subárbol derecho del hijo derecho de X.

Los casos 1 y 4 son simétricos respecto de X, al igual que los casos 2 y 3. Como consecuencia, sólo existen dos casos básicos, aunque desde la perspectiva de la programación, debamos seguir distinguiendo cuatro casos (y bastantes subcasos especiales).

El primer caso, en el que la inserción se produce en los «márgenes» (es decir, a la izquierda de la izquierda o a la derecha de la derecha), se recupera con una *rotación simple* del árbol. Una rotación simple intercambia los papeles de los padres y de los hijos, manteniendo la ordenación del árbol. El segundo caso, en el que la inserción se produce «dentro» (es decir, a la izquierda de la derecha o a la derecha de la izquierda), se resuelve de forma más compleja mediante una *rotación doble*. Éstas son las operaciones básicas sobre árboles, empleadas en numerosas ocasiones en los algoritmos de equilibrado de árboles. El resto de esta sección describe estas rotaciones y demuestra que ambas son suficientes para mantener la condición de equilibrio.

18.4.2 Rotación simple

La Figura 18.23 muestra la rotación simple que resuelve el caso 1. El gráfico que muestra el árbol antes del ajuste es el de la izquierda; a la derecha se muestra el árbol después de dicho ajuste. Veamos lo que vamos a hacer. El nodo k_2 incumple la propiedad de equilibrio AVL ya que su subárbol izquierdo es dos niveles más profundo que su subárbol derecho (en esta sección empleamos las líneas punteadas para marcar los niveles). La situación descrita es la única posible dentro del caso 1 que permite al nodo k_2 satisfacer la propiedad AVL antes de la inserción e incumplirla después. El subárbol A ha crecido en un nivel, provocando que sea, exactamente, dos niveles más profundo que C. B no puede estar al mismo nivel que el nuevo A, ya que si no k_2 no estaría bien equilibrado antes de la inserción. B tampoco puede estar al mismo nivel que C, ya que, de estarlo, k_1 sería el primer nodo en el camino que incumpliría la condición de equilibrio AVL (y estamos suponiendo que lo es k_2).

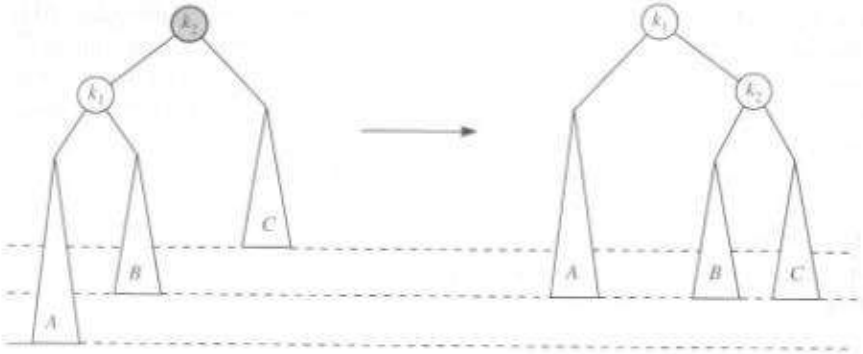


Figura 18.23 Rotación simple para tratar el caso 1.

Para reequilibrar de forma correcta el árbol, queremos subir A un nivel y bajar C otro tanto. Nótese que de hecho esto es algo más de lo que requiere la propiedad AVL. Para hacerlo, debemos reorganizar los nodos en un árbol de búsqueda equivalente, tal y como se muestra en la Figura 18.23. Ello queda ilustrado por el siguiente razonamiento abstracto: consideramos el árbol como una estructura flexible, ponemos un peso en el nodo k_2, cerramos los ojos, y dejamos que actúe la gravedad reequilibrando la balanza. El resultado es que k_1 será la nueva raíz.

La propiedad de los árboles binarios de búsqueda indica que en el árbol inicial, $k_2 > k_1$, así que en el nuevo árbol k_2 se convierte en el hijo derecho de k_1. A y C continúan como hijo izquierdo de k_1 e hijo derecho de k_2, respectivamente. El subárbol B, que contiene los elementos entre k_1 y k_2, puede colocarse en el nuevo árbol como hijo izquierdo de k_2, satisfaciéndose así todos los requerimientos de la ordenación.

Para hacer todo esto sólo se precisan los cambios de las referencias de los hijos mostrados en la Figura 18.24, que producen un nuevo árbol binario que ahora es un AVL. Esto es debido a que A sube un nivel, B permanece en el mismo y C desciende un nivel. k_1 y k_2 no sólo satisfacen la propiedad AVL, sino que además sus subárboles son de la misma altura. Más aún, la nueva altura del subárbol completo es *exactamente la misma* que la altura del subárbol previo a la inserción, que ha provocado el crecimiento de A. Así, no es necesario realizar ninguna actualización más de las alturas de los nodos que se encuentran en el camino hacia la raíz, y como consecuencia, *no se necesitan más rotaciones*. Esta rotación simple se utiliza mucho en otros algoritmos de equilibrado presentados en este capítulo. Debido a ello, la hacemos formar parte del paquete `Rotaciones`.

```
1  /**
2   * Rotación de un nodo de un árbol binario con hijo izquierdo.
3   * En los árboles AVL, es la rotación simple para el caso 1.
4   */
5  static NodoBinario conHijoIzquierdo( NodoBinario k2 )
6  {
7      NodoBinario k1 = k2.izquierdo;
8      k2.izquierdo = k1.derecho;
9      k1.derecho = k2;
10     return k1;
11 }
```

Figura 18.24 Código para la rotación simple (caso 1).

La Figura 18.25 muestra que después de la inserción de 1 en el árbol AVL, el nodo 8 está desequilibrado. Claramente estamos en el primer caso, ya que 1 está en el subárbol izquierdo del hijo izquierdo de 8. En consecuencia, realizamos una rotación simple entre 8 y 4, obteniéndose el árbol de la derecha. Como ya hemos mencionado anteriormente en esta sección, el caso 4 es simétrico a éste. La rotación adecuada se ilustra en la Figura 18.26, y el código que la implementa se muestra en la Figura 18.27. Este método también forma parte del paquete `Rotaciones`.

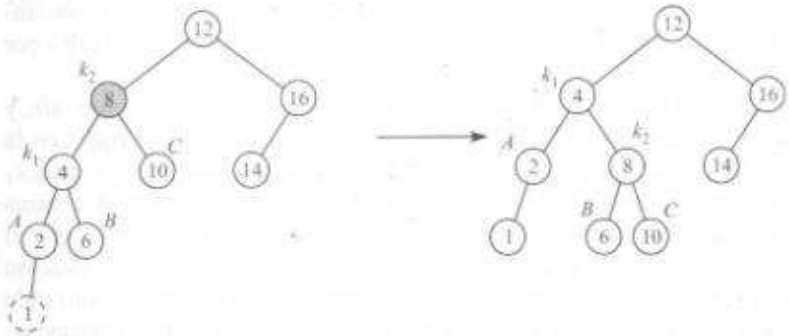


Figura 18.25 La rotación simple arregla el árbol AVL tras la inserción de 1.

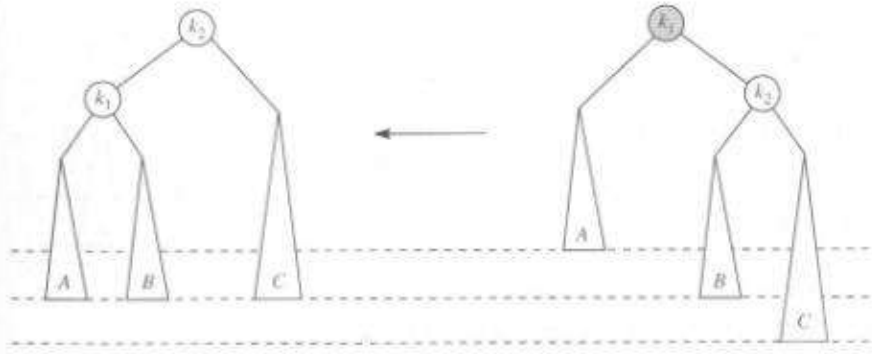


Figura 18.26 Rotación simple simétrica para resolver el caso 4.

```
1  /**
2   * Rotación de un nodo de un árbol binario con hijo derecho.
3   * En los árboles AVL, es la rotación simple para el caso 4.
4   */
5  static NodoBinario conHijoDerecho( NodoBinario k1 )
6  {
7      NodoBinario k2 = k1.derecho;
8      k1.derecho = k2.izquierdo;
9      k2.izquierdo = k1;
10     return k2;
11 }
```

Figura 18.27 Código para la rotación simple (caso 4).

18.4.3 Rotación doble

La rotación simple tiene un problema. Tal y como muestra la Figura 18.28, no funciona en el caso 2 (ni, por simetría, en el caso 3). El problema estriba en que el subárbol Q es demasiado profundo; una rotación simple no reduce lo suficiente su profundidad. La *rotación doble* que resuelve el problema se muestra en la Figura 18.29.

El hecho de que en la Figura 18.28 se inserte un elemento en el subárbol Q garantiza que no está vacío. Asumiremos que tiene una raíz y dos subárboles (posiblemente vacíos), así que podemos ver el árbol como cuatro subárboles conectados por tres nodos. Renombramos dichos subárboles como A, B, C y D. Como sugiere la Figura 18.29, exactamente uno de los árboles B o C es dos niveles más profundo que D, aunque no podemos asegurar cuál de ellos es. Esto carece de importancia por lo que, en la figura, tanto B como C se han dibujado 1,5 niveles por debajo de D.

Para recuperar el equilibrio, observamos que es imposible dejar k_3 como raíz, y además que la rotación entre k_3 y k_1 no funciona, tal y como se comprueba en la Figura 18.28. La única alternativa es colocar a k_2 como nueva raíz. Esto fuerza a k_1 a ser el hijo izquierdo de k_2 y a k_3 a ser el hijo derecho de k_2. Todo ello determina completamente las posiciones de los cuatro subárboles. Es fácil comprobar que el árbol resultante cumple la propiedad AVL. Además, como en el caso de la rotación simple, esto restablece la altura que tenía el árbol antes de la inserción, garantizando que la recuperación del equilibrio y la modificación de alturas han sido completadas.

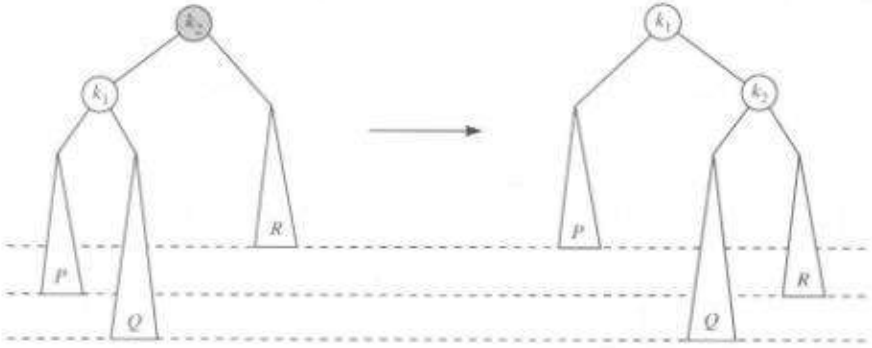


Figura 18.28 La rotación simple no arregla el caso 2.

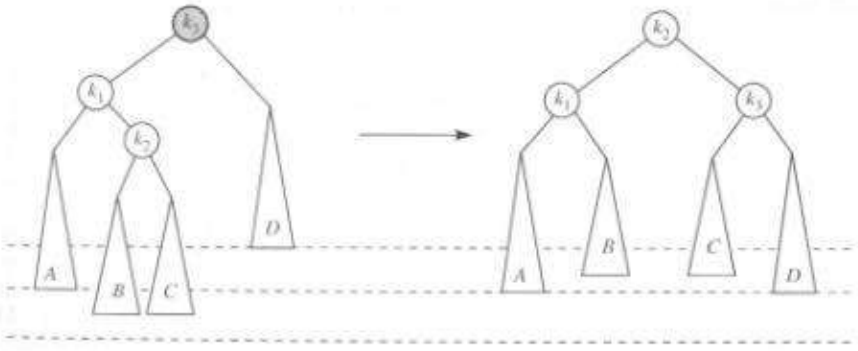


Figura 18.29 Rotación doble izquierda-derecha para arreglar el caso 2.

Como ejemplo, la Figura 18.30 muestra el resultado de insertar 5 en un árbol AVL. El desequilibrio de las alturas se produce en el nodo 8, por lo que estamos en el caso 2. Realizamos una rotación doble en ese nodo, generando el árbol de la derecha.

La Figura 18.31 muestra el caso simétrico 3, que también puede ser arreglado con una rotación doble. Por último, nótese que aunque la rotación doble parezca compleja, es equivalente a hacer lo siguiente:

- Una rotación simple entre el hijo de X y su nieto, seguida de
- una rotación simple entre X y su nuevo hijo.

Así es posible obtener un código muy compacto para implementar la rotación doble del caso 2, que se muestra en la Figura 18.32. El código para el caso simétrico 3 se encuentra en la Figura 18.33.

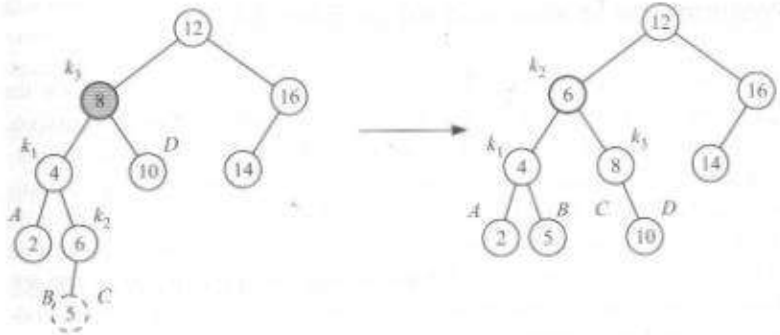


Figura 18.30 La rotación doble arregla el árbol AVL tras la inserción de 5.

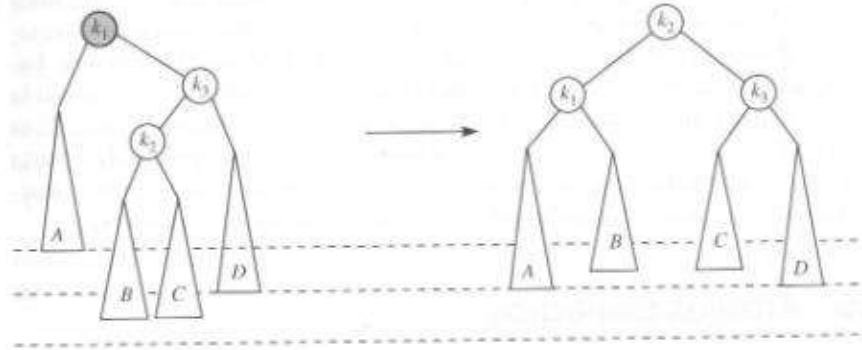


Figura 18.31 Rotación doble izquierda-derecha para arreglar el caso 3.

```
1  /**
2   * Rotación doble de un nodo de un árbol binario: primero
      el hijo izquierdo con su hijo derecho;
3   * después el nodo k3 con su nuevo hijo izquierdo.
4   * En los árboles AVL, es la rotación doble para resolver el caso 2.
5   */
6  static NodoBinario dobleConHijoIzquierdo( NodoBinario k3 )
7  {
8      k3.izquierdo = conHijoDerecho( k3.izquierdo );
9      return conHijoIzquierdo( k3 );
10 }
```

Figura 18.32 Código para la rotación doble (caso 2).

```
/**
 * Rotación doble de un nodo de un árbol binario: primero
 * el hijo derecho con su hijo izquierdo; después el nodo k1
 * con su nuevo hijo derecho.
 * En los árboles AVL, es la rotación doble para resolver el caso 3.
 */
static NodoBinario dobleConHijoDerecho( NodoBinario k1 )
{
    k1.derecho = conHijoIzquierdo( k1.derecho );
    return conHijoDerecho( k1 );
}
```

Figura 18.33 Código para la rotación doble (caso 3).

18.4.4 Resumen de la inserción en un árbol AVL

Concluiremos con un breve resumen de cómo se implementa la inserción en un árbol AVL. La forma más sencilla de hacerlo es emplear un algoritmo recursivo. Para insertar un nuevo nodo con clave X en un árbol AVL T, se le inserta recursivamente en el subárbol adecuado de T (llamado T_{LR}). Si la altura de T_{LR} no cambia, ya hemos acabado. En caso contrario, si se produce en T un desequilibrio de las alturas, realizamos la correspondiente rotación simple o doble, dependiendo de X y de las claves de T y T_{LR}, tras lo que habremos terminado (ya que la altura original es la misma que la que se obtiene tras la rotación). Esta descripción recursiva se ha explicado cuidadosamente, por resultar más fácil de entender y contener todos los elementos esenciales de cualquier implementación alternativa. Sin embargo, resulta mejorable. Por ejemplo, observamos que en cada nodo comparamos las alturas de sus dos subárboles. En general, es más eficiente almacenar directamente en el nodo el resultado de la comparación en lugar de guardar su altura. Esto evita cálculos repetitivos acerca del equilibrio. Por otra parte, en esta ocasión la recursión implica una sobrecarga substancial respecto a la versión iterativa. Esto es debido a que, hemos de recorrer el árbol hacia abajo para después volver hasta la raíz, en lugar de detenernos tan pronto como la rotación se realice. Como consecuencia, en la práctica, se emplean otros esquemas de equilibrio de árboles.

18.5 Árboles rojinegros

Una alternativa históricamente popular a los árboles AVL son los *árboles rojinegros*. Al igual que en los árboles AVL, las operaciones sobre los árboles rojinegros son logarítmicas en el caso peor. La principal ventaja de los árboles rojinegros es que las rutinas de inserción y de eliminación pueden efectuarse con un único recorrido descendente. Esto contrasta con los árboles AVL, en los que es necesario un recorrido descendente para establecer el lugar de la inserción y otro recorrido ascendente para actualizar las alturas de los nodos y, posiblemente, ajustar su equilibrio. Como resultado, una cuidadosa implementación no recursiva de los árboles rojinegros es más simple y eficiente que una implementación de los árboles AVL.

Un árbol rojinegro es un árbol binario de búsqueda que verifica las siguientes propiedades de orden:

1. Cada nodo está coloreado con los colores rojo o negro.
2. La raíz es negra.
3. Si un nodo es rojo, sus hijos deben ser negros.
4. Todos los caminos desde un nodo a una referencia `null` deben contener el mismo número de nodos negros.

En nuestras representaciones, en blanco y negro, de los árboles rojinegros, los nodos rojos se mostrarán sombreados. En la Figura 18.34 se presenta un árbol rojinegro. Todo camino desde la raíz a una referencia `null` contiene tres nodos negros.

Se puede demostrar por inducción que si cualquier camino desde la raíz a una referencia `null` contiene B nodos negros, entonces el árbol debe tener, al menos, $2^B - 1$ nodos negros. Más aún, como la raíz es negra y no pueden existir dos nodos consecutivos de color rojo en un camino, la altura de un árbol rojinegro es a lo sumo $2\log(N + 1)$. Como consecuencia, está garantizado que la búsqueda es una operación logarítmica.

La dificultad, como siempre, es que algunas operaciones pueden modificar el árbol y destruir así sus propiedades de coloreado. Esto dificulta la inserción y convierte a la eliminación en una operación especialmente complicada. En la próxima sección, se implementa la inserción y se estudia el algoritmo de eliminación.

18.5.1 Inserción ascendente

Recordemos que un nuevo elemento se inserta siempre en el árbol como una hoja. Si coloreamos el nuevo elemento de negro, estamos seguros de incumplir la cuarta propiedad, ya que así generamos un camino con más nodos negros. Como consecuencia, el nuevo nodo debe ser, en principio, rojo. De esta forma, si el padre es negro ya hemos acabado; así, la inserción de 25 en el árbol de la Figura 18.34 es trivial. Pero si el padre es rojo, estaríamos incumpliendo la tercera propiedad, al haber dos nodos rojos consecutivos. En este caso debemos ajustar el árbol para hacer que se cumpla la propiedad 3 sin que deje de cumplirse la cuarta propiedad. Las operaciones básicas que emplearemos son los cambios de color y las rotaciones.

Existen varios casos (cada uno con su correspondiente simétrico) que debemos considerar cuando el padre es rojo. En primer lugar, supongamos que el hermano del padre es negro (adoptamos el acuerdo de que las referencias `null` son negras).

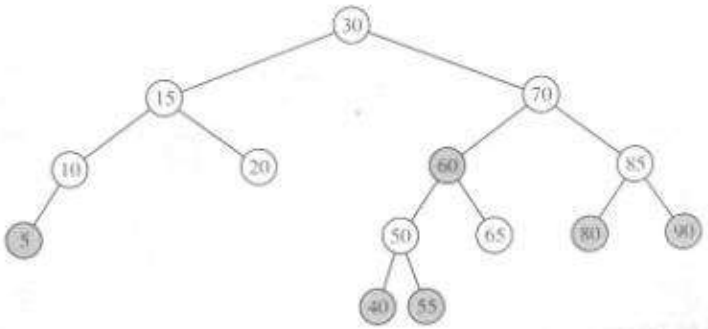


Figura 18.34 Ejemplo de árbol rojinegro: la secuencia de inserciones es 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5 y 55. Los nodos sombreados son rojos.

Esto se aplicaría en la inserción de 3 u 8, pero no en la inserción de 99. Sea X la nueva hoja añadida, P su padre, S el hermano del padre (si existe) y G el abuelo. En este caso sólo X y P son rojos. G es negro ya que en caso contrario *antes* de la inserción existirían dos nodos rojos consecutivos —un incumplimiento de la propiedad 3—. Adoptando la terminología de los árboles AVL, decimos que respecto a G, X es un nodo exterior o interior. Si X es un nieto exterior, la propiedad 3 se recupera mediante una rotación simple entre su padre y su abuelo, con algunos cambios de color. Si X es un nieto interior, es suficiente una rotación doble y algunos cambios de color. La rotación simple se muestra en la Figura 18.35 y la doble en la Figura 18.36. Aunque X es una hoja, hemos ilustrado un caso más general, que permite que X esté en medio del árbol. Esta rotación más general será empleada más tarde, en este mismo capítulo.

Antes de continuar, observemos por qué estas rotaciones son correctas. Necesitamos asegurar que no existen dos nodos rojos consecutivos. Por ejemplo, en la Figura 18.36 es fácil ver que las únicas posibles instancias de nodos rojos consecutivos se encontrarían entre P y uno de sus hijos, o entre G y C. Sin embargo, las raíces de A, B y C deben ser negras; de lo contrario en el árbol de partida se incumpliría la propiedad 3. En el árbol inicial, existe un único nodo negro en el camino desde la raíz del árbol hasta A, B y C, y dos nodos negros en el camino desde la raíz hasta D y E. Es fácil comprobar que éste es todavía el caso después de la rotación y la recoloración.

Pero, ¿qué sucede si S es rojo y se intenta insertar el elemento 79 en el árbol de la Figura 18.34? Entonces no funcionan ni la rotación simple ni la rotación doble, ya que ambas incluyen en sus resultados nodos rojos consecutivos. De hecho, es

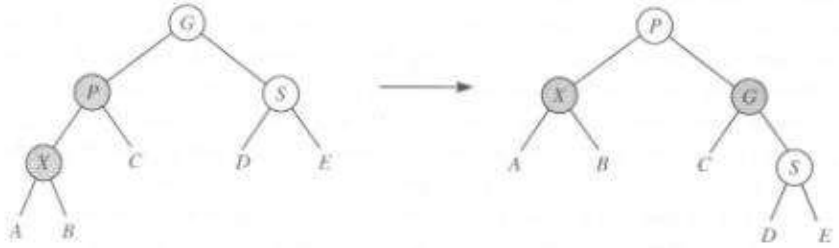


Figura 18.35 Si S es negro y X es un nieto externo, la propiedad 3 se recupera con una rotación simple entre su padre y su abuelo, con los cambios adecuados de color.

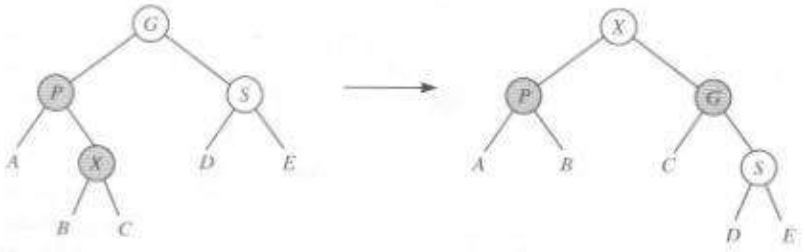


Figura 18.36 Si S es negro y X es un nieto interior, una rotación doble en la que intervengan X, su padre y su abuelo, junto a los cambios de color apropiados, solucionará el incumplimiento de la tercera propiedad.

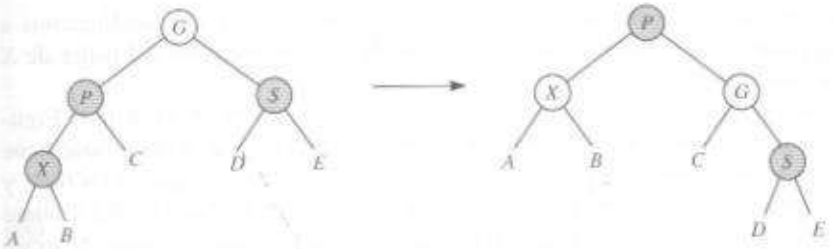


Figura 18.37 Si S es rojo, una rotación simple entre el padre y el abuelo, junto a los cambios de color apropiados, restablece la tercera propiedad entre X y P.

fácil comprobar que en este caso debe haber tres nodos en los caminos hasta D y E, y sólo uno de ellos puede ser negro. Esto nos indica que tanto S como la nueva raíz del subárbol deben ser rojas. Por ejemplo en la Figura 18.37, se muestra el caso de la rotación simple que se produce cuando X es un nieto exterior. Aunque, en este caso parece funcionar, aparece un problema: ¿qué sucede si el padre de la raíz del subárbol (es decir, el bisabuelo inicial de X) también es rojo? Podemos filtrar este mecanismo hacia atrás, hasta que lleguemos a la raíz (que se coloreará en negro) o ya no se tengan nodos rojos consecutivos. Pero entonces deberíamos realizar un recorrido ascendente hacia la raíz, igual que en los árboles AVL.

18.5.2 Árboles rojinegros descendentes

Para evitar la eventual necesidad de tener que hacer un doble recorrido del árbol, aplicamos un procedimiento descendente mientras se busca el punto de inserción. Más concretamente, se garantiza que cuando llegamos a una hoja e insertamos un nodo, S no será rojo. Entonces basta con añadir una hoja roja y, si es necesario, emplear una sola rotación (simple o doble). El proceso es conceptualmente sencillo.

En el recorrido hacia abajo, cuando se encuentra un nodo X con dos hijos rojos, convertimos X en rojo y sus dos hijos en negro. La Figura 18.38 muestra este cambio de color. Es sencillo comprobar que el número de nodos negros en los caminos por debajo de X permanece invariante. Sin embargo, si el padre de X es rojo, introducimos dos nodos rojos consecutivos. Pero en este caso, podemos realizar la rotación simple de la Figura 18.35 o la rotación doble de la Figura 18.36. Ahora bien, ¿qué sucedería si el hermano del padre de X fuese también rojo? En realidad, *esto no puede suceder*. Si en el camino hacia las hojas encontramos un nodo Y con dos hijos rojos, sabemos que los nietos de Y deben ser negros. Y como los hijos de Y se colorean en negro, incluso después de la rotación que puede pro-

Figura 18.38 Cambio de color: sólo si el padre de X es rojo se continúa el proceso con una rotación

ducirse, no encontraremos otro nodo rojo en dos niveles. Así, cuando llegamos a X, si el padre de X es rojo entonces no es posible que el hermano del padre de X sea rojo también.

Como ejemplo, supongamos que deseamos insertar 45 en el árbol de la Figura 18.34. En el camino hacia las hojas del árbol, encontramos el nodo 50 que tiene dos hijos rojos. Entonces hacemos un cambio de color, coloreando 50 en rojo, y 40 y 55 en negro. El resultado se muestra en la Figura 18.39. Sin embargo, ahora 50 y 60 son rojos. En consecuencia, hacemos una rotación simple (ya que 50 es un nodo exterior) entre 60 y 70, convirtiendo 60 en la raíz negra del subárbol derecho de 30 y coloreando 70 en rojo. Todo esto se muestra en la Figura 18.40. Entonces continuamos el proceso, realizando los mismos cambios si encontramos en el camino otros nodos con dos hijos rojos. En nuestro ejemplo no hay ninguno más.

Cuando alcanzamos una hoja, insertamos 45 como un nodo rojo, y como el padre es negro, ya hemos terminado. En la Figura 18.41 se muestra el árbol que se obtiene. Si el padre hubiese sido rojo, habríamos necesitado hacer una rotación.

Tal y como muestra la Figura 18.41, el árbol rojinegro que se obtiene está, por lo general, muy bien equilibrado. Las distintas pruebas sugieren que el número de nodos consultados en una búsqueda de un árbol rojinegro medio es casi idéntico al número de nodos consultados en los árboles AVL, aunque estáticamente las propiedades de equilibrio de los árboles rojinegros son ligeramente más débiles. La

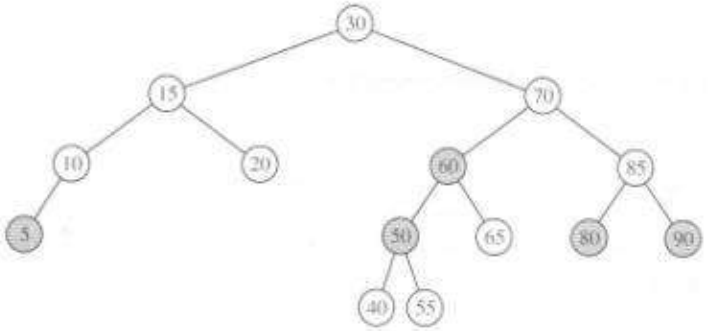


Figura 18.39 El cambio de color de 50 supone una incorrección; debido a que ésta se produce en un nodo exterior, para arreglarla basta una rotación simple.

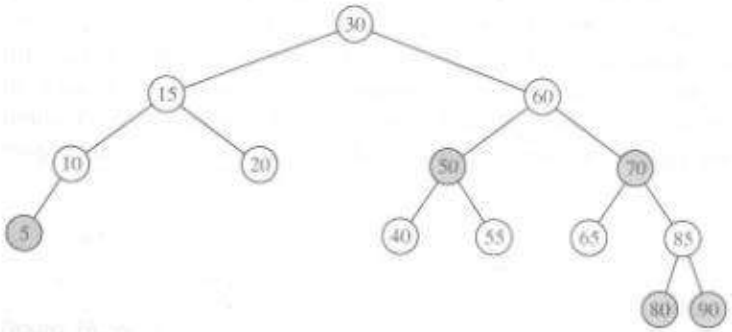


Figura 18.40 Resultado de la rotación simple que arregla el fallo producido en el nodo 50.

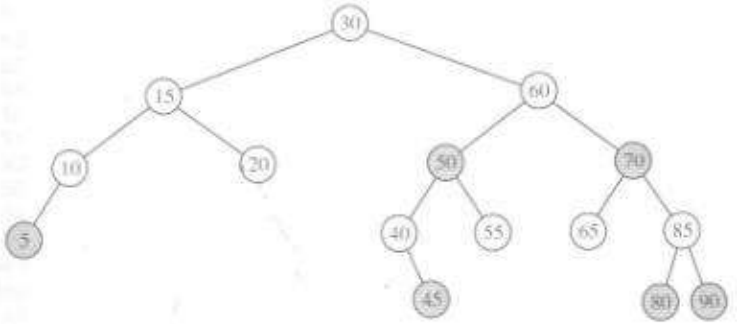


Figura 18.41 Inserción de 45 como nodo rojo.

ventaja de los árboles rojinegros es la sobrecarga relativamente pequeña que se necesita para realizar una inserción y el hecho de que, en la práctica, las rotaciones son bastante infrecuentes.

18.5.3 Implementación en Java

La implementación concreta es compleja no sólo por la gran cantidad de rotaciones posibles. También es complicada por la posibilidad de que algunos subárboles (como el subárbol derecho de 10) pueden estar vacíos y por el caso especial de la raíz (la cual, entre otras cosas, no tiene padre). Para eliminar los casos especiales, empleamos dos testigos o centinelas:

- Se emplea `nodoNulo` en lugar de la referencia `null`, `nodoNulo` es siempre de color negro.
- Se emplea `cabecera` como pseudo-raíz. Tendrá un valor clave $-\infty$ y un hijo derecho que apuntará a la verdadera raíz.

A causa de estos elementos, incluso las operaciones básicas necesitan ser modificadas. Como consecuencia, no tiene sentido emplear la herencia de la clase `ArbolBinarioBusqueda`. De modo que, esta clase está escrita partiendo de cero. Su sección pública se muestra en la Figura 18.42 y la sección privada en la Figura 18.43. Recordamos que, por defecto, el constructor de `NodoBinario` inicializa el atributo `color` a 1. Por tanto, `NEGRO` debe ser 1. Las líneas 59 y 60 declaran los centinelas que han sido explicados anteriormente. En la rutina `insertar` se emplean cuatro referencias: `actual`, `padre`, `abuelo` y `bisabuelo`. Su declaración como atributos estáticos (en las líneas 66 a 69) indica que son esencialmente variables globales. Esto es debido a que, como mostraremos más tarde, es conveniente tenerlos compartidos por las rutinas `insertar` y `reorientacion`. El método `eliminar` está sin implementar.

El resto de rutinas son similares a los métodos correspondientes de `ArbolBinarioBusqueda`, excepto en que se tratan adecuadamente los nodos centinelas. En concreto, el constructor debe codificarse con el valor $-\infty$, y `vaciar` y `esVacio`, mostrados en las líneas 45 a 48, necesitan emplear `nodoNulo` (en lugar de `null`) y `cabecera.derecho` (en lugar de la típica `raiz`).

```
package EstructurasDatos;

import Soporte.*; import Soporte.Comparable;
import Excepciones.*;

// Clase ArbolRojoNegro
//
// CONSTRUCCIÓN: con un centinela con valor menos infinito
//
// ***********************OPERACIONES PÚBLICAS**********************
// void insertar( x )       --> Inserta x
// void eliminar( x )       --> Elimina x
// void eliminarMin( )      --> Elimina el elemento más pequeño
// Comparable buscar( x )   --> Devuelve el elemento que ajusta con x
// Comparable buscarMin( )  --> Devuelve el elemento más pequeño
// Comparable buscarMax( )  --> Devuelve el elemento más grande
// boolean esVacio( )       --> Devuelve true si vacío; si no, false
// void vaciar( )           --> Elimina todos los elementos
// void imprimirArbol( )    --> Imprime el árbol de forma ordenada
// ***************************ERRORES*****************************
// Muchas rutinas lanzan ElementoNoEncontrado en condiciones degeneradas
// insertar lanza ElementoDuplicado si el elemento ya está en el árbol

/**
 * Implementa los árboles rojinegros.
 * Nótese que los "encajes" se basan en el método compara.
 */
public class ArbolRojoNegro implements ArbolBusqueda
{
    public ArbolRojoNegro( Comparable negInf )
      { /* Figura 18.44 */ }

    public void insertar( Comparable x ) throws ElementoDuplicado
      { /* Figura 18.48 */ }
    public void eliminar( Comparable x ) throws ElementoNoEncontrado
      { /* No se implementa */ }
    public void eliminarMin( ) throws ElementoNoEncontrado
      { /* No se implementa */ }
    public Comparable buscarMin( ) throws ElementoNoEncontrado
      { /* Figura 18.46 */ }
    public Comparable buscarMax( ) throws ElementoNoEncontrado
      { /* Similar a buscarMin */ }
    public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
      { /* Figura 18.47 */ }
    public boolean esVacio( )
      { return cabecera.derecho == nodoNulo; }
    public void vaciar( )
      { cabecera.derecho = nodoNulo; }
    public void imprimirArbol( )
      { imprimirArbol( cabecera.derecho ); }
```

Figura 18.42 Esqueleto de la clase `ArbolRojoNegro` (parte 1: sección pública).

El constructor de `ArbolRojoNegro` se muestra en la Figura 18.44. Únicamente crea los dos centinelas e inicializa las referencias de sus hijos `izquierdo` y `derecho` apuntando a `nodoNulo`. Es preciso un inicializador estático para crear e inicializar `nodoNulo`.

```
51      private void imprimirArbol( NodoBinario t )
52        { /* Figura 18.45 */ }
53      private void reorientacion( Comparable item )
54        { /* Figura 18.49 */ }
55      private NodoBinario rotar( Comparable item,
56                                 NodoBinario padre )
57        { /* Figura 18.50 */ }
58
59      private NodoBinario cabecera;
60      private static NodoBinario nodoNulo;
61
62      private static final int NEGRO = 1;     // NEGRO debe ser 1
63      private static final int ROJO  = 0;
64
65          // Empleados en la rutina de inserción y las auxiliares
66      private static NodoBinario actual;
67      private static NodoBinario padre;
68      private static NodoBinario abuelo;
69      private static NodoBinario bisabuelo;
70  }
```

Figura 18.43 Esqueleto de la clase `ArbolRojoNegro` (parte 2: sección privada).

```
1   /**
2    * Construye el árbol.
3    * @param negInf un valor menor o igual que todos los demás.
4    */
5   public ArbolRojoNegro( Comparable negInf )
6   {
7       cabecera           = new NodoBinario( negInf );
8       cabecera.izquierdo = cabecera.derecho = nodoNulo;
9   }
10
11  static                 // Inicializador estático de nodoNulo
12  {
13      nodoNulo           = new NodoBinario( null );
14      nodoNulo.izquierdo = nodoNulo.derecho = nodoNulo;
15  }
```

Figura 18.44 Constructor de `ArbolRojoNegro` e inicializador estático.

La Figura 18.45 muestra el sencillo cambio que supone el uso de los centinelas. El test sobre `null` es reemplazado por el test sobre `nodoNulo`. El método `buscarMin` se muestra en la Figura 18.46.

```
1   /**
2    * Método interno para imprimir un subárbol en orden.
3    * @param t la raíz del subárbol.
4    */
5   private void imprimirArbol( NodoBinario t )
6   {
7       if( t != nodoNulo )
8       {
9           imprimirArbol( t.izquierdo );
10          System.out.println( t.dato.toString( ) );
11          imprimirArbol( t.derecho );
12      }
13  }
```

Figura 18.45 `imprimirArbol` de la clase `ArbolRojoNegro`.

```
1  /**
2   * Busca el elemento más pequeño del árbol.
3   * @return el menor elemento.
4   * @exception ElementoNoEncontrado si el árbol es vacío.
5   */
6  public Comparable buscarMin( ) throws ElementoNoEncontrado
7  {
8    if( esVacio( ) )
9      throw new ElementoNoEncontrado( "buscarMin de ArbolRojoNegro" );
10
11   NodoBinario itr = cabecera.derecho;
12   while( itr.izquierdo != nodoNulo )
13     itr = itr.izquierdo;
14
15   return itr.dato;
16 }
```

Figura 18.46 Método `buscarMin` de `ArbolRojoNegro`; observe el uso de `cabecera` y `nodoNulo`.

La rutina `buscar`, mostrada en la Figura 18.47, emplea otro truco muy común. Antes de iniciar la búsqueda, colocamos `x` en el centinela `nodoNulo`. Hemos garantizado así el ajuste con `x`, incluso si `x` no es encontrado. Si el ajuste se produce en `nodoNulo`, concluimos que el elemento no ha sido encontrado. Emplearemos este truco en el método `insertar`.

```
1  /**
2  * Buscar un elemento en el árbol.
3  * @param x el elemento a buscar.
4  * @return el elemento buscado.
5  * @exception ElementoNoEncontrado si en el árbol no
6  *            no hay ningún elemento que ajuste con x.
7  */
8  public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
9  {
10   nodoNulo.dato = x;
11   actual = cabecera.derecho;
12
13   for( ; ; )
14   {
15     if( x.menorQue( actual.dato ) )
16       actual = actual.izquierdo;
17     else if( actual.dato.menorQue( x ) )
18       actual = actual.derecho;
19     else if( actual != nodoNulo )
20       return actual.dato;
21     else
22       throw new ElementoNoEncontrado( "buscar de ArbolRojoNegro" );
23     }
24 }
```

Figura 18.47 Método `buscar` de `ArbolRojoNegro`; obsérvese el uso de `cabecera` y `nodoNulo`.

El método `insertar`, se muestra en la Figura 18.48, se obtiene directamente de nuestra descripción. El bucle `while`, codificado en las líneas 12 a 21, desciende por el árbol y «repara» los nodos con dos hijos rojos llamando al método `reorientacion`, tal y como se muestra en la Figura 18.49. Para hacerlo, no sólo mantenemos información acerca del nodo actual, sino también de su padre, abuelo y bisabuelo. Nótese que después de la rotación, los valores almacenados en el abuelo y el bisabuelo no son correctos. Sin embargo, nos hemos asegurado que serán correctamente colocados la próxima vez que se necesiten.

Cuando el bucle termina, `x` es encontrado (indicado por `actual != nodoNulo`) o no. Si lo encontramos, se lanza una excepción en la línea 25. En caso contrario, `x` no está en el árbol y es preciso hacerle hijo de `padre`. Creamos un nuevo nodo (como el nuevo nodo `actual`), lo unimos al padre e invocamos a `reorientacion`, todo ello en las líneas 26 a 33.

El código empleado para realizar una rotación simple se muestra en el método `rotar` de la Figura 18.50. Debido a que el árbol obtenido debe unirse al padre, `rotar` tiene como parámetro el nodo padre. En lugar de guardar la información de qué tipo de rotación realizamos (esto es, si es hacia la derecha o hacia la izquier-

```
1  /**
2   * Inserta en el árbol.
3   * @param x el elemento a insertar.
4   * @exception ElementoDuplicado si un elemento que
5   *     ajusta con x ya está en el árbol.
6   */
7  public void insertar( Comparable x ) throws ElementoDuplicado
8  {
9      actual = padre = abuelo = cabecera;
10     nodoNulo.dato = x;
11
12     while( actual.dato.compara( x ) != 0 )
13     {
14         bisabuelo = abuelo; abuelo = padre; padre = actual;
15         actual = x.menorQue( actual.dato ) ?
16                  actual.izquierdo : actual.derecho;
17
18             // Comprueba si hay dos hijos ROJOs; resolviéndolo en su
19         if( actual.izquierdo.color == ROJO && actual.derecho.color
20             reorientacion( x );
21     }
22
23         // La inserción falla si el elemento ya está en el árbol
24     if( actual != nodoNulo )
25         throw new ElementoDuplicado( "insertar de ArbolRojoNegro" );
26     actual = new NodoBinario( x, nodoNulo, nodoNulo );
27
28         // Enlace al padre
29     if( x.menorQue( padre.dato ) )
30         padre.izquierdo = actual;
31     else
32         padre.derecho = actual;
33     reorientacion( x );
34 }
```

Figura 18.48 Rutina `insertar` de la clase `ArbolRojoNegro`.

```
1  /**
2   * Rutina invocada durante una inserción si un nodo tiene dos
3   * hijos ROJOs. Realiza cambios de color y rotaciones.
4   * @param item el elemento a insertar.
5   */
6  private void reorientacion( Comparable item )
7  {
8          // Cambio de color
9      actual.color = ROJO;
10     actual.izquierdo.color = NEGRO;
11     actual.derecho.color = NEGRO;
12
13     if( padre.color == ROJO ) // Tiene que rotar
14     {
15         abuelo.color = ROJO;
16         if( item.menorQue( abuelo.dato ) !=
17                            item.menorQue( padre.dato ) )
18             padre = rotar( item, abuelo ); // Comenzar rotación doble
19         actual = rotar( item, bisabuelo );
20         actual.color = NEGRO;
21     }
22     cabecera.derecho.color = NEGRO; // Colorea la raíz en NEGRO
23 }
```

Figura 18.49 Rutina `reorientacion`: invocada cuando un nodo tiene dos hijos rojos o cuando se inserta un nuevo nodo.

```
1  /**
2   * Rutina interna que realiza una rotación simple o doble.
3   * Invocada por reorientacion.
4   * @param item el elemento de reorientacion.
5   * @param padre el padre de la raíz del subárbol que debe rotarse.
6   * @return la raíz del subárbol rotado.
7   */
8  private NodoBinario
9  rotar( Comparable item, NodoBinario padre )
10 {
11     if( item.menorQue( padre.dato ) )
12         return padre.izquierdo = item.menorQue(
13                                      padre.izquierdo.dato ) ?
14             Rotaciones.conHijoIzquierdo( padre.izquierdo ) : // II
15             Rotaciones.conHijoDerecho( padre.izquierdo )   : // ID
16     else
17         return padre.derecho = item.menorQue(
18                                    padre.derecho.dato ) ?
19             Rotaciones.conHijoIzquierdo( padre.derecho )   : // DI
20             Rotaciones.conHijoDerecho( padre.derecho )     : // DD
21 }
```

Figura 18.50 Rutina para realizar la rotación apropiada.

da) mientras descendemos por el árbol, pasamos x como parámetro. Como se espera un número reducido de rotaciones a lo largo de la inserción, hacerlo de este modo es, además de simple, rápido.

`reorientacion` llama a `rotar` cuando es necesario, para efectuar una rotación simple o doble. Como una rotación doble no es más que la combinación de dos rotaciones simples, podemos preguntar si estamos en un caso interior y si es así, realizar una rotación extra entre el nodo actual y su padre (pasando su abuelo a la rutina `rotar`). En cualquier caso, hacemos una rotación entre el padre y el abuelo (pasando el bisabuelo a `rotar`). Esto es codificado, de forma sutil, en las líneas 16 a 19 de la Figura 18.49.

18.5.4 Eliminación descendente

La eliminación en los árboles rojinegros también puede implementarse de modo descendente. No es necesario decir que en este caso una implementación concreta es bastante complicada, ya que el mismo algoritmo `eliminar` de los árboles no equilibrados no es trivial. El algoritmo usual de eliminación en los árboles binarios de búsqueda elimina nodos que son hojas o tienen un único hijo. Recuerde que nunca se borran nodos con dos hijos; sólo se reemplaza su contenido.

Si el nodo que vamos a borrar es rojo no hay ningún problema. Sin embargo, si es negro su eliminación hará que se incumpla la cuarta propiedad. La solución al problema consiste en asegurar que cualquier nodo que eliminemos será rojo.

A lo largo de esta discusión, X será el nodo actual, T su hermano y P su padre. Comenzamos por colorear la raíz centinela en rojo. Según descendemos por el árbol, intentamos asegurar que X sea rojo. Cuando llegamos a un nuevo nodo, estamos seguros de que P es rojo (de forma inductiva, gracias al invariante que tratamos de mantener) y de que X y T son negros (ya que no podemos tener dos nodos rojos consecutivos).

Existen dos casos principales, además de las variantes simétricas usuales (que son omitidas).

En primer lugar supongamos que X tiene dos hijos rojos. Existen tres subcasos, los cuales dependen de los hijos de T:

1. T tiene dos hijos negros: cambio de color (Figura 18.51).
2. T tiene un hijo exterior rojo: realizar una rotación simple (Figura 18.52).
3. T tiene un hijo interior rojo: realizar una rotación doble (Figura 18.53).

Una observación detenida de las rotaciones nos conduce a la conclusión de que si T tiene dos hijos rojos, tanto una rotación simple como una rotación doble funcionarían correctamente, de forma que es más lógico decantarse por la rotación simple. Observe que si X es una hoja entonces sus dos hijos son negros, así que siempre podemos emplear alguno de estos tres mecanismos para conseguir que su color sea rojo.

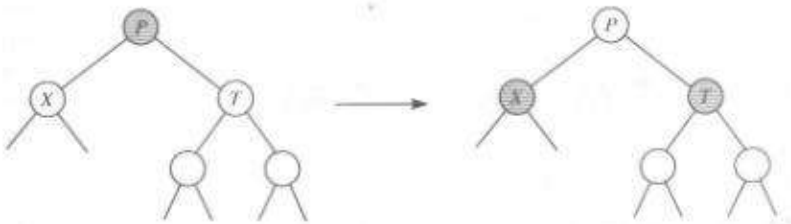


Figura 18.51 X tiene dos hijos negros cuyos hermanos son también negros: debemos hacer un cambio de color.

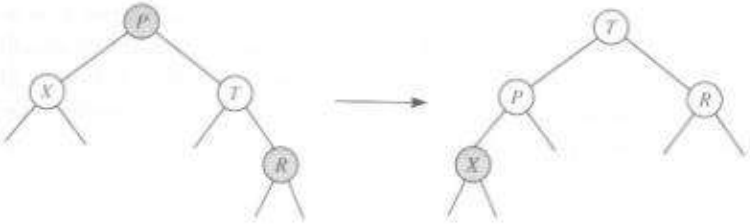


Figura 18.52 *X* tiene dos hijos negros y el hijo exterior de su hermano es rojo; debemos hacer una rotación simple.

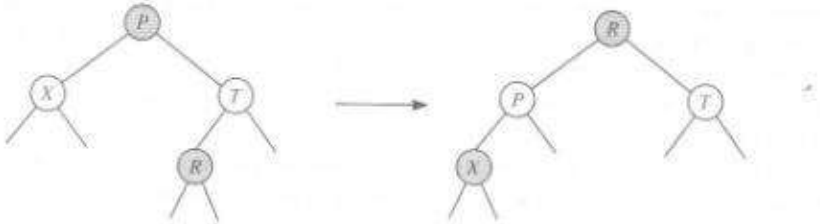


Figura 18.53 *X* tiene dos hijos negros y el hijo interior de su hermano es rojo; debemos hacer una rotación doble.

El segundo caso es que alguno de los hijos de *X* sea rojo. Nótese que debido a que las rotaciones del primer caso siempre colorean *X* de rojo, si *X* tiene un hijo de su mismo color, introduciríamos nodos rojos consecutivos. Por tanto, necesitamos una solución alternativa. En este caso, descendemos al siguiente nivel, obteniendo nuevos *X*, *T* y *P*. Si tenemos suerte, estaremos en un nodo rojo (esto puede suceder el 50 por ciento de las veces), y arreglamos el problema convirtiendo el nodo actual nuevo en rojo. En caso contrario, estamos en la situación mostrada en la Figura 18.54. Esto es, el *X* actual es negro, *T* es rojo y *P* es negro. Podemos realizar una rotación entre *T* y *P*, haciendo que el nuevo padre de *X* sea rojo y su nuevo abuelo sea negro. Ahora *X* ya no es rojo, pero estamos de nuevo en el punto de partida (aunque en un nivel inferior). Esto es suficientemente bueno, ya que prueba que podemos descender de modo iterativo por el árbol. Así, siempre que alcancemos un nodo rojo o un nodo con dos hijos negros hemos ganado. Esto está garantizado en el algoritmo de eliminación, ya que se alcanza una de las dos situaciones siguientes:

- *X* es una hoja. Esta situación ya fue considerada por el caso principal ya que *X* tiene dos hijos negros.

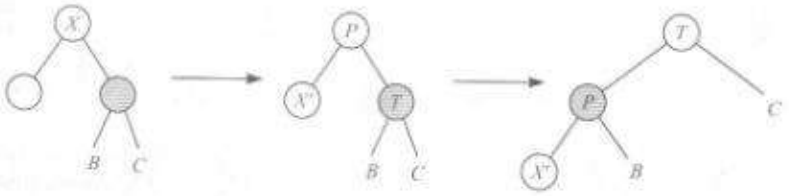


Figura 18.54 *X* es negro y al menos un hijo suyo es rojo; si descendemos al siguiente nivel y caemos en un nodo rojo, todo está bien; en caso contrario, realizamos una rotación entre su padre y un hermano.

- *X* tiene un solo hijo. Si éste es negro, se aplica el caso principal y si es rojo, eliminamos *X*, si es necesario, y coloreamos su hijo en negro.

En algunas ocasiones se utiliza como alternativa la *eliminación perezosa*, en la que los elementos eliminados se marcan simplemente como tales. Sin embargo, consume espacio de forma innecesaria y complica otras rutinas (*véase* el Ejercicio 18.24).

18.6 AA-Árboles

Debido a la gran cantidad de rotaciones necesarias, los árboles rojinegros necesitan de algunos trucos para su codificación, con el subsiguiente peligro de cometer errores. En particular, la operación `eliminar` es bastante complicada. Esta sección describe un tipo de árbol de búsqueda bastante simple pero con un equilibrado bastante competitivo, conocido como *AA-árbol*. Los AA-árboles son el método elegido cuando se necesitan árboles equilibrados, se permite una implementación más inmediata (con el subsiguiente coste) y se requieren eliminaciones. Los AA-árboles añaden una condición adicional a las impuestas por los árboles rojinegros: los hijos izquierdos no pueden ser rojos.

Esta restricción adicional simplifica en gran medida los algoritmos de los árboles rojinegros, por dos razones. En primer lugar, elimina aproximadamente la mitad de los casos de reestructuración. En segundo lugar, simplifica el algoritmo `eliminar`, al no tener que considerar uno de sus casos más complicados. Más exactamente, si un nodo interno tiene un único hijo, éste debe ser un hijo derecho rojo, ya que los hijos izquierdos rojos están ahora prohibidos y un hijo único negro incumpliría la cuarta propiedad de los árboles rojinegros. De este modo, siempre podemos reemplazar un nodo interno por el nodo más pequeño de su subárbol derecho. Este nodo más pequeño será una hoja o tendrá un hijo rojo, que puede ser fácilmente eliminado.

Para simplificar aún más la implementación, representamos la información del equilibrado de un modo más directo. En lugar de guardar un color en cada nodo, almacenamos el *nivel* de dicho nodo. El nivel de un nodo se define como sigue:

- El nivel es 1 si el nodo es una hoja.
- El nivel es el mismo que el de su padre si el nodo es rojo.
- El nivel es una unidad menor que el nivel de su padre si el nodo es negro.

El nivel representa el número de enlaces izquierdos que se encuentran en el camino hacia el centinela `nodoNulo`.

El resultado es un *AA-árbol*. Si traducimos las condiciones estructurales de colores a niveles, sabemos que el nivel del hijo izquierdo debe ser una unidad menor que su padre y el del hijo derecho debe ser cero o una unidad menor que el de su padre (pero no más). Llamaremos *enlaces horizontales* a las conexiones entre un nodo y un hijo suyo del mismo nivel. Entonces, las propiedades de color implican lo siguiente:

1. Los enlaces horizontales son referencias de lados derechos (ya que sólo los hijos derechos son rojos).
2. No existen dos enlaces horizontales consecutivos (ya que no pueden existir dos nodos rojos consecutivos).

3. Los nodos del nivel 2, o superiores, deben tener dos hijos.
4. Si un nodo no tiene un enlace horizontal derecho entonces sus dos hijos están al mismo nivel.

La Figura 18.55 muestra un ejemplo de AA-árbol. La raíz de dicho árbol es el nodo de clave 30. La búsqueda se realiza aplicando el algoritmo usual. Como ya es habitual, `insertar` y `eliminar` son más complejos, ya que los algoritmos naturales de los árboles de búsqueda pueden incumplir las propiedades de los enlaces horizontales. Sin embargo, y no sorprendentemente, las rotaciones pueden solucionar todos los problemas planteados por estas rutinas.

18.6.1 Inserción

La inserción de un nuevo elemento siempre se hace en el nivel más bajo. Como es habitual, esto puede causar problemas. En la Figura 18.55 la inserción de 2 podría crear un enlace horizontal izquierdo, mientras que la inserción de 45 generaría dos enlaces horizontales derechos consecutivos. Como consecuencia, después de insertar un nodo en el nivel más profundo, pueden necesitarse algunas rotaciones para restablecer las propiedades de los enlaces horizontales.

En ambos casos, una única rotación simple arregla el problema. Eliminamos los enlaces horizontales izquierdos haciendo una rotación entre el nodo y su hijo izquierdo y eliminamos los enlaces horizontales derechos consecutivos realizando una rotación entre el primero y el segundo de los (tres) nodos conectados mediante los dos enlaces. Estos procedimientos se llaman `giro` y `reparto`, respectivamente.

`giro` se muestra en la Figura 18.56 y `reparto` en la Figura 18.57. Un `giro` elimina un enlace horizontal izquierdo. Sin embargo, puede generar enlaces horizontales derechos consecutivos, ya que el hijo derecho de *X* puede ser también horizontal. En este caso, invocaremos primero a `giro` y después a `reparto`. Después de `reparto` se incrementa el nivel del nodo del centro. Esto puede causar problemas al padre inicial de *X*, al crear un enlace horizontal izquierdo o enlaces horizontales derechos consecutivos: ambos problemas pueden solucionarse aplicando la estrategia `giro/reparto` en el camino de regreso a la raíz. Esto se hace automáticamente si se usa recursión, así que una implementación recursiva de `insertar` es sólo dos llamadas más larga que la correspondiente rutina de los árboles de búsqueda no equilibrados.

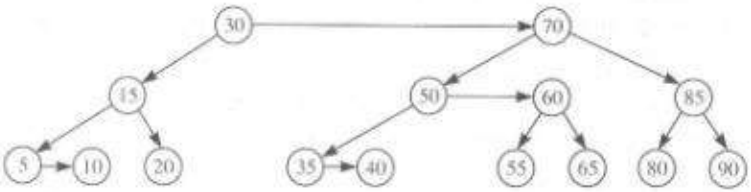


Figura 18.55 AA-árbol obtenido de la inserción de 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55 y 35.

Figura 18.56 `giro` es una rotación simple entre *X* y *P*.

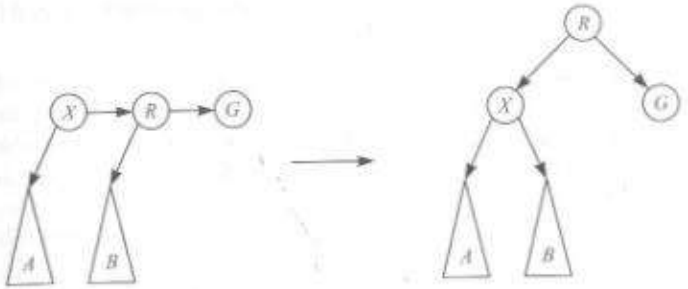


Figura 18.57 `reparto` es una rotación simple entre *X* y *R*; obsérvese que el nivel de *R* aumenta.

Para ver el algoritmo en acción, mostramos el resultado de insertar el elemento 45 en el AA-árbol de la Figura 18.55. En la Figura 18.58, cuando añadimos 45 en el último nivel, se forman dos enlaces horizontales consecutivos. Los pares `giro/reparto` se aplican en el camino de vuelta a la raíz, siempre que sea necesario. En particular, se necesita un `reparto` en el nodo 35 a causa de los enlaces horizontales derechos consecutivos. El resultado se muestra en la Figura 18.59. Cuando la recursión vuelve al nodo 50, encontramos un enlace horizontal izquierdo. Realizamos entonces un `giro` en 50 para eliminar el enlace horizontal (el resultado puede verse en la Figura 18.60) y un `reparto` en 40 para eliminar los enlaces horizontales derechos consecutivos. En la Figura 18.61 se muestra el resultado

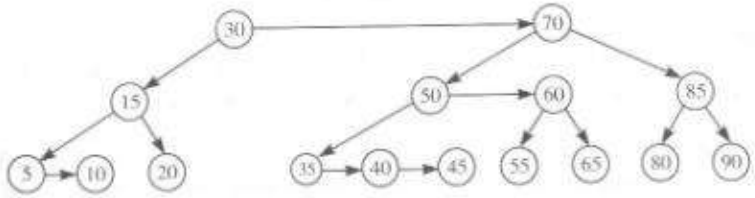


Figura 18.58 Situación después de insertar 45 en nuestro árbol ejemplo; aparecen dos enlaces horizontales derechos consecutivos tras el nodo 35.

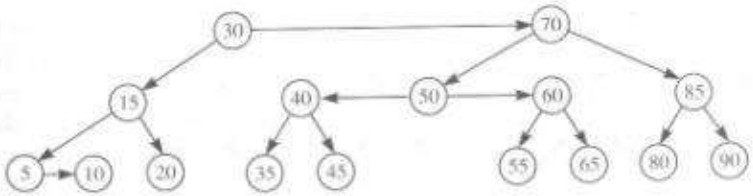


Figura 18.59 Situación después del `reparto` en 35: éste genera un enlace horizontal izquierdo en 50.

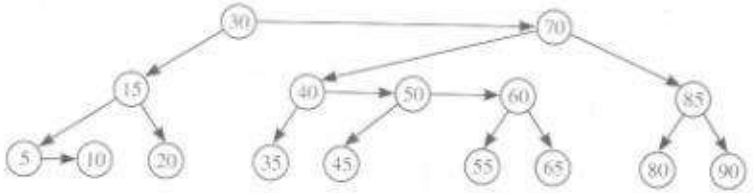


Figura 18.60 Situación después del `giro` en 50; éste genera dos nodos horizontales consecutivos comenzando en 40.

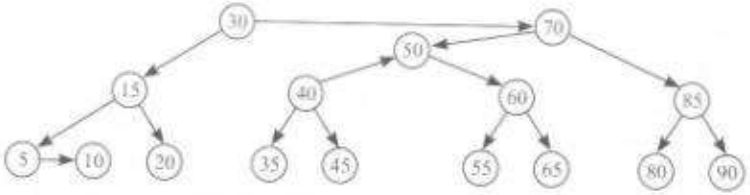


Figura 18.61 Situación después del `reparto` en 40; 50 está ahora al mismo nivel que 70, introduciéndose así un enlace horizontal izquierdo prohibido.

después del `reparto`: ahora el nodo 50 está en el nivel 3 y es el hijo horizontal izquierdo de 70. Necesitamos ejecutar ahora un nuevo par `giro/reparto`. El `giro` en 70 elimina el enlace horizontal izquierdo en el nivel superior, pero genera enlaces horizontales derechos consecutivos, tal y como se ilustra en la Figura 18.62. Cuando se aplica el último intercambio, los nodos horizontales consecutivos desaparecen y 50 se convierte en la nueva raíz del árbol. El resultado se muestra en la Figura 18.63.

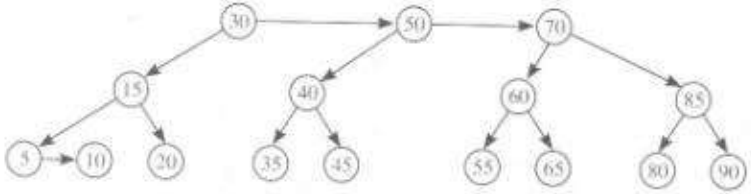


Figura 18.62 Situación después del `giro` en 70; éste crea dos enlaces horizontales consecutivos tras 30.

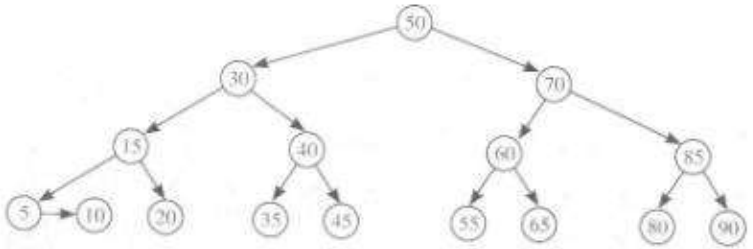


Figura 18.63 Situación después del `reparto` en 30; la inserción se ha completado.

18.6.2 Eliminación

En los árboles binarios de búsqueda ordinarios, el algoritmo `eliminar` se divide en tres casos: el elemento a eliminar puede ser una hoja, tener un solo hijo o tener dos hijos. En los AA-árboles tratamos el caso de un único hijo igual que el caso de dos hijos, ya que el primero de ellos sólo puede aparecer en el nivel 1. Más aún, el caso de dos hijos también es sencillo, si se garantiza que el nodo empleado como valor de reemplazamiento está en el nivel 1 y tiene, en el peor de los casos, un solo enlace horizontal derecho. De este modo, todo gira entorno a `eliminar` un nodo del nivel 1. Claramente, esto puede afectar al equilibrio del árbol (considere, por ejemplo, la eliminación de 10 en la Figura 18.63).

Sea T el nodo actual y considere que estamos empleando recursión. Si la eliminación ha reducido el nivel de uno de los hijos de T hasta dos unidades menos que el nivel de T, entonces éste también ha de ser reducido (sólo el hijo introducido en la llamada recursiva puede haber sido afectado, pero por simplicidad, no guardamos información de ello). Además, si T tiene un enlace horizontal derecho, entonces éste también debe descender de nivel. En este punto podemos tener hasta seis nodos en el mismo nivel: T, el hijo derecho horizontal de T, llamado R, los dos hijos de R, y sus hijos derechos horizontales. La Figura 18.64 muestra la situación posible más sencilla.

Después de eliminar el elemento 1, los nodos 2 y 5 se convierten en nodos del nivel-1. En primer lugar, arreglamos el enlace horizontal izquierdo que se encuentra ahora entre los nodos 5 y 3. Esto requiere esencialmente dos rotaciones: una entre los nodos 5 y 3 y otra entre los nodos 5 y 4. En este caso, el nodo actual T no interviene. Por otro lado, si la eliminación procede del lado derecho, entonces el nodo izquierdo de T puede convertirse en horizontal; esto requeriría una rotación doble similar (comenzando en T). Para evitar distinguir todos los casos, invocaremos siempre a `giro` tres veces. Una vez hecho esto, son suficientes dos llamadas a `reparto` para recolocar los arcos horizontales.

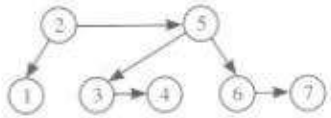


Figura 18.64 Cuando se elimina el elemento 1, todos los nodos están en el nivel 1, apareciendo así enlaces horizontales izquierdos.

18.6.3 Implementación en Java

El esqueleto de la clase de los AA-árboles se muestra en las Figuras 18.65 y 18.66. Gran parte de ella duplica implementaciones de árboles anteriores. Una vez más, empleamos el centinela `nodoNulo`; sin embargo, ahora no necesitamos una pseudo-raíz. El constructor, que no mostramos, genera el `nodoNulo`, igual que en el caso de los árboles rojinegros, y hace que `raiz` le apunte. `nodoNulo` está en el nivel 0. Las rutinas emplean diversos métodos privados auxiliares.

`insertar` se muestra en la Figura 18.67. Como hemos mencionado anteriormente en este capítulo, este método es idéntico al `insertar` recursivo de los ár-

```
package EstructurasDatos;

import Soporte.*; import Soporte.Comparable;
import Excepciones.*;

// Clase AA-Arbol
//
// CONSTRUCCIÓN: sin ninguna inicialización
//
// ********************OPERACIONES PÚBLICAS*********************
// Las mismas que en ArbolBinarioBusqueda; se omiten por brevedad
// ***************************ERRORES*****************************
// Muchas rutinas lanzan ElementoNoEncontrado en condiciones degeneradas
// Insertar lanza ElementoDuplicado si el elemento ya está en el árbol

/**
 * Implementa los AA-árboles.
 * Nótese que todas las comparaciones se basan en el método compara.
 */
public class AA_Arbol implements ArbolBusqueda
{
   public AA_Arbol( )
     { raiz = nodoNulo; }

   public void insertar( Comparable x ) throws ElementoDuplicado
     { raiz = insertar( x, raiz ); }
   public void eliminar( Comparable x ) throws ElementoNoEncontrado
     { raiz = eliminar( x, raiz ); }
   public void eliminarMin( ) throws ElementoNoEncontrado
     { eliminar( buscarMin( ) ); }
   public Comparable buscarMin( ) throws ElementoNoEncontrado
     { /* Implementación habitual; ver el código en Internet */ }
   public Comparable buscarMax( ) throws ElementoNoEncontrado
     { /* Implementación habitual; ver el código en Internet */ }
   public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
     { /* Implementación habitual; ver el código en Internet */ }
   public boolean esVacio( )
     { return raiz == nodoNulo; }
   public void vaciar( )
     { raiz = nodoNulo; }
   public void imprimirArbol( )
      { imprimirArbol( raiz ); }
```

Figura 18.65 Esqueleto de la clase de los AA_árboles (parte 1).

boles binarios de búsqueda, excepto por el hecho de que incluye una llamada a `giro` seguida de una llamada a `reparto`. En las Figuras 18.68 y 18.69, `giro` y `reparto` se implementan de una forma sencilla empleando las rotaciones de árboles de las que disponemos. Por último, `eliminar` se encuentra en la Figura 18.70.

Para ayudarnos en la tarea, mantenemos dos variables, `nodoBorrado` y `ultimoNodo`, que son permanentes ya que se declaran como estáticas. Cuando recorremos un hijo derecho, ajustamos `nodoBorrado`. Debido a que invocamos recursivamente a `eliminar` hasta que llegamos al final (no preguntamos acerca de la igualdad en el recorrido descendente), tenemos garantizado que si el elemento que debemos eliminar está en el árbol, `nodoBorrado` apuntará al nodo que lo contiene.

```
43    private NodoBinario
44    insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
45      { /* Figura 18.67 */ }
46    private NodoBinario
47    eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
48      { /* Figura 18.70 */ }
49    private void imprimirArbol( NodoBinario t )
50      { /* Implementación habitual; ver el código en Internet */ }
51    private NodoBinario giro( NodoBinario t )
52      { /* Figura 18.68 */ }
53    private NodoBinario reparto( NodoBinario t )
54      { /* Figura 18.69 */ }
55
56    private NodoBinario raiz;
57    private static NodoBinario nodoNulo;
58       static     // Inicializador estático de nodoNulo
59       {
60           nodoNulo = new NodoBinario( null );
61           nodoNulo.izquierdo = nodoNulo.derecho = nodoNulo;
62           nodoNulo.nivel = 0;
63       }
64
65    private static NodoBinario nodoBorrado;
66    private static NodoBinario ultimoNodo;
67 }
```

Figura 18.66 Esqueleto de la clase de los AA_árboles (parte 2).

```
1  /**
2   * Método interno para insertar en un subárbol.
3   * @param x el elemento a insertar.
4   * @param t la raíz del árbol.
5   * @return la nueva raíz.
6   * @exception ElementoDuplicado si el elemento que ajusta
7   *            con x ya está en el subárbol de raíz t.
8   */
9  private NodoBinario
10 insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
11 {
12     if( t == nodoNulo )
13         t = new NodoBinario( x, nodoNulo, nodoNulo );
14     else if( x.compara( t.dato ) < 0 )
15         t.izquierdo = insertar( x, t.izquierdo );
16     else if( x.compara( t.dato ) > 0 )
17         t.derecho = insertar( x, t.derecho );
18     else
19         throw new ElementoDuplicado( "insertar de AA_Arbol" );
20
21     t = giro( t );
22     t = reparto( t );
23     return t;
24 }
```

Figura 18.67 Método `insertar` de la clase `AA_Arbol`.

```
/**
 * Giro primitivo para AA-árboles.
 * @param t la raíz del árbol.
 * @return la nueva raíz tras la rotación.
 */
private NodoBinario giro( NodoBinario t )
{
    if( t.izquierdo.nivel == t.nivel )
        t = Rotaciones.conHijoIzquierdo( t );
    return t;
}
```

Figura 18.68 Método `giro` de la clase `AA_Arbol`.

```
/**
 * Reparto primitivo para los AA-árboles.
 * @param t la raíz del árbol.
 * @return la nueva raíz después de la rotación.
 */
private NodoBinario reparto( NodoBinario t )
{
    if( t.derecho.derecho.nivel == t.nivel )
    {
        t = Rotaciones.conHijoDerecho( t );
        t.nivel++;
    }
    return t;
}
```

Figura 18.69 Método `reparto` de la clase `AA_Arbol`.

Nótese que esta misma técnica puede emplearse en la rutina `buscar` para sustituir las comparaciones entre tres realizadas en cada nodo por comparaciones entre dos en cada nodo, más un test de igualdad adicional al final. `ultimoNodo` referencia el nodo del nivel–1 en el que finaliza la búsqueda. Como no nos detenemos hasta que llegamos al final, si el elemento se encuentra en el árbol, `ultimoNodo` apunta al nodo del nivel–1 que contiene el valor de reemplazamiento que debe ser eliminado.

Cuando termina una llamada recursiva, si estamos en el nivel 1, colocamos el valor del nodo en el nodo interno que debe ser sustituido, tras lo que podemos sobrepasar el nodo del nivel–1. En caso contrario, estamos en un nivel superior y necesitamos comprobar si se incumple la condición de equilibrio. Si es así, restablecemos el equilibrio y hacemos tres llamadas a `giro` y dos llamadas a `reparto`. Como hemos comentado anteriormente, esto garantiza que se recuperan las propiedades de los AA-árboles.

18.7 B-Árboles

Hasta este momento hemos supuesto que podemos almacenar las estructuras de datos completas en la memoria principal del computador. Supongamos ahora que tenemos más datos de los que podemos guardar en memoria principal, por lo que debemos almacenar los datos en disco. Cuando esto sucede las reglas del juego cambian, ya que a la hora de evaluar la complejidad de los algoritmos la notación *O* resulta insuficiente.

```
/**
 * Método interno para eliminar elementos de un subárbol.
 * @param x el elemento a eliminar.
 * @param t la raíz del árbol.
 * @return la nueva raíz.
 * @exception ElementoNoEncontrado si ningún elemento del árbol
 *            de raíz t ajusta con x.
 */
private NodoBinario
eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
{
    if( t != nodoNulo )
    {
        // Paso 1: Busca descendiendo por el árbol y actualiza
        //         ultimoNodo y nodoBorrado
        ultimoNodo = t;
        if( x.menorQue( t.dato ) )
            t.izquierdo = eliminar( x, t.izquierdo );
        else
        {
            nodoBorrado = t;
            t.derecho = eliminar( x, t.derecho );
        }
        // Paso 2: Si x está al final del árbol,
        //         lo eliminamos
        if( t == ultimoNodo )
        {
            if( nodoBorrado == nodoNulo ||
                x.compara( nodoBorrado.dato ) != 0 )
                throw new ElementoNoEncontrado( "de AA_Arbol eliminar" );
            nodoBorrado.dato = t.dato;
            nodoBorrado = nodoNulo;
            t = t.derecho;
        }
        // Paso 3: En otro caso, no estamos al final;
        //         debemos restablecer el equilibrio
        else
            if( t.izquierdo.nivel < t.nivel - 1 ||
                t.derecho.nivel < t.nivel - 1 )
            {
                if( t.derecho.nivel > --t.nivel )
                    t.derecho.nivel = t.nivel;
                t = giro( t );
                t.derecho = giro( t.derecho );
                t.derecho.derecho = giro( t.derecho.derecho );
                t = reparto( t );
                t.derecho = reparto( t.derecho );
            }
    }
    return t;
}
```

Figura 18.70 Eliminación en un AA-árbol.

El problema es que el análisis en notación *O* asume que el coste de todas las operaciones es similar. Pero, esto en particular deja de ser cierto cuando interviene la E/S de disco. Por ejemplo, una máquina 25-MIPS supuestamente ejecuta 25 millones de instrucciones por segundo. Esto supone ir muy deprisa, principalmente

por que la velocidad depende en gran medida de las propiedades eléctricas. Por contra, un disco es, en buena parte, un dispositivo mecánico. Por ello, su velocidad depende sobre todo del tiempo de giro y del tiempo que tarda en mover su cabeza lectora. La mayoría de discos giran a 3.600 RPM (discos más rápidos giran a 5.400 RPM). Así, en un minuto se realizan 3.600 revoluciones, o lo que es lo mismo, se produce una revolución cada 1/60 de segundo o 16.7 ms. En media, debemos esperar que, para encontrar lo que estamos buscando, el disco girará media vuelta, luego, si ignoramos otros factores, se tiene un tiempo de acceso de 8.3 ms. Ésta es una estimación generosa, pues los tiempos de acceso habituales varían entre 9 y 11 ms. Como consecuencia, podemos realizar unos 120 accesos a disco por segundo. Esto suena muy bien, hasta que lo comparamos con la velocidad del procesador. Lo que nos encontramos entonces es que 25 millones de instrucciones equivalen a 120 accesos a disco. O visto de otro modo, que un acceso a disco tiene el coste de 200.000 instrucciones. Desde luego, todos son cálculos aproximados, pero las velocidades relativas están muy claras: los accesos a disco son increíblemente costosos. Más aún, las velocidades de los procesadores están creciendo mucho más rápido que las velocidades de los discos (son sus *tamaños* los que sí están creciendo también deprisa), así que podemos permitirnos hacer un montón de cálculos si con ellos conseguimos ahorrarnos un acceso a disco. En la mayoría de los casos, es el número de accesos a disco el que gobierna el tiempo de ejecución. Así que si dividimos por dos el número de accesos a disco dividimos también por dos el tiempo de ejecución.

Veamos como se comporta normalmente un árbol de búsqueda sobre un disco. Supongamos que se desea tener acceso a los registros de conducción de los ciudadanos de Madrid. Asumimos que se tienen 2.000.000 elementos, que cada clave ocupa 32 bytes (es un nombre), y que cada registro supone 256 bytes. Asumimos que todo esto no cabe en la memoria principal y que somos 1 de los 20 usuarios de un sistema (luego disponemos de 1/20 de los recursos). Así, en un segundo, podemos ejecutar un millón de instrucciones o realizar seis accesos a disco.

Entonces el uso de un árbol binario de búsqueda no equilibrado puede suponer un desastre. En el peor de los casos, puede llegar a tener profundidad lineal, lo que nos llevaría a 2.000.000 accesos a disco. En media, una búsqueda exitosa requeriría 1,38 log N accesos a disco, y como log 2.000.000 $\approx$ 22, una búsqueda media necesitaría 31 accesos a disco o 5 segundos. El comportamiento de un árbol rojinegro es algo mejor. El caso peor de 2 log N es poco probable y el caso habitual está muy cercano a log N. Como consecuencia, un árbol rojinegro emplearía, en media, 23 accesos a disco, requiriendo 4 segundos.

Deseamos reducir el número de accesos a disco hasta una constante muy pequeña, como tres o cuatro. Estamos abiertos a escribir un código bastante complicado ya que las instrucciones máquina son, en este contexto, esencialmente gratis, a menos que abusemos de ellas de una forma ridículamente irrazonable. Debe quedar claro que un árbol binario de búsqueda no resolverá nunca el problema, ya que la profundidad habitual del árbol rojinegro está muy próxima a la altura óptima. No podemos ir más allá de log N empleando un árbol de búsqueda binario. La solución es intuitivamente sencilla: si tenemos mayor grado de ramificación, tenemos menor altura. Así, mientras un árbol binario perfecto de 31 nodos tiene 5 niveles, un árbol 5-ario de 31 nodos tiene sólo tres niveles, tal y como se muestra en la Figura 18.71. Un *árbol de búsqueda M-ario* permite ramificaciones de M hijos. A medida que el grado de ramificación aumenta, la profundidad disminuye. Cuando

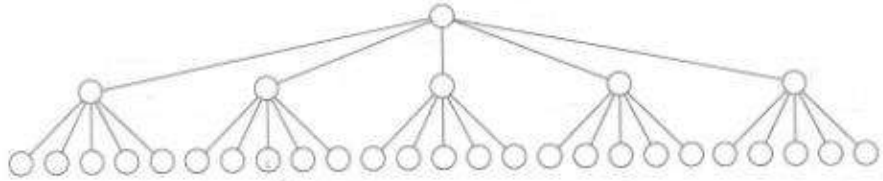


Figura 18.71 Un árbol 5-ario de 31 nodos tiene sólo tres niveles.

la altura de un árbol binario completo es, aproximadamente, $\log_2 N$, la altura de un árbol M-ario completo es, más o menos, $\log_M N$.

Podemos definir los árboles de búsqueda M-arios de forma similar a como definimos los árboles de búsqueda binarios. En este último caso, necesitamos una clave para decidir cuál de las dos ramas tomar. En un árbol de búsqueda M-ario, son necesarias $M - 1$ claves para decidir cuál de las ramas debemos utilizar. Para conseguir que este esquema sea eficiente en el peor de los casos, es preciso asegurarnos que el árbol M-ario esté equilibrado de alguna forma. En caso contrario, y al igual que los árboles binarios de búsqueda, podría degenerar en una lista enlazada. Ahora necesitamos una condición de equilibrio aún más restrictiva, ya que tampoco podemos permitirnos que nuestro árbol M-ario degenere en un árbol binario, ya que en tal caso volveríamos a los log N accesos.

Una forma de implementar todo esto es emplear *B-árboles*. En esta sección describimos el concepto básico de B-árbol[1]. Se conocen muchas variaciones y mejoras, y su implementación es ligeramente complicada, ya que es necesario distinguir bastantes casos. Sin embargo, es sencillo comprobar que, en principio, el uso de un B-árbol garantiza tener que realizar pocos accesos a disco.

Un B-árbol de orden M es un árbol M-ario que verifica las siguientes propiedades[2]:

1. Los datos se almacenan en las hojas.
2. Los nodos internos contienen $M - 1$ claves para guiar la búsqueda: la clave i representa la menor clave en el subárbol $i + 1$.
3. La raíz es una hoja o tiene entre 2 y M hijos.
4. Todos los nodos internos, excepto la raíz, tienen entre $\lceil M/2 \rceil$ y M hijos.
5. Todas las hojas se encuentran a la misma profundidad y tienen entre $\lceil L/2 \rceil$ y L hijos para un cierto valor fijo L (la discusión sobre cómo elegir L se realizará a continuación).

En la Figura 18.72 se muestra un ejemplo de B-árbol de orden 5. Nótese que todos los nodos internos tienen entre tres y cinco hijos (y por tanto, entre dos y cuatro claves); la raíz podría tener sólo dos hijos. Aquí tenemos $L = 5$. En este ejemplo L y M tienen los mismos valores, pero esto no es necesario, en general. Como L es 5, cada hoja guarda entre tres y cinco datos. Es necesario que los nodos estén medio llenos para poder garantizar que el B-árbol no degenerará en un simple árbol binario o ternario. Aunque existen varias definiciones de B-árboles en las que cambia esta estructura, en muchas de ellas en un grado muy pequeño, esta definición es una de las más extendidas.

[1] Lo que describimos aquí se conoce popularmente como B$^+$-árbol.

[2] Las reglas 3 y 5 deben relajarse durante las primeras L inserciones.

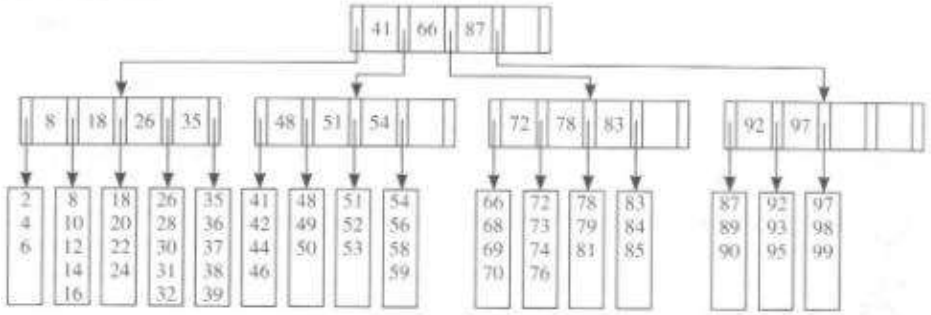


Figura 18.72 Un B-árbol de orden 5.

Cada nodo representa un bloque de disco, así que escogemos M y L basándonos en el tamaño de los elementos que deben guardarse. Como ejemplo, supongamos que un bloque almacena 8.192 bytes. En nuestro ejemplo cada clave ocupa 32 bytes. En un B-árbol de orden M, tendríamos $M - 1$ claves para un total de $32M - 32$ bytes, además de M ramas. Como cada rama es, esencialmente, el número de otro bloque de disco, podemos suponer que cada una de ellas ocupa 4 bytes. Es decir, las ramas ocupan $4M$ bytes. La memoria total necesaria para un nodo interno es $36M - 32$. El mayor valor de M para el que esta cifra no supera 8.192 es 228. Así que elegiremos $M = 228$. Como cada registro de datos es de 256 bytes, seríamos capaces de almacenar 32 de ellos en un bloque, por lo que $L = 32$. De este modo hemos garantizado que cada hoja tiene entre 16 y 32 registros de datos y que cada nodo interno (excepto la raíz) puede ramificarse en al menos 114 hijos. Como tenemos 2.000.000 registros, existen a lo sumo 62.500 hojas. Como consecuencia, en el caso peor, las hojas estarán en el nivel 4. Más concretamente, el número de accesos en el caso peor viene dado aproximadamente por $\log_{M/2} N$, más o menos 1. (Por ejemplo, la raíz y el primer nivel pueden almacenarse en la memoria principal, por lo que los accesos a disco sólo serían necesarios para el nivel 3 e inferiores.)

La cuestión pendiente es cómo insertar y eliminar elementos en un B-árbol. Las ideas básicas se esbozan a continuación. Observe que vuelven a aparecer muchos de los aspectos ya estudiados.

Comenzamos estudiando la inserción. Supongamos que se desea insertar el elemento 57 en el B-árbol de la Figura 18.72. Una búsqueda descendente por el árbol muestra que dicho elemento aún no está en el árbol y que podemos añadirlo en el nivel de las hojas como quinto hijo. Nótese que es posible que para hacerlo debamos reorganizar la información en la correspondiente hoja. Sin embargo, el coste de dicha reorganización es despreciable, si lo comparamos con el tiempo de los accesos a disco, en los que, en este caso, se incluye una escritura en disco.

Desde luego, esto es bastante sencillo ya que la hoja aún no está llena. Supongamos ahora que se desea insertar el elemento 55. La Figura 18.73 muestra el problema: la hoja donde queremos guardar 55 ya está llena. La solución es sencilla: como tenemos $L + 1$ elementos, los repartimos en dos hojas, en las que queda garantizado que podemos almacenar el menor número posible de elementos. En este caso, creamos dos nuevas hojas con tres elementos cada una. Se necesitan dos accesos a disco para escribir dichas hojas y un tercero para actualizar el padre. Note que en el padre cambian las dos claves y las dos ramas, pero lo hacen de una

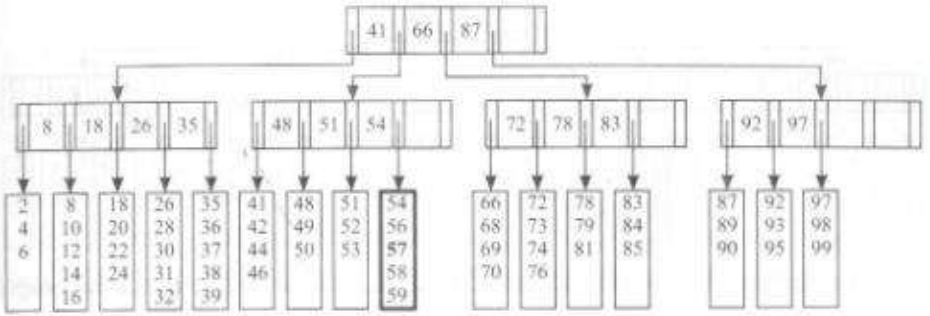


Figura 18.73 El B-árbol resultante tras de la inserción de 57 en el árbol de la Figura 18.72.

forma controlada, que puede calcularse fácilmente. El B-árbol obtenido se muestra en la Figura 18.74. Aunque la división de nodos consume tiempo, ya que necesita dos escrituras en disco adicionales, es una situación relativamente rara. Por ejemplo, si L es 32, cuando dividimos un nodo se crean dos hojas con 16 y 17 elementos, respectivamente. Para la hoja con 17 elementos, podemos realizar 15 inserciones más sin que sea necesario ningún desdoblamiento adicional. Visto de otro modo, por cada división de un nodo hay, aproximadamente, $L/2$ inserciones sin desdoblamiento.

La división en el ejemplo anterior del nodo resuelve el problema, ya que el padre no tiene completo su conjunto de hijos. Pero, ¿qué ocurriría en caso contrario? Supongamos que, por ejemplo, insertamos el elemento 40 en el B-árbol de la Figura 18.74. Debemos dividir la hoja que contiene las claves entre 35 y 39, y ahora 40, en dos nuevas hojas. Pero al hacerlo, el padre tendría seis hijos, y sólo se permite un máximo de cinco. La solución es dividir también el padre. El resultado se muestra en la Figura 18.75. Cuando el padre se divide, debemos actualizar los valores de las claves y también el padre del padre, empleando dos escrituras en disco adicionales (así que esta inserción cuesta cinco escrituras en disco). Sin embargo, una vez más, las claves cambian de una forma controlada, aunque ciertamente, debido a la gran cantidad de casos el código no es nada sencillo.

Cuando se divide un nodo interno, como sucede en este caso, su padre gana un hijo. ¿Qué ocurre si el padre ha llegado al límite permitido de hijos? En tal caso continuamos la división de nodos hasta que encontremos un padre que no necesite

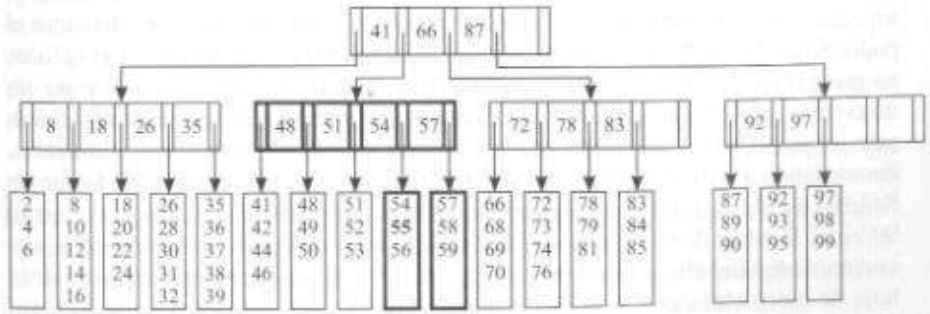


Figura 18.74 La inserción de 55 en el B-árbol de la Figura 18.73 provoca la división en dos nuevas hojas.

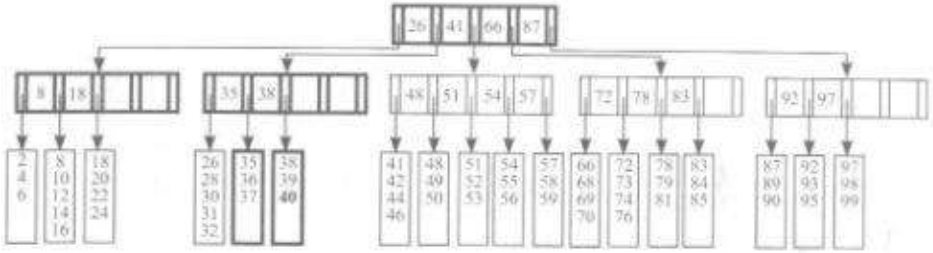


Figura 18.75 La inserción de 40 en el B-árbol de la Figura 18.74 provoca la división en dos nuevas hojas y la división del padre.

ser dividido o bien la raíz del árbol. Observe que esta idea ya ha sido utilizada en los árboles ascendentes rojinegros y en los AA-árboles. Si dividimos la raíz, obtenemos dos raíces. Obviamente, esto es inaceptable, pero en lugar de ello podemos generar una nueva raíz que ahora tenga como hijos las dos raíces previamente obtenidas. Ésta es la razón por la que se permite que la raíz pueda tener sólo dos hijos. Ésta es también la única manera en la que un B-árbol puede ganar altura. No es necesario comentar que la división de todos los nodos hasta llegar a la raíz es una situación excepcional. Esto es debido a que, por ejemplo, en un árbol con cuatro niveles la raíz tendrá que haber sido dividida tres veces a lo largo de toda la secuencia de inserciones (suponiendo que no se ha producido ninguna eliminación). De hecho, tener que dividir un nodo que no sea una hoja es una situación bastante rara.

Existen otras formas de manejar el desbordamiento en el número de hijos. Una técnica alternativa es dar un hijo en adopción a un vecino que tenga sitio para él. Por ejemplo, para insertar 29 en el B-árbol de la Figura 18.75, podemos hacer sitio moviendo el elemento 32 a la siguiente hoja. Esta técnica requiere una modificación del padre, ya que las claves se ven afectadas. Sin embargo, tiende a mantener llenos los nodos, ahorrando espacio a largo plazo.

Podemos realizar una eliminación buscando el elemento que debemos borrar y después eliminándolo. El problema es que si la hoja en la que se encuentra sólo tiene el mínimo número de elementos permitido, ahora nos quedamos por debajo de dicho mínimo. Podemos arreglar esta situación adoptando el hijo de un vecino, si es que el vecino no tiene también el mínimo número de hijos. Si no es así, podemos combinar la hoja con dicho vecino para obtener una hoja completa. Desafortunadamente, esto significa que el padre ha perdido un hijo. Si esto causa que el padre también esté bajo mínimos, reiteramos el mismo procedimiento. Este proceso puede repetirse a lo largo del camino hacia la raíz. La raíz no puede tener un único hijo (permitir tal cosa sería claramente una tontería), así que si, como resultado de la adopción, se queda con un solo hijo, lo que haremos será eliminarle, convirtiendo a su hijo en la nueva raíz del árbol. Ésta es la única manera de que un B-árbol pierda altura. Por ejemplo, supongamos que se desea eliminar el elemento 99 en el B-árbol de la Figura 18.75. Como la hoja tiene sólo dos elementos y el vecino tiene también el mínimo de tres, combinamos los elementos en una nueva hoja de cinco elementos. Como resultado, el padre tiene ahora dos hijos. Sin embargo, puede adoptar otro de un vecino, ya que éste tiene cuatro hijos. Así, ambos tendrán tres hijos. El resultado se muestra en la Figura 18.76.

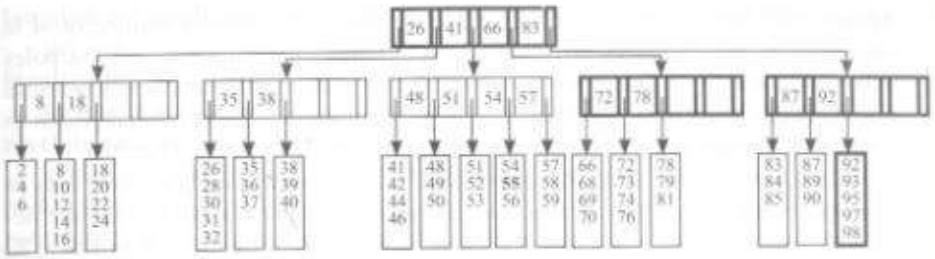


Figura 18.76 B-árbol resultante tras la inserción de 99 en el B-árbol de la Figura 18.75.

Resumen

Los árboles binarios de búsqueda son muy importantes dentro del diseño de algoritmos. Soportan casi todas las operaciones más útiles, y su coste logarítmico en media es muy reducido. Las implementaciones no recursivas de los árboles son algo más rápidas que las correspondientes versiones recursivas, pero estas últimas son más elegantes y más fáciles de entender y depurar. El problema que plantean los árboles de búsqueda es que su eficiencia depende en gran medida de la aleatoriedad de la entrada. Si no se da el caso, el tiempo de ejecución puede crecer extraordinariamente, hasta el punto de que los árboles de búsqueda se hacen más costosos que las listas enlazadas.

Hemos visto varias maneras de solucionar este problema. Todas ellas incluyen la reestructuración del árbol para asegurar que en cada nodo se da algún tipo de equilibrio. La reestructuración se realiza a través de rotaciones que preservan la propiedad de los árboles binarios de búsqueda. Para estos árboles, habitualmente el coste de una búsqueda es menor que el coste en un árbol binario de búsqueda no equilibrado, ya que el nodo central tiende a estar cerca de la raíz. Sin embargo, los costes de la inserción y la eliminación suelen ser algo mayores. Las variaciones en el equilibrio de los árboles se distinguen en la cantidad de esfuerzo necesario para implementar las operaciones que los modifican.

El esquema clásico es el árbol AVL, en el que para cada nodo, las alturas de sus hijos derecho e izquierdo deben diferir, a lo sumo, en una unidad. El problema que presentan en la práctica los árboles AVL es que exigen considerar gran cantidad de casos diferentes, haciendo que el coste de cada inserción o eliminación sea relativamente alto. En este capítulo se han estudiado dos alternativas. La primera son los árboles rojinegros descendentes. Su principal ventaja es que la recuperación de la condición de equilibrio puede implementarse en un único paso, descendiendo por el árbol, en sustitución de la solución tradicional, que desciende por el árbol para regresar después al punto de partida.

Esto conduce a un código más simple y a una ejecución más rápida de la que permiten los árboles AVL. La segunda alternativa son los AA-árboles, que son similares a los árboles rojinegros ascendentes. Su ventaja principal es una implementación recursiva relativamente sencilla de las rutinas de inserción y eliminación. Ambas emplean nodos centinelas para eliminar los casos especiales un tanto fastidiosos.

Se deben emplear árboles binarios de búsqueda no equilibrados cuando se está seguro de que los datos son razonablemente aleatorios y que la cantidad de los

mismos es relativamente pequeña. Se deben emplear los árboles rojinegros si la eficiencia es importante (y no necesitamos eliminar elementos). Los AA-árboles deben utilizarse si se quiere una implementación más sencilla que tenga una más que aceptable eficiencia. Los B-árboles se emplean cuando la cantidad de datos es demasiado grande como para almacenarlos juntos en la memoria principal.

El Capítulo 21 examina otra alternativa: el *árbol de ensanchamiento*. Ésta es una opción muy interesante frente a los árboles de búsqueda equilibrados que tiene además una codificación sencilla y resulta muy competitiva en la práctica. El capítulo siguiente estudia las tablas hash, que son un método completamente diferente de implementar las operaciones de búsqueda.

Elementos del juego

AA-árbol Árbol de búsqueda equilibrado empleado cuando se necesita un caso peor $O(\log N)$, se aceptan implementaciones poco elaboradas y se requiere poder eliminar elementos.

árbol AVL Árbol binario de búsqueda con la propiedad adicional de equilibrio que consiste en que, en cada nodo del árbol, las alturas de sus subárboles izquierdo y derecho difieren a lo sumo en una unidad. Como primer tipo de árbol equilibrado de búsqueda, tiene una gran importancia histórica. Además, ilustra muchas de las ideas que se emplean en otros esquemas.

árbol binario de búsqueda equilibrado Árbol que añade una propiedad estructural para garantizar profundidad logarítmica en el caso peor. Las actualizaciones son más lentas que en los árboles binarios de búsqueda, pero los accesos son más rápidos.

árbol binario de búsqueda Estructura de datos que permite la inserción, búsqueda y eliminación en un tiempo medio de $O(\log N)$. Para cualquier nodo del árbol, todos los nodos con una clave menor que él están en el subárbol izquierdo y todos los nodos con una clave mayor están el subárbol derecho. No se permite la existencia de duplicados.

árbol M-ario Árbol que permite ramificaciones en M hijos. A medida que el grado de ramificación crece, la profundidad disminuye.

árbol rojinegro Árbol de búsqueda equilibrado que es una buena alternativa al árbol AVL, ya que puede emplearse un único paso descendente durante las rutinas de inserción y de eliminación. Los nodos se colorean como rojos o negros de un modo restrictivo que garantiza la profundidad logarítmica. Una detallada codificación genera una implementación más rápida.

B-árbol La estructura de datos más popular para la búsqueda de datos en disco. Existen diversas variantes de la misma idea.

eliminación perezosa Método de eliminación que marca los elementos como borrados, pero no los elimina realmente.

enlace horizontal En un AA-árbol, es una conexión entre un nodo y un hijo suyo del mismo nivel. Un enlace horizontal debe ir siempre hacia la derecha y no pueden existir dos enlaces horizontales consecutivos.

`giro` y `reparto` Un `giro` elimina los enlaces horizontales izquierdos, realizando una rotación entre un nodo y su hijo izquierdo. Un `reparto` elimina enlaces horizontales consecutivos, realizando una rotación entre un nodo y su hijo derecho.

longitud del camino externo Suma del coste de acceso a todos los nodos externos del árbol de un árbol binario. Mide el coste de una búsqueda no exitosa.

longitud del camino interno Suma de las profundidades de los nodos de un árbol binario. Mide el coste de una búsqueda exitosa.

nivel de un nodo En un AA-árbol, es el número de enlaces izquierdos que hay en el camino desde el nodo hasta el centinela `nodoNulo`.

nodo externo Es el nodo `null`.

rotación doble Equivale a dos rotaciones simples.

rotación simple Intercambia los papeles de un padre y un hijo manteniendo la ordenación del árbol. Se emplea para recuperar el equilibrio de un árbol.

Errores comunes

1. Si se emplea un árbol de búsqueda no equilibrado cuando la secuencia de datos de entrada no es aleatoria se obtendrá una eficiencia pobre.
2. La implementación correcta de la operación `eliminar` presenta algunos trucos ingeniosos, especialmente en el caso de los árboles de búsqueda equilibrados.
3. La eliminación perezosa es una buena alternativa a la rutina `eliminar` estándar, pero entonces debemos modificar otros métodos, como `buscarMin`.
4. El código de los árboles de búsqueda equilibrados tiende casi siempre a presentar errores.
5. Si olvidamos devolver una referencia a la nueva raíz del subárbol en los métodos privados auxiliares `insertar` y `eliminar`, estaremos cometiendo un error. El valor de retorno debe ser asignado a `raiz`.
6. El empleo de centinelas con un código que no saque partido de ellos puede conducir a bucles infinitos. Un caso muy común es hacer un test acerca de `null` cuando se emplea el centinela `nodoNulo`.

En Internet

Todo el código de este capítulo forma parte del directorio **DataStructures**. Aquí están los nombres de los ficheros:

AATree.java	Contiene la implementación de los AA-árboles. Es la versión inglesa de la clase `AA_arbol`.
BinaryNode.java	Contiene la implementación de la clase `NodoBinario`.
BinarySearchTree.java	Contiene la implementación de los árboles binarios de búsqueda no equilibrados. Es la versión inglesa de la clase `ArbolBinarioBusqueda`.
BinarySearchTreeWithRank.java	Contiene los árboles binarios de búsqueda no equilibrados con búsqueda por posición. Es la versión inglesa de la clase `ABBConRango`.

RedBackTree.java	Contiene la implementación de los árboles rojinegros. Es la versión inglesa de la clase `ArbolRojoNegro`.
Rotations.java	Contiene las cuatro rotaciones básicas.

Ejercicios

Cuestiones breves

18.1. Muestre el resultado de insertar los elementos 3, 1, 4, 6, 9, 2, 5 y 7 en un árbol binario de búsqueda inicialmente vacío. Muestre después el resultado de borrar la raíz.

18.2. Dibuje todos los árboles binarios de búsqueda que pueden obtenerse a partir de la inserción de las permutaciones de 1, 2, 3 y 4. ¿Cuántos árboles son? ¿Cuál es la probabilidad de obtener cada uno de ellos si todas las permutaciones son igualmente probables?

18.3. Dibuje todos los árboles AVL que pueden obtenerse a partir de la inserción de las permutaciones de 1, 2 y 3. ¿Cuántos árboles son? ¿Cuál es la probabilidad de obtener cada uno de ellos si todas las permutaciones son igualmente probables?

18.4. Repita el ejercicio 18.3 para cuatro elementos.

18.5. Muestre el resultado de insertar 2, 1, 4, 5, 9, 3, 6 y 7 en un árbol AVL inicialmente vacío. Mostrar después el resultado si se hace en un árbol rojinegro descendente.

18.6. Repita los Ejercicios 18.3 y 18.4 para un árbol rojinegro.

18.7. Discuta las ventajas, y los inconvenientes, de evitar todas las excepciones y emplear `null` para indicar los errores.

Problemas teóricos

18.8. Demuestre el Teorema 18.2.

18.9. Muestre el resultado de insertar ordenadamente los elementos del 1 al 15 en un árbol AVL inicialmente vacío. Generalizar esto (con una demostración) para indicar qué es lo que ocurre cuando se insertan los elementos del 1 al $2^k - 1$ en un árbol AVL inicialmente vacío.

18.10. Diseñe un algoritmo para `eliminar` elementos en un árbol AVL.

18.11. Demuestre que la altura de un árbol rojinegro es a lo sumo $2 \log N$ (aproximadamente) y mostrar una secuencia de inserciones que alcance esta cota.

18.12. Muestre que cualquier árbol AVL puede colorearse como un árbol rojinegro. ¿Todos los árboles rojinegros satisfacen la propiedad de los árboles AVL?

18.13. Demuestre la corrección del algoritmo de eliminación de un AA-árbol.

18.14. Supongamos que el atributo `nivel` de un AA-árbol está representado por un `byte` 8-bit. ¿Cuál es el menor AA-árbol que podría desbordar el atributo `nivel` en la raíz?

18.15. Un B*-árbol de orden M es un B-árbol en el que cada nodo interior tiene entre $2M/3$ y M hijos. Las hojas se llenan en la misma medida. Describa un método para realizar la inserción en un B*-árbol.

Problemas prácticos

18.16. Implemente de forma recursiva `buscar`, `buscarMin` y `buscarMax`.

18.17. Implemente iterativamente `buscarKesimo` empleando la misma técnica que en la rutina `buscar` no recursiva.

18.18. Una representación alternativa que permite realizar la operación `buscarKesimo` es almacenar en cada nodo el tamaño de su subárbol izquierdo más 1. ¿Por qué representa esto una ventaja? Reescriba la clase de los árboles de búsqueda para emplear esta representación.

18.19. Escriba un método sobre árboles binarios de búsqueda que tenga como parámetros dos claves, `inferior` y `superior`, y escriba todos los elementos X que se encuentren dentro del rango especificado por dichas claves. El programa debe ejecutarse en un tiempo medio $O(K + \log N)$, donde K es el número de claves mostradas. Si K es pequeño, sólo debería examinarse una pequeña parte del árbol. Tiene que emplear un método recursivo oculto en lugar de iterar la búsqueda ordenada. También se pide acotar el tiempo de ejecución del algoritmo.

18.20. Escriba un método sobre árboles binarios de búsqueda que tenga como parámetros dos enteros, `inferior` y `superior`, y construya de forma óptima un `ABBConRango` equilibrado, que contenga todos los enteros entre `inferior` y `superior`, ambos inclusive. Todas las hojas deben estar en el mismo nivel (si el tamaño del árbol es una unidad menor que una potencia de 2) o en dos niveles consecutivos. *La rutina debe ejecutarse en tiempo lineal*. Pruebe el método resolviendo el problema Josephus (Sección 13.1).

18.21. Las rutinas para realizar las rotaciones dobles son ligeramente ineficientes ya que hacen cambios innecesarios sobre las referencias de los hijos. Reescriba dichas rutinas para evitar las llamadas a la rutina de la rotación simple.

18.22. Diseñe una implementación iterativa descendente de los AA-árboles. Comparar esta implementación con la del texto, discutiendo la simplicidad y eficiencia de ambas.

18.23. Escriba recursivamente los métodos `giro` y `reparto` de modo que en cada `eliminar` sólo se necesite una llamada a las mismas.

Problemas de programación

18.24. Rehaga la clase de los árboles binarios de búsqueda para implementar la eliminación perezosa. Nótese que esto afecta al resto de rutinas, especialmente a `buscarMin` y `buscarMax`, que ahora deben implementarse recursivamente.

18.25. Implemente los árboles binarios de búsqueda de modo que sólo se emplee una comparación por nivel en las rutinas `buscar`, `insertar` y `eliminar`.

18.26. Escriba un programa para evaluar empíricamente las siguientes estrategias de eliminación de nodos con dos hijos.

a) Sustituirlo por el nodo mayor, X, de T_L y eliminar recursivamente X.
b) Reemplazarlo alternativamente con el mayor nodo de T_L y el menor nodo de T_R, y eliminar recursivamente el nodo correspondiente.

c) Sustituirlo por el mayor nodo de T_L o el menor nodo de T_R, eliminando recursivamente el nodo adecuado, realizando la elección de forma aleatoria.

¿Qué estrategia obtiene los mejores resultados? ¿Cuál de ellas tarda menos tiempo de CPU en procesar una secuencia entera de operaciones?

18.27. Implemente el método `eliminar` para árboles rojinegros.

18.28. Implemente las operaciones de los árboles de búsqueda con búsqueda por posición para los árboles de búsqueda equilibrados que prefiera el lector.

18.29. Implemente los B-árboles sobre la memoria principal.

18.30. Implemente los B-árboles sobre ficheros en disco.

18.31. Diseñe un applet que ilustre las operaciones principales de los árboles binarios de búsqueda, tanto equilibrados como no equilibrados.

Bibliografía

Se puede encontrar más información acerca de los árboles binarios de búsqueda, y en particular sobre sus propiedades matemáticas, en los dos libros de Knuth [18 y 19].

Muchos artículos tratan el tema de la pérdida teórica de equilibrio en los árboles de búsqueda, causada por un algoritmo de eliminación sesgado. El artículo de Hibbard [16] propone el algoritmo original de la eliminación y establece que una eliminación conserva la aleatoriedad de los árboles. En [17] se realiza un análisis completo para árboles con tres nodos y en [3] se realiza dicho análisis para árboles con cuatro nodos. El artículo de Eppinger [10] muestra una evidencia empírica de la no aleatoriedad y los artículos de Culberson y Munro [7 y 8] muestran una evidencia analítica, aunque no una demostración completa para el caso general de inserciones y eliminaciones entremezcladas. La afirmación de que el nodo más profundo de un árbol binario de búsqueda aleatorio es tres veces más profundo que la media se demuestra en [11]; el resultado no es nada sencillo.

Los árboles AVL fueron propuestos por Adelson-Velskii y Landis [2]. Se puede encontrar un algoritmo de eliminación en [19]. Los análisis acerca de los costes de las búsquedas en un AVL están sin completar, pero pueden verse algunos resultados en [20]. El algoritmo descendente de los árboles rojinegros es de [15]; puede encontrarse una descripción más comprensible en [21]. En [12] se da una implementación de los árboles rojinegros descendentes sin nodos centinelas; además se presenta una demostración convincente de la poca utilidad de `nodoNulo`. El AA-árbol está basado en los B-árboles binarios simétricos discutidos en [4]. La implementación del texto es una adaptación de la descripción de [1]. En [13] pueden encontrarse multitud de tipos de árboles binarios de búsqueda equilibrados.

Los B-árboles aparecen por primera vez en [5]. La implementación descrita en el artículo original permite que los datos se almacenen tanto en los nodos internos como en las hojas. La estructura de datos descrita aquí se conoce en diversos contextos como B^+-árboles. En [9] puede encontrarse información acerca de los B*-árboles, descritos en el Ejercicio 18.15. En [6] se presenta un resumen de los distintos tipos de B-árboles. En [14] se muestran resultados empíricos sobre distintos esquemas empleados. En [12] puede verse una implementación en C++.

1. A. Andersson, «Balanced Search Trees Made Simple», *Proceedings of the Third Workshop on Algorithms and Data Structures* (1993), 61-71.
2. U. M. Adelson-Velskii y E. M. Landis, «An Algorithm for the Organization of Information», *Soviet Math. Doklady* **3** (1962), 1259-1263.
3. R. A. Baeza-Yates, «A Trivial Algorithm Whose Analysis Isn't: A Continuation», *BIT* **29** (1989), 88-113.
4. R. Bayer, «Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms», *Acta Informatica* **1** (1972), 290-306.
5. R. Bayer y E. M. McCreight, «Organization and Maintenance of Large Ordered Indices», *Acta Informatica* **1** (1972), 173-189.
6. D. Comer, «The Ubiquitous B-tree», *Computing Surveys* **11** (1979), 121-137.
7. J. Culberson y J. I. Munro, «Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations», *Computer Journal* **32** (1989), 68-75.
8. J. Culberson y J. I. Munro, «Analysis of the Standard Deletion Algorithm in Exact Fit Domain Binary Search Trees», *Algorithmica* **5** (1990), 295-311.
9. K. Culik, T. Ottman, y D. Wood, «Dense Multiway Trees», *ACM Transactions on Database Systems* **6** (1981), 486-512.
10. J. L. Eppinger, «An Empirical Study of Insertion and Deletion in Binary Search Trees», *Communications of the ACM* **26** (1983), 663-669.
11. P. Flajolet y A. Odlyzko, «The Average Height of Binary Search Trees and Other Simple Trees», *Journal of Computer and System Sciences* **25** (1982), 171-213.
12. B. Flamig, *Practical Data Structures in C++*, John Wiley and Sons, New York, New York (1994).
13. G. H. Gonnet y R. Baeza-Yates, *Handbook ofAlgorithms and Data Structures*, 2.ª ed., Addison-Wesley, Reading, Mass. (1991).
14. E. Gudes y S. Tsur, «Experiments with B-tree Reorganization», *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), 200-206.
15. L. J. Guibas y R. Sedgewick, «A Dichromatic Framework for Balanced Trees», *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8-21.
16. T. H. Hibbard, «Sorne Combinatorial Properties of Certain Trees with Applications to Searching and Sorting», *Journal of the ACM* **9** (1962), 13-28.
17. A. T. Jonassen y D. E. Knuth, «A Trivial Algorithm Whose Analysis Isn't», *Journal of Computer and System Sciences* **16** (1978), 301-322.

18. D. E. Knuth, *The Art of Computer Programming: Volume 1: Fundamental Algorithms*, 3.ª ed., Addison-Wesley, Reading, Mass. (1997).

19. D. E. Knuth, *The Art of Computer Programming: Volume 3: Sorting and Searching*, 2.ª ed., Addison-Wesley, Reading, Mass. (1997).

20. K. Melhorn, «A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions», *SIAM Journal on Computing* **11** (1982), 748-760.

21. R. Sedgewick, *Algorithms in C++*, Addison-Wesley, Reading, Mass. (1992).

CAPÍTULO

19 Tablas hash

En el Capítulo 18 hemos discutido los árboles binarios de búsqueda, los cuales permiten varias operaciones sobre un conjunto de elementos. Este capítulo discute las *tablas hash* o *tablas de localización*, que permiten únicamente un subconjunto de las operaciones permitidas por los árboles binarios de búsqueda. La implementación de las tablas hash se conoce usualmente como *hashing* o *localización*. La *localización* es la técnica empleada para realizar inserciones, eliminaciones y búsquedas con coste medio constante.

A diferencia de los árboles binarios de búsqueda, el tiempo medio de ejecución de las operaciones de las tablas hash se basa en propiedades estadísticas en lugar de en la espera de entradas aleatorias. Esta mejora se consigue perdiendo gran cantidad de información sobre el orden de los elementos; así, ciertas operaciones, como `buscarMin` o `buscarMax` y la impresión ordenada de la tabla, no pueden realizarse en tiempo lineal. Como consecuencia, la tabla hash y el árbol binario de búsqueda tienen aplicaciones diferentes y eficiencia distinta.

En este capítulo veremos:

- Varios métodos para implementar las tablas de localización.
- Algunas comparaciones analíticas entre dichos métodos.
- Diversas aplicaciones del hashing.
- Una comparación entre las tablas hash y los árboles binarios de búsqueda.

19.1 Ideas básicas

Las *tablas de localización* permiten consultar o eliminar cualquier elemento conociendo su nombre, de modo que lo que estamos implementando es un diccionario. Desearíamos ser capaces de ejecutar las operaciones básicas en tiempo constante, tal y como se ha conseguido en los casos de las pilas y de las colas. Debido a que el tipo de accesos está ahora mucho menos restringido, esto parece un objetivo imposible. Parece razonable que cuando el diccionario aumenta de tamaño, las búsquedas tardarán más tiempo, sin embargo, esto no tiene por qué ser necesariamente así.

Supongamos, por ejemplo, que todos los elementos son enteros no negativos pequeños, entre 0 y 65.535. Entonces podemos emplear un vector, indexado entre 0 y 65.535, para implementar las operaciones como sigue. En primer lugar, inicializamos todas las casillas del vector `a` con 0. Para realizar `insertar(i)` ejecutamos `a[i]++`. Observe que `a[i]` representa el número de veces que ha sido insertado `i`. Para ejecutar `buscar(i)` comprobamos que `a[i]` no sea 0. Para realizar `eliminar(i)`, nos aseguramos de haber encontrado `i`, y si es así, ejecutamos `a[i]--`. Claramente, el tiempo de ejecución de cada operación es constante. Tenemos además el coste de la inicialización del vector, pero ésta supone una cantidad de trabajo constante (65.536 asignaciones), pues sólo se ejecuta una vez.

Esta solución tiene dos problemas. En primer lugar, supongamos que manejamos enteros de 32 bits en lugar de enteros de 16 bits. En tal caso, el vector `a` debería almacenar 4 billones de elementos, lo que es impracticable. En segundo lugar, si los elementos no son enteros sino cadenas de caracteres (o cualquier otra cosa más general), no podríamos emplear directamente enteros para indexar el vector.

El segundo problema no lo es realmente. Igual que el número `1234` es una secuencia de dígitos `(1, 2, 3, 4)`, la cadena `"hola"` es la secuencia formada por los caracteres `'h'`, `'o'`, `'l'` y `'a'`. Tenemos que el número `1234` se puede recuperar mediante el valor de $1 \cdot 10^3 + 2 \cdot 10^3 + 3 \cdot 10^1 + 4 \cdot 10^0$. Recuerde, de la Sección 12.1, que un carácter puede ser representado con 7 bits como un número entre 0 y 127. Como un carácter es básicamente un entero pequeño, podemos interpretar una cadena como un entero. Una representación posible es $\texttt{'h'} \cdot 128^3 + \texttt{'o'} \cdot 128^3 + \texttt{'l'} \cdot 128^1 + \texttt{'a'} \cdot 128^0$. Esto permitiría una implementación mediante un vector básico, como la que hemos visto anteriormente.

El problema que conlleva esta estrategia es que la representación entera descrita genera enteros muy grandes: la representación de `'hola'` es 224.229.227 y cadenas más largas generan representaciones mucho mayores. Esto nos conduce de nuevo al primer problema: ¿Cómo evitamos el uso de un vector absurdamente grande?

Esto se consigue usando una función que asocia a números grandes (o cadenas interpretadas como números) otros más pequeños y manejables. La función que asocia a un elemento un índice pequeño se conoce como *función hash* o *función de localización*. Si `x` es un entero (no negativo) arbitrario entonces `x%tamanyoTabla` genera un número entre 0 y `tamanyoTabla-1`, adecuado para indexar una posición de un vector de tamaño `tamanyoTabla`. Si `s` es una cadena, podemos convertir `s` en un entero grande `x`, empleando el método sugerido anteriormente y después aplicar el operador `%` para obtener un índice adecuado. Así, si `tamanyoTabla` es 10.000, `"hola"` se indexaría por 9.227. La Sección 19.2 discute con detalle la implementación de funciones de localización para cadenas de caracteres.

El empleo de la función de localización introduce una complicación: es posible que a dos elementos distintos les corresponda la misma posición. Esto no podemos evitarlo, ya que existen muchos más elementos potenciales que posiciones. Cuando esto sucede, se produce una *colisión*. Existen muchos métodos para resolver con rapidez una colisión. Estudiaremos tres de los más simples: *exploración lineal*, *exploración cuadrática* y *encadenamiento separado*. Todos ellos son sencillos de implementar, pero tienen una eficiencia distinta, dependiendo del grado de ocupación del vector.

19.2 Función de localización

Calcular la función de localización para cadenas tiene una sutil complicación: la conversión de `s` a `x` genera un entero que es casi más grande de lo que puede almacenar convenientemente la máquina. Esto es debido a que $128^4 = 2^{28}$, que sólo se diferencia del mayor entero de una máquina 32-bit en un factor de 8. Como consecuencia, no podemos calcular la función de localización manejando directamente potencias de 128. En su lugar, nos basaremos en la siguiente observación. Un polinomio cualquiera

$$A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0 \tag{19.1}$$

puede evaluarse como

$$(((A_3)X + A^2)X + A_1)X + A_0 \tag{19.2}$$

Observe que en la Ecuación 19.2 evitamos el cálculo directo de X^i. Esto es conveniente por tres razones. En primer lugar, se evita un resultado intermedio demasiado grande, el cual, como veremos más tarde, provocaría en general, un desbordamiento. En segundo lugar, el cómputo en la ecuación sólo contiene tres productos y tres sumas; un polinomio de grado N se calcula con N productos y N sumas. Esto resulta más económico que el cálculo directo en la Ecuación 19.1. En tercer lugar, el cálculo se realiza de izquierda a derecha (A_3 corresponde a `'h'`, A_2 a `'o'` y así sucesivamente, y X es 128).

Permanece aún el problema de desbordamiento: como el resultado de los cálculos es el mismo, probablemente es demasiado grande. Sin embargo, sólo necesitamos el resultado módulo `tamanyoTabla`. Aplicando el operador `%` después de cada producto (o suma), conseguimos que los resultados intermedios sean pequeños[1]. La función de localización resultante se muestra en la Figura 19.1. Una característica algo incómoda de esta función es que el cálculo de módulo es costoso. Como el desbordamiento está permitido (y sus resultados son consistentes en una misma plataforma), podemos hacer que el cálculo de la función de localización sea algo más rápido realizando una sola operación módulo, justo antes del `return`. Desgraciadamente, la repetición del producto por 128 tiende a desplazar los caracteres del principio hacia la izquierda, no influyendo en el resultado. Para remediar esta situación, multiplicaremos por 37 en lugar de hacerlo por 128. Esto reduce el desplazamiento de los caracteres iniciales.

```
1  // Función de localización aceptable
2  public final static int hash( String clave, int tamanyoTabla )
3  {
4      int valorHash = 0;
5
6      for ( int i = 0; i < clave.length( ); i++ )
7          valorHash = valorHash * 128 + clave.charAt( i ) )
8                                                  % tamanyoTabla;
9      return valorHash;
10 }
```

Figura 19.1 Primer intento de implementación de una función de localización.

[1] En la Sección 7.4 se estudian las propiedades de la operación módulo.

El resultado se muestra en la Figura 19.2. Ésta no es necesariamente la mejor función posible. Es cierto además que en algunas aplicaciones (por ejemplo, cuando intervienen cadenas largas), podemos querer jugar con ella. Sin embargo, en la mayoría de los casos la función es bastante buena. Observe que el desbordamiento podría generar números negativos. Por ello, si el operador de módulo genera un valor negativo, lo convertimos en positivo (líneas 15 y 16). Nótese también que el resultado obtenido al permitir el desbordamiento y haciendo una sola operación de módulo final no es el mismo que el que se obtendría haciendo dicha operación tras cada paso. De modo que hemos alterado ligeramente la función original, pero esto no representa ningún problema, pues no se trataba de algo a mantener a ultranza.

Vale la pena insistir en que, aunque la eficiencia es una consideración importante durante el diseño de la función, perseguimos que ésta distribuya las claves de forma equitativa, a fin de reducir al máximo las colisiones. Como consecuencia, debemos ser cuidadosos en no llevar las optimizaciones demasiado lejos. Un ejemplo de ello es la función de localización de la Figura 19.3. En ella nos limitamos a añadir los caracteres a las claves y devolvemos el resultado módulo `tamanyoTabla`. ¿Qué podría ser más sencillo? La función es fácil de implementar y calcula el valor hash rápidamente. Sin embargo, si `tamanyoTabla` es muy grande, la función no distribuye bien las claves. Por ejemplo, supongamos que `tamanyoTabla` es 10.000. Supongamos también que las claves tienen, a lo sumo, 8 caracteres. Entonces, como un elemento ASCII del tipo `char` es un entero entre 0 y 127, la función sólo podría tomar valores entre 0 y 1.016 (127*8). Indudablemente, ésta no es una distribución uniforme. Toda la eficiencia ganada por la velocidad del cálculo de la función será más que contrarrestada por el esfuerzo que requiere resolver una gran cantidad de colisiones.

Por último, observe que 0 es un resultado posible de la función, por lo que las tablas se indexan comenzando en el 0.

```
1  /**
2   * Rutina de localización para objetos de la clase String.
3   * @param clave la cadena a procesar.
4   * @param tamanyoTabla el tamaño de la tabla hash.
5   * @return el valor hash.
6   */
7  public final static int hash( String clave, int tamanyoTabla )
8  {
9      int valorHash = 0;
10
11     for ( int i = 0; i < clave.length( ); i++ )
12         valorHash = 37 * valorHash + clave.charAt( i );
13
14         valorHash %= tamanyoTabla;
15         if ( valorHash < 0 )
16             valorHash += tamanyoTabla;
17
18         return valorHash;
19 }
```

Figura 19.2 Función de localización más rápida que saca provecho del desbordamiento.

```
1  // Función de localización pobre cuando tamanyoTabla es grande
2  public final static int hash( String clave, int tamanyoTabla)
3  {
4      int valorHash = 0;
5
6      for ( int i = 0; i < clave.length( ); i++ )
7          valorHash += clave.charAt( i );
8
9      return valorHash % tamanyoTabla;
10 }
```

Figura 19.3 Función de localización inadecuada cuando `tamanyoTabla` es grande.

19.3 Exploración lineal

Ahora que disponemos de una función de localización, necesitamos decidir lo que hacer cuando se produzca una colisión. Más concretamente, si a X le corresponde una posición que ya está ocupada, ¿dónde lo colocamos? La estrategia más simple posible es la *exploración lineal*, que consiste en buscar secuencialmente en el vector hasta que encontremos una posición vacía. La búsqueda rota desde la última posición hasta la primera, si ello es necesario. La Figura 19.4 muestra el resultado de insertar las claves 89, 18, 49, 58 y 9 en una tabla hash cuando se emplea exploración lineal. Suponemos que la función de localización devuelve la clave X módulo el tamaño de la tabla. La Figura 19.4 incluye los valores de la función de localización.

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

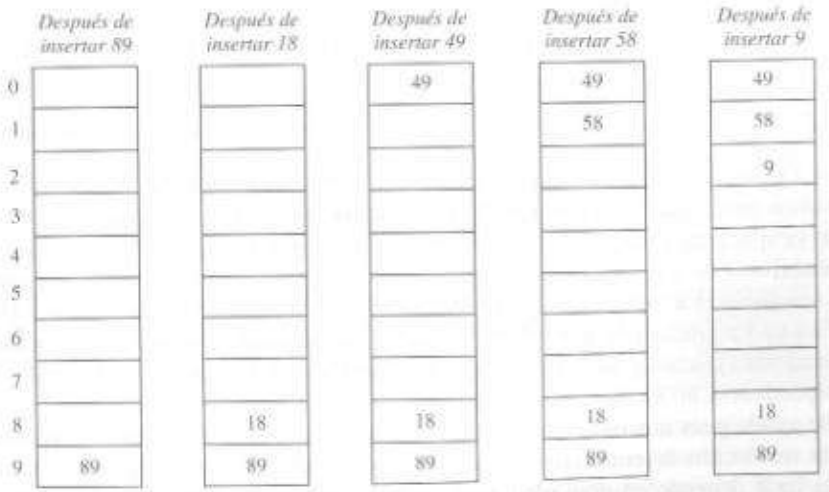


Figura 19.4 Tabla hash con exploración lineal tras cada inserción.

La primera colisión se produce cuando se inserta 49; este elemento se coloca en la siguiente posición vacía, que es la posición 0. Después, 58 colisiona con 18, 89 y 49 antes de encontrar una casilla vacía tres posiciones más allá, que es la posición 1. La colisión del elemento 9 se resuelve de modo similar. Siempre que la tabla sea suficientemente grande, podremos encontrar una casilla vacía. Sin embargo, el tiempo que se emplea en buscarla puede ser bastante elevado. Por ejemplo, si existe una sola casilla vacía en el vector, es posible que tengamos que examinar la tabla entera hasta llegar a ella. En media, podemos esperar tener que buscar en la mitad de la tabla hasta encontrar una casilla vacía. Esto está muy lejos del tiempo constante por acceso que deseamos. En contraposición, si suponemos que la tabla se va a mantener relativamente vacía, entonces las inserciones no serán tan costosas. Todo esto se discutirá en breve.

El algoritmo `buscar` sigue exactamente el mismo camino que el algoritmo `insertar`. Si llega a una casilla vacía , el elemento que estamos buscando no se encuentra en la tabla; en caso contrario, lo encontraremos. Por ejemplo, para encontrar 58, empezamos en la casilla 8 (tal y como indica la función hash). Aquí encontramos un elemento, pero como no es el buscado, pasamos a examinar la posición 0 y después la 1, hasta que lo encontramos. Si realizamos `buscar` con 19, se examinarían las casillas 9, 0, 1 y 2 antes de encontrar una celda vacía en la posición 3. Concluimos que no hemos encontrado 19.

La eliminación estándar no puede aplicarse, ya que al igual que sucede en los árboles binarios de búsqueda, un elemento de la tabla hash no sólo se representa a sí mismo, sino que conecta otros elementos haciendo de posición ocupada durante la resolución de conflictos. Así, si eliminamos 89 de la tabla hash, prácticamente todas las operaciones `buscar` posteriores fallarán. Como consecuencia, se implementa la eliminación perezosa, marcando los elementos como borrados de la tabla. Esta información se almacena en un atributo adicional. Cada elemento está *activo* o *borrado*.

19.3.1 Análisis de la exploración lineal

Para estimar la eficiencia de la exploración lineal, hacemos dos suposiciones:

1. La tabla hash es grande.
2. Cada intento en la tabla hash es independiente de los intentos anteriores.

La primera suposición es perfectamente razonable, ya que en otro caso no estaríamos manejando una tabla hash. La segunda suposición indica que si la fracción ocupada de la tabla es λ, entonces cada vez que examinamos una celda, la probabilidad de que esté ocupada es también λ, independientemente de los intentos anteriores. La independencia es una propiedad estadística importante que simplifica en un grado muy importante el análisis de sucesos aleatorios. Desgraciadamente, tal y como se discute en la Sección 19.3.2, dicha suposición de independencia no es sólo injustificada sino también errónea. A pesar de ello, nos es de ayuda pues nos indica hasta donde podemos llegar si somos más cuidadosos en la resolución de conflictos. Como ya hemos dicho antes, la eficiencia de una tabla hash depende en gran medida de lo llena que está la tabla. Su ocupación se indica mediante el factor de carga.

DEFINICIÓN: El *factor de carga* de una tabla hash es la fracción ocupada de la tabla. Denotamos el factor de carga mediante λ. Está entre 0 (vacía) y 1 (llena).

Ahora podemos realizar, en el Teorema 19.1, un análisis simple de la exploración lineal, aunque parcialmente incorrecto como consecuencia de nuestras observaciones sobre la independencia.

> *Si se supone independencia entre intentos, el número medio de celdas que se examinan en una inserción con exploración lineal es* $1/(1-\lambda)$.
>
> *En una tabla con factor de carga* λ*, la probabilidad de que una celda esté vacía es* $1-\lambda$*. Como consecuencia, el número esperado de intentos independientes hasta encontrar una celda vacía es* $1/(1-\lambda)$.

La demostración del Teorema 19.1 usa el hecho de que, si la probabilidad de que un suceso se produzca es p, se necesitan en media $1/p$ intentos para que dicho suceso se produzca, suponiendo que los intentos son independientes. Por ejemplo, el número esperado de lanzamientos de una moneda hasta que salga cara es dos, y la cantidad de tiros de un dado hasta que salga un 4 es seis. En ambos casos se asume independencia.

19.3.2 Lo que sucede realmente: la agrupación primaria

Desgraciadamente, la independencia asumida no se cumple. Esto se muestra en la Figura 19.5. En la parte superior tenemos el resultado de llenar la tabla hasta el 70 por ciento de su capacidad, cuando los intentos sucesivos son independientes. En el centro el correspondiente al uso de la exploración lineal. Observe los conjuntos de agrupaciones; éste es el fenómeno conocido como *agrupación primaria*.

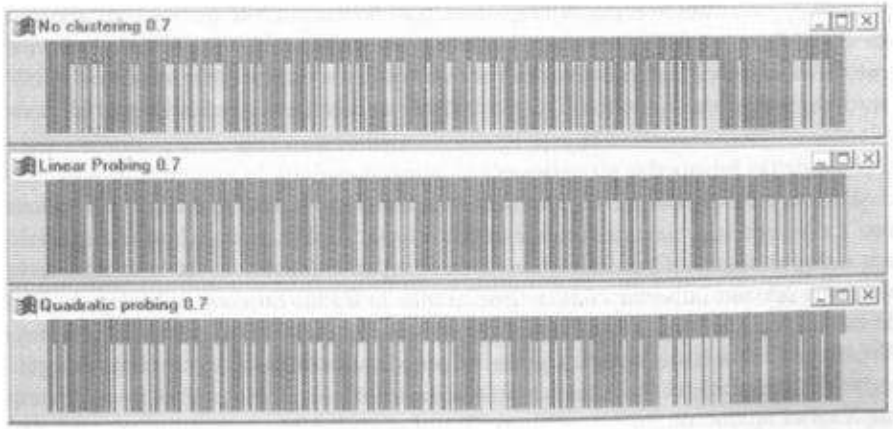


Figura 19.5 Ilustración de la agrupación primaria en la exploración lineal (en el centro) respecto a la no agrupación (arriba) y a la menos importante agrupación secundaria en la exploración cuadrática (abajo); las líneas más largas representan celdas ocupadas; el factor de carga es 0.7.

En la agrupación primaria se tienen bloques de celdas ocupadas. Como consecuencia, muchas claves requerirán un número excesivo de intentos para resolver el conflicto, para terminar añadiéndose a uno de los bloques. En la agrupación primaria, la falta de eficiencia no está sólo provocada por los elementos que colisionan por tener el mismo valor hash, sino también por aquéllos que colisionan en las posiciones alternativas de otros. El análisis matemático que se precisa para tener todo esto correctamente en cuenta es bastante complicado, pero ha sido realizado ya, obteniéndose el resultado del Teorema 19.2.

El número medio de celdas examinadas en una inserción con exploración lineal es cercano a $(1 + 1/(1 - \lambda)^2)/2$.

La demostración queda fuera de los objetivos del texto. Véase [6].

Para una tabla medio llena, tenemos que 2,5 es la cantidad media de celdas examinadas durante una inserción. Esto es muy similar a lo que el análisis anterior ha indicado. Las diferencias se hacen mayores cuando λ se acerca a 1. Por ejemplo, si la tabla está ocupada en un 90 por ciento, entonces $\lambda = 0{,}9$. El análisis sugiere que deberían ser examinadas diez celdas. Esto es bastante, aunque podría resultar asumible. Sin embargo, según el Teorema 19.2, la respuesta real es que necesitaríamos examinar unas 50 celdas. Esto es ciertamente excesivo, particularmente porque se trata sólo de un valor medio, por lo que pueden existir inserciones todavía peores.

19.3.3 Análisis de la operación `buscar`

El coste de una inserción puede emplearse para acotar el coste de `buscar`. Existen dos tipos de operaciones `buscar`: exitosas y no exitosas. Una operación `buscar` no exitosa es sencilla de analizar. La secuencia de celdas que son examinadas durante una búsqueda sin éxito de X es exactamente la misma que las que serían consultadas en `insertar` X. De este modo obtenemos una respuesta inmediata para el coste de las búsquedas no exitosas.

Para operaciones `buscar` exitosas, las cosas son ligeramente más complicadas. La Figura 19.4 muestra una tabla con $\lambda = 0{,}5$. En tal caso, el coste medio de una inserción es 2,5. El coste medio de `buscar` un elemento recién insertado sería entonces 2,5, sin importar cuántas inserciones le sigan. En contrapartida, el coste de buscar el primer elemento insertado en la tabla es siempre el de 1,0 intentos. Así, en una tabla con $\lambda = 0{,}5$, algunas búsquedas son simples y otras más complicadas. En concreto, el coste de una búsqueda exitosa de X es igual al coste de buscar X en el mismo momento en el que X es insertado. Para encontrar el coste medio de una búsqueda exitosa en una tabla con factor de carga λ, debemos calcular el coste medio de una inserción haciendo la media sobre todos los factores de carga que nos llevan a λ. Con esta base, podemos calcular el tiempo medio de una búsqueda con exploración lineal.

El número medio de celdas examinadas durante una búsqueda sin éxito empleando exploración lineal es cerca de $(1 + 1/(1 - \lambda)^2)/2$. *El número medio de celdas examinadas durante una búsqueda con éxito es aproximadamente* $(1 + 1/(1 - \lambda))/2$.

El coste de una búsqueda sin éxito es el mismo que el coste de una inserción. Para las búsquedas exitosas, calculamos el coste medio de una inserción sobre una secuencia de ellas. Como la tabla es grande, podemos calcular esta media evaluando $S(\lambda) = \frac{1}{\lambda}\int_{x=0}^{\lambda} I(x)\,dx$. *En otras palabras, el coste medio de una búsqueda exitosa en una tabla con factor de carga* λ *es igual al coste de una inserción en una tabla con factor de carga x, haciendo la media entre los valores* 0 *y* λ. *Del Teorema 19.2 podemos derivar la siguiente ecuación:*

$$
\begin{aligned}
S(\lambda) &= \frac{1}{\lambda}\int_{x=0}^{\lambda} \frac{1}{2}\left(1 + \frac{1}{(1-x)^2}\right)dx \\
&= \frac{1}{2\lambda}\left(x + \frac{1}{(1-x)}\right)\Bigg|_{x=0}^{\lambda} \\
&= \frac{1}{2\lambda}\left(\left(\lambda + \frac{1}{(1-\lambda)}\right) - 1\right) \\
&= \frac{1}{2}\left(\frac{2-\lambda}{1-\lambda}\right) \\
&= \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)
\end{aligned}
$$

Podemos aplicar la misma técnica para obtener el coste de un `buscar` exitoso, bajo la hipótesis de independencia (empleando $I(x) = 1/(1 - x)$ en el Teorema 19.3). Si no hay ningún tipo de agrupación, el coste medio de dicha operación empleando la exploración lineal sería $-\ln(1 - \lambda)/\lambda$. Si el factor de carga es 0,5, entonces el número medio de intentos para una búsqueda exitosa con exploración lineal es 1,5, mientras el análisis teórico produciría un valor de 1,4. Observe que esta media no depende de la ordenación de las claves, sino sólo de la equidad de la función de localización. Nótese también que aunque manejemos funciones con buenas propiedades, existirán siempre, tanto secuencias de pruebas más largas como otras más cortas, que contribuyen a la media. Por ejemplo, existen, con toda certeza, secuencias de longitud 4, 5 y 6, incluso en una tabla hash medio vacía. Determinar la longitud de la mayor de las secuencias de pruebas es un cálculo realmente complicado, ya que la agrupación primaria no sólo incrementa su longitud sino que aumenta la probabilidad de que aparezca. El problema principal que presenta la agrupación primaria es que la eficiencia empeora extraordinariamente en la inserción en tablas hash con factores de carga elevados. Además, aumenta la probabilidad de algunas de las secuencias más largas (esto es, aquéllas que se encuentran en el extremo superior de la media).

Para reducir el número de intentos, necesitamos un esquema de resolución de conflictos que evite la agrupación primaria. Observe que si la tabla está medio vacía, eliminar los efectos de la agrupación primaria podría evitar, en media, la mitad de los intentos en el caso de una inserción o de una búsqueda sin éxito y una décima parte en el caso de una búsqueda exitosa. A pesar de que resulte razonable intentar reducir la probabilidad de las secuencias más largas, *la exploración lineal es una estrategia bastante útil*. Como es tan fácil de implementar, cualquier otro método que se utilice para eliminar la agrupación primaria debería tener una complejidad similar. En caso contrario, invertiríamos demasiado tiempo intentando evitar una pequeña fracción del número de pruebas. Una alternativa razonable es la *exploración cuadrática*.

19.4 Exploración cuadrática

La *exploración cuadrática* es un método de resolución de conflictos que elimina el problema de la agrupación primaria provocado por la exploración lineal. Su nombre se debe al uso de la fórmula $F(i) = i^2$ en la eliminación de los conflictos. Más concretamente, si la función de localización devuelve como resultado H y la búsqueda en la celda indexada por H no nos conduce a ninguna conclusión, entonces se consultan sucesivamente las celdas $H + 1^2$, $H + 2^2$, $H + 3^2$, ..., $H + i^2$, (empleando circularidad). Esto es claramente diferente a la exploración lineal que consultaría las celdas $H + 1$, $H + 2$, $H + 3$, ..., $H + i$.

La Figura 19.6 muestra la tabla que se obtiene al emplear exploración cuadrática en lugar de exploración lineal, durante la inserción de la secuencia de la Figura 19.4. Cuando 49 colisiona con 89, la primera alternativa es visitar la celda siguiente. Dicha casilla está vacía, así que podemos colocar en ella el 49. A continuación, 58 colisiona con 8. Se examina la celda en la novena posición (la siguiente), pero se produce otra colisión. En la siguiente casilla examinada, que está $2^2 = 4$ posiciones más allá *de la posición hash original*, se encuentra una posición vacante. Así colocamos, en el segundo intento, el elemento 58. Lo mismo sucede para el 9. Observe que los lugares alternativos para los elementos cuyo valor hash es 8 y los lugares alternativos para los elementos cuyo valor hash es 9 no son los mismos. La larga secuencia de intentos realizada para insertar el elemento 58 no afecta a la inserción posterior de 9. Esto contrasta con lo que sucedía cuando empleábamos exploración lineal.

Antes de codificar los algoritmos debemos considerar algunos detalles:

- En la exploración lineal, cada intento examina una celda diferente. ¿Garantiza la exploración cuadrática que cuando se examina una celda, ésta no ha sido consultada previamente en el proceso actual? ¿Garantiza la exploración cuadrática que durante la inserción de X, X es siempre insertado si la tabla no está aún completa?
- La exploración lineal se implementa fácilmente. La exploración cuadrática parece precisar de las operaciones multiplicación y módulo. ¿Esta aparente complejidad adicional hace que la exploración cuadrática sea ineficiente?
- ¿Qué sucede (en ambas exploraciones, lineal y cuadrática) si el factor de carga es demasiado elevado? ¿Podemos expandir dinámicamente la tabla, de modo similar a cómo se hace con las estructuras de datos basadas en vectores?

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

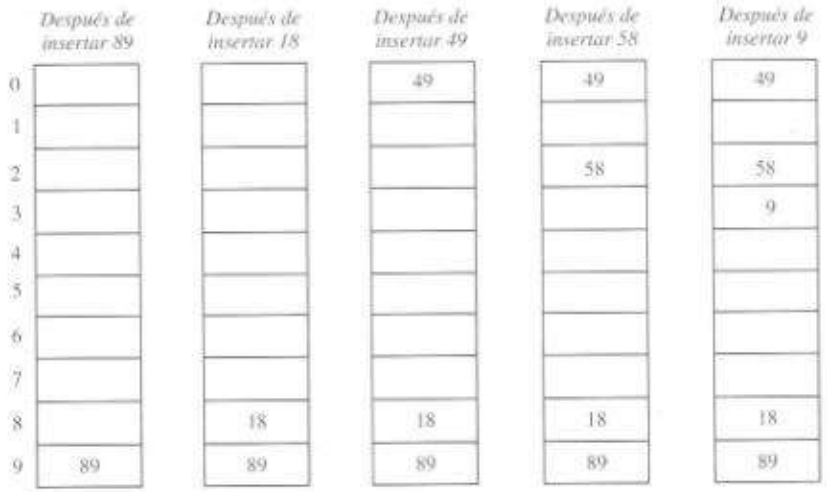


Figura 19.6 Tabla hash con exploración cuadrática tras cada inserción (nótese que el tamaño de la tabla ha sido elegido inadecuadamente ya que no es un número primo).

Afortunadamente, las noticias son relativamente buenas en todos los puntos. Si el tamaño de la tabla es primo y el factor de carga no excede nunca 0,5, sabemos que siempre insertaremos el nuevo elemento X y que ninguna celda será consultada dos veces durante un acceso. Sin embargo, para garantizar que se cumplen estas propiedades, debemos asegurar que el tamaño de la tabla es un número primo, tal y como muestra el Teorema 19.4. Por completitud, la Figura 19.7 muestra una rutina que genera números primos. Dicha rutina emplea el algoritmo de la Figura 9.8.

```
1  /**
2   * Método interno para generar un número primo
3   * mayor o igual que n. Véase la Figura 9.8 para encontrar esPrimo.
4   */
5  private final static int siguientePrimo( int n )
6  {
7      if( n % 2 == 0 )
8          n++;
9
10     for( ; !esPrimo( n ); n += 2 )
11         ;
12
13     return n;
14 }
```

Figura 19.7 Rutina empleada en la exploración cuadrática para encontrar un primo mayor o igual que N.

Si se emplea exploración cuadrática y el tamaño de la tabla es un número primo, siempre podemos insertar un nuevo elemento en la tabla si ésta se encuentra medio vacía. Además, durante una operación de inserción no se examina ninguna celda dos veces.

Sea M el tamaño de la tabla. Supongamos que M es un número primo mayor que 3. Mostraremos en primer lugar, que las primeras $\lfloor M/2 \rfloor$ posiciones alternativas son distintas. Dos de estas posiciones son $H + i^2 \pmod M$ y $H + j^2 \pmod M$, donde $0 < i, j \leq \lfloor M/2 \rfloor$. Supongamos que ambas son iguales, pero que $i \neq j$. Entonces

$$H + i^2 \equiv H + j^2 \pmod M$$
$$i^2 \equiv j^2 \pmod M$$
$$i^2 - j^2 \equiv 0 \pmod M$$
$$(i - j)(i + j) \equiv 0 \pmod M.$$

Como M es primo, se sigue que $i + j$ o $i - j$ es divisible por M. Como i y j son distintos y su suma es menor que M, no puede darse ninguna de estas alternativas, de modo que se llega a una contradicción. Como consecuencia, las primeras $\lfloor M/2 \rfloor$ posiciones alternativas son todas distintas, de modo que una inserción siempre tendrá éxito si la tabla está al menos media vacía.

Si la tabla está algo más que media llena, la inserción puede fallar (aunque esto es extremadamente improbable). En la práctica, no nos planteamos la posibilidad de errores durante las inserciones, ya que ello implicaría que hemos realizado demasiados intentos. Recordemos que en cada inserción esperamos realizar cerca de 2 o 2,5 intentos, mientras que para fallar en una de ellas con una tabla de tamaño 100.000 se necesitarían 50.000 intentos. Además, si el tamaño de la tabla es siempre un número primo y el factor de carga permanece por debajo de 0,5, hemos garantizado absolutamente el éxito de la inserción. Si el tamaño de la tabla no es primo, el número de posiciones alternativas puede verse reducido dramáticamente. Por ejemplo, si el tamaño de la tabla es 16, entonces las únicas posiciones alternativas están a 1, 4 y 9 casillas de la inicial. Una vez más esto no es considerable: aunque no hemos garantizado $\lfloor M/2 \rfloor$ intentos, en la mayoría de las ocasiones tendremos muchos más de las que esperábamos necesitar. A pesar de todo, es mejor jugar sobre seguro y emplear la teoría como guía para escoger los parámetros. Más aún, se ha demostrado empíricamente que los tamaños primos son los ideales para las tablas hash, ya que eliminan algo de la no aleatoriedad ocasionalmente introducida por las funciones de localización.

El segundo aspecto importante es la eficiencia. Recordemos que para un factor de carga de 0,5, la eliminación de la agrupación primaria evita en media sólo 0,5 de las pruebas en una inserción y 0,1 en una búsqueda con éxito. Tenemos otro beneficio adicional: encontrar una secuencia de pruebas larga es significativamente menos probable. Sin embargo, si realizar una prueba en la exploración cuadrática es dos veces más costoso, el esfuerzo no valdrá la pena. La exploración lineal se implementa mediante una sencilla suma (de 1), un test para comprobar si necesitamos circularidad, y una sustracción (en el caso de que se precise circularidad). La fórmula para la exploración cuadrática indica que se necesita sumar 1 (para pasar de $i - 1$ a i), una multiplicación (para calcular i^2), otra suma y una operación módulo. Ciertamente, estos cálculos parecen demasiado costosos para ser empleados en la práctica. Sin embargo, disponemos del truco siguiente, explicado en el Teorema 19.5.

La exploración cuadrática puede implementarse sin costosas multiplicaciones ni divisiones.

Sea H_{i-1} la posición computada más reciente (H_0 es la posición hash inicial), y H_i la posición que estamos intentado calcular. Entonces se tiene

$$H_i = H_0 + i^2 \pmod M$$
$$H_{i-1} = H_0 + (i-1)^2 \pmod M. \tag{19.3}$$

Si restamos ambas ecuaciones en la Ecuación 19.3, obtenemos

$$H_i - H_{i-1} = i^2 - (i-1)^2 \pmod M$$
$$H_i = H_{i-1} + 2i - 1 \pmod M. \tag{19.4}$$

La Ecuación 19.4 nos indica que podemos calcular el nuevo valor H_i a partir del valor previo H_{i-1} sin elevar i al cuadrado. Aunque aún tenemos una multiplicación, ésta es por 2, implementada trivialmente en la mayoría de los computadores mediante el intercambio de bits. La operación módulo no es realmente necesaria, ya que el valor de $2i - 1$ debe ser menor que M. Así, si la añadimos a H_{i-1}, el resultado seguirá siendo menor que M (en cuyo caso no necesitamos el módulo) o ligeramente mayor (en cuyo caso, podemos calcular el módulo restándole M).

El Teorema 19.5 muestra que podemos calcular la siguiente posición que debemos examinar empleando una suma (para incrementar i), un cambio de bits (para evaluar $2i$), una sustracción por 1 (para evaluar $2i - 1$), otra suma (para incrementar la posición inicial con $2i - 1$), un test para comprobar si es necesaria la circularidad, y una poco probable sustracción para implementar la operación módulo. Como consecuencia, la diferencia entre la exploración lineal y la exploración cuadrática es, por cada prueba, un cambio de bits, una sustracción por 1 y una suma. Esto parece ser menor que el coste de realizar una prueba adicional cuando se manejan claves complicadas (como cadenas de caracteres).

El último detalle a considerar es la expansión dinámica. Si el factor de carga es superior a 0,5, duplicaremos el tamaño de la tabla hash. Esto nos lleva a considerar varios puntos. En primer lugar, ¿cuánto nos costará encontrar otro número primo? La respuesta es que es fácil encontrarlos. Sólo tenemos que examinar $O(\log N)$ números antes de encontrar uno que sea primo. Como consecuencia, la rutina de la Figura 19.7 es muy eficiente. Como el test de primalidad tiene un coste máximo de $O(N^{1/2})$, la búsqueda de un número primo tiene un coste máximo de $O(N^{1/2} \log N)$[2]. Esto es mucho menos que el coste $O(N)$ de copiar los contenidos de la tabla vieja en la nueva.

[2] Esta rutina también es necesaria si incluimos un constructor que permita al usuario especificar el tamaño de la tabla hash. La implementación de la tabla hash es la responsable de asegurar que se emplea un número primo

Una vez que se ha asignado memoria al vector de mayor tamaño, ¿debemos copiar en él toda la información? La respuesta es no. Un nuevo vector implica una nueva función hash, de modo que no podemos emplear las posiciones antiguas. Debemos buscar cada elemento en la tabla vieja, calcular su nuevo valor hash e insertarlo en la nueva tabla. Este proceso recibe el nombre de *rehashing*. La sección siguiente muestra que es sencillo implementar el rehashing en Java.

```
package EstructurasDatos;

import Soporte.*;
import Excepciones.*;

// Clase abstracta ExploracionTablaHash
//
// CONSTRUCCIÓN: sin ninguna inicialización
//
// ********************OPERACIONES PÚBLICAS*********************
// void insertar( x )    --> Inserta x
// void eliminar( x )    --> Elimina x
// Hashable buscar( x ) --> Devuelve el elemento que ajusta con x
// void vaciar( )        --> Elimina todos los elementos
// int hash( String str, int tamanyoTabla )
//                       --> Método para indexar cadenas en una tabla hash
// ***************************ERRORES*************************
// buscar y eliminar lanzan ElementoNoEncontrado
// insertar sobreescribe el valor anterior en caso de repetición; no error

/**
 * Implementación de las tablas hash mediante tablas con exploración.
 * Es una clase abstracta que debe ser extendida para
 * implementar un algoritmo de exploración concreto, como
 * la exploración cuadrática.
 * Nótese que todos los "ajustes" se basan en el método equals.
 */
public abstract class ExploracionTablaHash implements TablaHash
{
    /**
     * Método abstracto para la resolución de conflictos.
     * Es el único método que debe ser rescrito en cada clase.
     * @param x el elemento a insertar.
     * @return la posición donde finaliza la búsqueda.
     */
    protected abstract int buscarPos( Hashable x );

    public final static int hash( String clave, int tamanyoTabla )
      { /* Figura 19.2 */ }
    public ExploracionTablaHash( )
      { /* Figura 19.10 */ }
    public final vaciar( )
      { /* Figura 19.10 */ }
    public final Hashable buscar( Hashable x ) throws ElementoNoEncontrado
      { /* Figura 19.11 */ }
    public final void eliminar( Hashable x ) throws ElementoNoEncontrado
      { /* Figura 19.12 */ }
    public final void insertar( Hashable x )
      { /* Figura 19.13 */ }
```

Figura 19.8 Esqueleto de la clase de las tablas hash con exploración (parte 1).

19.4.1 Implementación en Java

Ahora ya estamos preparados para presentar una implementación completa en Java de las tablas hash con exploración cuadrática. Emplearemos herencia y definiremos en primer lugar una clase abstracta que implemente las tablas hash. Después extenderemos dicha clase codificando un método de resolución de conflictos mediante exploración cuadrática. Recordemos que el interfaz `Hashable` necesita la implementación de una función hash adecuada. El esqueleto de la clase se muestra en las Figuras 19.8 y 19.9. A continuación se describe detalladamente cada rutina de la misma.

La tabla hash consiste en un vector de referencias `EntradaHash`; cada referencia es `null` o apunta a un objeto que almacena un elemento y un atributo que indica si esa entrada está activa o se ha eliminado. El vector se declara en la línea 53 como `protected`, de modo que el método de resolución de conflictos, implementado en las clases derivadas, pueda acceder a él. Necesitamos conocer el número de elementos insertados en la tabla hash (incluyendo los elementos marcados como eliminados). Dicho número se almacena en `tamanyoActual`, declarado en la línea 54.

El resto de la clase contiene las declaraciones de las rutinas de la tabla hash. La operación `buscar` devuelve el elemento encontrado en la búsqueda de `x`. Si `x` no se encuentra en la tabla, se lanza una excepción. El método de inserción añade un nuevo elemento a la tabla; si dicho elemento ya está en ella, el valor anterior se sobrescribe con el nuevo.

Las rutinas de inicialización de la tabla hash se muestran en la Figura 19.10. En ellas no hay ningún detalle notable. Observe que inicialmente todas las referencias de `vector` son `null`. Las rutinas de búsqueda se muestran en la Figura 19.11.

```
50      private static final int TAMANYO_TABLA_POR_DEFECTO = 11;
51
52          /** El vector de elementos. */
53      protected EntradaHash [ ] vector; // El vector de elementos
54      private int tamanyoActual;  // El número de celdas ocupadas
55
56      private final void crearVector( int tamanyoVector )
57        { /* Figura 19.10 */ }
58      private final void confirmaBuscar( int posicionActual,
59                                         String mensaje ) throws ElementoNoEncontrado
60        { /* Figura 19.11 */ }
61  }
62
63  // La entrada básica almacenada en ExploracionTablaHash
64  class EntradaHash
65  {
66      Hashable elemento;    // El elemento
67      boolean estaActivo;   // false si se elimina
68
69      public EntradaHash( Hashable e )
70        { this( e, true ); }
71
72      public EntradaHash( Hashable e, boolean i )
73        { elemento = e; estaActivo = i; }
74  }
```

Figura 19.9 Esqueleto de la clase de las tablas hash con exploración (parte 2).

```
/**
 * Método interno para asignar memoria a vector.
 * @param tamanyoVector el tamaño del vector.
 */
private final void crearVector( int tamanyoVector )
{
    vector = new EntradaHash[ tamanyoVector ];
}

/**
 * Construye la tabla hash.
 */
public ExploracionTablaHash( )
{
    crearVector( TAMANYO_TABLA_POR_DEFECTO );
    vaciar( );
}

/**
 * Vacía de forma lógica la tabla hash.
 */
public final void vaciar( )
{
    tamanyoActual = 0;
    for( int i = 0; i < vector.length; i++ )
        vector[ i ] = null;
}
```

ura 19.10 Inicialización de la tabla hash.

```
/**
 * Busca un elemento en una tabla hash.
 * @param x el elemento a buscar.
 * @return el elemento que ajusta con el buscado.
 * @exception ElementoNoEncontrado si no hay en la tabla
 *     ningún elemento que ajuste con x.
 */
public final Hashable
buscar( Hashable x ) throws ElementoNoEncontrado
{
    int posicionActual = buscarPos( x );
    confirmaBuscar( posicionActual, "buscar de ExploracionTablaHash" );
    return vector[ posicionActual ].elemento;
}

/**
 * No hace nada si posicionActual existe y está activa.
 * En caso contrario, lanza una excepción.
 * @param posicionActual el resultado de la llamada a buscarPos.
 * @param mensaje la cadena para construir la excepción.
 * @exception ElementoNoEncontrado si la celda está inactiva.
 */
private final void confirmaBuscar( int posicionActual,
                                   String mensaje ) throws ElementoNoEncontrado
{
    if( vector[ posicionActual ] == null ||
        vector[ posicionActual ].estaActivo == false )
        throw new ElementoNoEncontrado( mensaje );
}
```

ra 19.11 Método buscar de la clase de las tablas hash con exploración.

Todas ellas emplean el método abstracto `buscarPos`. Debemos extender la clase abstracta `ExploracionTablaHash` implementando dicho método. Las rutinas `buscar` son sencillas de codificar. Un elemento es encontrado si el resultado de `buscarPos` es una celda activa (esto es, no es `null` y tampoco está marcada como eliminada). De este modo, si `buscarPos` se detiene en una celda activa debe existir un elemento que ajuste con el buscado. Esta comprobación se hace en el método privado `confirmaBuscar`. De modo similar, la rutina `eliminar` de la Figura 19.12 es sencilla. Comprobamos si `buscarPos` nos conduce a una celda activa. Si es así, la celda se marca como eliminada; en caso contrario, hemos acabado.

La rutina de inserción se muestra en la Figura 19.13. En la línea 9 invocamos al método `buscarPos`. `buscarPos` nos devuelve la posición en la que insertar `x`.

```
1  /**
2   * Elimina elementos de la tabla hash.
3   * @param x el elemento a eliminar.
4   * @exception ElementoNoEncontrado si no se encuentra
5   *     en la tabla hash ningún elemento que ajuste con x.
6   */
7  public final void eliminar( Hashable x ) throws ElementoNoEncontrado
8  {
9      int posicionActual = buscarPos( x );
10     confirmaBuscar( posicionActual, "eliminar de
                                    ExploracionTablaHash" );
11     vector[ posicionActual ].estaActivo = false;
12 }
```

Figura 19.12 Rutina `eliminar` para las tablas hash con exploración.

```
1  /**
2   * Inserción en la tabla hash. Si el elemento ya está en
3   * en la tabla, lo reemplaza por el nuevo elemento.
4   * @param x el elemento a insertar.
5   */
6  public final void insertar( Hashable x )
7  {
8          // Inserta x como activo
9      int posicionActual = buscarPos( x );
10     vector[ posicionActual ] = new EntradaHash( x, true );
11     if( ++tamanyoActual < vector.length / 2 )
12         return;
13
14         // CÓDIGO DE REHASHING
15     EntradaHash [ ] vectorAntiguo = vector;
16
17         // Crea una nueva tabla vacía con tamaño doble
18     crearVector( siguientePrimo( 2 * vectorAntiguo.length ) );
19     tamanyoActual = 0;
20
21         // Copia la tabla
22     for( int i = 0; i < vectorAntiguo.length; i++ )
23         if( vectorAntiguo[ i ] != null && vectorAntiguo[ i ].estaActivo
24             insertar( vectorAntiguo[ i ].elemento );
25 }
```

Figura 19.13 Rutina de inserción de la clase de las tablas hash con exploración, que incluye el código de rehashing.

```
1  // Implementación de las tablas hash con exploración cuadrática.
2  public class TablaExploracionCuadratica extends ExploracionTablaHash
3  {
4      protected final int buscarPos( Hashable x )
5      {
6          int colision = 0;
7          int posicionActual = x.hash( vector.length );
8
9          while( vector[ posicionActual ] != null &&
10                !vector[ posicionActual ].elemento.equals( x ) )
11         {
12             posicionActual += 2 * ++colision - 1;
13             if( posicionActual >= vector.length )
14                 posicionActual -= vector.length;
15         }
16
17         return posicionActual;
18     }
19 }
```

Figura 19.14 Clase que implementa la exploración cuadrática.

La inserción se realiza en la línea 10, en la que construimos una nueva `EntradaHash` y la añadimos al vector. Observe que esto sobrescribe cualquier valor que ajuste con `x` almacenado previamente. En la línea 11 ajustamos `tamanyoActual` y terminamos, a menos que sea necesario realizar una reevaluación de la función de localización (*rehashing*). Así, el código restante implementa el rehashing.

En la línea 15 se guarda una referencia a la tabla original. Después creamos una nueva tabla vacía de tamaño doble, en las líneas 18 y 19. Por último, recorremos el vector antiguo insertando sus elementos activos en la nueva tabla. La rutina `insertar` emplea una nueva función de localización (ya que el vector tiene un tamaño diferente) y resuelve de forma automática todas las colisiones. Podemos estar seguros de que la llamada recursiva a `insertar` (en la línea 24) no fuerza un nuevo rehash. Como alternativa, podemos reemplazar la línea 24 con dos líneas de código entre sus correspondientes llaves (*véase* el Ejercicio 19.14).

De momento, nada de lo que hemos hecho depende de la exploración cuadrática. La Figura 19.14 implementa dicha estrategia de resolución de conflictos extendiendo `ExploracionTablaHash` y codificando el método `buscarPos` mediante el algoritmo de exploración cuadrática. La búsqueda en la tabla continúa hasta que se encuentra una celda vacía o un elemento que ajuste. La metodología descrita en el Teorema 19.5 se implementa en las líneas 12 a 14. Observe que si `x` está marcado como eliminado, su posición será devuelta por `buscarPos`. `confirmaBuscar` lanzará una excepción siempre que determine que `x` no es un elemento activo.

19.4.2 Análisis de la exploración cuadrática

El análisis matemático riguroso de la exploración cuadrática todavía no ha podido ser realizado, debido a su complejidad. Aunque esta estrategia elimina la agrupación primaria, los elementos indexados en la misma posición probarán las mismas celdas alternativas. Esto se conoce con el nombre de *agrupación secundaria*. Debido a ello, de nuevo, no podemos suponer la independencia de intentos sucesivos. Sin embargo, la agrupación secundaria produce una separación mucho menor de los resultados teóricos. Los resultados de las simulaciones sugieren que, por lo general, sólo se incrementa en 1/2 el número de intentos por búsqueda, y esto sólo llega a suceder en tablas con factores de carga elevados. La Figura 19.5 muestra la diferencia entre la exploración lineal y la cuadrática, poniendo de manifiesto que el grado de agrupación es menor en la exploración cuadrática que en la lineal.

Existen varias técnicas para eliminar la agrupación secundaria. La más conocida es el *doble hashing*, que emplea una segunda función de localización para la resolución de conflictos. Más concretamente, realizamos intentos a distancia $Hash_2(X)$, $2Hash_2(X)$ y así sucesivamente. Esta segunda función de localización debe reunir ciertas propiedades (por ejemplo, no puede devolver 0), y debemos asegurar que puede llegar a examinar todas las celdas. Una función como $Hash_2(X) = R - (X \bmod R)$, donde R es un primo menor que M, generalmente funcionará bien. El doble hashing es muy interesante teóricamente hablando, ya que puede demostrarse que realiza el mismo número de intentos que se obtienen en el análisis aleatorio teórico de la exploración lineal. Sin embargo, es algo más complicado de implementar que la exploración cuadrática, y como veremos, algunos de sus detalles requieren especial atención.

Aparentemente no existe ningún argumento de peso para no emplear en la práctica la estrategia de exploración cuadrática, salvo el gasto que supone mantener la tabla medio vacía. Además, éste sólo sería relevante en otros lenguajes de programación, distintos a Java, en los que es posible que los elementos que almacenemos sean de gran tamaño.

19.5 Hashing enlazado

Una alternativa a la exploración cuadrática muy conocida es el *hashing enlazado* o *hashing abierto*. En el hashing enlazado, se maneja un vector de listas enlazadas: $L_0, L_1, \ldots, L_{M-1}$. La función de localización indica en qué lista debemos insertar el elemento X y después, durante la ejecución de una operación `buscar`, nos dice en qué lista se encuentra el elemento X. La idea subyacente es que, aunque la búsqueda en una lista enlazada es una operación lineal, si las listas son lo suficientemente cortas, el tiempo de búsqueda será reducido. En particular, supongamos que el factor de carga, que ahora se define como N/M, es λ. Nótese que en el hashing enlazado, el factor de carga no está acotado por 1,0. La longitud media de cada lista es λ, lo que hace que el número medio de intentos durante una inserción o una búsqueda sin éxito sea λ. El número de intentos correspondiente a una búsqueda exitosa es $1 + \lambda/2$. Esto es debido a que una búsqueda exitosa siempre se realizará en una lista no vacía, y es de esperar que tengamos que recorrer la mitad de sus elementos. El coste relativo de una búsqueda exitosa respecto a una no exitosa puede parecer anómalo cuando $\lambda < 2$, ya que la primera de ellas es más costosa que la segunda. Sin embargo, esto tiene sentido ya que muchas de las búsquedas sin éxito toparían con una lista vacía.

Un factor de carga muy corriente es 1,0; un factor de carga menor no aumenta de modo significativo la eficiencia, y además necesita de un espacio extra. El atractivo del hashing enlazado es que la eficiencia no se ve afectada por un moderado incremento del factor de carga, además de que podemos evitar el rehashing.

Éste es un detalle importante en los lenguajes en los que no se puede realizar la expansión dinámica de los vectores. Además, el número de pruebas de una búsqueda es menor que en la exploración cuadrática, en particular en las búsquedas no exitosas.

Podemos implementar el hashing enlazado empleando las clases de listas enlazadas existentes. Sin embargo, como en esta ocasión el nodo cabecera representa un gasto inútil de espacio, pues no se necesita para nada, podemos optar por no reutilizar componentes, implementando las listas como si fuesen pilas. El esfuerzo realizado en la codificación resulta gratamente clarificador. Además, el gasto de memoria se reduce a una referencia por nodo, además de una referencia adicional por lista; por ejemplo, cuando el factor de carga es 1,0 tenemos dos referencias por elemento. Este hecho puede ser importante en otros lenguajes de programación si los elementos a guardar son de gran tamaño. En nuestro caso, tenemos las mismas dificultades que en las implementaciones de las pilas basadas en vectores y en listas enlazadas.

Resumen

Las tablas hash pueden emplearse para que el coste medio de las operaciones `insertar` y `buscar` sea constante. Cuando utilizamos tablas hash, es especialmente importante prestar atención a detalles como el factor de carga; en caso contrario, las cotas constantes de tiempo no son de aplicación. También es importante elegir cuidadosamente la función de localización, sobre todo cuando las claves que se manejan no son enteros ni cadenas cortas. Debe elegirse una función fácilmente calculable con una buena distribución de sus valores.

Normalmente, en el hashing enlazado el factor de carga está muy cercano a 1, aunque la eficiencia no empeora sensiblemente cuando este factor crece mucho. En la exploración cuadrática, el tamaño de la tabla debe ser un número primo y el factor de carga no debería exceder 0,5. El rehashing debe emplearse para permitir que la tabla crezca manteniendo el factor de carga adecuadamente. Esto es importante si hay escasez de espacio y no es posible declarar inicialmente una tabla hash de gran tamaño.

A pesar de los resultados obtenidos para las tablas hash, podemos seguir empleando árboles binarios de búsqueda para implementar las operaciones `insertar` y `buscar`. Aunque las cotas del tiempo medio obtenidas son $O(\log N)$, los árboles binarios de búsqueda permiten disponer adicionalmente de diversas rutinas que precisan de un cierto orden y son, por tanto, más potentes. Si empleamos una tabla hash no podemos buscar de forma eficiente el menor de los elementos almacenados ni extender la tabla para permitir computaciones sobre el i-ésimo elemento de la tabla respecto del orden. Además, no podemos buscar eficientemente una cadena a menos que la conozcamos de forma exacta. Un árbol binario de búsqueda puede encontrar rápidamente todos los elementos que se encuentran en un rango determinado, mientras que esto no lo puede hacer una tabla hash. Por otra parte, en la práctica la cota $O(\log N)$ no resulta mucho mayor que $O(1)$, especialmente porque los árboles binarios no necesitan ni de multiplicaciones ni de divisiones.

Por otra parte, el caso peor para el hashing suele provenir de una incorrecta elección de los mecanismos que intervienen, mientras que una entrada ordenada puede provocar el mal comportamiento de los árboles binarios. Los árboles binarios de búsqueda son algo costosos de implementar. Así que, si no necesitamos ninguna información de orden y existe alguna posibilidad de que aparezcan entradas ordenadas, la tabla hash es la estructura de datos que debemos emplear.

Las aplicaciones del hashing son muy numerosas. Los compiladores emplean tablas hash para tratar las variables declaradas en el código. Dicha estructura de datos se llama *tabla de símbolos*. Las tablas hash son las más adecuadas en este caso ya que sólo se necesitan operaciones de `insertar` y `buscar`. Normalmente, los identificadores son cortos, de modo que la función hash puede computarse rápidamente. Observe que en esta aplicación, la mayoría de las búsquedas tienen éxito.

Otro uso muy común de las tablas hash son los juegos de computador. A medida que el programa busca entre las distintas líneas de juego, lleva la cuenta de las posiciones que ha visto mediante la computación de una función hash basada en la posición (y almacenando el movimiento hecho para esa posición). Si una de dichas posiciones se repite, el programa puede evitar la repetición de (costosos) cálculos haciendo una transposición de movimientos. Esta estructura común a todos los juegos de computador se llama *tabla de transposición*. Esto ya se vio en la Sección 10.2, cuando se implementó el juego de las tres en raya.

Un tercer uso del hashing son los correctores ortográficos. Si la detección de los errores prima sobre su corrección, se puede hacer un hashing previo sobre un diccionario completo, pudiéndose comprobar la corrección de las palabras en tiempo constante. Las tablas hash son adecuadas para ello ya que no es importante mantener ordenadas las palabras. Basta con imprimir los errores en el mismo orden en el que se producen.

Esto completa la discusión de los mecanismos básicos de búsqueda. El siguiente capítulo estudia los montículos binarios, los cuales implementan las colas de prioridad, permitiendo así el acceso eficiente a los elementos más pequeños de una colección.

Elementos del juego

agrupación primaria Problema de la exploración lineal que afecta a la eficiencia. Aparecen grandes grupos de celdas ocupadas, haciendo que las inserciones en ellos sean más costosas. Además, dichas inserciones provocan que los grupos de celdas ocupadas crezcan aún más.

agrupación secundaria Agrupación que se produce cuando los elementos indexados en la misma posición examinan las mismas celdas alternativas. Supone una desviación menor de los resultados teóricos.

colisión Se produce cuando a dos o más elementos de una tabla hash les corresponde en ella la misma posición. Este problema es inevitable ya que habitualmente hay más elementos que posiciones.

eliminación perezosa Técnica de marcar los elementos como eliminados en lugar de borrarlos físicamente. Es necesaria en las tablas hash con exploración.

exploración cuadrática Resolución de conflictos que examina las celdas 1, 4, 9 y así sucesivamente, a partir de la posición inicial.

exploración lineal Técnica para evitar los conflictos, que examina secuencialmente el vector hasta que se encuentra una celda vacía.

factor de carga Número de elementos de la tabla hash dividido por el tamaño del vector de la tabla hash. En una tabla hash con exploración, el factor de carga toma valores entre 0 (vacía) y 1 (llena). En el hashing enlazado puede ser mayor que 1.

función hash Función que convierte un elemento en un entero adecuado para indexar la posición del vector en el que se almacena dicho elemento. Si la función hash es inyectiva, podemos acceder al elemento a través de su índice en el vector. Normalmente no son inyectivas, por lo que varios elementos pueden colisionar en el mismo índice.

hashing Implementación de las tablas hash para realizar inserciones, eliminaciones y búsquedas.

hashing doble Técnica de hashing que no sufre de agrupación secundaria. Para la resolución de conflictos emplea una segunda función hash.

hashing enlazado Alternativa eficiente, hablando en términos de espacio, a la exploración cuadrática, en la que se mantiene un vector de listas enlazadas. Es menos sensible a factores de carga elevados. Presenta algunos de los inconvenientes de las implementaciones de las pilas basadas en vectores y en listas enlazadas.

tabla hash Tabla empleada para implementar un diccionario cuyas operaciones se realizan en tiempo constante.

Errores comunes

1. La función de localización devuelve un elemento de tipo `int`. Debido a que los cálculos intermedios pueden provocar un desbordamiento, la variable local debe comprobar que el resultado de la operación de módulo no es negativo, evitando así la devolución de un valor fuera de rango.
2. La eficiencia de una tabla hash con exploración empeora sensiblemente a medida que el factor de carga se aproxima a 1,0. No debe permitirse que esto suceda, haciendo un rehashing cuando el factor de carga se aproxime a 0,5.
3. La eficiencia de todos los métodos de hashing depende del uso de una buena función de localización. Un error muy usual es emplear una función inadecuada.

En Internet

La tabla hash con exploración cuadrática se encuentra en el directorio **DataStructures**. En el directorio **Chapter19** podemos encontrar una colección de ficheros para generar la Figura 19.5.

HashEntry.java	Contiene la implementación de `EntradaHash`.
ProbingHashTable.java	Contiene la implementación de la clase abstracta de las tablas hash con exploración. Es la versión inglesa de la clase `ExploracionTablaHash`.
QuadraticProbingTable.java	Contiene la implementación de las tablas hash con exploración cuadrática. Es la versión inglesa de la clase `TablaExploracionCuadratica`.

Ejercicios

Cuestiones breves

19.1. ¿Cuáles son los índices del vector de una tabla hash de tamaño 11?

19.2. ¿Cuál es el tamaño apropiado de una tabla hash si el número de elementos que hay en ella es 10?

19.3. Explique cómo se realizaría la eliminación en las tablas con exploración y con hashing enlazado.

19.4. ¿Cuál es el número esperado de pruebas en una búsqueda con y sin éxito en una tabla hash con exploración lineal y factor de carga 0,25?

19.5. Dada la entrada (4.371, 1.323, 6.173, 4.199, 4.344, 9.679, 1.989), una tabla de tamaño fijo 10 y la función hash $H(X) = X \bmod 10$, muestre

a) la tabla hash con exploración lineal resultante.
b) la tabla hash con exploración cuadrática resultante.
c) la tabla con hashing enlazado resultante.

19.6. Muestre el resultado de hacer un rehashing en cada una de las tablas del Ejercicio 19.5. El tamaño de la nueva tabla debe ser un número primo.

19.7. La rutina `esVacia` aún no ha sido implementada. Impleméntela de modo que devuelva la expresión `tamanyoActual == 0`.

Problemas teóricos

19.8. Una estrategia alternativa de resolución de conflictos consiste en definir una secuencia, $F(i) = R_i$, donde $R_0 = 0$ y $R_1, R_2, \ldots, R_{M-1}$ es una secuencia aleatoria de los $M - 1$ primeros enteros (recuerde que el tamaño de la tabla es M).

a) Demuestre que bajo esta estrategia, las colisiones pueden resolverse siempre que la tabla no esté llena.
b) ¿Podría eliminar esta estrategia la agrupación primaria?
c) ¿Podría eliminar esta estrategia la agrupación secundaria?
d) Si el factor de carga de la tabla es λ, ¿cuál es el coste en tiempo de una inserción?
e) La generación de una permutación aleatoria mediante el algoritmo de la Sección 9.4 implica un gran número de (costosas) llamadas al generador de números aleatorios. Dé un algoritmo eficiente para la generación de permutaciones aleatorias que evite las llamadas al generador de números aleatorios.

19.9. Si el rehashing se produce tan pronto como el factor de carga se aproxima a 0,5, en adelante cuando se inserte un nuevo elemento, el factor de carga estará entre 0,25 y 0,5. ¿Cuál es el factor de carga esperado en esta situación? En concreto, ¿es o no cierto que el factor de carga medio es 0,375?

19.10. Cuando se ejecuta el proceso del rehashing, se hacen $O(N)$ intentos para insertar N elementos. Dé una estimación del número de intentos (es decir, N o $2N$ o algo similar) necesario. *Indicación:* Calcule el coste medio de una inserción en la nueva tabla. En dichas inserciones el factor de carga varía desde 0 a 0,25.

19.11. Bajo ciertas hipótesis, el coste esperado de una inserción en una tabla hash con agrupación secundaria viene dado por $1/(1-\lambda) - \lambda - \ln(1-\lambda)$. Desgraciadamente, esta fórmula no es ajustada para la exploración cuadrática. Pese a ello, asumiendo que lo fuera, determine lo siguiente:

a) El coste esperado de una búsqueda sin éxito.
b) El coste esperado de una búsqueda exitosa.

19.12. Empleamos una tabla hash con exploración cuadrática para almacenar 10.000 objetos de tipo `String`. Supongamos que el factor de carga es 0,4 y que la longitud media de las cadenas es 8. Determine lo siguiente:

a) El tamaño de la tabla hash.
b) La cantidad de memoria empleada para almacenar los 10.000 objetos de tipo `String`.
c) La cantidad de memoria adicional empleada por la tabla hash.
d) La memoria total que necesita la tabla hash.
e) El coste en espacio.

Problemas prácticos

19.13. Implemente la exploración lineal.

19.14. Implemente en la clase de las tablas hash con exploración, el código del rehashing sin realizar ninguna llamada recursiva a `insertar`.

19.15. Experimente con la tabla hash que examina un carácter aleatorio en una cadena. ¿Es ésta una opción mejor que la adoptada en el texto?

19.16. Modifique la clase de las tablas hash de modo que la operación `esVacia` se ejecute en tiempo constante.

19.17. Modifique el algoritmo de eliminación de modo que el factor de carga siempre se encuentre por debajo de 1/8, y se ejecute el rehashing para mantener la tabla medio llena. Debe mantenerse un campo de datos adicional. ¿Por qué?

Prácticas de programación

19.18. Elija un diccionario de gran tamaño. Cree una tabla hash dos veces más grande que el diccionario. Aplique a cada palabra la función de localización descrita en el texto, almacénela y cuente el número de veces que se indexa cada posición. De este modo se obtendrá una distribución. Cierto porcentaje de las posiciones no se indexará nunca, algunas lo serán una vez, otras dos veces y así sucesivamente. Compare esta distribución con la que se obtiene con valores aleatorios, uniformemente distribuidos, utilizando la discusión sobre ellos de la Sección 9.3.

19.19. Realice simulaciones para comparar la eficiencia del hashing con los resultados teóricos. Declare una tabla hash con exploración. Inserte en ella 10.000 enteros generados aleatoriamente y calcule el número medio de pruebas realizadas. Éste es el coste medio de una búsqueda exitosa. Repita el test varias veces para obtener una media ajustada y realice las pruebas para las exploraciones lineal y cuadrática, con factores de carga 0,1, 0,2, ..., 0,9. Debe declararse la tabla de modo que nunca sea necesario el rehashing. Por ejemplo, en el test con factor de carga 0,4 debería declararse una tabla de tamaño aproximado 25.000 (ajústese para que sea un número primo).

19.20. Compare el tiempo necesario para las búsquedas exitosas y las inserciones en una tabla hash con hashing enlazado y factor de carga 1 y una tabla hash con exploración cuadrática y factor de carga 0,5. Realice las pruebas con enteros, cadenas y registros complicados en los que la clave sea una cadena.

19.21. Un programa en BASIC consiste en una serie de instrucciones, numeradas todas ellas en orden ascendente. El control se transmite empleando una instrucción *goto* o *gosub* y un número de instrucción. Escriba un programa que lea un programa correcto en BASIC y numere de nuevo sus instrucciones de modo que la primera empiece en el número F y cada instrucción tenga un número D mayor que la anterior. Debe asumirse un límite máximo de N instrucciones, pero los números de las mismas en la entrada podrían ser tan grandes como un entero de 32 bits. Su programa debe ejecutarse en tiempo lineal.

19.22. Diseñe un applet que ilustre el funcionamiento de la exploración cuadrática.

Bibliografía

A pesar de la aparente simplicidad del hashing, gran parte de su análisis es muy complicado y aún hay muchas cuestiones sin resolver. Existen diversos temas teóricos muy interesantes en relación con el intento de disminuir la probabilidad del mal funcionamiento del método de hashing.

Un artículo pionero sobre rehashing es [11]. En [6] puede encontrarse valiosa información acerca de este tema, incluyendo un análisis del hashing cerrado con exploración lineal. El hashing doble se analiza en [5] y [7]. Otro esquema de resolución de conflictos es el *hashing coalescente*, descrito en [12]. Un excelente resumen sobre el tema puede encontrarse en [8]; [9] realiza sugerencias y advertencias sobre la elección de las funciones de localización. En [4] pueden encontrarse precisos resultados analíticos y de simulación acerca de todos los métodos descritos en este capítulo. Yao [13] muestra que el hashing uniforme, en el que no existe ningún tipo de agrupación, es óptimo respecto al coste de una búsqueda exitosa.

Si se conocen con antelación las claves de entrada, entonces existe una función hash perfecta que evita cualquier colisión [1]. Otros esquemas de hashing algo más complicados, en los que el caso peor no sólo depende de la entrada sino también de ciertos números aleatorios elegidos, aparecen en [2] y [3]. Estos esquemas garantizan que en el caso peor se produce un número constante de colisiones

(aunque la elaboración de la función hash puede necesitar gran cantidad de tiempo en el improbable caso de que se utilicen números aleatorios inadecuados). Son muy útiles en la implementación en hardware de las tablas hash.

En [10] puede encontrarse otro método para resolver el Ejercicio 19.8.

1. J. L. Carter y M. N. Wegman, «Universal Classes of Hash Functions», *Journal of Computer and System Sciences* **18** (1979), 143-154.
2. M. Dietzfelbinger, A. R. Karlin, K. Melhorn, F. Meyer auf def Heide, H. Rohnert, y R. E. Tarjan, «Dynamic Perfect Hashing: Upper and Lower Bounds», *SIAM Journal on Computing* **23** (1994), 738-761.
3. R. J. Enbody y H. C. Du, «Dynamic Hashing Schemes», *Computing Surveys* **20** (1988), 85-113.
4. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2.ª ed., Addison-Wesley, Reading, Mass. (1991).
5. L. J. Guibas y E. Szemeredi, «The Analysis of Double Hashing», *Journal of Computer and System Sciences* **16** (1978), 226-274.
6. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2.ª ed., Addison-Wesley, Reading, Mass. (1997).
7. G. Lueker y M. Molodowitch, «More Analysis of Double Hashing», *Combinatorica* **13** (1993), 83-96.
8. W. D. Maurer y T. G. Lewis, «Hash Table Methods», *Computing Surveys* **7** (1975), 5-20.
9. B. J. McKenzie, R. Harries, y T. Bell, «Selecting a Hashing Algorithm», *Software-Practice and Experience* **20** (1990), 209-224.
10. R. Morris, «Scatter Storage Techniques», *Communications of the ACM* **11** (1968), 38-44.
11. W. W. Peterson, «Addressing for Random Access Storage», *IBM Journal of Research and Development* **1** (1957), 130-146.
12. J. S. Vitter, «Implementations for Coalesced Hashing», *Information Processing Letters* **11** (1980), 84-86.
13. A. C. Yao, «Uniform Hashing Is Optimal», *Journal of the ACM* **32** (1985), 687-693.

CAPÍTULO

20 Una cola de prioridad: el montículo binario

La cola de prioridad es una estructura de datos fundamental que sólo permite el acceso al elemento mínimo. En este capítulo se estudia una implementación de la cola de prioridad, el elegante *montículo binario*. El montículo binario soporta la inserción de nuevos elementos y la eliminación del elemento mínimo en tiempo logarítmico en el caso peor. Solamente usa un vector y es muy simple de implementar.

En este capítulo estudiaremos:

- Las propiedades básicas de los montículos binarios.
- Cómo se pueden codificar las operaciones `insertar` y `eliminarMin` para que se ejecuten en tiempo logarítmico.
- Un algoritmo lineal de construcción de montículos.
- Una implementación en Java.
- Un algoritmo de ordenación fácilmente implementable: el *método del montículo* (*heapsort*), que se ejecuta en un tiempo $O(N \log N)$ sin requerir ninguna memoria adicional.
- Cómo se pueden utilizar los montículos para implementar la ordenación externa.

20.1 Ideas básicas

Como ya se indicó en la Sección 6.8, la cola de prioridad soporta acceso y eliminación del elemento mínimo, usando, respectivamente, `buscarMin` y `eliminarMin`. Podríamos usar una lista enlazada, llevando a cabo las inserciones por delante de la lista en tiempo constante, pero entonces encontrar y/o eliminar el elemento mínimo requeriría un recorrido lineal de la lista. Alternativamente, podríamos obligar a que la lista estuviera siempre ordenada, lo cual haría que el acceso y la eliminación del elemento mínimo fueran más baratos, pero entonces las inserciones serían lineales.

Otra forma de implementar las colas de prioridad consiste en usar un árbol binario de búsqueda, en el que las dos operaciones mencionadas requieren, en media, un tiempo $O(N \log N)$. Sin embargo, ésta es habitualmente una mala elección,

pues normalmente la entrada no es lo suficientemente aleatoria. Podríamos usar un árbol binario de búsqueda equilibrado, pero las estructuras estudiadas en el Capítulo 18 son tediosas de implementar y en la práctica resultan poco eficientes. (Sin embargo, en el Capítulo 21 se estudia una estructura de datos, el árbol de ensanchamiento, que ha demostrado ser una buena alternativa en algunas situaciones.)

Puesto que la cola de prioridad soporta solamente algunas de las operaciones de los árboles de búsqueda, no debería resultar más costosa de implementar que éstos. Por otra parte, la cola de prioridad es más potente que una simple cola, porque podemos implementar una cola usando una cola de prioridad de la siguiente manera: insertamos cada elemento junto con una indicación del momento de inserción, con lo que una operación `eliminarMin` en base al momento de inserción, implementa en realidad una operación `quitarPrimero`. En consecuencia, es razonable esperar que la implementación tenga propiedades intermedias entre las de las colas y las de los árboles de búsqueda. Esto último se consigue con el montículo binario, el cual:

- se puede implementar usando un simple vector (al igual que la cola),
- soporta `insertar` y `eliminarMin` en tiempo $O(\log N)$ en el caso peor (un término medio entre el rendimiento de la cola y el del árbol binario de búsqueda), y
- soporta `insertar` en tiempo constante en el caso medio y `buscarMin` en tiempo constante en el caso peor (como la cola).

Al igual que los árboles de búsqueda equilibrados del Capítulo 18, el montículo binario tiene dos propiedades, una relacionada con su estructura y otra relacionada con la ordenación. Y al igual que los árboles de búsqueda equilibrados, una operación sobre un montículo binario puede destruir alguna de estas propiedades, por lo que la aplicación de una operación sobre un montículo binario no debe terminar hasta que ambas propiedades se restablezcan, lo cual, afortunadamente, resulta fácil de conseguir. (En este capítulo, el término *montículo* se refiere siempre al montículo binario.)

20.1.1 Propiedad estructural

La única estructura que proporciona cotas de tiempo logarítmicas es el árbol, por lo que parece natural que el montículo organice sus datos como tal. Como queremos que la cota logarítmica se mantenga en el caso peor, el árbol debería estar equilibrado.

Un *árbol binario completo* es un árbol completamente lleno, con la excepción del nivel inferior, que debe llenarse de izquierda a derecha. La característica distintiva del árbol binario completo es que no faltan nodos en el árbol. En la Figura 20.1 se muestra un ejemplo de árbol binario completo con diez elementos. Si el nodo J fuera hijo derecho de E, el árbol ya no sería completo, pues faltaría un nodo.

Los árboles completos tienen algunas propiedades que son muy útiles. En primer lugar, su altura (longitud del camino más largo) es a lo sumo $\lfloor \log N \rfloor$. Esto es así, porque un árbol completo de altura H tiene entre 2^H y $2^{H+1} - 1$ nodos. Esto significa que podemos esperar un comportamiento logarítmico en el caso peor, si restringimos los cambios en la estructura a los caminos de la raíz a una hoja.

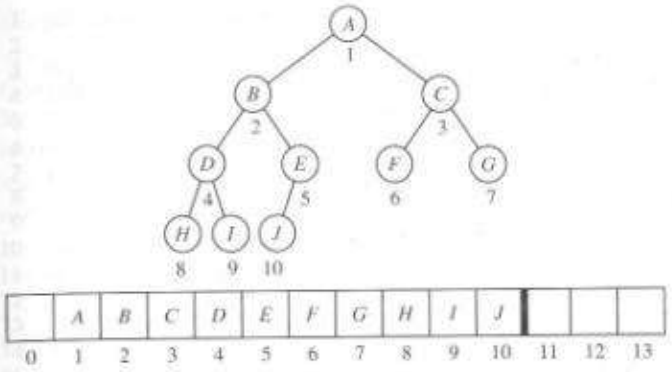


Figura 20.1 Árbol binario completo y su representación mediante un vector.

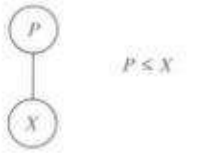
En segundo lugar, pero no por ello menos importante, en un árbol binario completo no se necesitan referencias a los hijos `izquierdo` y `derecho`, ya que, como se muestra en la Figura 20.1, podemos representarlo sin ambigüedad almacenando su recorrido por niveles en un vector. Colocamos la raíz en la posición 1 (la posición 0 se deja libre por una razón que se explicará más adelante). Necesitamos también mantener un entero que nos dice cuántos nodos hay actualmente en el árbol. Entonces, dado un elemento situado en la posición i del vector, sabemos que su hijo izquierdo se encuentra en la posición $2i$, aunque si dicha cantidad sobrepasa el número de nodos en el árbol, entonces sabemos que no existe tal hijo izquierdo. Análogamente, el hijo derecho se encuentra inmediatamente a continuación del hijo izquierdo, es decir, en la posición $2i + 1$, de nuevo comprobándose su existencia mediante una comparación con el número de nodos del árbol. Finalmente, su padre se encuentra en la posición $\lfloor i/2 \rfloor$. Observemos que todos los nodos, excepto la raíz, tienen un padre. Si la raíz tuviera un padre, el cálculo lo situaría en la posición 0. Por eso reservamos dicha posición para colocar un elemento falso que sirva como padre de la raíz, lo que simplificará algunas operaciones.

El uso de un vector para almacenar un árbol recibe el nombre de *representación implícita*. Como resultado de usar esta implementación, no solamente no son necesarias las referencias a los hijos, sino que además las operaciones de recorrido de los árboles son muy simples y muy probablemente lo bastante rápidas en la mayoría de los computadores. El montículo consistirá en un vector de objetos y un entero que representa el tamaño actual del montículo.

En este capítulo, para hacer los algoritmos más sencillos de visualizar, los montículos se dibujan como árboles, aunque la implementación de estos árboles utilizará un vector. No usamos la representación implícita para todos los árboles de búsqueda, ya que esto supondría algunos problemas adicionales, que se esbozan en el Ejercicio 20.8.

20.1.2 Propiedad de ordenación de los montículos

La propiedad que permite realizar rápidamente operaciones sobre los montículos es la *propiedad de ordenación de los montículos*. Puesto que queremos encontrar rápidamente el elemento mínimo, tendría sentido que el elemento más pequeño se



$P \leq X$

Figura 20.2 Propiedad de ordenación de los montículos.

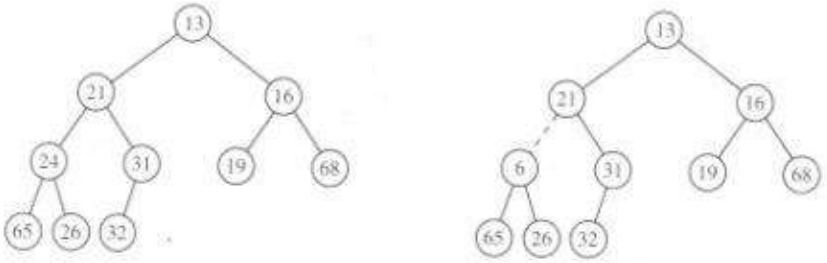


Figura 20.3 Dos árboles completos (sólo el de la izquierda es un montículo).

encontrara situado en la raíz. Si consideramos que cualquier subárbol también debería ser un montículo (recursivamente), entonces cada nodo debe ser menor que todos sus descendientes. Aplicando este razonamiento obtenemos la propiedad de ordenación de los montículos.

PROPIEDAD DE ORDENACIÓN DE UN MONTÍCULO

En un montículo, para cada nodo X con padre P, se cumple que el dato en P es menor o igual que el dato en X.

En la Figura 20.2 se ilustra la propiedad de ordenación de los montículos. En la Figura 20.3, el árbol de la izquierda es un montículo, pero el de la derecha no lo es (la línea punteada muestra el lugar en el que se viola la propiedad). Observe que la raíz no tiene padre. En la representación implícita, podemos colocar el valor $-\infty$ en la posición 0 para eliminar este caso especial cuando implementamos los montículos. Por la propiedad de ordenación, vemos que el menor elemento siempre se puede encontrar en la raíz, de modo que `buscarMin` se puede ejecutar en tiempo constante. Los *montículos maximales* soportan acceso al máximo en lugar de al mínimo. Para implementarlos basta con hacer unos mínimos cambios.

20.1.3 Operaciones permitidas

Ahora que hemos fijado la representación, podemos empezar a escribir código. Ya sabemos que nuestro montículo soporta las operaciones básicas `insertar`, `buscarMin` y `eliminarMin`, y también las habituales `esVacio` y `vaciar`. Añadiremos además algunas operaciones más. La Figura 20.4 muestra la sección pública de la clase e ilustra algunos de estos métodos adicionales. La Figura 20.5 contiene la parte privada de la clase.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase MonticuloBinario
7  //
8  // CONSTRUCCIÓN: con un centinela con un valor menos infinito
9  //
10 // ******************OPERACIONES PÚBLICAS*********************
11 // void insertar( x )        --> Inserta x
12 // Comparable eliminarMin( ) --> Elimina y devuelve el menor
13 // Comparable buscarMin( )   --> Devuelve el menor elemento
14 // boolean esVacia( )        --> Devuelve true si está vacío
15 // void vaciar( )            --> Elimina todos los elementos
16 // void introducir( x )      --> Inserta x (perezosamente)
17 // ************************ERRORES**************************
18 // buscarMin y eliminarMin lanzan DesbordamientoInferior si vacío
19
20 /**
21  * Implementa un montículo binario.
22  * Permite inserción perezosa y proporciona un método lineal de
23  * construcción de montículos.
24  * Observe que todas las comparaciones están basadas en compara.
25  */
26 public class MonticuloBinario implements ColaPrioridad
27 {
28     public MonticuloBinario( Comparable infNeg )
29       { /* Figura 20.7 */ }
30     public void insertar( Comparable x )
31       { /* Figura 20.12 */ }
32     public void introducir( Comparable x )
33       { /* Figura 20.12 */ }
34     public Comparable buscarMin( ) throws DesbordamientoInferior
35       { /* Figura 20.6 */ }
36     public Comparable eliminarMin( ) throws DesbordamientoInferior
37       { /* Figura 20.16 */ }
38     public boolean esVacia( )
39       { return tamActual == 0; }
40     public void vaciar( )
41       { tamActual = 0; }
```

Figura 20.4 Esqueleto de la clase `MonticuloBinario` (parte 1: sección pública).

```
42     private int tamActual;     // Número de elementos del montículo
43     private boolean ordenado; // True si cumple propiedad ordenación
44     private Comparable [ ] vector; // El vector del montículo
45     private static final int CAPACIDAD = 11;
46
47     private void obtenerVector( int nuevoTamMax )
48       { /* Figura 20.8 */ }
49     private void comprobarTam( )
50       { /* Figura 20.11 */ }
51     private void hundir( int hueco )
52       { /* Figura 20.17 */ }
53     private void arreglarMonticulo( )
54       { /* Figura 20.19 */ }
55 }
```

Figura 20.5 Esqueleto de la clase `MonticuloBinario` (parte 2: sección privada).

Comenzamos examinando los métodos públicos. El constructor se declara en la línea 28, mientras que el método `insertar` aparece en la 30. Añade un nuevo elemento `x` al montículo, realizando las operaciones necesarias para mantener la propiedad de ordenación del montículo. En la línea 32 se añade un método nuevo, `introducir`, el cual añade un nuevo elemento `x` al montículo, pero no garantiza que se mantenga la propiedad de ordenación del montículo. ¿Para qué querríamos `introducir` un elemento en el montículo? La respuesta es que `introducir` es un método mucho más sencillo de implementar que `insertar`. Por supuesto, tan pronto como realizamos una operación `introducir`, ya no tenemos un montículo, por lo que no debemos esperar que funcionen correctamente las operaciones `buscarMin` o `eliminarMin`. Por ejemplo, si queremos `introducir` un nuevo mínimo, no tenemos garantías de que se situará en la raíz (de hecho es de esperar que no lo haga).

En consecuencia, usar `introducir` parece una idea tonta. Pero no lo es, porque hay muchas aplicaciones en las que podemos tener que `introducir` muchos elementos antes de que se produzca la siguiente llamada a `eliminarMin`. En tal caso, no necesitamos mantener el montículo ordenado hasta que se produzca la llamada a `eliminarMin`. La operación `arreglarMonticulo`, declarada en la línea 53, reestablece el orden en el montículo en tiempo lineal, sin importar lo desordenado que se encuentre. Luego, por ejemplo, si queremos colocar N elementos en el montículo antes de ejecutar la primera operación `eliminarMin`, es más eficiente ejecutar N operaciones `introducir` y una `arreglarMonticulo` que N operaciones `insertar`. Sin embargo, no usaremos `introducir` en sustitución de sólo unos pocos `insertar`. El método `arreglarMonticulo` es privado, y su uso es transparente.

El resto de las operaciones son como esperamos. En la línea 34 se declara `buscarMin`, que devuelve el elemento mínimo. El método `eliminarMin`, en la línea 36, devuelve y elimina el elemento mínimo. También se proporcionan, en las líneas 38 a 41, los habituales `esVacio` y `vaciar`.

Una vez procesamos una operación `introducir`, el usuario no puede realizar una operación `buscarMin` ni `eliminarMin` sin realizar antes una operación `arreglarMonticulo`. En lugar de confiar en que el usuario haga esta llamada, mantenemos un atributo `ordenado`, que toma el valor `false` cuando una aplicación de `introducir` produce una violación del orden en el montículo. La Figura 20.6 muestra que `buscarMin` comprueba si se satisface la propiedad de ordenación de montículo, y en caso de no ser así, produce automáticamente una llamada a `arreglarMonticulo` para restablecer la propiedad.

```
1  /**
2   * Busca el elemento más pequeño en la cola de prioridad.
3   * @return el elemento más pequeño.
4   * @exception DesbordamientoInferior si la cola está vacía.
5   */
6  public Comparable buscarMin( ) throws DesbordamientoInferior
7  {
8      if( esVacia( ) )
9          throw new DesbordamientoInferior ( "Montículo vacío" );
10     if( !ordenado )
11         arreglarMonticulo( );
12     return vector[ 1 ];
13 }
```

Figura 20.6 Rutina `buscarMin`.

```
1  /**
2   * Construye el montículo binario.
3   * @param infNeg un valor menor o igual que todos los demás.
4   */
5  public MonticuloBinario( Comparable infNeg )
6  {
7      tamActual = 0;
8      ordenado = true;
9      obtenerVector( CAPACIDAD );
10     vector[ 0 ] = infNeg;
11 }
```

Figura 20.7 Constructor para `MonticuloBinario`.

```
1  /**
2   * Método privado para reservar memoria para el vector.
3   * Incluye una posición extra para el centinela.
4   * @param nuevoTamMax la capacidad del montículo.
5   */
6  private void obtenerVector( int nuevoTamMax )
7  {
8      vector = new Comparable[ nuevoTamMax + 1 ];
9  }
```

Figura 20.8 Reserva de memoria para el vector (método llamado por el constructor y durante la duplicación del tamaño del vector).

El resto de atributos son los que ya sospechábamos: un vector al que se asigna memoria dinámicamente y un entero que almacena el tamaño actual del montículo. El constructor, mostrado en la Figura 20.7, inicializa los atributos, asignando `true` a `ordenado` y reservando después memoria para el vector mediante una llamada a `obtenerVector`, en la línea 9. Después, en la línea 10, se coloca el centinela en la posición 0. El usuario debe proporcionar al constructor `infNeg` como un parámetro. El cuerpo de `obtenerVector`, mostrado en la Figura 20.8, consta de una sola línea.

20.2 Implementación de las operaciones básicas

La propiedad de ordenación parece buena hasta el momento, puesto que permite un rápido acceso al elemento mínimo. Ahora debemos mostrar que podemos soportar eficientemente las operaciones `insertar` y `eliminarMin`, haciéndolo en un tiempo logarítmico. Es fácil, conceptual y prácticamente, realizar las dos operaciones requeridas. Todo el trabajo implicado se encuentra en mantener la propiedad de ordenación del montículo.

20.2.1 `insertar`

Para insertar un elemento X en el montículo, primero debemos añadir un nodo al árbol. La única opción es crear un hueco en la siguiente posición disponible; en caso contrario, el árbol no sería completo y violaríamos la propiedad estructural.

Si X se puede colocar en ese hueco sin violar la propiedad de ordenación del montículo, lo colocamos ahí y hemos terminado. Si no, desplazamos el elemento situado en el padre del nodo a dicho nodo, moviendo así el hueco hacia la raíz. Continuamos con este proceso hasta que se pueda colocar X en el hueco. La Figura 20.9 muestra que para insertar 14, creamos un hueco en la siguiente posición disponible. Como insertar 14 en el hueco, produciría una violación del orden, colocamos 31 en el hueco. En la Figura 20.10 se continúa con esta estrategia hasta encontrar la posición correcta para 14.

Llamamos a esta estrategia general *reflotamiento*, porque el nuevo elemento es reflotado hasta encontrar la posición adecuada. La rutina, `comprobarTam`, mostrada en la Figura 20.11, duplicará el tamaño del vector cuando ello sea necesario. La Figura 20.12 muestra las rutinas que añaden elementos al montículo. La rutina `introducir` es bien corta; simplemente añade el nuevo elemento `x` en la siguiente posición disponible. Si se produce una violación del orden, se le asigna `false` a `ordenado`. El método `insertar` implementa el reflotamiento usando un bucle. La instrucción de la línea 30 incrementa el tamaño actual y coloca el hueco en el nodo que acabamos de añadir. Ejecutamos el bucle de la línea 31 mientras el elemento del nodo padre sea mayor que `x`. La línea 32 mueve el elemento del padre al hueco tras lo cual la tercera componente del bucle `for` reflota el hueco hacia el padre. Cuando el bucle termina, la línea 33 coloca `x` en el hueco. El centinela situado en la posición O garantiza la terminación del bucle `for`.

El tiempo necesario para realizar la inserción podría ser $O(\log N)$ si el elemento a insertar fuera el nuevo mínimo, porque en tal caso se reflotaría hasta la raíz. En media, el reflotamiento termina pronto: se ha demostrado que en media se requieren 2,6 comparaciones para llevar a cabo una inserción, de modo que en media `insertar` mueve un elemento 1,6 niveles.

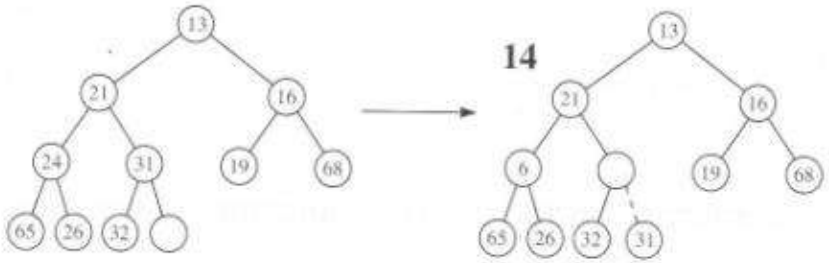


Figura 20.9 Intento de inserción de 14, creando el hueco y flotándolo hacia arriba.

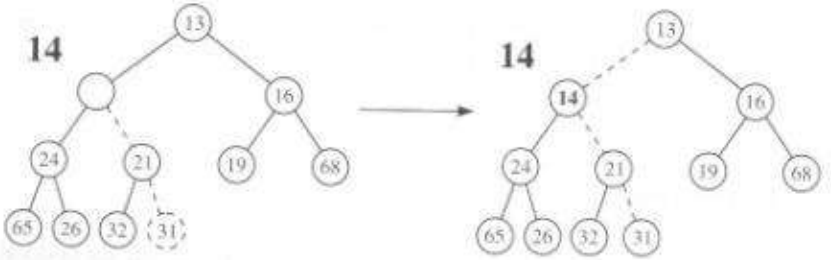


Figura 20.10 Los dos pasos que faltan para insertar 14 en el montículo de la figura anterior.

```
1  /**
2   * Método privado que duplica el vector cuando está lleno.
3   */
4  private void comprobarTam( )
5  {
6      if( tamActual == vector.length - 1 )
7      {
8          Comparable [ ] vectorAnt = vector;
9          obtenerVector( tamActual * 2 );
10         for( int i = 0; i < vectorAnt.length; i++ )
11             vector[ i ] = vectorAnt[ i ];
12     }
13 }
```

Figura 20.11 Método privado `comprobarTam` que dobla el tamaño del vector si es necesario.

```
1  /**
2   * Inserta en la cola de prioridad, sin mantener
3   * el orden del monticulo. Se permiten elementos duplicados.
4   * @param x el elemento a insertar.
5   */
6  public void introducir( Comparable x )
7  {
8      comprobarTam( );
9      vector[ ++tamActual ] = x;
10     if( x.menorQue( vector[ tamActual / 2 ] ) )
11         ordenado = false;
12 }
13
14 /**
15  * Inserta en la cola de prioridad, manteniendo el orden.
16  * Se permiten elementos duplicados.
17  * @param x el elemento a insertar.
18  */
19 public void insertar( Comparable x )
20 {
21     if( !ordenado )
22     {
23         introducir( x );
24         return;
25     }
26
27     comprobarTam( );
28
29         // Reflotamiento
30     int hueco = ++tamActual;
31     for( ; x.menorQue( vector[ hueco / 2 ] ); hueco /= 2 )
32         vector[ hueco ] = vector[ hueco / 2 ];
33     vector[ hueco ] = x;
34 }
```

Figura 20.12 Métodos `introducir` e `insertar`.

20.2.2 eliminarMin

La operación `eliminarMin` se realiza de forma similar a las inserciones. Como ya hemos visto, encontrar el mínimo es fácil; la parte difícil es eliminarlo. Cuando se elimina el mínimo, se crea un hueco en la raíz. Puesto que el montículo tiene ahora un elemento menos, la propiedad estructural nos dice que se debe eliminar el último nodo. La Figura 20.13 muestra la situación: el elemento mínimo es 13, la raíz tiene un hueco y el último elemento debe colocarse en algún lugar del montículo.

Si el último elemento se pudiera colocar en el hueco, ya habríamos terminado. Pero esto es imposible, a menos que el tamaño del árbol sea 2 o 3, porque los elementos de la parte inferior son más grandes que los del segundo nivel. Debemos hacer lo mismo que en la inserción: poner algún elemento en el hueco y después mover el hueco. La única diferencia está en que en `eliminarMin`, nos movemos hacia abajo. Para hacer esto, buscamos el hijo del hueco con el elemento más pequeño. Si ese elemento es menor que el que estamos intentando colocar, movemos el hijo al hueco, empujando así el hueco un nivel hacia abajo. En la Figura 20.14, colocamos el hijo más pequeño (14) en el hueco, moviendo el hueco un nivel hacia abajo. Repetimos esto de nuevo, colocando 19 en el hueco y creando un nuevo hueco un nivel más abajo. Entonces colocamos 26 en el hueco y creamos un nuevo hueco en el último nivel. Finalmente, podemos colocar 31 en el hueco, como se muestra en la Figura 20.15. Es fácil ver que ésta es una operación logarítmica en el caso peor. Llamamos a este proceso *hundimiento*. No es sorprendente que el hundimiento raramente termine antes de bajar uno o dos niveles, por lo que la operación es también logarítmica en el caso medio.

La Figura 20.16 muestra el método `eliminarMin`. Las comprobaciones de montículo vacío y de ordenación se realizan automáticamente a través de la llamada

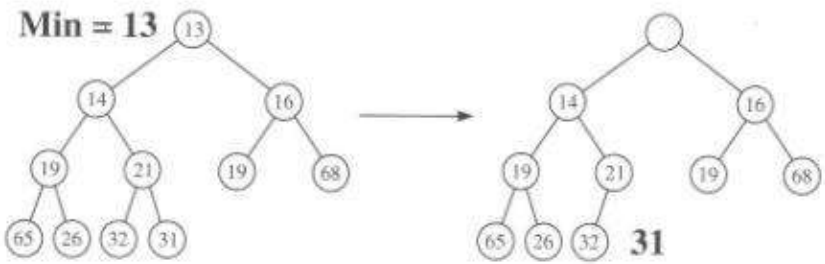


Figura 20.13 Creación del hueco en la raíz cuando se elimina el elemento mínimo.

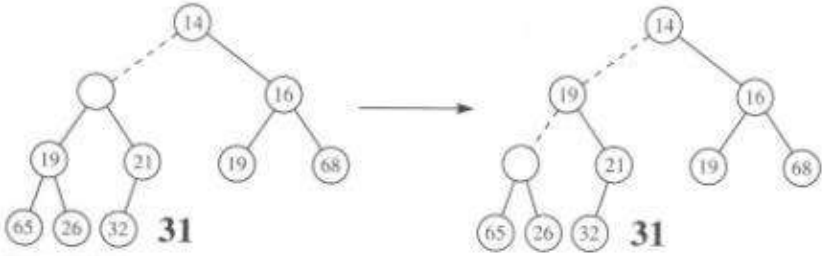


Figura 20.14 Los dos pasos siguientes de `eliminarMin`.

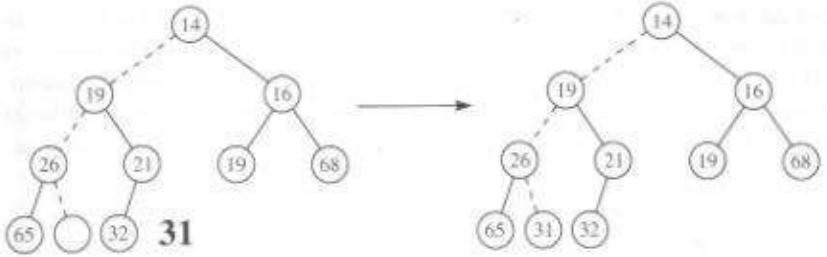


Figura 20.15 Últimos dos pasos de `eliminarMin`.

```
1  /**
2   * Elimina el elemento mínimo de la cola de prioridad.
3   * @exception DesbordamientoInferior si la cola está vacía.
4   */
5  public Comparable eliminarMin( ) throws DesbordamientoInferior
6  {
7      Comparable elemMin = buscarMin( );
8      vector[ 1 ] = vector[ tamActual-- ];
9      hundir( 1 );
10
11     return elemMin;
12 }
```

Figura 20.16 Método `eliminarMin`.

a `buscarMin` en la línea 7. El verdadero trabajo lo lleva a cabo `hundir`, que se muestra en la Figura 20.17. El código aquí mostrado es similar en espíritu al del reflotamiento en el método `insertar`. Debido a que hay dos hijos en lugar de un padre, el código se complica un poco más. El método `hundir` toma un único argumento que nos dice dónde se va a colocar el hueco. Entonces el elemento situado en el hueco se elimina y comienza el hundimiento.

```
1  /**
2   * Método interno para hundir en el montículo.
3   * @param hueco el índice donde comienza el hundimiento.
4   */
5  private void hundir( int hueco )
6  {
7      int hijo;
8      Comparable tmp = vector[ hueco ];
9
10     for( ; hueco * 2 <= tamActual; hueco = hijo )
11     {
12         hijo = hueco * 2;
13         if( hijo != tamActual &&
14                 menorQue( vector[ hijo + 1 ],
15                           vector[ hijo ] ) )
16             hijo++;
17         if( vector[ hijo ].menorQue( tmp ) )
18             vector[ hueco ] = vector[ hijo ];
19         else
20             break;
21     }
22     vector[ hueco ] = tmp;
23 }
```

Figura 20.17 Método `hundir` usado en `eliminarMin` y `arreglarMonticulo`.

En el caso de `eliminarMin`, `hueco` estará en la posición 1. El bucle `for` de la línea 10 termina cuando no hay hijo izquierdo. La tercera componente mueve el hueco al hijo. En las líneas 13 a 15 se busca el hijo con el elemento más pequeño. Nótese que hemos de ser cuidadosos porque el último nodo de un montículo de tamaño par es hijo único, por lo que no siempre podemos asumir que hay dos hijos. En base a ello realizamos el test de la línea 13.

20.3 `arreglarMonticulo`: construcción en tiempo lineal del montículo

La operación `arreglarMonticulo` toma un árbol completo sin orden y restablece el mismo. Queremos que sea una operación lineal, pues en tal caso se podrían hacer N inserciones en tiempo $O(N \log N)$. Esperamos poder alcanzar un tiempo $O(N)$ porque N inserciones sucesivas requieren un tiempo total $O(N)$ en media, basándonos en el resultado del final de la Sección 20.2.1. N inserciones sucesivas llevan a cabo más trabajo de lo que necesitamos, ya que tras cada inserción se mantiene el orden, mientras que ahora esto es solamente necesario en un instante concreto.

La solución abstracta más sencilla se obtiene viendo el montículo como una estructura definida recursivamente, como se muestra en la Figura 20.18. Comenzaremos por llamar recursivamente a `arreglarMonticulo` sobre los submontículos izquierdo y derecho. Tras ello, tenemos garantizado que la propiedad de ordenación del montículo se mantiene en todas partes excepto, quizás, en la raíz. Podemos establecer el orden completamente llamando a `hundir` con la raíz como argumento. La rutina recursiva trabaja garantizando que cuando aplicamos `hundir(i)`, todos los descendientes del nodo `i` han sido procesados por la correspondiente llamada recursiva a `hundir`. Sin embargo, la recursión no es necesaria, como se hace evidente a partir del comentario siguiente: si llamamos a `hundir` sobre cada nodo en sentido inverso al recorrido por niveles, cuando realicemos la llamada con `i` se habrán procesado todos los descendientes del nodo `i` con una llamada a `hundir`. Esto nos conduce a un algoritmo increíblemente simple para `arreglarMonticulo`, que se muestra en la Figura 20.19. Observemos que no hace falta ejecutar `hundir` sobre las hojas, por lo que comenzamos por el nodo de mayor índice que no sea una hoja.

El primer árbol de la Figura 20.20 es el árbol sin ordenar. Los otros siete árboles en las Figuras 20.20 a 20.23 muestran el resultado de cada una de las siete lla-

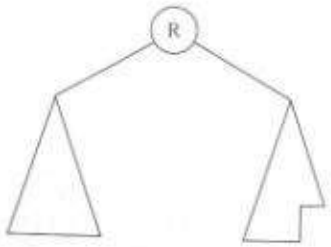


Figura 20.18 Visión recursiva del montículo.

```
1  /**
2   * Restablece la propiedad de ordenación del montículo tras una
3   * serie de operaciones introducir. Se ejecuta en tiempo lineal.
4   */
5  private void arreglarMonticulo( )
6  {
7      for( int i = tamActual / 2; i > 0; i-- )
8          hundir( i );
9      ordenado = true;
10 }
```

Figura 20.19 Implementación del método lineal `arreglarMonticulo`.

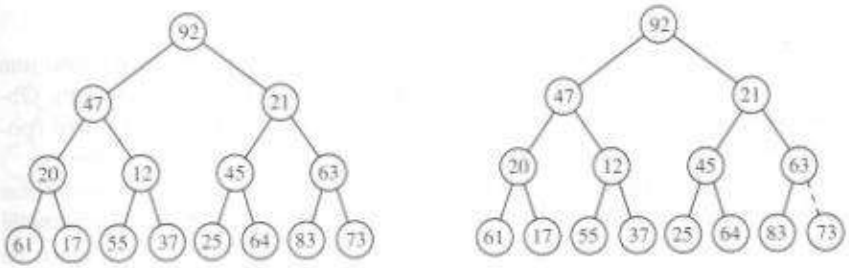


Figura 20.20 Montículo inicial (a la izquierda); y tras la llamada a `hundir(7)` (a la derecha).

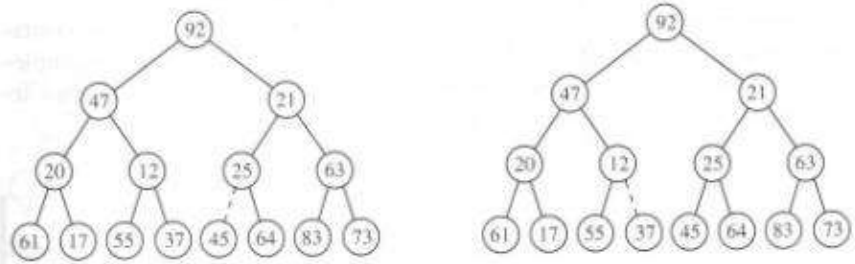


Figura 20.21 Montículo tras la llamada a `hundir(6)` (a la izquierda); y tras la llamada a `hundir(5)` (a la derecha).

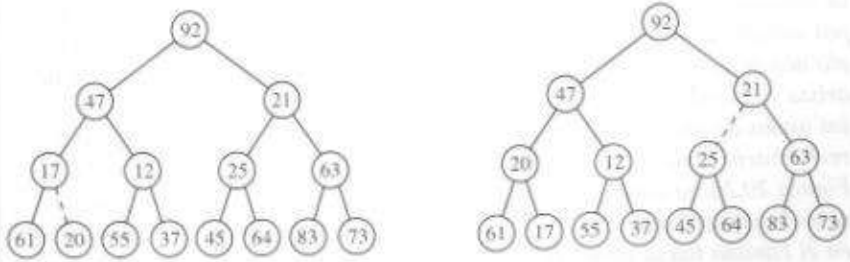


Figura 20.22 Montículo tras la llamada a `hundir(4)` (a la izquierda); y tras la llamada a `hundir(3)` (a la derecha).

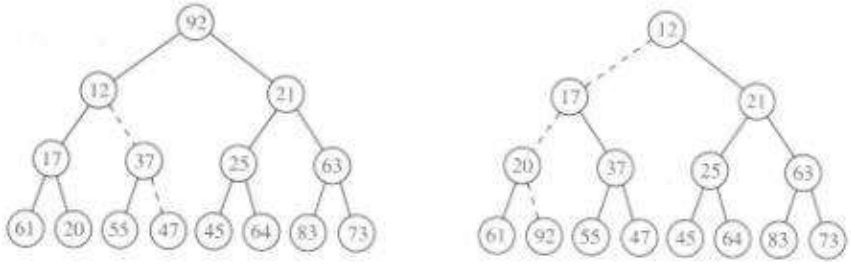


Figura 20.23 Montículo tras la llamada a `hundir(2)` (a la izquierda); y tras la llamada a `hundir(1)` (a la derecha), terminando así la ejecución de `arreglarMonticulo`.

madas a `hundir`. Cada línea punteada se corresponde con dos comparaciones: una para buscar el hijo menor y la otra para comparar el hijo menor con el nodo. Observemos que hay solamente diez líneas punteadas en el algoritmo completo (podría haber habido sólo una más), lo que se corresponde con 20 comparaciones.

Para acotar el tiempo de ejecución de `arreglarMonticulo`, debemos acotar el número de líneas punteadas, lo cual se puede conseguir calculando la suma de las alturas de todos los nodos del montículo, que es igual al máximo número de líneas punteadas. Es de esperar que este número sea pequeño porque la mitad de los nodos son hojas, las cuales tienen altura 0, y una cuarta parte tiene altura 1. Luego, solamente una cuarta parte de los nodos (los que no están incluidos en los dos casos anteriores) pueden contribuir en más de 1 unidad de altura. En particular, hay solamente un nodo que contribuye con la máxima altura, $\lfloor \log N \rfloor$.

Para obtener una cota de tiempo lineal para `arreglarMonticulo`, necesitamos demostrar que la suma de las alturas de los nodos de un árbol binario completo es $O(N)$. Esto se demuestra en el Teorema 20.1. Demostramos la cota para árboles perfectos usando un argumento de marcado.

Dado un árbol binario perfecto de altura H que contiene $N = 2^{H+1} - 1$ nodos, se cumple que la suma de alturas de sus nodos es $N - H - 1$.

Usamos un argumento de marcado del árbol. (También se podría hacer un cálculo más directo usando la fuerza bruta, como en el Ejercicio 20.10.) Para cada nodo en el árbol con altura h, marcamos h aristas del árbol en la siguiente forma: bajamos por la arista izquierda y después solamente por aristas derechas, marcando cada arista por la que pasamos. Un ejemplo nos lo da un árbol perfecto de altura 4. Los nodos de altura 1 tienen su arista izquierda marcada, como se muestra en la Figura 20.24. Después, los nodos de altura 2 tienen marcada una arista izquierda y luego una derecha hacia abajo en el árbol, lo cual se muestra en la Figura 20.25. En la Figura 20.26, se marcan tres aristas para cada nodo de altura 3: la primera es la arista izquierda que sale del nodo, y después las aristas derechas en el camino hacia abajo en el árbol. Finalmente, en la Figura 20.27 vemos que se han marcado 4 aristas: la izquierda que sale del nodo y después las tres derechas siguientes en el camino hacia abajo en el árbol.

Observe que una arista nunca se marca dos veces y que se marcan todas las aristas excepto las del camino de la derecha. Puesto que hay $N - 1$ aristas (a cada nodo excepto a la raíz le llega una arista) y H aristas en el camino de la derecha, el número de aristas marcadas es $N - 1 - H$, lo que concluye la demostración del teorema.

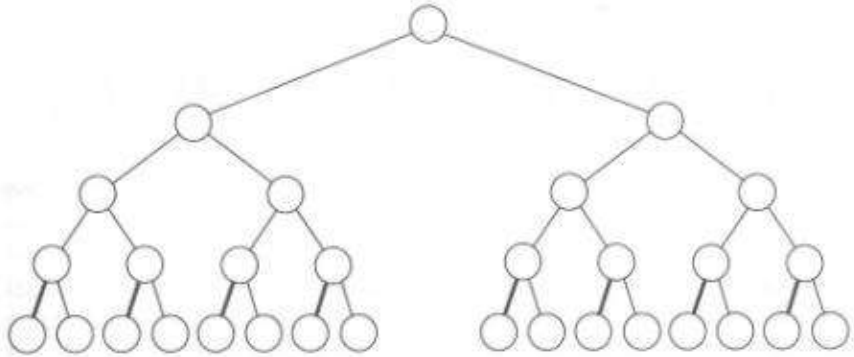


Figura 20.24 Marcado de las aristas izquierdas para los nodos de altura 1.

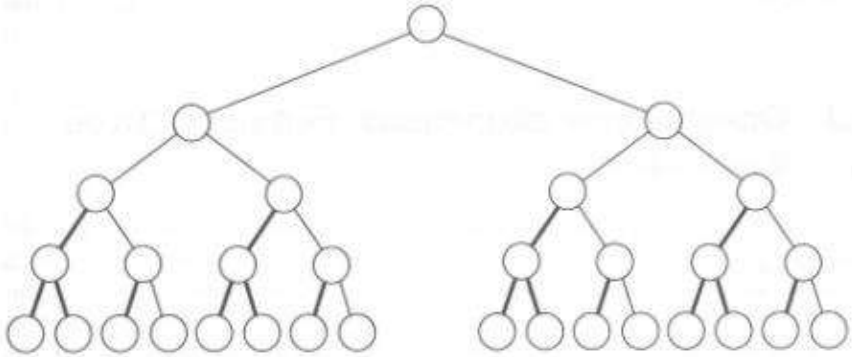


Figura 20.25 Marcado de la primera arista izquierda y de la derecha situada a continuación, para los nodos de altura 2.

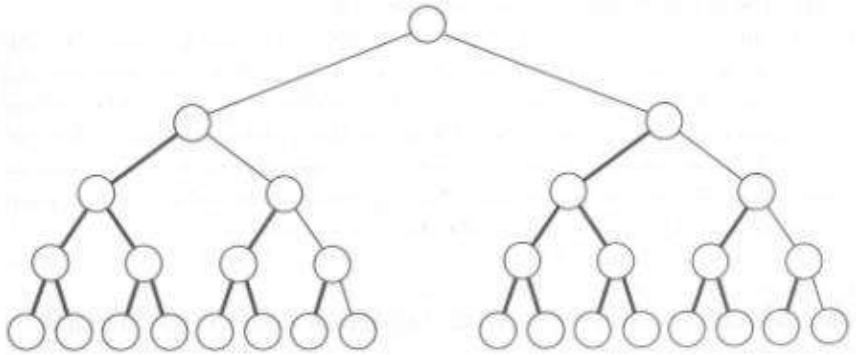


Figura 20.26 Marcado de la primera arista izquierda y de las dos aristas derechas situadas a continuación, para los nodos de altura 3.

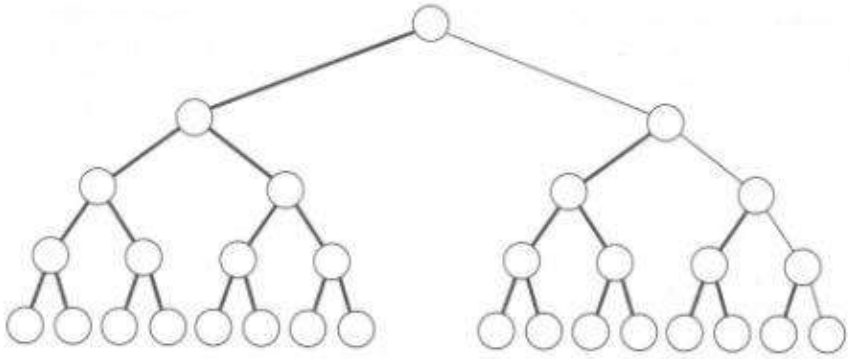


Figura 20.27 Marcado de la primera arista izquierda y de las tres aristas derechas situadas a continuación, para los nodos de altura 4.

Un árbol binario completo no es un árbol perfecto, pero el resultado que hemos obtenido es una cota superior de la suma de las alturas de los nodos de un árbol binario completo. Puesto que un árbol completo tiene entre 2^H y $2^{H+1} - 1$ nodos, este teorema implica que la suma es $O(N)$. Un argumento más cuidadoso establece que la suma de las alturas es $N - v(N)$, donde $v(N)$ es el número de unos en la representación binaria de N. Se deja como Ejercicio 20.12 la demostración de este resultado.

20.4 Operaciones avanzadas: `reducirClave` y `mezclar`

En el Capítulo 22 se estudian unas colas de prioridad que soportan dos operaciones adicionales. La operación `reducirClave`reduce el valor de un elemento en la cola de prioridad. Se supone conocida la posición del elemento. En un montículo binario, esto se puede conseguir fácilmente reflotando hasta restablecer el orden en el montículo. Sin embargo, debemos tener cuidado, ya que hemos supuesto que la posición de cada elemento se almacena de forma separada. Todos los elementos implicados en el reflotamiento ven sus posiciones modificadas. El método `reducirClave`es útil en la implementación de algoritmos sobre grafos (por ejemplo el algoritmo de Dijkstra, visto en la Sección 14.3).

El algoritmo `mezclar` combina dos colas de prioridad. Debido a que el montículo está basado en un vector, lo mejor que podemos esperar conseguir con una mezcla es copiar los elementos del montículo más pequeño en el grande y realizar después algunos arreglos. Aun así, esto llevará un tiempo al menos lineal por operación. Si usáramos árboles generales con nodos conectados mediante referencias a los hijos, podríamos reducir la cota a coste logarítmico por operación. La operación de mezcla se usa en el diseño avanzado de algoritmos.

20.5 Ordenación interna: método del montículo

Se puede usar una cola de prioridad para ordenar N elementos de la siguiente forma:

1. Insertar cada elemento en un montículo binario.
2. Extraer cada elemento llamando a `eliminarMin` N veces. El resultado está ordenado.

Usando las observaciones de la sección anterior, podemos implementar este proceso de forma más eficiente:

1. `introducir` cada elemento en un montículo binario.
2. Aplicar `arreglarMonticulo`.
3. Llamar a `eliminarMin` N veces; los elementos saldrán del montículo en orden.

El paso 1 requiere un tiempo total lineal y el paso 2 también requiere un tiempo lineal. En el paso 3, cada llamada a `eliminarMin` exige un tiempo logarítmico, con lo que N llamadas tardan un tiempo $O(N \log N)$. En consecuencia, tenemos un algoritmo de ordenación $O(N \log N)$ en el caso peor, lo que es al menos igual de bueno que un algoritmo basado en comparaciones, como se vió en la Sección 8.8. Un problema del algoritmo tal y como lo hemos presentado es que la ordenación de un vector requiere la construcción de una estructura de datos montículo, que ya conlleva en sí misma la sobrecarga de implementarse como un vector. Sería preferible emular el montículo, en lugar de usar toda la parafernalia de la clase de los montículos. Supondremos para el resto de esta discusión que vamos a hacer esto.

Pero, incluso aunque no usemos directamente la clase de los montículos, tenemos el problema de tener que utilizar un segundo vector, ya que tenemos que almacenar en ese segundo vector el orden en el que los elementos van saliendo del equivalente al montículo, para después copiarlo en el original. Se ha doblado la cantidad de memoria utilizada, lo cual puede ser crítico en algunas aplicaciones. Observemos que el tiempo extra utilizado en copiar el segundo vector en el primero, es solamente $O(N)$, por lo que a diferencia de lo que sucedía en la ordenación por mezcla, el vector adicional no afecta de forma significativa al tiempo de ejecución. El problema es el espacio.

Una forma inteligente de evitar el uso de un segundo vector hace uso del hecho de que después de cada operación `eliminarMin`, el montículo tiene un elemento menos. En consecuencia, la última celda del vector está disponible para almacenar el elemento eliminado. Por ejemplo, supongamos que tenemos un montículo de 6 elementos. La primera operación `eliminarMin` produce A_1. Ahora el montículo tiene 5 elementos, por lo que podemos colocar A_1 en la posición 6. La siguiente operación `eliminarMin` produce A_2. Puesto que el montículo tiene ahora solamente 4 elementos, podemos colocar A_2 en la posición 5.

Bajo esta estrategia, el vector contendrá, después de la última operación `eliminarMin`, los elementos en orden *decreciente*. Si queremos el vector en orden *creciente*, podemos cambiar la propiedad de ordenación de forma que cada elemento padre sea mayor que sus hijos. En tal caso tendríamos un montículo maximal. Por ejemplo, supongamos que queremos ordenar la secuencia 59, 36, 58, 21, 41, 97, 31, 16, 26, 53. Después de haber introducido los elementos en el montículo y de haber aplicado `arreglarMonticulo`, obtenemos lo reflejado en la Figura 20.28. (Observemos que en este caso no hay centinela: asumimos que los datos comienzan en la posición 0, como es habitual en otros algoritmos de ordenación del Capítulo 8.)

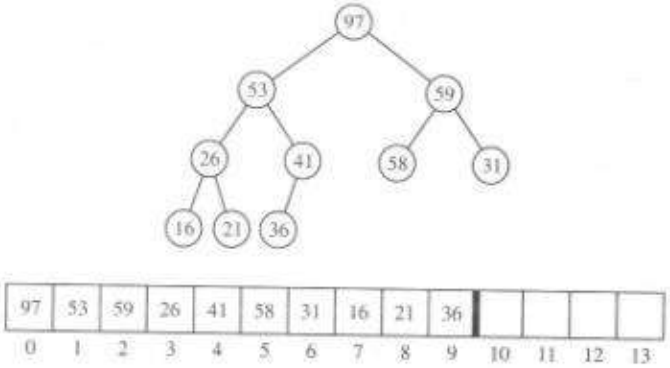


Figura 20.28 Montículos maximal tras la aplicación de `arreglarMonticulo`.

La Figura 20.29 muestra el montículo resultante tras la primera aplicación de `eliminarMax`. El último elemento del montículo es 21; 97 se ha colocado en una parte del vector que técnicamente ya no forma parte del montículo.

La Figura 20.30 muestra que tras la segunda aplicación de `eliminarMax`, 16 se convierte en el último elemento. Ahora quedan solamente ocho elementos en el montículo. El máximo elemento, 59, que ha sido eliminado, se coloca en una posición del vector que ya no pertenece al montículo. Después de ejecutar otras siete operaciones `eliminarMax` más, el montículo contendrá sólo un elemento, pero los elementos que aparecen en el vector se encuentran ordenados de forma creciente.

La implementación del método del montículo (`ordenaViaMonticulo`) es simple porque las operaciones básicas se corresponden con las operaciones sobre los montículos. Hay tres pequeñas diferencias. En primer lugar, puesto que estamos usando un montículo maximal, necesitamos invertir todas las comparaciones, pasando de > a <. En segundo lugar, ya no podemos asumir que hay un centinela en la posición 0, porque todos los demás algoritmos de ordenación almacenan datos en la posición 0 y `ordenViaMonticulo` no debería ser diferente en este sentido. Aunque ya no se necesita el centinela (porque no hay operaciones de reflotamiento), su ausencia afecta a los cálculos de las posiciones de los hijos y del padre. En

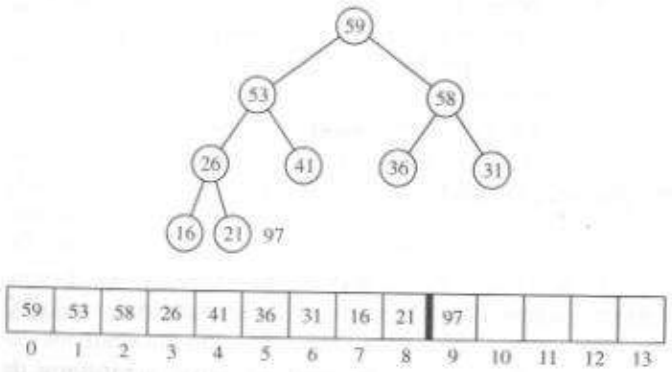


Figura 20.29 Montículo después de la primera aplicación de `eliminarMax`.

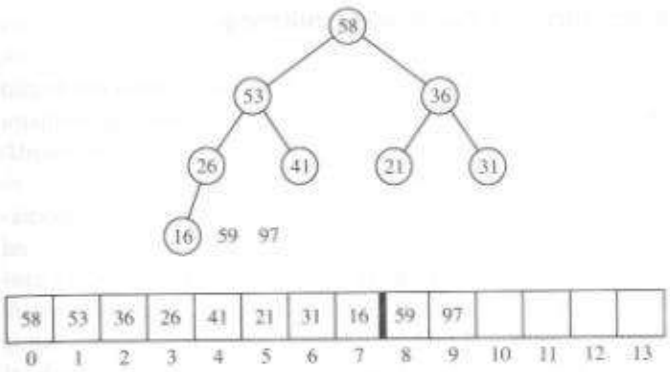


Figura 20.30 Montículo después de la segunda aplicación de `eliminarMax`.

```
1  /**
2   * Método estándar de ordenación mediante un montículo
3   * @param v un vector de objetos de la clase Comparable
4   */
5  public static void ordenaViaMonticulo( Comparable [ ] v )
6  {
7      for( int i = v.length / 2; i >= 0; i-- )  // Construcción
8          hundir2( v, i, v.length );
9      for( int i = v.length - 1; i > 0; i-- )   // eliminarMax
10     {
11         intercambiarReferencias( v, 0, i );
12         hundir2( v, 0, i );
13     }
14 }
```

Figura 20.31 Rutina `ordenaViaMonticulo`.

concreto, dado un nodo en la posición i, su padre está en la posición $(i - 1)/2$, su hijo izquierdo en la $2i + 1$, y el derecho en la siguiente al hijo izquierdo. En tercer lugar, `hundir2` necesita ser informado del tamaño actual del montículo (que se reduce en 1 en cada iteración de `eliminarMax`). La implementación de `hundir2` se deja como Ejercicio 20.20. Suponiendo que hemos escrito `hundir2`, entonces el algoritmo `ordenViaMonticulo` se puede expresar fácilmente, como se muestra en la Figura 20.31.

20.6 Ordenación externa

Hasta el momento, todos los algoritmos de ordenación examinados requieren que la entrada quepa en la memoria principal. Sin embargo, hay aplicaciones para las que la entrada es demasiado grande para ser cargarda completa en memoria. Esta sección estudia algoritmos *de ordenación externa*, diseñados para tratar entradas de gran tamaño.

20.6.1 Por qué necesitamos nuevos algoritmos

La mayor parte de los algoritmos de ordenación interna toman ventaja del hecho de que se puede acceder directamente a la memoria. El método Shellsort compara los elementos `v[i]` y `v[i-desp]` en una unidad de tiempo. El método del montículo compara `v[i]` y `v[hijo=i*2]` en una unidad de tiempo.

El algoritmo de ordenación rápida, con selección del pivote mediante la mediana de tres, requiere la comparación de `v[primero]`, `v[centro]` y `v[ultimo]` en un número constante de unidades de tiempo. Si la entrada se encuentra en una cinta, todas esas operaciones pierden la eficiencia requerida, pues solamente podemos acceder a los elementos en una cinta de forma secuencial. Incluso si los datos están en un disco, hay una notable pérdida de eficiencia en la práctica debido al retraso producido por el necesario giro del disco y el movimiento de la cabeza hasta el lugar adecuado.

Para ver lo lentos que pueden llegar a ser los accesos externos, podríamos crear un fichero aleatorio grande pero no lo suficiente como para no caber en memoria principal. Cuando leemos el fichero y lo ordenamos usando un algoritmo eficiente, el tiempo para leer la entrada es significativo en comparación con el tiempo que se tarda en ordenarla, incluso aunque la operación de ordenación requiere un tiempo $O(N \log N)$ (o incluso peor para el método Shellsort) y la lectura de la entrada sea sólo $O(N)$.

20.6.2 Modelo de ordenación externa

La amplia variedad de dispositivos de almacenamiento masivo hacen que la ordenación externa dependa del dispositivo mucho más que la ordenación interna. Los algoritmos considerados aquí trabajan sobre cintas, que constituyen probablemente el medio de almacenamiento más restrictivo. Puesto que el acceso a un elemento en una cinta se realiza recorriendo toda la cinta hasta la posición adecuada, sólo se puede acceder de forma eficiente a una cinta de forma secuencial (en cualquier sentido).

Supondremos que disponemos al menos de tres cintas para llevar a cabo la ordenación. Necesitamos dos para realizar una ordenación eficiente, la tercera simplifica las cosas. Si solamente tuviéramos una cinta, tendríamos un grave problema: cualquier algoritmo requeriría $\Omega(N)$ accesos a la cinta.

20.6.3 El algoritmo sencillo

El algoritmo básico de ordenación externa usa la rutina de mezcla usada en mergesort. Supongamos que tenemos cuatro cintas, $A1$, $A2$, $B1$ y $B2$, que son dos cintas de entrada y dos de salida. Dependiendo del punto en el que nos encontremos en el algoritmo, las cintas A y B son de entrada o de salida. Supondremos que los datos están inicialmente en $A1$, y además que la memoria puede almacenar (y ordenar) M registros de una vez. El primer paso natural es leer M registros de una vez de la cinta de entrada, ordenar internamente estos registros y después escribir los registros ordenados en $B1$ o en $B2$ (de forma alternada). Cada conjunto de registros ordenados se llama una *carrera*. Cada vez que está hecho esto, rebobinamos todas las cintas. Supongamos que tenemos la misma entrada que en nuestro ejemplo del Shellsort. La configuración inicial se muestra en la Figura 20.32. Si $M = 3$, después de construir las carreras, las cintas contendrán los datos como se muestra en la Figura 20.33.

Ahora $B1$ y $B2$ contienen un grupo de carreras. Tomamos las primeras carreras de cada cinta, las mezclamos y escribimos el resultado, que es una carrera el doble de larga, en $A1$. Después tomamos la siguiente carrera de las dos cintas, las mezclamos y escribimos el resultado en $A2$. Continuamos con este proceso, alternando la salida sobre $A1$ y $A2$ hasta que o bien $B1$ o bien $B2$ esté vacía. En este punto, o bien las dos están vacías o hay una (probablemente corta) carrera restante. En este último caso, copiamos esta carrera en la cinta apropiada. Rebobinamos todas las cintas y repetimos los mismos pasos, esta vez usando las cintas A como entrada y las B como salida. Esto nos conducirá a carreras de longitud $4M$. Continuamos con este proceso hasta obtener una carrera de longitud N, en cuyo momento la carrera representa la entrada ordenada. Las Figuras 20.34 a 20.36 muestran cómo trabaja este proceso sobre nuestro ejemplo de entrada.

El algoritmo requerirá $\lceil \log (N/M) \rceil$ pasadas, más la primera para la construcción de las carreras iniciales. Por ejemplo, si tenemos 10 millones de registros de 128 bytes cada uno y 4 MB de memoria principal, entonces la primera pasada creará 320 carreras. Necesitaríamos 9 pasadas más para completar la ordenación. Esta fórmula también nos indica que nuestro ejemplo de la Figura 20.33 requiere $\lceil \log (13/3) \rceil$ pasadas más, es decir 3.

A1	81	94	11	96	12	35	17	99	28	58	41	75	15
A2													
B1													
B2													

Figura 20.32 Configuración inicial de la cinta.

A1													
A2													
B1	11	81	94	17	28	99	15						
B2	12	35	96	41	58	75							

Figura 20.33 Distribución de las carreras de longitud 3 en dos cintas.

A1	11	12	35	81	94	96	15						
A2	17	28	41	58	75	99							
B1													
B2													

Figura 20.34 Cintas después de la primera ronda de mezclas (la longitud de la carrera es 6).

A1												
A2												
B1	11	12	17	28	35	41	58	75	81	94	96	99
B2	15											

Figura 20.35 Cintas después de la segunda ronda de mezclas (la longitud de la carrera es 12).

A1	11	12	15	17	28	35	41	58	75	81	94	96	99
A2													
B1													
B2													

Figura 20.36 Cintas después de la tercera ronda de mezclas.

20.6.4 Mezcla multiaria

Si tuviéramos cintas adicionales, entonces podríamos esperar reducir el número de pasadas requeridas para ordenar nuestra entrada. Hacemos esto extendiendo la mezcla básica de dos cintas a una mezcla K-aria.

Mezclar dos carreras consiste en recorrer cada cinta de entrada hasta el principio de las carreras a mezclar. Entonces se toma el elemento más pequeño de los dos (aquellos sobre los que nos encontramos en cada una de las cintas) y se coloca en una cinta de salida avanzando la correspondiente cinta de entrada. Si hay K cintas de entrada, la estrategia trabaja de la misma forma; la única diferencia está en que es ligeramente más complicado encontrar el elemento más pequeño entre K elementos. Podemos encontrar el más pequeño usando una cola de prioridad. Para obtener el siguiente elemento a escribir en la cinta de salida, llevamos a cabo una operación `eliminarMin`. Se avanza la correspondiente cinta de entrada, y si aún no se ha completado la carrera de esa cinta de entrada, insertamos el nuevo elemento en la cola de prioridad. La Figura 20.37 muestra cómo se distribuye la entrada del ejemplo anterior en tres cintas. Las Figuras 20.38 y 20.39 muestran las dos pasadas de la mezcla 3-aria que completan la ordenación.

A1								
A2								
A3								
B1	11	81	94	41	58	75		
B2	12	35	96	15				
B3	17	28	99					

Figura 20.37 Distribución inicial de carreras de longitud 3 en tres cintas.

A1	11	12	17	28	35	81	94	96	99
A2	15	41	58	75					
A3									
B1									
B2									
B3									

Figura 20.38 Situación después de una ronda de mezcla 3-aria (la longitud de la carrera es 9).

A1												
A2												
A3												
B1	11	12	15	17	28	35	41	58	75	81	94	96	99
B2													
B3													

Figura 20.39 Situación después de dos rondas de mezclas 3-arias.

Después de la fase de construcción de las carreras iniciales, el número de pasadas requeridas usando una mezcla K-aria es $\lceil \log_K (N/M) \rceil$, porque la longitud de las carreras se hace K veces más grande cada vez. Para nuestro ejemplo, se tiene $\lceil \log_3 13/3 \rceil = 2$. Si tuviéramos 10 cintas, entonces $K = 5$, y en tal caso, en el ejemplo grande de la sección anterior, las 320 carreras requerirían $\log_5 320 = 4$ pasadas.

20.6.5 Mezcla multifase

La estrategia de la mezcla K-aria desarrollada en la sección anterior requiere el uso de $2K$ cintas. Esto podría ser prohibitivo en algunas aplicaciones. Es posible hacerlo con solamente $K + 1$ cintas, a lo que se llama *mezcla multifase*. Un ejemplo de ello sería realizar una mezcla 2-aria usando solamente tres cintas.

Supongamos que tenemos tres cintas —$T1$, $T2$ y $T3$— y un fichero de entrada en $T1$ que produce 34 carreras. Una opción consiste en colocar 17 carreras en cada una de las cintas $T2$ y $T3$. Entonces podríamos mezclar este resultado en $T1$, obteniendo entonces una cinta con 17 carreras. El problema está en que puesto que todas las carreras están en una sola cinta, ahora debemos poner algunas de ellas en la cinta $T2$ para poder realizar la siguiente mezcla. La forma lógica de hacer esto consiste en copiar las primeras ocho carreras de $T1$ en $T2$ y después realizar la mezcla.

	Carrera	Después					
	Const.	T3+T2	T1+T2	T1+T3	T2+T3	T1+T2	T1+T3	T2+T3
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	2

Figura 20.40 Número de carreras usando la mezcla multifase.

Esto tiene el efecto de añadir media pasada extra por cada pasada realizada. La cuestión es, ¿podemos hacerlo mejor?

Un método alternativo consiste en dividir las 34 carreras de forma no uniforme. Supongamos que colocamos 21 carreras en $T2$ y 13 en $T3$. Entonces podríamos mezclar 13 carreras produciendo el resultado sobre $T1$ antes de que $T3$ se vacíe. Entonces podemos rebobinar $T1$ y $T3$, y mezclar después $T1$ con 13 carreras, y $T2$ con 8 carreras produciendo el resultado sobre $T3$. Tras ello mezclaríamos $T1$ y $T3$, y así sucesivamente. La Figura 20.40 muestra el número de carreras en cada cinta después de cada pasada.

La distribución original de las carreras supone una gran diferencia. Por ejemplo, si se colocan 22 carreras en $T2$ y 12 en $T3$, después de la primera mezcla, obtenemos 12 carreras en $T1$ y 10 en $T2$. Tras una nueva mezcla hay 10 carreras en $T1$ y 2 carreras en $T3$. En este punto, la ejecución se hace más lenta porque solamente podemos mezclar dos conjuntos de carreras antes de agotar $T3$. Entonces $T1$ tiene 8 carreras y $T2$ tiene 2 carreras. De nuevo solamente podemos mezclar dos conjuntos de carreras, obteniendo $T1$ con 6 carreras y $T3$ con 2 carreras. Después de tres pasadas más, $T2$ tiene dos carreras y las demás cintas están vacías. Debemos copiar 1 carrera en otra cinta. Después mezclamos y terminamos.

Se observa que la primera distribución considerada es óptima. Si el número de carreras es un número de Fibonacci F_N, entonces la mejor forma de distribuirlas consiste en dividirlas en dos números de Fibonacci, F_{N-1} y F_{N-2}. En caso contrario, la cinta debería rellenarse con carreras falsas para conseguir un número de carreras igual a un número de Fibonacci. Los detalles de cómo colocar el conjunto inicial de carreras en las cintas se deja como Ejercicio 20.19. Podemos extender esto a las mezclas K-arias, en las que necesitaríamos para la distribución óptima números de Fibonacci de orden K. Un número de Fibonacci de orden K se define como la suma de los K números de Fibonacci de orden K anteriores, como indican las siguientes ecuaciones:

$$F^{(K)}(N) = F^{(K)}(N-1) + F^{(K)}(N-2) + \cdots + F^{(K)}(N-K)$$

$$F^{(K)}(0 \leqslant N \leqslant K-2) = 0$$

$$F^{(K)}(K-1) = 1$$

20.6.6 Selección del reemplazo

La última cuestión que consideramos es la construcción de las carreras. La estrategia usada hasta el momento es sencilla: leemos tantos elementos como sea posible y los ordenamos, escribiendo el resultado en una cinta. Ésta parece la mejor aproximación posible, hasta que nos damos cuenta de que tan pronto como se escribe el primer elemento en la cinta de salida, la memoria que estaba utilizando está disponible para otro elemento. Si el siguiente elemento de la cinta es mayor que el que acaba de escribirse en la cinta, entonces se puede incluir en la carrera.

Usando esta observación, podemos dar un algoritmo para producir carreras, llamado habitualmente *selección del reemplazo*. Inicialmente se leen M elementos y se colocan en una cola de prioridad (en la memoria principal), usando operaciones `introducir` y una operación `arreglarMonticulo` al final. Llevamos a cabo una operación `eliminarMin`, escribiendo el elemento menor en la cinta de salida. Leemos el siguiente elemento de la cinta de entrada. Si es mayor que el elemento que acabamos de escribir, podemos añadirlo a la cola de prioridad usando una operación `insertar`; en caso contrario, no puede formar parte de la carrera actual. Puesto que la cola de prioridad tiene un elemento menos, este elemento se almacenará en el espacio que no forma parte del montículo (espacio muerto) de la cola de prioridad hasta que se complete la carrera, y se utilizará en la siguiente. Almacenar un elemento en el espacio muerto es exactamente lo que se hacía en heapsort. Continuamos haciendo esto hasta que el tamaño de la cola de prioridad se haga 0, en cuyo momento se acaba la carrera. Comenzamos una nueva carrera construyendo una nueva cola de prioridad utilizando una operación `arreglarMonticulo` sobre todos los elementos situados en el espacio muerto.

La Figura 20.41 muestra la construcción de la carrera para el pequeño ejemplo que hemos estado utilizando, con $M = 3$. Los elementos reservados para la siguiente carrera aparecen sombreados. Los elementos 11, 94 y 81 se colocan usando `arreglarMonticulo`. Se produce la salida de 11 y entonces se coloca 96 en el montículo mediante una inserción, pues es mayor que 11. El elemento 81 sale a

	3 elementos del vector del montículo				
	vector(1)	vector(2)	vector(3)	Salida	Siguiente elemento Lectura
Carrera 1	11	94	81	11	96
	81	94	96	81	12
	94	96	12	94	35
	96	35	12	96	17
	17	35	12	Final de la carrera	Reconstrucción
Carrera 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75
	58	99	75	58	15
	75	99	15	75	Final de la cinta
	99		15	99	
			15	Final de la carrera	Reconstrucción
Carrera 3	15				15

Figura 20.41 Ejemplo de construcción de una carrera.

continuación, y entonces se lee 12. Como 12 es menor que 81, no puede incluirse 1 en la carrera actual, por lo que se coloca en el espacio muerto del montículo. El montículo ahora solamente contiene 94 y 96. Después de producirse su salida, sólo tenemos elementos en el espacio muerto, por lo que construimos un nuevo montículo comenzando así con la segunda carrera. En este ejemplo, la selección del reemplazo produce solamente 3 carreras, comparadas con las 5 obtenidas mediante ordenación. Gracias a ello, una mezcla 3-aria acaba en una pasada en lugar de en dos. Si la entrada está distribuida aleatoriamente, se puede demostrar que la selección del reemplazo puede producir carreras de longitud media $2M$. Para nuestro ejemplo grande, tendríamos 160 carreras en lugar de 320, por lo que una mezcla 5-aria aun requeriría cuatro pasadas. En este caso no nos hemos ahorrado ninguna pasada, aunque si tuviéramos suerte podríamos ahorrárnosla de tener 125 carreras o menos. Puesto que la ordenación externa tarda tanto, cada pasada que nos ahorremos puede tener una gran influencia en el tiempo total de ejecución.

Como hemos visto, es posible que la selección del reemplazo no sea mejor que el algoritmo estándar. Pero, la entrada de la que partimos está con frecuencia casi ordenada, en cuyo caso la selección del reemplazo produce unas pocas carreras, anormalmente largas. Este tipo de entrada es habitual en las ordenaciones externas y hace que la selección del reemplazo sea extremadamente valiosa.

Resumen

En este capítulo hemos presentado una implementación elegante de la cola de prioridad. El montículo binario usa solamente un vector, y soporta las operaciones básicas en tiempo logarítmico, en el caso peor. El montículo nos facilita un algoritmo de ordenación conocido, el *método del montículo* o *heapsort*. Los Ejercicios 20.22 y 20.23 le piden comparar la eficiencia del heapsort con la del quicksort. Generalmente hablando, heapsort es más lento que el quicksort, pero es más sencillo de implementar. Finalmente, hemos visto que las colas de prioridad son estructuras de datos importantes en la ordenación externa.

Esto completa la implementación de las estructuras de datos fundamentales clásicas. En la Parte V se examinan estructuras de datos más sofisticadas, comenzando con el árbol de ensanchamiento, un árbol binario de búsqueda con algunas propiedades especiales.

Elementos del juego

árbol binario completo Árbol completamente lleno, sin que falte ningún nodo. Un montículo es un árbol binario completo, lo que permite su representación mediante un vector y garantiza una profundidad logarítmica.

carrera Grupo ordenado en la ordenación externa. Al final de la ordenación tenemos solamente una carrera.

heapsort Algoritmo basado en la idea de que se puede usar una cola de prioridad para ordenar elementos en un tiempo $O(N \log N)$.

hundimiento La eliminación del elemento mínimo implica colocar el último elemento en un hueco creado en la raíz. El hueco se va hundiendo en el árbol a través de los hijos menores hasta que se pueda colocar el elemento sin violar la propiedad de ordenación del montículo.

mezcla multiaria Mezcla K-aria que reduce el número de pasadas. La implementación obvia usa $2K$ cintas.

mezcla multifase Implementa una mezcla K-aria usando $K + 1$ cintas.

montículo binario Estructura clásica usada para implementar las colas de prioridad. Tiene dos propiedades: una estructural y otra de ordenación.

montículo maximal Montículo que soporta accesos al máximo en lugar de al mínimo.

operación `arreglarMonticulo` Proceso de restablecer la propiedad de ordenación de un montículo. Se puede hacer en tiempo lineal aplicando una rutina de hundimiento a todos los nodos en sentido inverso al recorrido por niveles.

operación `introducir` Operación que añade un elemento, pero que al contrario que `insertar`, no garantiza que se mantenga la ordenación del montículo. Es útil si pretendemos añadir muchos elementos antes de acceder al mínimo.

ordenación externa Método de ordenación usado cuando la cantidad de datos es demasiado grande para caber en la memoria principal.

propiedad de ordenación del montículo Propiedad que se cumple en un montículo (minimal), que consiste en que el elemento del padre de un nodo nunca es mayor que el elemento del propio nodo.

reflotamiento La inserción se implementa creando un hueco en la primera posición disponible, flotándolo después hasta que se pueda colocar en él el nuevo elemento sin que ello suponga una violación del orden respecto a su padre.

representación implícita Uso de un vector para almacenar un montículo.

selección del reemplazo La longitud de las carreras construidas inicialmente puede ser mayor que la cantidad de memoria principal disponible. Si podemos almacenar M elementos en la memoria principal, con esta técnica podemos esperar carreras de longitud $2M$.

Errores comunes

1. El montículo binario requiere un centinela en la posición 0. Un error común es utilizar un centinela inapropiado.
2. La parte más difícil del montículo binario es el caso del hundimiento cuando solamente existe uno de los hijos. Ya que es una situación rara, es difícil descubrir una implementación incorrecta.
3. Si se ha ejecutado una operación `introducir`, la propiedad de ordenación del montículo debe restablecerse antes de acceder al elemento mínimo.
4. Para heapsort, los datos comienzan en la posición 0, por lo que los hijos del nodo i se encuentran en las posiciones $2i + 1$ y $2i + 2$.

En Internet

El montículo binario está disponible en el directorio **DataStructures**. El nombre del fichero es el siguiente:

BinaryHeap.java Contiene la implementación del montículo binario.

Ejercicios

Cuestiones breves

20.1. Describa las propiedades estructural y de ordenación de los montículos binarios.

20.2. En un montículo binario, para un elemento en la posición i, ¿dónde se encuentran su padre y sus hijos izquierdo y derecho?

20.3. Muestre los resultados de insertar los elementos 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13 y 2, uno a uno, en un montículo inicialmente vacío. Después muestre el resultado de usar en su lugar el algoritmo lineal `arreglarMonticulo`.

20.4. ¿Dónde podría haber estado la undécima línea punteada en las Figuras 20.20 a 20.23?

20.5. Un montículo maximal soporta las operaciones `insertar`, `eliminarMax` y `buscarMax` (pero no `eliminarMin` o `buscarMin`). Describa en detalle cómo se pueden implementar los montículos maximales.

20.6. Muestre el resultado del algoritmo de ordenación heapsort sobre la entrada del Ejercicio 20.3 después de la construcción inicial y de dos operaciones posteriores `eliminarMax`.

20.7. ¿Es heapsort un algoritmo de ordenación estable? Es decir, si hay elementos duplicados, ¿retienen dichos elementos duplicados la ordenación inicial entre ellos?

Problemas teóricos

20.8. Un árbol binario completo con N elementos usa las posiciones 1 a N de un vector. Determine lo largo que puede llegar a ser el vector en los siguientes casos:

a) Un árbol binario con dos niveles adicionales (es decir, está ligeramente desequilibrado).
b) Un árbol binario con el nodo más profundo a profundidad $2 \log N$.
c) Un árbol binario con el nodo más profundo a profundidad $4.1 \log N$.
d) El peor caso de árbol binario.

20.9. Demuestre las siguientes afirmaciones con respecto al elemento máximo en un montículo:

a) Debe ser una de las hojas.
b) Hay exactamente $\lceil N/2 \rceil$ hojas.
c) Se debe examinar cada hoja para encontrarlo.

20.10. Demuestre el Teorema 20.1 usando directamente una suma. Haga lo siguiente:

a) Demuestre que hay 2^i nodos de altura $H - i$.
b) Escriba la ecuación que define la suma de las alturas usando la parte (a).
c) Evalúe la suma de la parte (b).

20.11. Verifique que la suma de las alturas de un árbol binario perfecto es $N - v(N)$, donde $v(N)$ es el número de unos en la representación binaria de N.

20.12. Demuestre la cota del Ejercicio 20.11 usando un argumento de inducción.

20.13. En el heapsort se usan, en el caso peor, $O(N \log N)$ comparaciones. Derive la constante multiplicativa (es decir, decida si son $N \log N$, $2N \log N$ o $3N \log N$, etc.).

20.14. Demuestre que hay entradas que fuerzan al método `hundir2` de heapsort a bajar todo el camino hasta una hoja. *Pista:* trabaje hacia atrás.

20.15. Un *d-montículo* es una estructura de datos implícita similar al montículo binario, con la diferencia de que los nodos tienen d hijos. Un d-montículo es, por tanto, menos profundo que un montículo binario, pero encontrar el hijo menor requiere examinar d hijos en lugar de dos. Con esto en mente determine el tiempo de ejecución (en términos de d y de N) de las operaciones `insertar` y `eliminarMin` para un d-montículo.

20.16. Un *montículo mini-max* es una estructura de datos que soporta `eliminarMin` y `eliminarMax` con coste logarítmico. La estructura es idéntica a la del montículo binario. La propiedad de ordenación del montículo mini-max es que para cada nodo X a profundidad par, el dato almacenado en X es el menor de su subárbol, mientras que para cada nodo X a profundidad impar, el dato almacenado en X es el mayor de su subárbol. La raíz se encuentra a profundidad par. Haga lo siguiente:

a) Dibuje un posible montículo mini-max para los elementos 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10. Observe que hay muchos montículos posibles.
b) Determine cómo encontrar los elementos máximo y mínimo.
c) Proporcione un algoritmo para insertar un nuevo nodo en un montículo mini- max.
d) Proporcione un algoritmo para ejecutar `eliminarMin` y `eliminarMax`.
e) Proporcione un algoritmo para ejecutar `arreglarMonticulo` en tiempo lineal.

20.17. Un *2-D-montículo* es una estructura de datos que permite que cada nodo tenga dos claves individuales. Se puede ejecutar `eliminarMin` con respecto a cualquiera de dichas claves. La propiedad de ordenación de un 2-D-montículo consiste en que para cada nodo X a profundidad par, el elemento almacenado en X tiene la clave #1 más pequeña de su subárbol, mientras que para cada nodo X a profundidad impar, el elemento almacenado en X tiene la clave #2 más pequeña de su subárbol. Haga lo siguiente:

a) Dibuje un posible 2-D-montículo para los elementos (1, 10), (2, 9), (3, 8), (4, 7) y (5, 6).
b) Explique cómo encontrar el elemento con mínima clave #1.
c) Explique cómo encontrar el elemento con mínima clave #2.
d) Proporcione un algoritmo para insertar un nuevo elemento en el 2-D-montículo.
e) Proporcione un algoritmo para ejecutar `eliminarMin` con respecto a cualquiera de las claves.
f) Proporcione un algoritmo para ejecutar `arreglarMonticulo` en tiempo lineal.

20.18. Un *montárbol* es un árbol binario de búsqueda en el que cada nodo almacena un elemento, dos hijos y una prioridad aleatoria generada en la construcción del nodo. Los nodos del árbol obedecen el orden habitual de los árboles binarios de búsqueda, pero también deben mantener la ordenación de tipo montículo con respecto a las prioridades. El montárbol es una buena alternativa a los árboles binarios equilibrados porque el equilibrio se basa en prioridades aleatorias en lugar de en los valores de sus elementos. En consecuencia los resultados de los árboles binarios de búsqueda para el caso medio se aplican también aquí. Haga lo siguiente:

a) Muestre que una colección de elementos distintos, cada uno de los cuales tiene una prioridad distinta, se puede representar mediante un único montárbol.
b) Muestre cómo llevar a cabo una inserción en un montárbol usando un algoritmo ascendente.
c) Muestre cómo llevar a cabo una inserción en un montárbol usando un algoritmo descendente.
d) Muestre cómo llevar a cabo una eliminación en un montárbol.

20.19. Explique cómo colocar el conjunto inicial de carreras en dos cintas cuando el número de cintas no es un número de Fibonacci.

Problemas prácticos

20.20. Escriba la rutina `hundir2` con la siguiente declaración. (Recuerde que el montículo maximal comienza en la posición 0, no en la 1):

```
private static void hundir2( Comparable [ ] v,
                             int indice, int tamanyo );
```

Prácticas de programación

20.21. Escriba un programa para comparar el tiempo de ejecución de N operaciones `introducir` seguidas por una operación `arreglarMonticulo` con el de N operaciones `insertar` separadas. Ejecute su programa para entradas ordenadas, ordenadas de forma inversa y aleatorias.

20.22. Implemente tanto heapsort como quicksort y compare su eficiencia sobre entradas ordenadas y aleatorias. Use en las pruebas distintos tipos de datos.

20.23. Supongamos que tenemos un hueco en el nodo X. La rutina `hundir2` compara con el hijo de X y después mueve el hijo hacia arriba si es mayor (en el caso de un montículo maximal) que el elemento a colocar, empujando por tanto el hueco hacia abajo. La rutina termina cuando es seguro colocar el elemento a insertar en el hueco. Consideremos la siguiente estrategia alternativa para `hundir2`. Movemos los elementos hacia arriba y el hueco hacia abajo lo más lejos posible, sin comprobar que se puede insertar el nuevo elemento. Esto colocaría la nueva celda en una de las hojas y probablemente violaría el orden. Para arreglar el orden, reflote la nueva celda en la forma habitual. Se espera que el reflotamiento se repetirá en media solamente uno o dos niveles. Escriba una rutina para introducir esta idea. Compare el tiempo de ejecución con el de la implementación estándar de heapsort.

20.24. Implemente una ordenación externa.

20.25. Diseñe un applet que ilustre el funcionamiento del montículo binario.

Bibliografía

El montículo binario se describió por primera vez, en el contexto del algoritmo heapsort, en [8]. El algoritmo lineal `arreglarMonticulo` se ha tomado de [4]. En [7] se proporcionan resultados precisos del número de comparaciones y movimientos de datos usados por heapsort en los casos peor, mejor y medio. En los Capítulos 21 y 22 se discuten implementaciones avanzadas de las colas de prioridad. La ordenación externa se discute en detalle en [6]. El Ejercicio 20.15 se resuelve en [5], el 20.16 en [2] y el 20.17 en [3]. Los montárboles se describen en [1].

1. C. Aragon y R. Seidel, «Randomized Search Trees», *Algorithmica* **16** (1996), 464-497.
2. M. D. Atkinson, J. R. Sack, N. Santoro, y T. Strothotte, «Min-Max Reaps and Generalized Priority Queues», *Communications of the ACM* **29** (1986), 996-1000.
3. Y. Ding y M. A. Weiss, «The k-d Heap: An Efficient Multi-dimensional Priority Queue», *Proceedings of the Third Workshop on Algorithms and Data Structures* (1993), 302-313.
4. R. W. Floyd, «Algorithm 245: Treesort 3», *Communications of the ACM* **7** (1964), 701.
5. D. B. Johnson, «Priority Queues with Update and Finding Minimum Spanning Trees», *Information Processing Letters* **4** (1975), 53-57.
6. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2.ª ed., Addison-Wesley, Reading, Mass. (1997).
7. R. Schaffer y R. Sedgewick, «The Analysis of Heapsort», *Journal of Algorithms* **14** (1993), 76-100.
8. J. W. J. Williams, «Algorithm 232: Heapsort», *Communications of the ACM* **7** (1964), 347-348.

Parte V

Estructuras de datos avanzadas

CAPÍTULO

21 Árboles de ensanchamiento

Este capítulo describe una notoria estructura de datos denominada *árbol de ensanchamiento*[1]. Los árboles de ensanchamiento soportan todas las operaciones de los árboles de búsqueda binarios, pero no garantizan un rendimiento $O(\log N)$ en el caso peor. En cambio, sus cotas son *amortizadas*, lo que significa que aunque las operaciones individuales pueden ser costosas, se garantiza que cualquier secuencia de operaciones se comportará como si cada operación en la secuencia exhibiera un comportamiento logarítmico. Como quiera que ésta es una garantía más débil que la proporcionada por los árboles de búsqueda equilibrados, para soportar cada elemento en el árbol sólo necesitaremos el dato y dos referencias por nodo. Además, las operaciones son algo más sencillas de implementar. Como veremos a lo largo de este capítulo, los árboles de ensanchamiento tienen algunas otras propiedades interesantes.

En este capítulo veremos:

- La descripción de los conceptos de amortización y auto-ajustamiento.
- El algoritmo del árbol básico ensanchado de forma ascendente y la demostración del coste amortizado logarítmico por operación.
- Cómo pueden implementarse los árboles de ensanchamiento utilizando un algoritmo descendente, con una implementación completa de los árboles de ensanchamiento (incluyendo el algoritmo de eliminación).
- Comparaciones entre los árboles de ensanchamiento y otras estructuras de datos.

21.1 Auto-ajustamiento y análisis amortizado

Aunque los árboles de búsqueda equilibrados proporcionan un tiempo de ejecución por operación logarítmico en el caso peor, tienen varias limitaciones:

- Los árboles de búsqueda equilibrados necesitan almacenar un atributo extra por nodo con información de equilibrio.

[1] *N. del T.:* En inglés, *splay tree*.

- Son complicados de implementar; como resultado, las inserciones y eliminaciones son costosas y propensas a errores.
- No ganamos nada cuando trabajamos con entradas sencillas.

Examinemos las consecuencias de cada una de estas deficiencias. En primer lugar, los árboles de búsqueda equilibrados requieren almacenar un atributo extra. Aunque en teoría éste puede ser tan pequeño como un único bit (como en los árboles rojinegros), en la práctica, para satisfacer las restricciones hardware, el atributo utilizará un entero completo. A pesar de todo, en una época en la que las memorias de los computadores son ya enormes uno puede preguntarse si debe preocuparse demasiado de la memoria. La respuesta en la mayoría de los casos es probablemente no, si no fuera porque el mantenimiento del atributo extra requiere un código más complicado y tiende a llevarnos a tiempos mayores de ejecución y a un número mayor de errores. De hecho, es difícil comprobar si la información de equilibrio de los árboles de búsqueda es correcta, ya que los errores sólo nos conducen a un árbol desequilibrado. Si sólo de forma muy esporádica cometiésemos tales errores *leves*, puede ser muy difícil encontrarlos. Por tanto, como cuestión práctica, los algoritmos que nos permitan eliminar dichas complicaciones, sin con ello sacrificar el rendimiento, se merecen una consideración seria.

Hay una segunda razón por la cual podemos sospechar que el rendimiento de la búsqueda equilibrada podría mejorarse. Tenemos que su rendimiento en el caso peor, en promedio y en el caso mejor son esencialmente idénticos. Un ejemplo es la operación `buscar` sobre algún elemento X. Es razonable esperar que el coste de una primera ejecución de `buscar` sea logarítmico; pero un segundo acceso inmediato a X debería ser menos costoso que el primero. En un árbol rojinegro esto no es cierto. También esperaríamos que si realizamos accesos a X, Y y Z, entonces un nuevo acceso a la misma terna de elementos debería ser más rápido. Esto es importante debido a la *regla del 90-10*. Esta regla, sugerida por estudios empíricos, afirma que, en la práctica, el 90 por ciento de los accesos que se requieren corresponden sólo al 10 por ciento de los datos. En consecuencia, queremos un acceso rápido para dicho 90 por ciento.

La regla del 90-10 se ha utilizado durante muchos años en sistemas de discos de entrada salida. Una memoria de respaldo (*caché*) almacena en memoria principal el contenido de algunos bloques de disco. Nuestra esperanza es que cuando se haga una petición de acceso a disco, el bloque pueda encontrarse en la cache de memoria principal y así se ahorre el coste de un acceso a disco. Por supuesto, sólo unos pocos bloques de disco pueden almacenarse en memoria. Aún así, almacenar los bloques de disco más recientemente accedidos permite una mejora considerable en el rendimiento, pues frecuentemente los mismos bloques son accedidos una y otra vez. Los navegadores de Internet utilizan la misma idea: una caché almacena localmente las páginas Web visitadas últimamente.

21.1.1 Cotas de tiempo amortizadas

Estamos pidiendo mucho: queremos evitar la información de equilibrio y ser capaces de sacar partido de la regla del 90-10. Ello nos conduce a pensar que podríamos tener que renunciar a alguna de las propiedades de los árboles de búsqueda equilibrados.

Elegimos sacrificar el rendimiento logarítmico en el caso peor. Ya que no queremos mantener información sobre el equilibrio, este sacrificio parece inevitable. Sin embargo, no podemos admitir el rendimiento típico de un árbol de búsqueda desequilibrado. Hay, en cambio, un compromiso razonable: un tiempo $O(N)$ para un acceso aislado puede ser aceptable, siempre y cuando no ocurra muy a menudo. En particular, si M operaciones (empezando con la primera operación) tardan un tiempo total $O(M \log N)$ en el caso peor, entonces el hecho de que algunas operaciones sean costosas no tiene graves consecuencias. Cuando para una secuencia de operaciones podemos demostrar una cota, en el caso peor, mejor que la correspondiente cota obtenida considerando cada operación por separado, el tiempo de ejecución se denomina *amortizado*. En el ejemplo anterior, tenemos un coste amortizado logarítmico. Es decir, algunas operaciones pueden tardar más que un tiempo logarítmico, pero estamos seguros de que se compensan con otras operaciones menos costosas que ocurrieron anteriormente en la secuencia.

Sin embargo, una cota amortizada no es siempre aceptable. En concreto, si una operación mala consume demasiado tiempo, realmente necesitamos una cota en el caso peor en vez de una cota amortizada. Aún así, en muchos casos las estructuras de datos se utilizan como parte de un algoritmo y sólo es importante la cantidad total de tiempo consumida por la estructura en el transcurso de la ejecución del algoritmo.

Ya hemos visto un ejemplo de cota amortizada. Cuando implementamos la duplicación del vector en una pila o cola, el coste de una sola operación es constante cuando no es necesaria la duplicación, pero es $O(N)$ cuando sí lo es. Sin embargo, cualquier secuencia de M operaciones sobre la cola o pila garantiza un coste total $O(M)$, lo que significa un coste amortizado constante por operación. El hecho de que la duplicación del vector sea costosa no tiene consecuencias, porque su coste puede distribuirse entre muchas operaciones anteriores que no fueron costosas.

21.1.2 Una estrategia simple de auto-ajustamiento (que no funciona)

En un árbol binario de búsqueda, no podemos limitarnos a almacenar los elementos frecuentemente accedidos en una tabla. Esto sólo funcionaba en el caso del acceso a memoria externa debido a que la técnica del uso de la memoria de respaldo se beneficia de la gran diferencia entre el tiempo de un acceso a memoria y de un acceso a disco. Ya que el coste de un acceso en un árbol binario de búsqueda es proporcional a la profundidad del nodo accedido, podemos intentar reestructurar el árbol moviendo hacia la raíz los elementos más frecuentemente accedidos. Aunque esto supone un tiempo extra en la primera operación `buscar`, a la larga podría merecer la pena.

La forma más fácil de desplazar un elemento hacia la raíz consiste en intercambiarlo continuamente con su padre hasta que llegue a la raíz. Entonces, si accedemos al elemento por segunda vez, el segundo acceso es muy barato. Incluso si se realizaran otras operaciones antes del siguiente acceso, este elemento seguirá estando cerca de la raíz y por tanto, será encontrado rápidamente. Esta estrategia se denomina *estrategia de rotación hacia la raíz*. En la Figura 21.1 se muestra la aplicación de esta estrategia al nodo 3 [2].

[2] Una inserción cuenta como un acceso. Por tanto, un elemento siempre se insertaría como una hoja e inm
plazado hacia la raíz. Una búsqueda sin éxito cuenta como un acceso a la hoja donde termina su búsqueda.

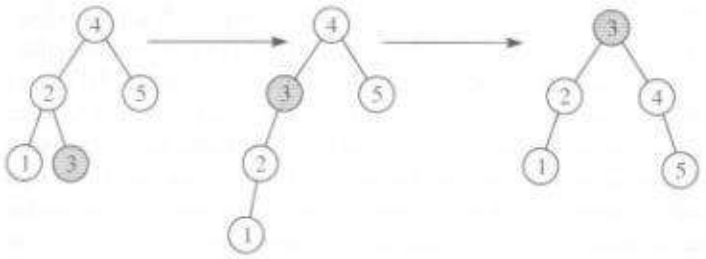


Figura 21.1 Estrategia de rotación hacia la raíz aplicada cuando se accede al nodo 3.

El resultado de la rotación es que un futuro acceso (durante algún tiempo) al nodo 3 será barato. Desgraciadamente, en el proceso de mover el nodo 3 hacia arriba dos niveles, los nodos 4 y 5 se mueven hacia abajo un nivel. Esto significa que si los patrones de acceso no siguen la regla del 90-10, es posible que se produzca una larga secuencia de accesos malos. Como resultado, la estrategia de rotación hacia la raíz no tendría un comportamiento amortizado logarítmico; lo cual es difícilmente aceptable. Un caso malo se presenta en el Teorema 21.1.

Existen secuencias arbitrariamente largas para las cuales M accesos con rotación hacia la raíz utilizan un tiempo $\Theta(MN)$.

Consideremos el árbol formado al insertar las claves 1, 2, 3, ..., N, en un árbol inicialmente vacío. Esto produce un árbol formado exclusivamente por hijos izquierdos. Esto no es de momento malo, ya que el tiempo de construcción del árbol es $O(N)$. Como ilustra la Figura 21.2, cada nuevo nodo añadido se convierte en un hijo de la raíz, tras lo cual sólo se necesita una rotación para colocar el nuevo elemento en la raíz. La parte negativa, como ilustra la Figura 21.3, es que el acceso al nodo con clave 1 consume N unidades de tiempo. Una vez que las rotaciones han terminado, el acceso al nodo con clave 2 consume N unidades de tiempo, y el acceso al nodo 3 consume $N - 1$ unidades de tiempo. Acceder a las N claves en orden requiere un total $N + \sum_{i=2}^{N} i = \Theta(N^2)$. Después de acceder a todos los nodos, el árbol ha vuelto a su estado inicial, y podemos repetir la secuencia. Por ello tenemos un coste amortizado $\Theta(N)$, lo cual, ciertamente, no es una gran cosa.

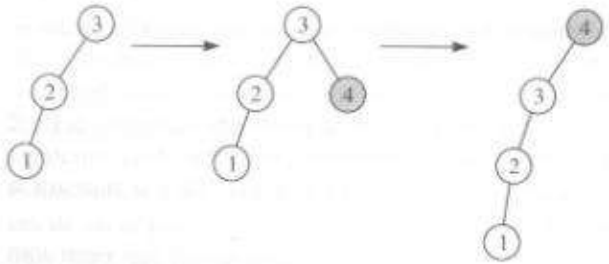


Figura 21.2 Inserción de 4 utilizando la rotación hacia la raíz.

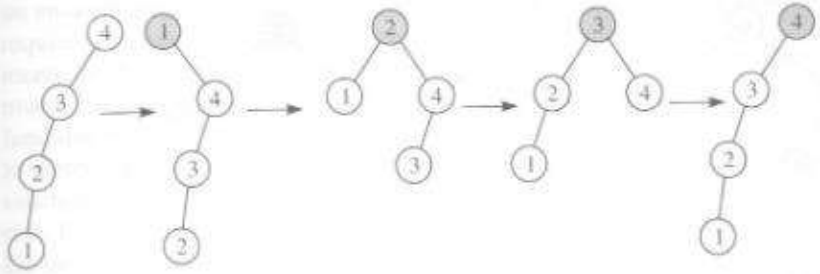


Figura 21.3 Ciertos accesos secuenciales a los elementos requieren un tiempo total cuadrático.

21.2 Árboles básicos de ensanchamiento ascendente

Lograr un coste amortizado logarítmico parece imposible, porque cuando movemos un elemento hacia la raíz vía rotaciones, otros elementos pasan a estar más profundos. Parece que si no se guarda información sobre el equilibrio siempre podría haber algunos nodos muy profundos. Sorprendentemente, podemos arreglar de forma sencilla la estrategia de rotación hacia la raíz, para conseguir una cota amortizada logarítmica. La estrategia de rotación hacia la raíz se denomina *ensanchamiento*. Su implementación nos lleva a los *árboles de ensanchamiento ascendente*.

La estrategia de ensanchamiento es parecida a la de rotación hacia la raíz, con una sutil diferencia. Ahora seguimos rotando de forma ascendente a lo largo del camino de acceso (más adelante presentaremos una estrategia descendente). Sea X un nodo, distinto de la raíz, en el camino de acceso en el que estamos rotando. Si el padre de X es la raíz del árbol, simplemente rotamos X y la raíz, como muestra la Figura 21.4. Ésta es la última rotación, y coloca X en la raíz. Observe que esto es exactamente lo que se haría en la rotación hacia la raíz. Éste es el caso *zig*.

En otro caso, X tiene un padre P y un abuelo G, y entonces hay dos casos a considerar, junto con sus simétricos. El primer caso es el denominado *zig-zag*, que corresponde al caso interior en los árboles AVL. Aquí X es un hijo derecho y P es un hijo izquierdo (o viceversa). Realizamos una doble rotación, exactamente análoga a la doble rotación de los árboles AVL, como muestra la Figura 21.5. Observemos que ya que una doble rotación equivale a dos rotaciones ascendentes, este caso no se diferencia de lo que haríamos en la rotación hacia la raíz. En la Figura 21.1 el ensanchamiento en el nodo 3 corresponde a una simple rotación zig-zag.

El último caso es el caso *zig-zig*, que corresponde al caso externo de los árboles AVL. Aquí, X y P son ambos hijos izquierdos o hijos derechos. En este caso, transformamos el árbol de la izquierda de la Figura 21.6 en el árbol de la derecha.

Figura 21.4 Caso zig (rotación simple normal).

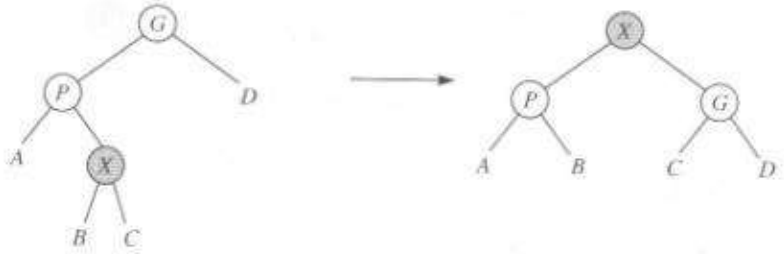


Figura 21.5 Caso zig-zag (igual que una rotación doble); se omite el caso simétrico.

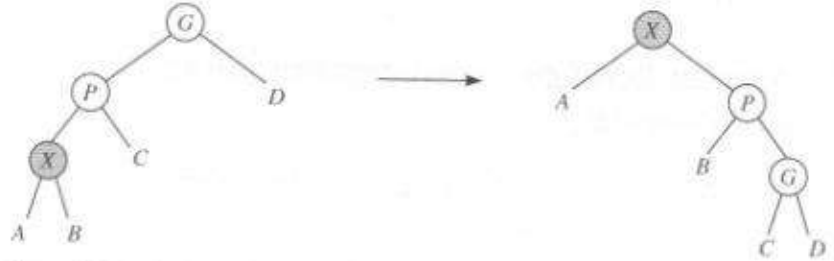


Figura 21.6 Caso zig-zig (exclusivo de los árboles de ensanchamiento); se omite el caso simétrico.

Observemos que esto difiere de lo que se hace en la rotación hacia la raíz. La rotación hacia la raíz rota entre X y P y después entre X y G. La rotación de ensanchamiento zig-zig rota entre P y G y después entre X y P.

El cambio en el caso zig-zig parece bastante pequeño; sorprende un tanto que tenga tanta repercusión. Para ver la diferencia entre el ensanchamiento y la rotación hacia la raíz, consideremos la secuencia que provocaba el mal resultado del Teorema 21.1. De nuevo, insertamos las claves 1, 2, 3, ..., N en un árbol inicialmente vacío en un tiempo total lineal y obtenemos un árbol desequilibrado, sólo con hijos izquierdos. Sin embargo, como muestra la Figura 21.7, el resultado de

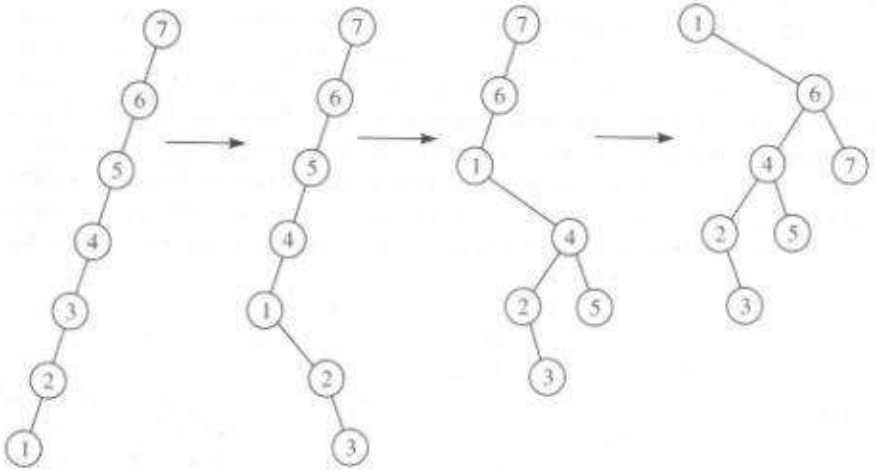


Figura 21.7 Resultado del ensanchamiento en el nodo 1 (tres zig-zig y un zig).

un ensanchamiento es algo mejor. Después del ensanchamiento en el nodo 1, que requiere N accesos a nodos, un ensanchamiento del nodo 2 consistirá aproximadamente en $N/2$ accesos, en vez de en $N - 1$ accesos. El ensanchamiento no sólo mueve el nodo accedido a la raíz, también divide aproximadamente por 2 la profundidad de la mayoría de los nodos en el camino de acceso. En contrapartida, sólo algunos nodos sombreados se profundizan, y como mucho dos niveles. Un ensanchamiento posterior en el nodo 2 llevará a los nodos del camino a $N/4$ de la raíz. Esto se repite, hasta que la profundidad llega a ser, aproximadamente, log N. De hecho, un complicado análisis demuestra que lo que era un caso malo para el algoritmo de rotación hacia la raíz, es ahora un caso bueno para el ensanchamiento: el acceso secuencial de los N elementos en el árbol ensanchado precisa un tiempo total que es sólo $O(N)$. Por tanto, ganamos bastante, al menos con entradas sencillas. La Sección 11.4 demuestra, mediante cuentas sutiles, que ahora ya no hay secuencias de accesos malas.

21.3 Operaciones básicas de los árboles de ensanchamiento

Como se mencionó anteriormente, después de cada acceso se realiza una operación de ensanchamiento. Cuando se realiza una inserción, realizamos un ensanchamiento. Como resultado, el nuevo elemento insertado se convierte en la raíz del árbol. De no hacerlo, podríamos gastar un tiempo cuadrático en la inserción de N elementos en el árbol.

En la operación `buscar`, hacemos en ensanchamiento desde el último nodo accedido durante la búsqueda. Si la búsqueda tiene éxito, entonces el elemento encontrado será ensanchado y se convertirá en la nueva raíz. Si la búsqueda no tiene éxito, será ensanchado el último nodo accedido antes de alcanzar una referencia `null` y se convertirá en la nueva raíz. Este proceso es necesario; de otra forma, si reiterásemos la búsqueda del elemento 0 en el árbol inicial de la Figura 21.7 utilizaríamos un tiempo lineal por operación. De igual manera, las operaciones `buscarMin` y `buscarMax` realizarán un ensanchamiento después de acceder al árbol.

Las operaciones interesantes son las eliminaciones. Recordemos que `eliminarMin` y `eliminarMax` son operaciones importantes de las colas de prioridad. Con árboles de ensanchamiento, estas operaciones son sencillas. Implementamos `eliminarMin` de la siguiente manera. Primero realizamos `buscarMin`, lo que hace que el menor elemento se coloque en la raíz, con lo que por las propiedades de los árboles de búsqueda, no habrá ningún hijo izquierdo. Podemos utilizar el hijo derecho como nueva raíz. De forma similar, `eliminarMax` puede implementarse llamando a `buscarMax` y haciendo que la nueva raíz sea el hijo izquierdo después del ensanchamiento.

Hasta la operación `eliminar` es sencilla. Para realizar la eliminación, accedemos al elemento que queremos eliminar. Esto lo coloca en la raíz. Si es borrado, obtenemos dos subárboles I y D (izquierdo y derecho). Si buscamos el mayor elemento en I, utilizando la operación `buscarMax`, entonces el mayor elemento se colocará en la raíz de I y no tendrá hijo derecho. Terminamos la operación `eliminar` haciendo que D sea el hijo derecho de la raíz de I. Un ejemplo de esta operación se muestra en la Figura 21.8.

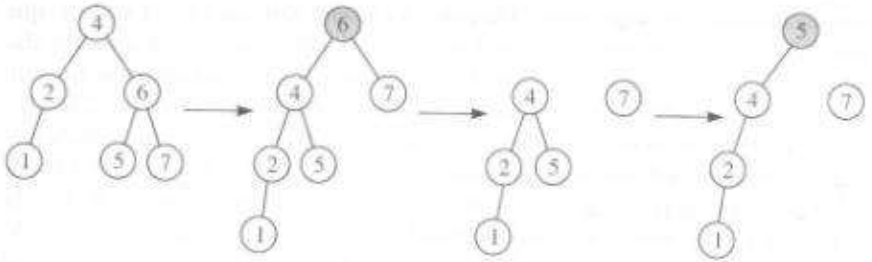


Figura 21.8 La operación `eliminar` aplicada al nodo 6; primero el 6 se ensancha hacia la raíz; la eliminación deja dos subárboles; se realiza un `buscarMax` en el subárbol izquierdo, elevando el 5 a la raíz de un árbol izquierdo; entonces se puede añadir el hijo derecho (no mostrado).

El coste de la operación `eliminar` es el de dos ensanchamientos. Todas las demás operaciones tienen el coste de un ensanchamiento. Por tanto, necesitamos analizar el coste de una serie de pasos de ensanchamiento. La próxima sección muestra que el coste amortizado de un ensanchamiento es, como mucho, el de $3 \log N + 1$ rotaciones. Entre otras cosas, esto significa que no debemos preocuparnos de que el algoritmo de eliminación descrito anteriormente pueda dar lugar a comportamientos sesgados. La cota amortizada de los árboles de ensanchamiento asegura que cualquier secuencia de M ensanchamientos necesitará, como mucho, $3M \log N + M$ rotaciones. En consecuencia, una secuencia de M operaciones que empiece con un árbol vacío exigirá, como mucho, un tiempo $O(M \log N)$.

21.4 Análisis del ensanchamiento ascendente

El análisis de los árboles de ensanchamiento es complicado porque cada ensanchamiento puede necesitar desde unas pocas hasta $O(N)$ rotaciones. Es más, al contrario de lo que sucede con los árboles de búsqueda equilibrados, cada ensanchamiento cambia la estructura del árbol. Esta sección prueba que el coste amortizado de un ensanchamiento es, como mucho, el de $3 \log N + 1$ rotaciones. La cota amortizada de los árboles de ensanchamiento asegura que cualquier secuencia de M ensanchamientos necesitará, como mucho, $3M \log N + M$ rotaciones. En consecuencia, una secuencia de M operaciones que empiece con un árbol vacío exigirá, como mucho, un tiempo $O(M \log N)$.

Para probar esta cota, introducimos una función de contabilidad denominada *función de potencial*. La función de potencial no se calcula en absoluto por el algoritmo. Es sólo un mecanismo de contabilidad virtual para establecer la cota de tiempo requerida. Su elección no es obvia, obteniéndose generalmente tras una gran cantidad de pruebas y errores.

Para cada nodo i en el árbol de ensanchamiento, sea $S(i)$ el número de descendientes de i (incluyendo al propio i). Nuestra función de potencial será la suma, sobre todos los nodos i en el árbol T, del logaritmo de $S(i)$. Más exactamente,

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

Para simplificar la notación, tomaremos $R(i) = \log S(i)$. Esto nos conduce a

$$\Phi(T) = \sum_{i \in T} R(i).$$

$R(i)$ representa el *rango* del nodo i. Recordemos que ni los rangos ni los tamaños son computados por los algoritmos de los árboles de ensanchamiento (a menos, por supuesto, que se necesiten estadísticas). Observemos que el rango de la raíz es $\log N$. Además, cuando se realiza una rotación zig, sólo se modifican los rangos de los dos nodos involucrados en la rotación. Cuando se realiza una rotación zig-zig o zig-zag, sólo cambia el rango de tres nodos. Y finalmente, cada ensanchamiento simple está formado por una serie de rotaciones zig-zig o zig-zag, seguidas, eventualmente, por una rotación zig. Cada rotación zig-zig o zig-zag puede contarse como dos rotaciones simples.

Sea X el nodo desde donde se ensancha el árbol y r el número total de rotaciones realizadas durante el ensanchamiento. Sea Φ_i la función de potencial del árbol inmediatamente después del i-ésimo ensanchamiento. En particular, Φ_0 es el potencial antes del primer ensanchamiento.

Si el i-ésimo ensanchamiento utiliza r_i rotaciones, se tiene

$$\Phi_i - \Phi_{i-1} + r_i \leq 3 \log N + 1.$$

Antes de probar el Teorema 21.2, veamos lo que significa. El coste de M ensanchamientos puede tomarse como $\Sigma_{i=1}^{M} r_i$ rotaciones. Si los M ensanchamientos son consecutivos (es decir, no se entremezclan inserciones o eliminaciones), entonces el potencial del árbol después del i-ésimo ensanchamiento es el mismo que antes del $(i+1)$-ésimo ensanchamiento. Por tanto, podemos utilizar M veces el Teorema 21.2 para obtener la secuencia de ecuaciones en la Ecuación 21.1

$$\begin{aligned}
\Phi_1 - \Phi_0 + r_1 &\leq 3 \log N + 1 \\
\Phi_2 - \Phi_1 + r_2 &\leq 3 \log N + 1 \\
\Phi_3 - \Phi_2 + r_3 &\leq 3 \log N + 1 \\
&\cdots \\
\Phi_M - \Phi_{M-1} + r_M &\leq 3 \log N + 1
\end{aligned} \tag{21.1}$$

Estas ecuaciones se solapan, de modo que si sumamos todas ellas, obtenemos

$$\Phi_M - \Phi_0 + \Sigma_{i=1}^{M} r_i \leq (3 \log N + 1)M, \tag{21.2}$$

lo que acota el número total de rotaciones de la siguiente manera

$$\Sigma_{i=1}^{M} r_i \leq (3 \log N + 1)M - (\Phi_M - \Phi_0).$$

Consideremos ahora lo que ocurre cuando se entremezclan las inserciones con las búsquedas. Merece la pena observar que el potencial de un árbol vacío es 0. Cuando se inserta un nodo en el árbol como una hoja, antes del ensanchamiento el potencial del árbol se ve incrementado en como mucho $\log N$ (esto se demostrará

en breve). Supongamos que se utilizan r_i rotaciones en la inserción y que el potencial antes de la inserción es Φ_{i-1}. Después de la inserción el potencial valdrá como mucho $\Phi_{i-1} + \log N$. Después del ensanchamiento que mueve el nodo insertado a la raíz, el nuevo potencial satisfará

$$\begin{aligned} \Phi_i - (\Phi_{i-1} + \log N) + r_i &\leqslant 3 \log N + 1 \\ \Phi_i - \Phi_{i-1} + r_i &\leqslant 4 \log N + 1. \end{aligned} \tag{21.3}$$

Supongamos que se realizan F búsquedas e I inserciones, y que Φ_i representa el potencial después de la i-ésima operación. Entonces, ya que cada búsqueda está gobernada por el Teorema 21.2 y cada inserción está gobernada por la Ecuación 21.3, el proceso de suma por solapamientos nos conduce a

$$\Sigma_{i=1}^{M} r_i \leqslant (3 \log N + 1)F + (4 \log N + 1)I - (\Phi_M - \Phi_0). \tag{21.4}$$

Por otra parte, antes de la primera operación el potencial es 0 y, ya que nunca puede ser negativo, tenemos $\Phi_M - \Phi_0 \geqslant 0$. En consecuencia, obtenemos

$$\Sigma_{i=1}^{M} r_i \leqslant (3 \log N + 1)F + (4 \log N + 1)I. \tag{21.5}$$

lo que muestra que el coste por operación de cualquier secuencia de búsquedas e inserciones es, como mucho, logarítmico. Ya que una eliminación equivale a dos ensanchamientos, ella también es logarítmica. Con todo ello podemos probar las dos afirmaciones pendientes, a saber, el Teorema 21.2 y el hecho de que una inserción de un nodo añade como mucho log N al potencial. Ambos teoremas se prueban utilizando argumentos de despliegue. Nos ocupamos primero en el Teorema 21.3 de la afirmación sobre la inserción.

La inserción como una hoja del N-ésimo nodo en un árbol agrega, como mucho, log N *al potencial del árbol.*

Los únicos nodos cuyos rangos se ven afectados son aquéllos en el camino desde la hoja insertada a la raíz. Sean $S_1, S_2, \ldots, S_k$ sus tamaños antes de la inserción y observemos que $S_k = N - 1$ y $S_1 < S_2 < \cdots < S_k$. Tomamos como $S_1', S_2', \ldots, S_k'$ los tamaños después de la inserción. Claramente, $S_i' \leqslant S_{i+1}'$ para $i < k$, ya que $S_i' = S_i + 1$. En consecuencia, $R_i' \leqslant R_{i+1}'$. Por tanto, el cambio en el potencial es

$$\Sigma_{i=1}^{k} (R_i' - R_i) \leqslant R_k' - R_k + \Sigma_{i=1}^{k-1} (R_{i+1} - R_i) \leqslant \log N - R_1 \leqslant \log N.$$

Para probar el Teorema 21.2, descomponemos cada paso de ensanchamiento en sus componentes zig, zig-zag y zig-zig, y establecemos una cota del coste de cada tipo de rotación. Por despliegue de estas cotas, obtenemos una cota para el ensanchamiento. Antes de continuar, necesitamos un resultado técnico, el Teorema 21.4.

Si $a + b \leqslant c$ y a y b son ambos enteros positivos, entonces

$$\log a + \log b \leqslant 2 \log c - 2.$$

En virtud de la desigualdad aritmético-geométrica, tendremos

$$\sqrt{ab} \leqslant (a + b)/2.$$

Por tanto $\sqrt{ab} \leqslant c/2$. Elevando al cuadrado ambos lados obtenemos $ab \leqslant c^2/4$ y tomando logaritmos a ambos lados queda demostrado el teorema.

Ahora estamos listos para probar el Teorema 21.2.

21.4.1 Demostración de la cota de ensanchamiento

En primer lugar, si el nodo a ensanchar es la raíz, no se producirán rotaciones y no hay ningún cambio de potencial. De modo que el resultado del teorema se tendría trivialmente, por lo que podemos asumir que hay al menos una rotación. Sea X el nodo involucrado en el ensanchamiento. Necesitamos probar que si se realizan r rotaciones (un ziz-zig o zig-zag cuenta como dos rotaciones), entonces r más el cambio de potencial es, como mucho, $3 \log N + 1$. Sea Δ el cambio de potencial causado por cualquiera de los pasos del ensanchamiento, zig, zig-zag, o zig-zig. Sean $R_i(X)$ y $S_i(X)$ el rango y el tamaño de cada nodo X inmediatamente antes de un paso del ensanchamiento y sean $R_f(X)$ y $S_f(X)$ el rango y el tamaño de cada nodo X inmediatamente después del paso del ensanchamiento. A continuación se presentan las cotas que serán probadas.

En el caso de un paso zig que asciende al nodo X, $\Delta \leqslant 3(R_f(X) - R_i(X))$, mientras que para los otros dos pasos, $\Delta \leqslant 3(R_f(X) - R_i(X)) - 2$. Cuando sumamos estas cotas sobre todos los pasos que comprenden un ensanchamiento, la suma da lugar a la cota deseada. Estas cotas se prueban de forma separada en los Teoremas del 21.5 al 21.7. Entonces, la demostración del Teorema 21.2 se completa aplicando un razonamiento de sumas solapadas.

Para un paso zig, $\Delta \leqslant 3(R_f(X) - R_i(X))$.

Como se ha mencionado antes en esta sección, los únicos nodos cuyos rangos cambian en un paso zig son X y P. En consecuencia, el cambio de potencial es $R_f(X) - R_i(X) + R_f(P) - R_i(P)$. De la Figura 21.4 deducimos $S_f(P) < S_i(P)$; por lo que se sigue que $R_f(P) - R_i(P) < 0$. En consecuencia, el cambio de potencial satisface $\Delta \leqslant R_f(X) - R_i(X)$. Ya que $S_f(X) > S_i(X)$, se sigue que $R_f(X) - R_i(X) > 0$, por lo que $\Delta \leqslant 3(R_f(X) - R_i(X))$.

Los pasos zig-zag y zig-zig son más complicados, porque se ven afectados los rangos de tres nodos. Probamos primero el caso zig-zag.

Para un paso zig-zag, $\Delta \leq 3(R_f(X) - R_i(X)) - 2$.

Como antes, tenemos tres cambios, por lo que el cambio de potencial viene dado por

$$\Delta = R_f(X) - R_i(X) + R_f(P) - R_i(P) + R_f(G) - R_i(G).$$

A partir de la Figura 21.5 deducimos $S_f(X) = S_i(G)$, por lo que los rangos de ambos nodos deben ser iguales. Entonces obtenemos

$$\Delta = -R_i(X) + R_f(P) - R_i(P) + R_f(G).$$

Además $S_i(P) \geq S_i(X)$, por lo que, $R_i(P) \geq R_i(X)$. Realizando esta sustitución y reorganizando términos obtenemos

$$\Delta \leq R_f(P) + R_f(G) - 2R_i(X). \quad \textbf{(21.6)}$$

A partir de la Figura 21.5, $S_f(P) + S_f(G) \leq S_f(X)$. Aplicando el teorema 21.4, obtenemos $\log S_f(P) + \log S_f(G) \leq 2 \log S_f(X) - 2$, y por la definición del rango, esto se convierte en

$$R_f(P) + R_f(G) \leq 2R_f(X) - 2. \quad \textbf{(21.7)}$$

Sustituyendo la Ecuación 21.7 en la Ecuación 21.6 obtenemos

$$\Delta \leq 2R_f(X) - 2R_i(X) - 2. \quad \textbf{(21.8)}$$

Como en el caso de la rotación zig, $R_f(X) - R_i(X) > 0$, por lo que podemos sumarlo al lado derecho de la Ecuación 21.8, y factorizar, para obtener la desigualdad deseada

$$\Delta \leq 3(R_f(X) - R_i(X)) - 2.$$

Finalmente, probamos la cota para el caso zig-zig.

Para cada paso zig-zig se tiene $\Delta \leq 3(R_f(X) - R_i(X)) - 2$.

Como antes, tenemos tres cambios, por lo que el cambio de potencial viene dado por

$$\Delta = R_f(X) - R_i(X) + R_f(P) - R_i(P) + R_f(G) - R_i(G).$$

De la Figura 21.6 deducimos $S_f(X) = S_i(G)$, por lo que los rangos de ambos nodos son iguales. Obtenemos entonces

$$\Delta = -R_i(X) + R_f(P) - R_i(P) + R_f(G).$$

De forma análoga obtenemos $R_i(P) > R_i(X)$ y $R_f(P) < R_f(X)$. Realizando esta sustitución y reorganizando términos obtenemos

$$\Delta < R_f(X) + R_f(G) - 2R_i(X). \quad \textbf{(21.9)}$$

De la Figura 21.6, $S_i(X) + S_f(G) \leq S_f(X)$, por lo que al aplicar el Teorema 21.4 nos queda

$$R_i(X) + R_f(G) \leq 2R_f(X) - 2. \quad \textbf{(21.10)}$$

Reorganizando la Ecuación 21.10, obtenemos

$$R_f(G) \leq 2R_f(X) - R_i(X) - 2. \quad \textbf{(21.11)}$$

Cuando sustituimos en la Ecuación 21.11 obtenemos

$$\Delta \leq 3(R_f(X) - R_i(X)) - 2.$$

Una vez que hemos obtenido cotas para cada uno de los pasos del ensanchamiento, podemos finalmente completar la demostración del Teorema 21.2.

Sea $R_0(X)$ el rango de X antes del ensanchamiento, y $R_i(X)$ el rango de X después del i-ésimo paso del ensanchamiento. Antes del último paso del ensanchamiento, todos los pasos han sido zig-zag o zig-zig. Supongamos que hay k de estos pasos. Entonces el número total de rotaciones realizadas en este punto es 2k. El cambio total del potencial es

$$\Sigma_{i=1}^{k} (3(R_i(X) - R_{i-1}(X)) - 2).$$

Esta suma nos conduce a $3(R_k(X) - R_0(X)) - 2k$. En este punto, el número total de rotaciones más el cambio total de potencial está acotado por $3R_k(X)$, ya que 2k términos se cancelan y el rango inicial de X es no negativo. Si la última rotación es un zig-zig o un zig-zag, entonces un nuevo proceso de sumas solapadas nos conduce a un total de 3R(raíz). Observemos que aquí, el -2 en el incremento de potencial cancela dos rotaciones. Por su parte, esto no ocurre en el caso zig, por lo que obtendríamos 3R(raíz) + 1. Ya que, en el peor caso, el rango de la raíz es $\log N$, el número total de rotaciones más el cambio de potencial durante el ensanchamiento es, como mucho, $3 \log N + 1$.

La demostración de la cota de los árboles de ensanchamiento, aunque compleja, ilustra algunos puntos interesantes. Primero, el caso zig-zig es aparentemente el más costoso, ya que contribuye con la constante dominante tres, mientras que el caso zig-zag lo hace con dos. La demostración no funcionaría si intentáramos aplicarla al caso de la rotación hacia la raíz. Esto se debe a que, en el caso zig, el número de rotaciones más el cambio de potencial es $R_f(X) - R_i(X) + 1$. El 1 final no se elimina al realizar sumas solapadas, por lo que no podríamos probar una cota logarítmica. Esto es correcto, pues ya sabíamos que en este caso no era posible obtener una cota logarítmica.

La técnica de análisis amortizado es muy interesante, y se han desarrollado algunos principios generales para formalizar el marco. En la bibliografía se pueden encontrar más detalles al respecto.

21.5 Árboles de ensanchamiento descendente

Una implementación directa de la estrategia de ensanchamiento ascendente requiere una pasada hacia abajo para realizar el acceso seguida de una segunda pasada hacia arriba. Esto puede hacerse, bien manteniendo referencias al padre, almacenando el camino de acceso en una pila, o utilizando un truco más inteligente para almacenar el camino, utilizando referencias disponibles en los nodos accedidos. Desgraciadamente, todos estos métodos requieren una sustancial sobrecarga, siendo necesario tratar muchos casos especiales. Recuerde, de la Sección 18.5, que es mejor implementar los árboles de búsqueda utilizando una única pasada de arriba abajo. Para evitar casos especiales podemos utilizar nodos fantasma. Esta sección describe una estructura de *árbol de ensanchamiento descendente* que preserva la cota amortizada logarítmica. El procedimiento descendente es más rápido en la práctica y utiliza sólo un espacio extra constante. Es el método recomendado por los creadores de los árboles de ensanchamiento.

La idea básica en la que se basan los árboles de ensanchamiento descendente es que al descender en el árbol durante la búsqueda de un nodo X, debemos tomar los nodos que encontramos en el camino de acceso y moverlos a ellos y a sus subárboles del camino. Además, para garantizar la cota amortizada debemos realizar algunas rotaciones.

En cualquier instante en medio del ensanchamiento, hay un nodo actual X que es la raíz de su subárbol; éste se representa en los diagramas como el árbol central. El árbol L almacena los nodos menores que X, mientras que el árbol D almacena los nodos que son mayores que X. Inicialmente, X es la raíz del árbol T, y L y R son vacíos. Descendiendo por el árbol dos niveles cada vez, encontramos un par de nodos. Dependiendo de si esos dos nodos son menores o mayores que X, se colocan en L o R, junto con los subárboles que no están en el camino de acceso. Así, el nodo actual en el camino de búsqueda es siempre la raíz del árbol central. Cuando finalmente alcanzamos X, podemos unir L y R al fondo del árbol central. Como resultado, X se habrá movido a la raíz. Entonces, la cuestión es cómo se colocan los nodos en L y R, y cómo se realiza la concatenación final. Esto es lo que ilustran los árboles de la Figura 21.9. Como de costumbre, se omiten los casos simétricos.

En todos los dibujos, X es el nodo actual, Y su hijo y Z su nieto (en caso de que exista un nodo aplicable; el significado preciso de «aplicable» se hará más claro en la discusión del caso zig.)

Si la rotación debiera ser un zig, entonces el árbol cuya raíz es Y se convertiría en la nueva raíz del árbol central. X y el subárbol B se enlazan dando lugar al hijo izquierdo del menor elemento de R; y el hijo izquierdo de X se hace `null` [3]. Como resultado, X es el nuevo menor elemento de R, lo que hace que nuevas adiciones sean más fáciles.

Observemos que no es necesario que Y sea una hoja para que se aplique el caso zig. Si el nodo solicitado se encuentra en Y, el caso zig se aplicará, incluso si Y tiene hijos. El caso zig también se aplica si el elemento solicitado es menor que Y e Y no tiene hijo izquierdo, aunque Y tenga hijo derecho, y también en los casos simétricos.

[3] ntado aquí, el nodo más pequeño de R no tiene una referencia izquierda `null`, porque no es necesario

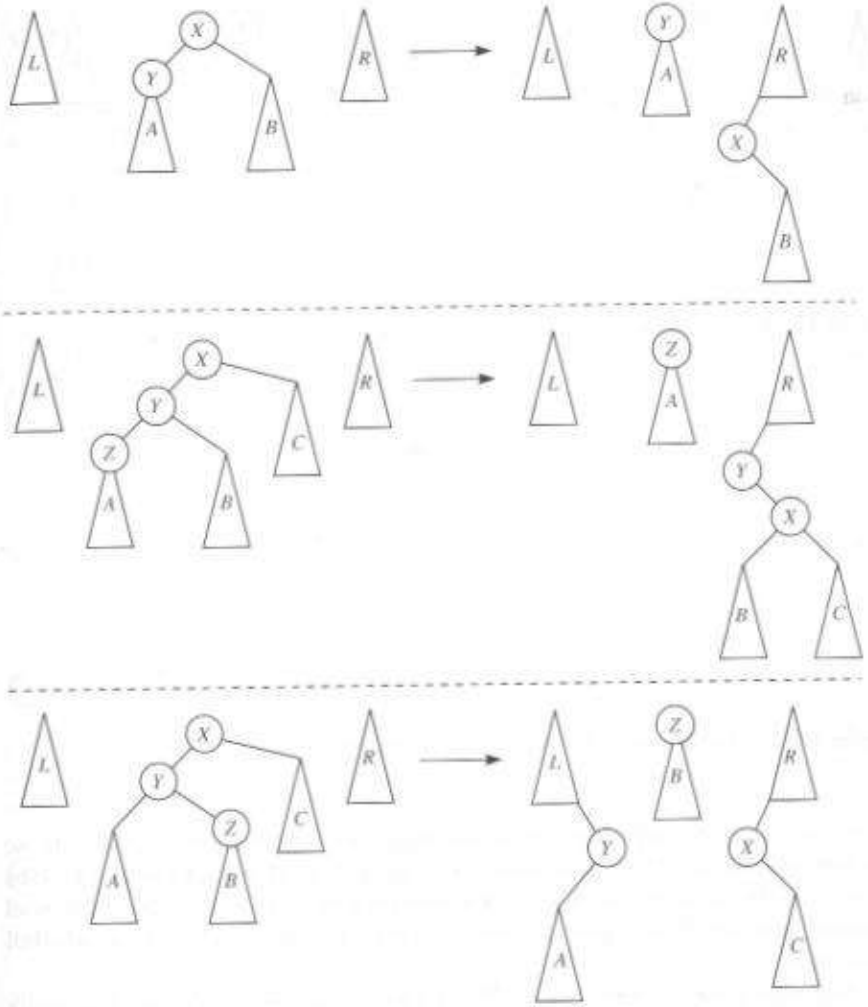


Figura 21.9 Rotaciones del ensanchamiento descendente: zig (arriba), zig-zig (en el centro) y zig-zag (abajo).

Algo similar ocurre en el caso zig-zig. El punto crucial es que se realiza una rotación entre X e Y. El caso zig-zag lleva el elemento Z del fondo arriba en el árbol central y enlaza los subárboles X e Y a R y L, respectivamente. Observemos que Y se convierte en el elemento mayor de L.

El caso zig-zag puede simplificarse algo porque no se realizan rotaciones. En vez de hacer que Z pase a ser la raíz del árbol central, tomaremos Y como raíz. Esto se muestra en la Figura 21.10. Esto simplifica el código, pues la acción en el caso zig-zag se hace idéntica al caso zig. Esto puede parecer ventajoso, ya que comprobar en qué estado estamos consume tiempo. La desventaja es que descender sólo un nodo da lugar a más iteraciones en el procedimiento de ensanchamiento.

Una vez hemos realizado el último paso del ensanchamiento, L, R y el árbol central se combinan para formar un único árbol, como muestra la Figura 21.11. Observe que el resultado es diferente al obtenido por el ensanchamiento ascendente. Sin embargo, las cotas amortizadas $O(\log N)$ se conservan (*véase* el Ejercicio 21.3).

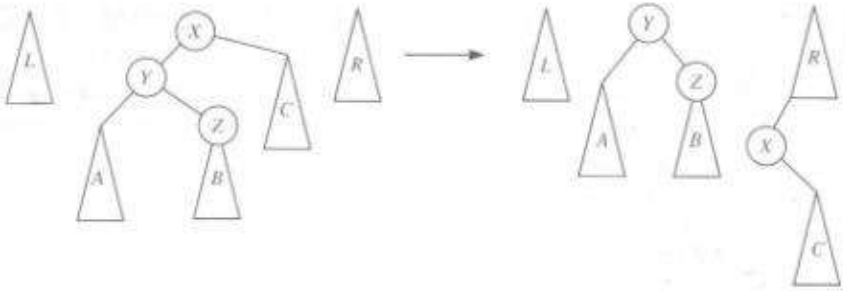


Figura 21.10 Zig-zag descendente simplificado.

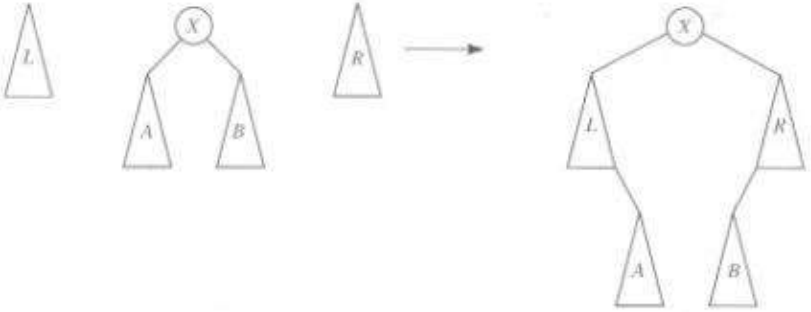


Figura 21.11 Montaje final del ensanchamiento descendente.

Un ejemplo del algoritmo de ensanchamiento descendente simplificado se muestra en la Figura 21.12. Intentamos acceder al 19 en el árbol. El primer paso es un zig-zag. De acuerdo con una versión simétrica de la Figura 21.20, traemos el subárbol con raíz 25 a la raíz del árbol central y enlazamos el 12 y su subárbol izquierdo a *L*.

A continuación, hacemos un zig-zig: el 15 se eleva a la raíz del árbol central y se realiza una rotación entre el 20 y el 25, enlazándose el árbol resultado a *R*. La búsqueda del 19 acaba con un zig. La nueva raíz del árbol central es 18, y 15 y su subárbol izquierdo se enlazan como hijo derecho del nodo mayor de *L*. El montaje, de acuerdo con la Figura 21.11, concluye el proceso de ensanchamiento.

21.6 Implementación de los árboles con ensanchamiento descendente

El esqueleto de la clase de los árboles con ensanchamiento se muestra en la Figura 21.13. Para eliminar molestos casos especiales, mantenemos un centinela estático `nodoNulo`. El centinela se coloca en memoria y se inicializa durante la carga de la clase, como muestra la Figura 21.14.

La Figura 21.15 muestra el método de inserción de un elemento `x`. Un nuevo nodo (`nodoNuevo`) se coloca en memoria y si el árbol es vacío se crea un árbol con un nodo. En otro caso, ensanchamos desde `x`. Si el elemento en la nueva raíz

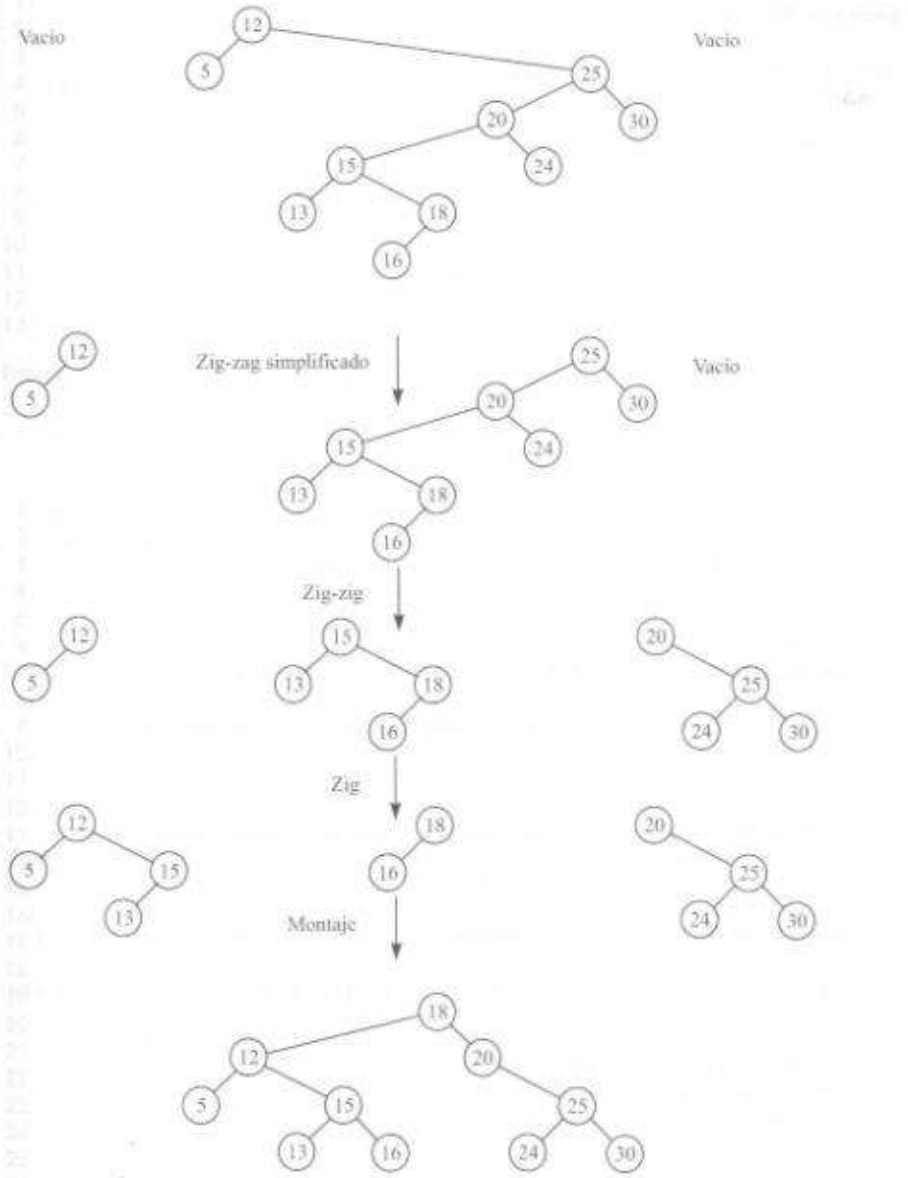


Figura 21.12 Pasos de un ensanchamiento descendente (accediendo al 19 del primer árbol).

del árbol es igual a `x`, tenemos un duplicado. En este caso, no queremos insertar `x` por lo que se lanza una excepción. Si la nueva raíz contiene un valor que es mayor que `x`, entonces la nueva raíz y su subárbol derecho se convierten en el un subárbol derecho de `nodoNuevo` y el subárbol izquierdo de la raíz se convierte en el subárbol izquierdo de `nodoNuevo`. Se aplica un razonamiento similar si la nueva raíz contiene un valor menor que `x`. En cualquier caso, `nodoNuevo` se asigna a `raiz` para indicar que es la nueva raíz.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase ArbolEnsanchamiento
7  //
8  // CONSTRUCCION: sin inicializador
9  //
10 // ******************OPERACIONES PÚBLICAS************************
11 // void insertar( x )       --> Inserta x
12 // void eliminar( x )       --> Elimina x
13 // void eliminarMin( )      --> Elimina el menor elemento
14 // Comparable buscar( x )   --> Devuelve el elemento que casa con x
15 // Comparable buscarMin( )  --> Devuelve el menor elemento
16 // Comparable buscarMax( )  --> Devuelve el mayor elemento
17 // boolean esVacio( )       --> Devuelve true si vacío; si no, false
18 // void vaciar( )           --> Elimina todos los elementos
19 // void imprimirArbol( )    --> Imprimir el árbol en orden
20 // ************************ERRORES*******************************
21 // Muchas rutinas lanzan ElementoNoEncontrado en casos degenerados
22 // insertar lanza ElementoDuplicado si el elemento ya está
23
24 /**
25  * Implementa un árbol con ensanchamiento descendente.
26  * Las comparaciones se basan en el método compara.
27  */
28 public class ArbolEnsanchamiento implements ArbolBusqueda
29 {
30     public ArbolEnsanchamiento ( )
31       { /* Figura 21.14 */ }
32     public void insertar( Comparable x ) throws ElementoDuplicado
33       { /* Figura 21.15 */ }
34     public void eliminar( Comparable x ) throws ElementoNoEncontrado
35       { /* Figura 21.16 */ }
36     protected NodoBinario ensanchamiento( Comparable x, NodoBinario t )
37       { /* Figura 21.18 */ }
38     public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
39       { /* Figura 21.17 */ }
40     public void eliminarMin( ) throws ElementoNoEncontrado
41       { eliminar( buscarMin( ) ); }
42     public boolean esVacio( )
43       { return raiz == nodoNulo; }
44     public void vaciar( )
45       { raiz = nodoNulo; }
46
47     // buscarMin, buscarMax e imprimirArbol están disponibles en Internet
48
49     private NodoBinario raiz;
50 }
```

Figura 21.13 Esqueleto de la clase de árboles con ensanchamiento descendente.

La Figura 21.16 muestra el algoritmo de eliminación de un árbol de ensanchamiento. Puede resultar chocante que el procedimiento de eliminación sea más corto que el correspondiente procedimiento de inserción. El método `buscar` se muestra en la Figura 21.17. Después del ensanchamiento, el elemento solicitado o bien es la raíz o no está en el árbol, por lo que el método es muy corto. Lo único

```
1  /**
2   * Construye el árbol.
3   */
4  public ArbolEnsanchamiento( )
5  {
6      raiz = nodoNulo;
7  }
8  private static NodoBinario nodoNulo;
9      static               // Inicializador estático para nodoNulo
10     {
11         nodoNulo = new NodoBinario( null );
12         nodoNulo.izquierdo = nodoNulo.derecho = nodoNulo;
13     }
```

Figura 21.14 Construcción del árbol de ensanchamiento, incluyendo la inicialización estática.

```
1  /**
2   * Inserta en el árbol.
3   * @param x el elemento a insertar.
4   * @exception ElementoDuplicado si un elemento que casa
5   *       con x ya está en el árbol.
6   */
7  public void insertar( Comparable x ) throws ElementoDuplicado
8  {
9      NodoBinario nodoNuevo = new NodoBinario( x );
10
11     if( raiz == nodoNulo )
12     {
13         nodoNuevo.izquierdo = nodoNuevo.derecho = nodoNulo;
14         raiz = nodoNuevo;
15     }
16     else
17     {
18         raiz = ensanchamiento( x, raiz );
19         if( x.menorQue( raiz.dato ) )
20         {
21             nodoNuevo.izquierdo = raiz.izquierdo;
22             nodoNuevo.derecho = raiz;
23             raiz.izquierdo = nodoNulo;
24             raiz = nodoNuevo;
25         }
26         else
27         if( raiz.dato.menorQue( x ) )
28         {
29             nodoNuevo.derecho = raiz.derecho;
30             nodoNuevo.izquierdo = raiz;
31             raiz.derecho = nodoNulo;
32             raiz = nodoNuevo;
33         }
34         else
35             throw new ElementoDuplicado( "Insertar en el árbol" );
36     }
37 }
```

Figura 21.15 Inserción en el árbol con ensanchamiento descendente.

```
/**
 * Eliminación del árbol.
 * @param x el elemento a eliminar.
 * @exception ElementoNoEncontrado si no hay un
 *            elemento que case con x en el árbol.
 */
public void eliminar( Comparable x ) throws ElementoNoEncontrado
{
    NodoBinario nuevoArbol;

        // Si x se encuentra, estará en la raíz
    raiz = ensanchamiento( x, raiz );
    if( raiz.dato.compara( x ) != 0 )
        throw new ElementoNoEncontrado( "Eliminar del árbol" );

    if( raiz.izquierdo == nodoNulo )
        nuevoArbol = raiz.derecho;
    else
    {
        // Buscar el máximo del subárbol izquierdo
        // Ensancharlo hacia la raíz; enlazar hijo derecho
        nuevoArbol = raiz.izquierdo;
        nuevoArbol = ensanchamiento( x, nuevoArbol );
        nuevoArbol.derecho = raiz.derecho;
    }
    raiz = nuevoArbol;
}
```

Figura 21.16 Eliminación en el árbol de ensanchamiento descendente.

```
/**
 * Buscar un elemento en el árbol.
 * @param x el elemento a buscar.
 * @return el elemento encontrado.
 * @exception ElementoNoEncontrado si no hay un
 *            elemento que case con x en el árbol.
 */
public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
{
    raiz = ensanchamiento( x, raiz );

    if( esVacio( ) || raiz.dato.compara( x ) != 0 )
        throw new ElementoNoEncontrado( "Buscar en el árbol" );

    return raiz.dato;
}
```

Figura 21.17 Método buscar de los árboles de ensanchamiento.

que nos falta es la rutina de ensanchamiento descendente. Se implementa como método protegido para que, si se desea, nuestra clase pueda ser extendida con un algoritmo de ensanchamiento diferente.

Nuestra implementación, mostrada en la Figura 21.18, utiliza una cabecera con referencias izquierda y derecha que a la postre contendrá las raíces de los árboles izquierdo y derecho. `cabecera` es un atributo estático, ya que queremos colocarlo en memoria sólo una vez durante la secuencia de ensanchamientos. Estos árboles

```
private static NodoBinario cabecera = new NodoBinario( null );

/**
 * Método interno para realizar el ensanchamiento.
 * El último nodo accedido se convierte en la raíz.
 * @param x el elemento al que se refiere el ensanchamiento.
 * @param t la raíz del subárbol que hay que ensanchar.
 * @return el subárbol después del ensanchamiento.
 */
protected NodoBinario ensanchamiento( Comparable x, NodoBinario t )
{
    NodoBinario maxArbolIzq, minArbolDer;

    cabecera.izquierdo = cabecera.derecho = nodoNulo;
    maxArbolIzq = minArbolDer = cabecera;

    nodoNulo.dato = x; // Garantiza un emparejamiento

    for( ; ; )
        if( x.menorQue( t.dato ) )
        {
            if( x.menorQue( t.izquierdo.dato ) )
                t = Rotaciones.conHijoIzquierdo( t );
            if( t.izquierdo == nodoNulo )
                break;
            // Enlazar derecho
            minArbolDer.izquierdo = t;
            minArbolDer = t;
            t = t.izquierdo;
        }
        else if( t.dato.menorQue( x ) )
        {
            if( t.derecho.dato.menorQue( x ) )
                t = Rotaciones.conHijoDerecho( t );
            if( t.derecho == nodoNulo )
                break;
            // Enlazar izquierdo
            maxArbolIzq.derecho = t;
            maxArbolIzq = t;
            t = t.derecho;
        }
        else
            break;

    maxArbolIzq.derecho = t.izquierdo;
    minArbolDer.izquierdo = t.derecho;
    t.izquierdo = cabecera.derecho;
    t.derecho = cabecera.izquierdo;
    return t;
}
```

Figura 21.18 Algoritmo de ensanchamiento descendente.

son inicialmente vacíos, por lo que la cabecera se utiliza para señalar al mínimo y máximo nodo del árbol izquierdo y derecho, respectivamente, en este estado inicial. De esta manera, el código puede evitar comprobaciones de árboles vacíos. La primera vez que el árbol izquierdo deja de ser vacío, la referencia derecha de la cabecera se inicializará y no cambiará en el futuro. Por tanto, contendrá la raíz

del hijo izquierdo al final de la búsqueda descendente. De forma similar, la referencia izquierda de la cabecera contendrá a la postre la raíz del árbol derecho.

Antes del montaje al final del ensanchamiento, `cabecera.izquierdo` y `cabecera.derecho` referenciarán a *R* y *L*, respectivamente (esto no es un error tipográfico, siga los enlaces). Tenga en cuenta que estamos utilizando el ensanchamiento simplificado.

21.7 Comparaciones de los árboles de ensanchamiento con otros árboles de búsqueda

La implementación en la sección anterior sugiere que los árboles de ensanchamiento no son tan complicados como los árboles rojinegros y son casi tan simples como los AA-árboles. ¿Merece la pena utilizarlos? La respuesta definitiva está aún por obtener, pero parece que si los accesos no son aleatorios, los árboles de ensanchamiento tienen en efecto muy buen rendimiento en la práctica. Es más, algunas propiedades relativas a su buen rendimiento pueden probarse analíticamente. Los accesos no aleatorios incluyen aquéllos que siguen la regla del 90-10, así como diversos casos especiales como accesos secuenciales, accesos por los dos extremos y patrones de accesos que son típicos de las colas de prioridad durante ciertos tipos de simulación de eventos. Los ejercicios le piden estudiar esto con más detalle.

Sin embargo, los árboles de ensanchamiento no son perfectos. Un problema es que, debido al ensanchamiento, la operación `buscar` es costosa. En consecuencia, cuando las secuencias de accesos son aleatorias y uniformes, los árboles de ensanchamiento no son tan buenos como otros árboles equilibrados.

Resumen

Este capítulo describe los árboles de ensanchamiento, los cuales son una reciente alternativa a los árboles de búsqueda equilibrados. Los árboles de ensanchamiento tienen algunas propiedades notables que pueden ser demostradas, incluyendo un coste amortizado logarítmico por operación. Otras propiedades se indican en los ejercicios. Algunos estudios han sugerido que los árboles de ensanchamiento pueden utilizarse en un amplio rango de aplicaciones debido a su aparente habilidad de adaptarse a las secuencias de accesos sencillas.

El próximo capítulo describe dos estructuras de colas de prioridad que, como los árboles de ensanchamiento, tienen rendimientos pobres en el caso peor, pero buenos rendimientos amortizados. Una de éstas, el montículo de emparejamientos, parece ser una elección excelente en algunas aplicaciones.

Elementos del juego

análisis amortizado Método de análisis que acota el coste de una secuencia de operaciones y distribuye equitativamente el coste entre las operaciones de la secuencia.

árbol de ensanchamiento ascendente Árbol en el cual los elementos son rotados hacia la raíz utilizando un método algo más complicado que el utilizado en la rotación sencilla hacia la raíz.

árbol de ensanchamiento descendente Tipo de árbol de ensanchamiento más eficiente en la práctica que su contrapartida ascendente, como ocurrió en el caso de los árboles rojinegros.

ensanchamiento Estrategia de rotación hacia la raíz que permite obtener una cota amortizada logarítmica.

estrategia de rotación hacia la raíz Estrategia que reorganiza un árbol binario de búsqueda después de cada acceso de forma que desplaza los elementos más frecuentemente accedidos cerca de la raíz.

función de potencial Herramienta de contabilidad utilizada para probar una cota amortizada.

rango En el análisis de un árbol de ensanchamiento, el logaritmo del tamaño de un nodo.

regla del 90-10 Regla que afirma que el 90 por ciento de los accesos corresponden al 10 por ciento de los datos. Los árboles de búsqueda equilibrados no sacan partido de esta regla.

zig y **zig-zag** Los casos zig y zig-zag son idénticos a los casos de la rotación hacia la raíz. El caso zig se utiliza cuando *X* es un hijo de la raíz, y el zig-zag cuando *X* es un nodo más interno.

zig-zig El caso zig-zig es exclusivo de los árboles de ensanchamiento. Se utiliza cuando *X* es un nodo externo.

Errores comunes

1. Después de cada acceso se tiene que realizar un ensanchamiento, aun en el caso de los que no tengan éxito, pues si no, no son válidas las cotas de rendimiento.
2. El código es un tanto rebuscado. Sea cuidadoso.

En Internet

La clase de árboles de ensanchamiento está disponible en Internet en el directorio **DataStructures**. El nombre del fichero es el siguiente:

SplayTree.java	Traducido como `ArbolEnsanchamiento.java`, contiene la implementación de la clase de los árboles de ensanchamiento.

Ejercicios

Cuestiones breves

21.1. Muestre el resultado de insertar los valores 3, 1, 4, 5, 2, 9, 6 y 8 en un

a) árbol de ensanchamiento ascendente.
b) árbol de ensanchamiento descendente.

21.2. Muestre el resultado de eliminar el 3 del árbol del Ejercicio 21.1 en ambos casos.

Problemas teóricos

21.3. Demuestre que el coste amortizado de los árboles de ensanchamiento descendente es $O(\log N)$.

21.4. Demuestre que si todos los nodos de un árbol de ensanchamiento son accedidos en orden secuencial, el árbol resultante estará formado por una cadena de hijos izquierdos.

21.5. Supongamos que, en un intento de ahorrar tiempo, ensanchamos sólo en una de cada dos operaciones. ¿Sigue siendo logarítmico el coste amortizado?

21.6. Los nodos del 1 al $N = 1.024$ forman un árbol de ensanchamiento de hijos izquierdos.

a) ¿Cuál es la longitud exacta del camino interno del árbol?
b) Calcule la longitud del camino interno después de las operaciones `buscar(1)`, `buscar(2)`, `buscar(3)`, cuando se utiliza un ensanchamiento ascendente.

21.7. Cambiando la función de potencial, se pueden obtener diferentes cotas para el ensanchamiento. Sea $P(i)$ una función peso asignada a cada nodo del árbol y $S(i)$ la suma de los pesos de todos los nodos en el árbol cuya raíz es i, incluyendo a i. El caso especial $P(i) = 1$ para todos los nodos corresponde a la función utilizada en la demostración de la cota del ensanchamiento. Demuestre los siguientes teoremas:

a) El tiempo total de acceso es $O(M + (M + N)\log N)$.
b) Si q_i es el número total de veces que se accede al nodo i y $q_i > 0$ para todo i, el tiempo de acceso total es $O(M + \Sigma_{i=1}^{N} q_i \log(M/q_i))$.

Problemas prácticos

21.8. Implemente la clase de las colas de prioridad utilizando un árbol de ensanchamiento.

21.9. Modifique los árboles de ensanchamiento para que soporten accesos indexados según el orden de los elementos en el árbol.

Prácticas de programación

21.10. Compare empíricamente los árboles de ensanchamiento descendente que se implementan en la Sección 21.6 con los árboles originales de ensanchamiento ascendente discutidos en la Sección 21.5.

21.11. Al contrario que los árboles de búsqueda equilibrados, los árboles de ensanchamiento incurren en una sobrecarga durante la operación `buscar` que es indeseable si la secuencia de acceso es suficientemente aleatoria. Experimente con una estrategia que ensanche en la operación `buscar` sólo después de rebasar cierta profundidad d en la búsqueda descendente. El ensanchamiento no mueve el elemento buscado a la raíz, sino al punto de profundidad d donde empezó el ensanchamiento.

21.12. Compare empíricamente una implementación de las colas de prioridad utilizando un árbol de ensanchamiento con un montículo binario. Utilice los siguientes modelos de entrada:

a) Inserciones y operaciones `eliminarMin` aleatorias.
b) Operaciones de `insertar` y `eliminarMin` correspondientes a una simulación dirigida por eventos.
c) Operaciones de `insertar` y `eliminarMin` correspondientes al algoritmo de Dijkstra.

21.13. Diseñe un applet que ilustre las operaciones de los árboles de ensanchamiento. Puede utilizar el ensanchamiento ascendente o descendente.

Bibliografía

Los árboles de ensanchamiento se describen en el artículo [3]. El concepto de análisis amortizado se discute en el artículo [4] y también con gran detalle en [5]. Una comparación entre los árboles de ensanchamiento y los árboles AVL se presenta en [1]. [2] muestra que los árboles de ensanchamiento se comportan bien en algunos tipos de simulación dirigida por eventos.

1. J. Bell y G. Gupta, «An Evaluation of Self-Adjusting Binary Search Tree Techniques», *Software-Practice and Experience* **23** (1993), 369-382.
2. D. W. Jones, «An Empirical Comparison of Priority-Queue and Event-Set Implementations», *Communications of the ACM* **29** (1986), 300-311.
3. D. D. Sleator y R. E. Tarjan, «Self-adjusting Binary Search Trees», *Journal of te ACM* **32** (1985), 652-686.
4. R. E. Tarjan, «Amortized Computational Complexity», *SIAM Journal on Algebraic and Discrete Methods* **6** (1985), 306-318.
5. M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, Benjamin/Cummings Publishing Co., Redwood City, Calif. (1994).

CAPÍTULO

22 Colas de prioridad con mezcla

Este capítulo examina las colas de prioridad que soportan una operación adicional, `mezclar`. La operación `mezclar`, muy importante en el diseño avanzado de algoritmos, combina dos colas de prioridad en una (y destruye las iniciales). Representaremos las colas de prioridad como árboles generales. Esto hace que la operación `reducirClave` sea algo más simple, lo que es importante en algunas aplicaciones.

En este capítulo veremos:

- Una discusión acerca de los *montículos sesgados*, que no son más que colas de prioridad combinables implementadas mediante árboles binarios.
- Una discusión acerca de los *montículos de emparejamientos*, que son colas de prioridad combinables basadas en los árboles M-arios. Los montículos de emparejamientos son una alternativa práctica a los montículos binarios, aun cuando la operación `mezclar` no sea necesaria.

22.1 Los montículos sesgados

Un *montículo sesgado* es un tipo particular (en el sentido que precisaremos más adelante) de árbol binario con ordenación de montículos. En dichos árboles no se impone en principio ninguna restricción estructural, por lo que, a diferencia de lo que sucede en los montículos o en los árboles binarios de búsqueda equilibrados, no existe ninguna garantía de que la profundidad del árbol sea logarítmica. De esta forma, los montículos sesgados resultan similares a los árboles de ensanchamiento.

22.1.1 La mezcla es importante

Si para representar una cola de prioridad empleamos un árbol binario con orden y no restringido estructuralmente, entonces la operación de mezclar se convierte en una rutina imprescindible. Esto es debido a que a partir de ella podemos realizar otras operaciones como las siguientes:

- `h.insertar( x )`: crea un árbol con un solo nodo que contiene a `x` y lo combina con la cola de prioridad.
- `h.buscarMin( )`: devuelve el elemento que se encuentra en la raíz.
- `h.eliminarMin( )`: elimina la raíz y mezcla sus hijos izquierdo y derecho.
- `h.reducirClave( p, nuevoValor )`: asumiendo que `p` es una referencia a un nodo de la cola de prioridad, podemos disminuir apropiadamente la clave de `p` y desligarlo de su padre. Esto nos devuelve dos colas de prioridad que pueden ser combinadas. Observe que `p` (que significa posición) no cambia (al contrario que en la operación equivalente de los montículos binarios).

En consecuencia, sólo es necesario mostrar cómo implementar la mezcla, ya que las otras operaciones resultan triviales. La operación `reducirClave` es muy importante en algunas aplicaciones avanzadas. Mostramos una de ellas en la Sección 14.3, en la que se discutió el algoritmo de Dijkstra para calcular el camino mínimo en un grafo. La operación `reducirClave` no se emplea en nuestra implementación ya que es complicado mantener la posición de cada elemento dentro del montículo binario. En un montículo de emparejamientos, la posición puede mantenerse como una referencia al correspondiente nodo del árbol, y a diferencia de lo que sucedía en los montículos binarios, la posición nunca cambia.

Esta sección discute una implementación de las colas de prioridad con mezcla que emplea árboles binarios: los montículos sesgados. En primer lugar, se muestra que si no nos importa demasiado la eficiencia, la mezcla de dos árboles con ordenación de montículos es sencilla. A continuación, se realiza una pequeña modificación (el montículo sesgado) que evita las ineficiencias obvias del algoritmo original. Por último, se realiza una demostración de que la operación `mezclar` para los montículos sesgados es logarítmica en coste amortizado y se comenta la relevancia práctica de este resultado.

22.1.2 Mezcla simple de árboles con ordenación de montículos

Supongamos que tenemos dos árboles con orden, H_1 y H_2, que debemos mezclar. Claramente, si alguno de los dos árboles es vacío, el otro es el resultado de la mezcla. En caso contrario, para mezclar los dos árboles comparamos sus raíces. La mezcla se realiza de forma recursiva combinando el árbol de mayor raíz con el subárbol derecho [1] del árbol de menor raíz.

La Figura 22.1 muestra que el efecto de esta estrategia recursiva es que los caminos derechos de las dos colas de prioridad se combinan para formar la nueva cola de prioridad. Todos los nodos del camino derecho mantienen su subárbol izquierdo inicial, y sólo cambian los nodos del camino derecho. Observe que en realidad los árboles que aparecen como argumento en el ejemplo mostrado en la Figura 22.1 no podrían obtenerse empleando únicamente inserciones y mezclas, ya que, tal y como hemos comentado, durante una mezcla no pueden añadirse hijos izquierdos. El efecto consiguiente es que, lo que aparenta ser un árbol binario con ordenación de montículos es en realidad una secuencia ordenada que se corresponde con el camino derecho del árbol. De este modo, todas las operaciones tienen coste lineal.

demos emplear cualquiera de los subárboles. Aquí se emplea el derecho por simplicidad.

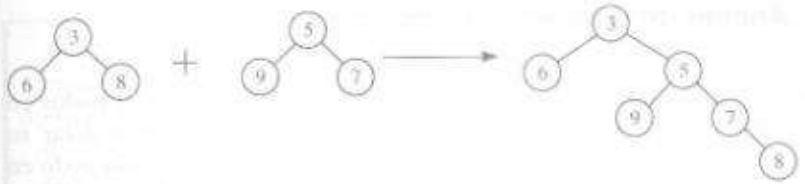


Figura 22.1 Mezcla simplista de dos árboles con ordenación de montículos: se mezclan los caminos derechos.

Afortunadamente, una sencilla modificación asegura que el camino derecho no va a crecer más.

22.1.3 El montículo sesgado: una modificación sencilla

La mezcla en la Figura 22.1 crea un árbol de mezcla temporal. Una modificación simple es la siguiente: antes de finalizar la mezcla, intercambiamos el hijo derecho e izquierdo de cada nodo del camino derecho del árbol temporal. En principio, sólo aquellos nodos que estaban en los caminos derechos iniciales estarán en el camino derecho del árbol temporal. Como resultado del intercambio, mostrado en la Figura 22.2, estos nodos formarán el camino izquierdo del árbol resultante. Cuando se realizan las mezclas de este modo, el tipo de árboles con ordenación de montículos resultante se conoce como *montículo sesgado*.

Una descripción recursiva es como sigue: sea L el árbol con la raíz más pequeña y R el otro árbol. Se realiza entonces el siguiente proceso:

1. Si uno de los árboles es vacío, el otro es el resultado de la mezcla.
2. En caso contrario, tomamos como *Temp* el subárbol derecho de L.
3. El subárbol izquierdo de L se convierte en su nuevo subárbol derecho.
4. Se calcula el resultado de la mezcla recursiva de *Temp* y R, que se convierte en el nuevo subárbol izquierdo de L.

Por medio del intercambio de hijos confiamos en que la longitud del camino derecho no crecerá excesivamente. Por ejemplo, si mezclamos un par de árboles con un único camino derecho prolongado, los nodos que intervienen en dichos caminos no reaparecerán en ningún camino derecho, al menos durante un tiempo. Aunque es posible obtener árboles en los que cada nodo aparece en un camino derecho, esto sólo puede obtenerse como resultado de un gran número de mezclas relativamente eficientes. En la siguiente sección, demostramos rigurosamente este hecho estableciendo que el coste amortizado de la operación de mezcla es logarítmico.

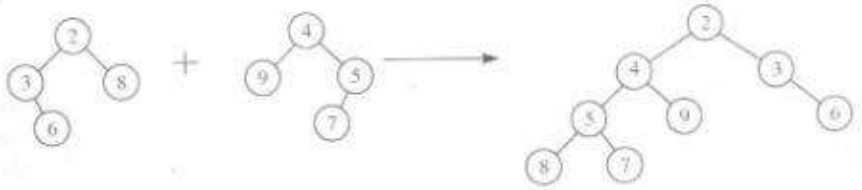


Figura 22.2 Mezcla de montículos sesgados: se mezclan los caminos derechos y el resultado se convierte en el camino izquierdo.

22.1.4 Análisis del montículo sesgado

Supongamos que tenemos dos montículos, H_1 y H_2, que tienen r_1 y r_2 nodos en sus respectivos caminos derechos. Entonces el tiempo empleado en realizar su mezcla es proporcional a $r_1 + r_2$. Cuando pagamos una unidad por cada nodo en los caminos derechos, el coste de una mezcla es proporcional a lo abonado. Como los árboles no tienen ninguna restricción estructural, todos los nodos en ambos árboles pueden caer en sus caminos derechos. Esto genera una cota $\Theta(N)$ para el caso peor de la mezcla (el Ejercicio 22.4 pide la construcción de un ejemplo). En contraposición, como mostraremos muy pronto, el tiempo amortizado de la mezcla de montículos sesgados es $O(\log N)$.

Al igual que en los árboles de ensanchamiento, la demostración se hace introduciendo una función de potencial que cancela el coste variable de las operaciones de los montículos sesgados. Queremos que la función de potencial aumente en $O(\log N) - (r_1 + r_2)$ de modo que el coste total de una mezcla y el cambio de potencial sólo sea $O(\log N)$. Si el potencial es mínimo antes de la primera operación, aplicando la técnica de sumas solapadas tenemos garantizado que el gasto total de M operaciones es $O(M \log N)$, igual que en los árboles de ensanchamiento.

Lo que se necesita entonces es alguna clase de función de potencial que capture el efecto de las operaciones de los montículos sesgados. Encontrar una función tal es bastante complicado. Sin embargo, una vez encontrada, la demostración es relativamente corta.

DEFINICIÓN: Un nodo p es *pesado* si el tamaño de su subárbol derecho es mayor que el tamaño del subárbol izquierdo; en caso contrario, se dice que es *ligero*. En particular, cuando los dos subárboles de un nodo tienen el mismo tamaño tenemos que es ligero.

En la Figura 22.3, antes de la mezcla, los nodos 3 y 4 son pesados. Después de la mezcla, sólo el nodo 3 es pesado. Se pueden comprobar fácilmente tres hechos. En primer lugar, como resultado de la mezcla sólo los nodos del camino derecho pueden ver modificado su carácter pesado/ligero, ya que ningún otro nodo ve alterados sus subárboles. En segundo lugar, toda hoja es ligera. En tercer lugar, el número de nodos ligeros en el camino derecho de un nodo N es, a lo sumo, $\lfloor \log N \rfloor + 1$. Esto es debido a que el hijo derecho de un nodo ligero tiene menos de la mitad del tamaño que su padre, por lo que puede aplicarse el principio de la división por la mitad. El $+1$ adicional viene como resultado de que las hojas sean ligeras. A partir de estos preliminares, podemos demostrar los Teoremas 22.1 y 22.2.

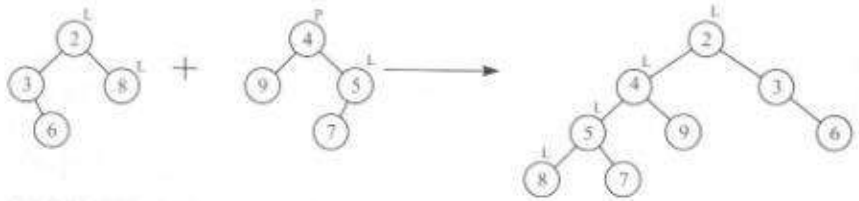


Figura 22.3 Cambio en el estado pesado/ligero después de la mezcla.

Sean H_1 y H_2 dos montículos sesgados, con N_1 y N_2 nodos, respectivamente, y sea N su tamaño combinado (esto es, $N_1 + N_2$). Supongamos que el camino derecho de H_1 tiene l_1 nodos ligeros y h_1 nodos pesados, es decir $l_1 + h_1$ nodos en total. Por su parte, el camino derecho de H_2 tiene l_2 nodos ligeros y h_2 nodos pesados, es decir $l_2 + h_2$ nodos en total. Si se define el potencial como el número total de nodos pesados en la colección de montículos sesgados que estamos manejando, entonces el coste de la mezcla es, a lo sumo, $2 \log N + (h_1 + h_2)$, pero el cambio en el potencial es, como mucho, $2 \log N - (h_1 + h_2)$.

El coste de la mezcla es el número de nodos en los caminos derechos, o sea, $l_1 + l_2 + h_1 + h_2$. Como el número de nodos ligeros es logarítmico, $l_1 \leq \lfloor \log N_1 \rfloor + 1$ y $l_2 \leq \lfloor \log N_2 \rfloor + 1$. De modo que $l_1 + l_2 \leq \log N_1 + \log N_2 + 2 \leq 2 \log N$, donde la última desigualdad se obtiene a partir del Teorema 21.4. Con todo ello, el coste de la mezcla es, a lo sumo, $2 \log N + (h_1 + h_2)$. La cota del cambio de potencial se deduce del hecho de que sólo los nodos que intervienen en la mezcla pueden cambiar su estado pesado/ligero, y de que cualquier nodo pesado del camino debe convertirse en ligero, ya que sus hijos se intercambian. Incluso si todos los nodos ligeros se convirtieran en pesados, el cambio de potencial estaría aún limitado por $l_1 + l_2 - (h_1 + h_2)$. Empleando el argumento anterior, esto es, a lo sumo, $2 \log N - (h_1 + h_2)$.

El coste amortizado de los montículos sesgados para las operaciones `mezclar`, `insertar` *y* `eliminarMin` *es, a lo sumo, $4 \log N$.*

Sea Φ_i el potencial de la colección de montículos sesgados inmediatamente después de la i-ésima operación. Observe que $\Phi_0 = 0$ y $\Phi_i \geq 0$. Una inserción comienza por crear un árbol con un único nodo, cuya raíz es, por definición, ligera. De este modo, no se altera el potencial anterior a la mezcla subsiguiente. Una operación `eliminarMin` *comienza por desechar la raíz de un árbol, y como consecuencia de lo cual no puede aumentar el potencial (podría, de hecho, reducirse). Tras ello se produce la mezcla de sus hijos. En definitiva, todo queda reducido a considerar los costes de la mezcla. Sea c_i el coste de la mezcla que se produce como resultado de la i-ésima operación. Entonces $c_i + \Phi_i - \Phi_{i-1} \leq 4 \log N$. Aplicando la técnica de sumas solapadas sobre M operaciones, obtenemos $\sum_{i=1}^{M} c_i \leq 4 M \log N$, ya que la diferencia $\Phi_M - \Phi_0$ no es negativa.*

El montículo sesgado es un destacado ejemplo de una estructura relativamente simple, cuyo análisis no lo es en absoluto. Sin embargo, es cierto que dicho análisis es sencillo de concluir una vez que la función de potencial adecuada ha sido identificada. Desgraciadamente, aún no disponemos de ningún tipo de teoría general que nos permita elegir las funciones de potencial adecuadas para cada problema. Normalmente, es necesario probar con muchas funciones antes de dar con la adecuada.

Un comentario final resulta oportuno. Aunque la descripción inicial del algoritmo emplea recursión y la recursión permite obtener un código más sencillo, ésta no puede emplearse en la práctica. Esto es debido a que el coste lineal en el caso peor puede provocar un desbordamiento de la pila, si implementáramos el algoritmo con recursión. Como consecuencia, debe implementarse la correspondiente versión iterativa del algoritmo. En lugar de alargarnos en el estudio de todos estos detalles técnicos un tanto enojosos, preferimos presentar una estructura de datos alternativa un poco más complicada: el *montículo de emparejamientos*. Esta estructura de datos no ha sido todavía completamente analizada, pero parece comportarse muy bien en la práctica.

22.2 Los montículos de emparejamientos

El *montículo de emparejamientos* es un árbol M-ario con ordenación de montículos, no restringido estructuralmente. Cumple la propiedad de que el coste de todas las operaciones, salvo la eliminación, es constante en el caso peor. Aunque `eliminarMin` es lineal en el caso peor, puede comprobarse que cualquier *secuencia* de operaciones de los montículos de emparejamientos tiene coste amortizado logarítmico. Se ha conjeturado, pero no demostrado, que pueden conseguirse mejores resultados. Más exactamente, que todas las operaciones, excepto `eliminarMin`, tienen coste amortizado constante, mientras que `eliminarMin` tiene coste amortizado logarítmico.

La Figura 22.4 muestra un montículo de emparejamientos abstracto. Su implementación usa la representación primer hijo/siguiente hermano, presentada en el Capítulo 17. La operación `reducirClave`, tal y como mostraremos más tarde, necesita que cada nodo contenga una referencia adicional. Cada nodo que sea el hijo más izquierdo de otro contiene una referencia a su padre; en otro caso, el nodo es un hermano derecho y contiene una referencia a su hermano izquierdo. En la Figura 22.5 se muestra esta representación, en la que la línea más oscura indica que dos referencias (una en cada dirección) conectan los pares de nodos.

22.2.1 Operaciones del montículo de emparejamientos y teoría

En principio, las operaciones básicas de los montículos de emparejamientos son sencillas. Ésta es la razón por la que se comportan bien en la práctica. Para combinar dos montículos de emparejamientos, convertimos el montículo con la raíz mayor en el primer hijo del montículo con la raíz menor. La inserción es un caso

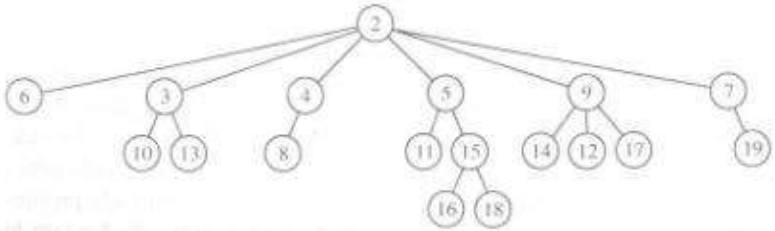


Figura 22.4 Representación abstracta de un montículo de emparejamientos simple.

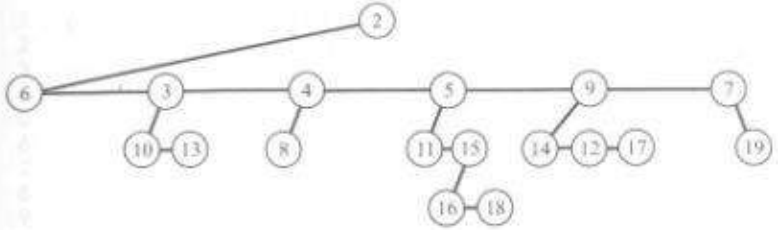


Figura 22.5 Representación del montículo de emparejamientos de la Figura 22.4; la línea oscura representa un par de referencias que conecta los nodos en ambas direcciones.

particular de mezcla. Para realizar una operación `reducirClave`, reducimos el valor del nodo correspondiente. Debido a que no mantenemos en todos los nodos referencias a sus padres, no sabemos si esto incumple el orden del montículo. Por este motivo, desligamos de su padre el nodo ajustado y completamos `reducirClave` combinando los dos montículos de emparejamientos obtenidos. La Figura 22.5 muestra cómo desligar un nodo de su padre, eliminándolo de lo que es, básicamente, una lista enlazada de hijos. Además se verifica un hecho importante: todas las operaciones descritas tienen un coste constante. Sin embargo, cuando consideramos `eliminarMin` no tenemos tanta suerte.

Para realizar `eliminarMin`, debemos eliminar la raíz del árbol, creando una colección de montículos. Si la raíz tiene c hijos, la combinación de todos ellos en un único montículo requiere $c - 1$ mezclas. Como consecuencia, si hay muchos hijos, el proceso requerirá bastante tiempo. En particular, si la secuencia de inserción fuera 1, 2, ..., N, entonces es sencillo comprobar que el primero de ellos estará en la raíz y que el resto serán sus hijos. Consecuentemente, `eliminarMin` es $O(N)$. Lo mejor que podemos esperar es preparar las mezclas de modo que no se repitan instancias costosas de la operación `eliminarMin`.

La variante más simple y práctica de todas las propuestas es la *mezcla en dos pasadas*. En primer lugar mezclamos pares de hijos yendo de izquierda a derecha[2]. Después de este primer recorrido, se ha reducido a la mitad el número de árboles a combinar. El segundo recorrido se hace de derecha a izquierda. En cada uno de los pasos combinamos el árbol situado más a la derecha que nos queda de la primera pasada con el obtenido después de la mezcla actual. Por ejemplo, si tenemos hijos desde c_1 hasta c_8, el primer recorrido realiza las mezclas c_1 y c_2, c_3 y c_4, c_5 y c_6, y c_7 y c_8. Los resultados son d_1, d_2, d_3 y d_4. Realizamos la segunda pasada mezclando d_3 y d_4; d_2 se combina con este resultado, y d_1 se mezcla con el resultado de dicha combinación, completándose así la rutina `eliminarMin`. La Figura 22.6 muestra el resultado de `eliminarMin` sobre el montículo de emparejamientos de la Figura 22.5.

Son posibles también otras estrategias de mezcla. Por ejemplo, podemos insertar en una cola cada subárbol (correspondiente a un hijo), sacar repetidamente de la cola dos árboles e insertar de nuevo en la cola el resultado de su mezcla. Después de $c - 1$ mezclas, sólo permanece un árbol en la cola y ése es el resultado de `eliminarMin`. Sin embargo, usar una pila en lugar de una cola puede estropear el

[2] Debemos tener cuidado si el número de hijos es impar. Cuando esto sucede, completamos el primer paso c
hijo con el árbol obtenido de la mezcla de más a la derecha.

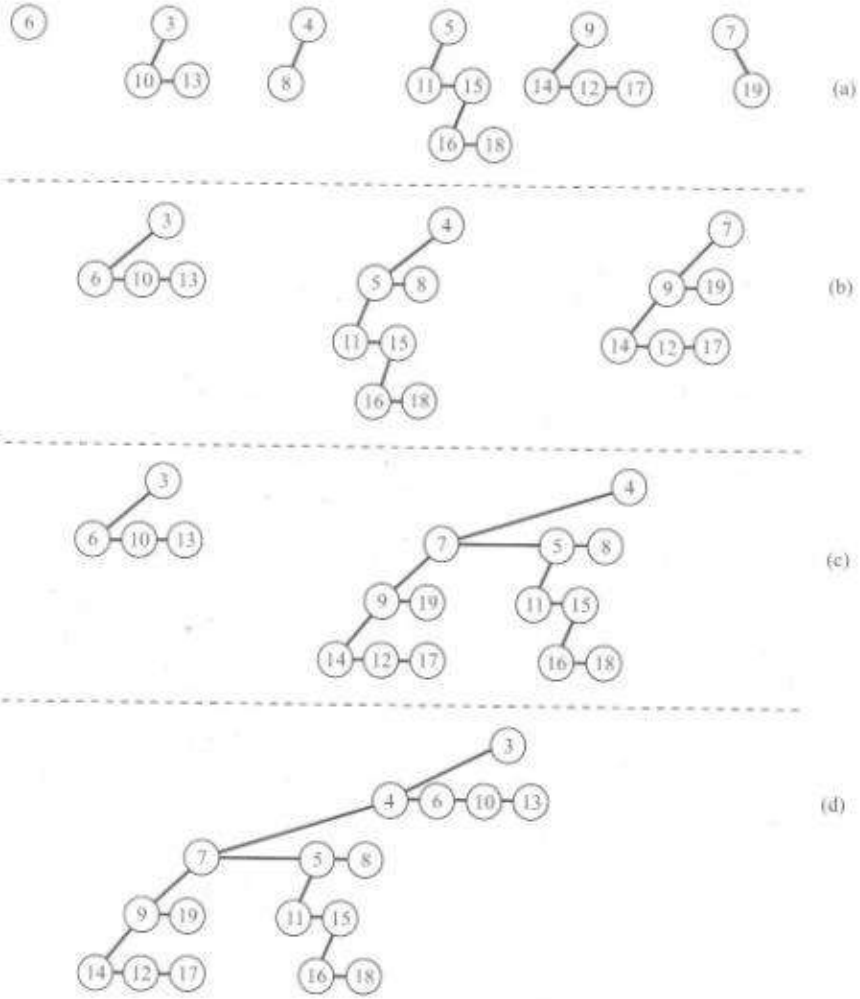


Figura 22.6 Recombinación de hermanos después de `eliminarMin`: en cada mezcla el árbol de mayor raíz se convierte en el hijo izquierdo del árbol de menor raíz: (a) los árboles obtenidos; (b) tras el primer paso; (c) tras la primera mezcla del segundo paso; (d) tras la segunda mezcla del segundo paso.

proceso, ya que puede suceder que la raíz del árbol que resulte tenga $c - 1$ hijos. Si esto sucede repetidas veces en una misma secuencia de operaciones, la rutina `eliminarMin` tendrá, por cada operación, un coste amortizado lineal, en lugar de logarítmico. El Ejercicio 22.8 pide al lector que construya una secuencia con estas características.

22.2.2 Implementación del montículo de emparejamientos

La funcionalidad de `MonticuloEmparejamientos` se describe en la Figura 22.7 y el esqueleto de dicha clase se muestra en la Figura 22.8. El nodo básico de un montículo de emparejamientos, `NodoEmparejamientos`, se puede ver en la

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase MonticuloEmparejamientos
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ************************OPERACIONES PÚBLICAS**********************
11 // void insertar( x )          --> Inserta x
12 // NodoEmparejamientos anyadirElem( x )
13                                --> Inserta x y devuelve su posición
14 // Comparable eliminarMin( ) --> Devuelve y elimina el menor element
15 // Comparable buscarMin( )    --> Devuelve el menor elemento
16 // boolean esVacio( )         --> Devuelve true si vacío; si no, fals
17 // void vaciar( )             --> Elimina todos los elementos
18 // void reducirClave( NodoEmparejamientos p, nuevoValor )
19 //                            --> Disminuye el valor en el nodo p
20 // ***************************ERRORES********************************
21 // buscarMin y eliminarMin lanzan DesbordamientoInferior sobre el vacío
```

Figura 22.7 Descripción de la clase `MonticuloEmparejamientos`.

```
1  /**
2   * Implementa un montículo de emparejamientos.
3   * Permite la operación reducirClave, pero necesita del uso
4   * de anyadirElem en lugar de insertar. Siempre se mantiene
5   * el orden del montículo; no se permiten operaciones perezosas.
6   */
7  public class MonticuloEmparejamientos implements ColaPrioridad
8  {
9      public MonticuloEmparejamientos( )
10       { vaciar( ); }
11     public Comparable buscarMin( ) throws DesbordamientoInferior
12       { /* Figura 22.10 */ }
13     public void insertar( Comparable x )
14       { /* Figura 22.11 */ }
15     public NodoEmparejamientos anyadirElem( Comparable x )
16       { /* Figura 22.11 */ }
17     public Comparable eliminarMin( ) throws DesbordamientoInferior.
18       { /* Figura 22.12 */ }
19     public void reducirClave( NodoEmparejamientos p, Comparable nuevoValor )
20                                                   throws ValorIlegal
21       { /* Figura 22.13 */ }
22     public boolean esVacio( )
23       { return tamanyoActual == 0; }
24     public void vaciar( )
25       { tamanyoActual = 0; raiz = null; }
26
27     private int  tamanyoActual;   // Número de elementos del montículo
28     private NodoEmparejamientos raiz;
29
30     private NodoEmparejamientos
31      comparaYEnlaza( NodoEmparejamientos primero, NodoEmparejamientos segundo
32       { /* Figura 22.15 */ }
33     private NodoEmparejamientos
34      combinaHermanos( NodoEmparejamientos primerHermano )
35       { /* Figura 22.16 */ }
36 }
```

Figura 22.8 Esqueleto de la clase de los montículos de emparejamientos.

Figura 22.9, y consiste en un elemento y tres referencias. Dos de ellas son el hijo izquierdo y el hermano siguiente. La tercera referencia es `prev`, que apunta a su padre, si el nodo es el primer hijo, o a un hermano izquierdo, en caso contrario. La clase `NodoEmparejamientos` es pública, por lo que se pueden emplear referencias hacia sus objetos, pero sus atributos y el constructor son de acceso amistoso y no son accesibles a los usuarios de la clase.

Los atributos de `NodoEmparejamientos` son, una referencia al nodo raíz (`raiz`) y un entero que representa el número de elementos almacenados en el montículo. Este último atributo se utiliza para simplificar la operación `reducirClave`. `MonticuloEmparejamientos` debe implementar el interfaz `ColaPrioridad`. Como resultado, el comportamiento de `insertar` no puede cambiar. Sin embargo, deseamos devolver una referencia al nuevo `NodoEmparejamientos` para utilizarla al invocar a `reducirClave`. Por este motivo implementamos `anyadirElem`. Esta rutina invoca a `insertar` y devuelve el nuevo `NodoEmparejamientos` recién creado. Para hacerlo, comparte el atributo `nuevoNodo` (que se describe en la implementación de `insertar`). No se incluye ningún método `mezclar`; su implementación es sencilla, y se deja como ejercicio al lector en el Ejercicio 22.10.

`buscarMin` se implementa en la Figura 22.10. Como el valor mínimo se encuentra en la raíz, esta rutina se codifica fácilmente. El método `insertar`, mostrado en la Figura 22.11, crea un árbol con un único nodo y lo combina con `raiz`

```
1  package EstructurasDatos;
2  import Soporte.*; import Soporte.Comparable;
3
4  /**
5   * Clase pública para utilizarla con MonticuloEmparejamientos. Sólo es
6   * pública para permitir devolver referencias a reducirClave.
7   * No tiene métodos públicos ni atributos públicos.
8   * @see MonticuloEmparejamientos
9   */
10 public class NodoEmparejamientos
11 {
12     /**
13      * Construye el NodoEmparejamientos.
14      * @param elElemento el valor almacenado en el nodo.
15      */
16     NodoEmparejamientos( Comparable elElemento )
17     {
18         dato             = elElemento;
19         izquierdo        = null;
20         siguienteHermano = null;
21         prev             = null;
22     }
23
24         // Acceso amistoso; accesible por otras rutinas del paquete
25     Comparable dato;
26     NodoEmparejamientos izquierdo;
27     NodoEmparejamientos siguienteHermano;
28     NodoEmparejamientos prev;
29 }
```

Figura 22.9 Clase `NodoEmparejamientos`.

```
1  /**
2   * Busca el menor elemento de la cola de prioridad.
3   * @return el menor elemento.
4   * @exception DesbordamientoInferior si la cola de prioridad está vacía.
5   */
6  public Comparable buscarMin( ) throws DesbordamientoInferior
7  {
8      if( esVacio( ) )
9          throw new DesbordamientoInferior( "Montículo vacío" );
10     return raiz.dato;
11 }
```

Figura 22.10 Método `buscarMin` de la clase de los montículos de emparejamientos.

```
1  private NodoEmparejamientos nuevoNodo = null; // Último nodo insertado
2
3  /**
4   * Inserción en la cola de prioridad.
5   * Se permiten duplicados.
6   * @param x el elemento a insertar.
7   */
8  public void insertar( Comparable x )
9  {
10     nuevoNodo = new NodoEmparejamientos( x );
11
12     tamanyoActual++;
13     if( raiz == null )
14         raiz = nuevoNodo;
15     else
16         raiz = comparaYEnlaza( raiz, nuevoNodo );
17 }
18
19 /**
20  * Inserción en la cola de prioridad y devolución de un NodoEmparejamientos
21  * que puede emplear reducirClave.
22  * Se permiten duplicados.
23  * @param x el elemento a insertar.
24  * @return el nodo que contiene el elemento recién insertado.
25  */
26 public NodoEmparejamientos anyadirElem( Comparable x )
27 {
28     insertar( x );
29     return nuevoNodo;
30 }
```

Figura 22.11 Métodos `insertar` y `anyadirElem` de la clase de los montículos de emparejamientos.

para obtener un nuevo árbol. Como hemos mencionado anteriormente en esta misma sección, `anyadirElem` invoca a `insertar` y devuelve una referencia al nuevo nodo recién creado. Observe que debemos considerar el caso especial de insertar un elemento en un árbol vacío.

En la Figura 22.12 se implementa el método `eliminarMin`. Si el montículo de emparejamientos está vacío, tenemos un error. En caso contrario, después de guardar el valor de la raíz, en la línea 12 llamamos a `combinaHermanos` para

```
1  /**
2   * Elimina el menor elemento de la cola de prioridad.
3   * @exception DesbordamientoInferior si la cola de prioridad está vacía.
4   */
5  public Comparable eliminarMin( ) throws DesbordamientoInferior
6  {
7      Comparable x = buscarMin( );
8
9      if( raiz.izquierdo == null )
10         raiz = null;
11     else
12         raiz=combinaHermanos( raiz.izquierdo );
13
14     tamanyoActual--;
15
16     return x;
17 }
```

Figura 22.12 Método `eliminarMin` de la clase de los montículos de emparejamientos.

combinar los subárboles de las raíces y enlazar el resultado a la nueva raíz. Si no hay subárboles que mezclar, en la línea 10 inicializamos `raiz` a `null`. Las rutinas `insertar` y `eliminarMin` ajustan adecuadamente `tamanyoActual`.

```
1  /**
2   * Cambia el valor del elemento guardado en el montículo
    de emparejamientos.
3   * @param p cualquier nodo devuelto por anyadirElem.
4   * @param nuevoValor el nuevo valor, que debe ser menor que
5   *    el valor almacenado ahora.
6   * @exception ValorIlegal si nuevoValor es mayor que
7   *    el valor almacenado ahora.
8   */
9  public void reducirClave( NodoEmparejamientos p, Comparable nuevoValor )
10                                        throws ValorIlegal
11 {
12     if( p.dato.menorQue( nuevoValor ) )
13         throw new ValorIlegal( "ReducirClave incorrecto" );
14     p.dato = nuevoValor;
15     if( p != raiz )
16     {
17         if( p.siguienteHermano != null )
18             p.siguienteHermano.prev = p.prev;
19         if( p.prev.izquierdo == p )
20             p.prev.izquierdo = p.siguienteHermano;
21         else
22             p.prev.siguienteHermano = p.siguienteHermano;
23         p.siguienteHermano = null;
24
25         raiz = comparaYEnlaza( raiz, p );
26     }
27 }
```

Figura 22.13 Método `reducirClave` de la clase de los montículos de emparejamientos.

La rutina `reducirClave` se implementa en la Figura 22.13. Nótese que si el nuevo valor es mayor que el inicial, entonces podríamos destruir el orden del montículo. Pero no hay ningún modo de saberlo sin examinar antes los hijos. Como puede haber bastantes, hacerlo sería muy ineficiente. Como consecuencia, supondremos que es un error intentar aumentar la clave empleando `reducirClave`. (El Ejercicio 22.9 pide al lector que describa un algoritmo para `aumentarClave`.) Después de realizar este test, disminuimos el valor del nodo. Si dicho nodo es la raíz, ya hemos terminado. En caso contrario, eliminamos el nodo de la lista de hijos en la que se encuentra, por medio del código en las líneas 17 a 23. Después de hacer esto, combinamos el árbol resultante con la raíz.

Las dos rutinas restantes son `comparaYEnlaza`, que combina dos árboles, y `combinaHermanos`, que combina todos los hermanos dado el primero de ellos. La Figura 22.14 muestra cómo se combinan dos submontículos. El procedimiento se generaliza para permitir que el segundo de los submontículos pueda tener hermanos (esto es necesario para el segundo recorrido de la mezcla en dos pasadas). Como ya hemos mencionado en este capítulo, el submontículo con la raíz más grande se convierte en el hijo situado más a la izquierda del otro submontículo. El código se muestra en la Figura 22.15. Nótese que existen muchos puntos en los que se comprueba que una referencia no es `null` antes de acceder a su atributo `prev`. Esto sugiere que quizás sería útil tener un centinela `nodoNulo`, tal y como es costumbre en las implementaciones avanzadas de los árboles de búsqueda. Esto se deja como Ejercicio 22.13.

Por último, en la Figura 22.16 se implementa `combinaHermanos`. Empleamos el vector `vectorArboles` para almacenar los subárboles. En el peor de los casos, tendremos $N - 1$ hermanos, por lo que el vector emplea `tamanyoActual` para determinar su capacidad. Comenzamos por separar los subárboles y guardarlos en `vectorArboles`, empleando para ello el bucle de las líneas 18 a 24. Suponiendo que tenemos más de un hermano para combinar, en las líneas 28 a 30 realizamos el recorrido de izquierda a derecha. Terminamos el proceso de mezcla con el recorrido de derecha a izquierda en las líneas 41 a 43. Una vez que hemos terminado, el resultado se encuentra en la posición 0 del vector, y puede ser devuelto.

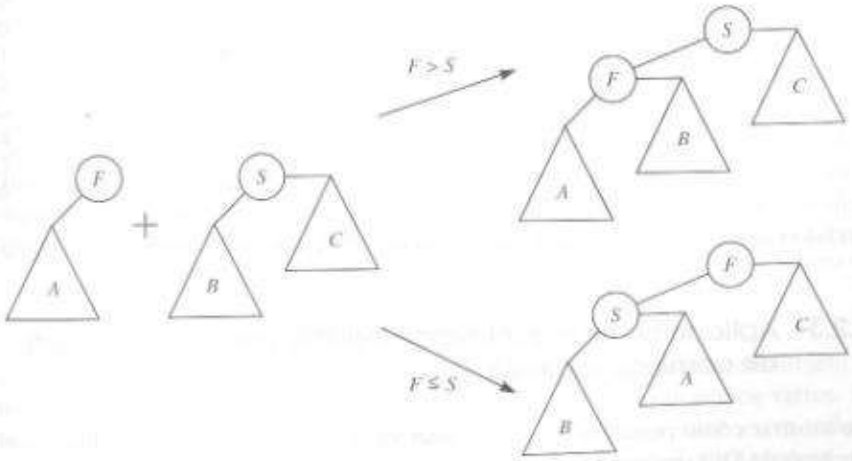


Figura 22.14 `comparaYEnlaza` combina dos árboles.

```
/**
 * Método interno básico para mantener el
 * orden en el montículo.
 * Enlaza primero y segundo para mantener el orden.
 * @param primero la raíz del árbol 1, no debe ser null.
 *     primero.siguienteHermano DEBE ser null en la entrada.
 * @param segundo la raíz del árbol 2, puede ser null.
 * @return el resultado de la mezcla.
 */
private NodoEmparejamientos
comparaYEnlaza( NodoEmparejamientos primero, NodoEmparejamientos segundo )
{
    if( segundo == null )
        return primero;

    if( segundo.dato.menorQue( primero.dato ) )
    {
        // Enlaza primero como el hijo más izquierdo de segundo
        segundo.prev = primero.prev;
        primero.prev = segundo;
        primero.siguienteHermano = segundo.izquierdo;
        if( primero.siguienteHermano != null )
            primero.siguienteHermano.prev = primero;
        segundo.izquierdo = primero;
        return segundo;
    }
    else
    {
        // Enlaza segundo como el hijo más izquierdo de primero
        segundo.prev = primero;
        primero.siguienteHermano = segundo.siguienteHermano;
        if( primero.siguienteHermano != null )
            primero.siguienteHermano.prev = primero;
        segundo.siguienteHermano = primero.izquierdo;
        if( segundo.siguienteHermano != null )
            segundo.siguienteHermano.prev = segundo;
        primero.izquierdo = segundo;
        return primero;
    }
}
```

Figura 22.15 Rutina `comparaYEnlaza`.

En la práctica, el almacenamiento dinámico (y la recogida de basura) del vector puede ser extremadamente costoso y en muchas ocasiones no será necesario. Puede ser mejor usar un objeto estático con un determinado tamaño inicial, expandiéndolo cuando sea necesario. Esto se propone al lector en el Ejercicio 22.11.

22.2.3 Aplicación: el algoritmo de Dijkstra para la obtención de caminos mínimos

Para mostrar cómo puede emplearse la operación `reducirClave`, reescribimos el algoritmo de Dijkstra estudiado en la Sección 14.3. Recordemos que manejamos una cola de prioridad de objetos de tipo `Camino`, ordenados según el atributo

```
/**
 * Método interno que implementa la mezcla en dos pasadas.
 * @param primerHermano la raíz;
 *     se supone que no es null.
 */
private NodoEmparejamientos
combinaHermanos( NodoEmparejamientos primerHermano )
{
    if( primerHermano.siguienteHermano == null )
        return primerHermano;

        // Creación del vector
    NodoEmparejamientos [ ] vectorArboles =
                    new NodoEmparejamientos[ tamanyoActual ];

        // Almacenamiento de los árboles en el vector
    int numeroHermanos = 0;
    for( ; primerHermano != null; numeroHermanos++ )
    {
        vectorArboles[ numeroHermanos ] = primerHermano;
        primerHermano.prev.siguienteHermano = null;  // rompe los enlac
        primerHermano = primerHermano.siguienteHermano;
    }
    vectorArboles[ numeroHermanos ] = null;

    // Combinación de pares de subárboles, de izquierda a derecha
    int i = 0;
    for( ; i+1 < numeroHermanos; i += 2 )
        vectorArboles[ i ] = comparaYEnlaza( vectorArboles[ i ],
                                             vectorArboles[ i + 1 ] );

        // j es el resultado del último comparaYEnlaza.
        // Si el número de árboles es impar, coge el último de ellos
    int j = i - 2;
    if( j == numeroHermanos - 3 )
        vectorArboles[ j ] = comparaYEnlaza( vectorArboles[ j ],
                                             vectorArboles[ j + 2 ]

        // Recorrido de derecha a izquierda, combinando el último
        // con el siguiente al último. El resultado es el nuevo últi
    for( ; j >= 2; j -= 2 )
        vectorArboles[ j - 2 ] = comparaYEnlaza( vectorArboles[ j - 2 ],
                                                 vectorArboles[ j ] );

    return vectorArboles[ 0 ];
}
```

Figura 22.16 El núcleo del algoritmo de mezcla: implementa una mezcla en dos pasadas para combinar todos los hermanos, a partir del primero de ellos.

`dist`. En cada momento, por cada vértice del grafo, necesitamos un solo objeto de tipo `Camino` en la cola de prioridad, pero por comodidad mantenemos varios. En esta sección, retocaremos el código de modo que, cuando un vértice w vea reducida su distancia, se buscará su posición en la cola de prioridad, realizándose una operación `reducirClave` sobre su correspondiente objeto `Camino`.

El código modificado se muestra en la Figura 22.17. Los cambios son relativamente pequeños. En primer lugar, en la línea 5 declaramos que `mm` es un montículo de emparejamientos en lugar de un montículo binario. Manejamos también un vector de referencias, `posicionesMonticulo`, que apuntan a nodos de

```
1  // Algoritmo de Dijkstra empleando montículos de emparejamientos
2  private boolean dijkstraMezcla( int nodoInicio )
3  {
4    int v, w;
5    MonticuloEmparejamientos mm = new MonticuloEmparejamientos( );
6    Camino vrec;
7    NodoEmparejamientos [ ] posicionesMonticulo;
8
9    limpiarDatos( );
10   posicionesMonticulo = new NodoEmparejamientos[ numVertices ];
11   for ( int i = 0; i < numVertices; i++ )
12       posicionesMonticulo[ i ] = null;
13   tabla[ nodoInicio ].dist = 0;
14   mm.insertar( new Camino( nodoInicio, 0 ) );
15
16   try
17   {
18      while( !mm.esVacio( ) )
19      {
20         vrec = (Camino) mm.eliminarMin( );
21         v = vrec.dest;
22
23         ListaIter p = new ListaEnlazadaIter( tabla[v].ady );
24         for( ; p.estaDentro( ); p.avanzar( ) )
25         {
26            w       = ( (Arista)p.recuperar( ) ).dest;
27            int cvw = ( (Arista)p.recuperar( ) ).coste;
28
29            if( cvw < 0 )
30               return false;
31            if( tabla[ w ].dist > tabla[ v ].dist+cvw )
32            {
33               tabla[ w ].dist = tabla[ v ].dist+cvw;
34               tabla[ w ].ant = v;
35
36               Camino nuevoValor = new Camino( w, tabla[ w ].dist );
37               if( posicionesMonticulo[ w ] == null )
38                  posicionesMonticulo[ w ] =
39                                    mm.anyadirElem( nuevoValor );
40               else
41                  mm.reducirClave( posicionesMonticulo[ w ],
42                                   nuevoValor );
43            }
44         }
45      }
46   }
47   catch( Exception e ) { }   // Esto no puede suceder
48   return true;
49 }
```

Figura 22.17 Algoritmo de Dijkstra que emplea montículos de emparejamientos y la rutina `reducirClave`.

montículos de emparejamientos. Inicialmente todas las referencias son `null` (líneas 11 y 12). Cada vez que se inserta un elemento en el montículo de emparejamientos, modificamos el vector `posicionesMonticulo`. Esto se hace en la línea 38[3]. El algoritmo se ha simplificado. Ahora llamamos a `eliminarMin` en tanto y cuando el montículo no esté vacío, en lugar de hacerlo repetidas veces hasta que aparece un nodo no consultado. Como consecuencia, ya no es necesario el atributo `eliminado`. Compare las líneas 18 a 21 con el correspondiente código de la Figura 14.29. Todo lo que queda por comentar son las actualizaciones tras la línea 31, que indican que se ha realizado un cambio. Si el vértice no ha sido insertado nunca en la cola de prioridad, lo insertamos por primera vez, actualizando el vector `posicionesMonticulo`. En caso contrario, en la línea 41 invocamos la rutina `reducirClave`.

Que la implementación del algoritmo de Dijkstra que emplea montículos binarios sea más rápida que la que emplea montículos de emparejamientos depende de varios factores. Un estudio, detallado en las referencias, indica que el montículo de emparejamientos es ligeramente mejor que el montículo binario cuando ambos se implementan cuidadosamente. Los resultados dependen en gran medida de los detalles de codificación y del número de llamadas a `reducirClave`. Son necesarios estudios más profundos para decidir cuándo son más adecuados en la práctica los montículos de emparejamientos.

Resumen

Este capítulo describe dos estructuras de datos que permiten la mezcla de sus elementos, y son eficientes en términos de coste amortizado: el montículo sesgado y el montículo de emparejamientos. Ambas son de sencilla implementación, ya que no tienen ninguna restricción estructural. El montículo de emparejamientos parece tener un cierto interés en la práctica, pero su análisis completo continúa siendo un intrincado problema abierto.

El siguiente capítulo, que es ya el último de este texto, describe una estructura de datos que se emplea para manipular conjuntos disjuntos y cuyo análisis amortizado también es interesante.

Elementos del juego

mezcla en dos pasadas El orden en el que se combinan los montículos de emparejamientos es importante. El algoritmo más sencillo para hacerlo es la mezcla en dos pasadas, en la que los montículos se mezclan en pares, de izquierda a derecha. Después se realiza un nuevo recorrido de derecha a izquierda para concluir la mezcla.

montículo de emparejamientos Árbol M-ario con ordenación de montículos y sin restricciones estructurales. Su análisis no está completo, pero parece comportarse bastante bien en la práctica.

[3] El `nodoInicio` no estará incluido en el montículo de emparejamientos, por lo que en la línea 14 no emple

montículo sesgado Árbol binario con ordenación de montículos. No tiene ninguna condición de equilibrio, pero todas sus operaciones tienen un coste amortizado logarítmico.

Errores comunes

1. Una implementación recursiva de los montículos sesgados no puede emplearse en la práctica, ya que la profundidad de la recursión puede ser lineal.
2. En la clase de los montículos sesgados debemos ser cuidadosos para actualizar adecuadamente las referencias `prev`.
3. Deben realizarse a lo largo del código de los montículos de emparejamientos ciertos tests para comprobar que las referencias no son `null`.
4. Tras la realización de una mezcla, dos montículos de emparejamientos no deben compartir ningún nodo.

En Internet

La clase de los montículos de emparejamientos está disponible en el directorio **DataStructures**. La Figura 22.17 forma parte de la clase `Grafo` explicada en el Capítulo 14 (**Graph.java** está en el directorio **Part3**).

PairHeap.java	Contiene la implementación de los montículos de emparejamientos. Es la versión inglesa de la clase `MonticuloEmparejamientos`.
PairNode.java	Contiene la definición de los nodos de los montículos de emparejamientos. Es la versión inglesa de `NodoEmparejamientos`.

Ejercicios

Cuestiones breves

22.1. Muestre el montículo sesgado resultante a partir de las siguientes secuencias de inserción:

a) 1, 2, 3, 4, 5, 6, 7
b) 4, 3, 5, 2, 6, 7, 1

22.2. Muestre el montículo de emparejamientos resultante a partir de las siguientes secuencias de inserción:

a) 1, 2, 3, 4, 5, 6, 7
b) 4, 3, 5, 2, 6, 7, 1

22.3. Muestre el resultado de dos operaciones `eliminarMin` sobre cada uno de los montículos de los Ejercicios 22.1 y 22.2.

Problemas teóricos

22.4. Dé una secuencia de operaciones que conduzca a una operación `mezclar` que requiera tiempo lineal para demostrar que la cota amortizada logarítmica de las operaciones del montículo sesgado no es una cota para el caso peor.

22.5. Muestre que las operaciones `reducirClave` y `aumentarClave` pueden implementarse en los montículos sesgados con un coste amortizado logarítmico.

22.6. Describa un algoritmo `arreglarMonticulo` de coste lineal en la clase de los montículos sesgados.

22.7. Muestre que guardando el tamaño del camino derecho de cada nodo del árbol podemos imponer una condición de equilibrio que permite que todas las operaciones tengan coste logarítmico en el caso peor. Esta estructura de datos se conoce como *montículo izquierdista*.

22.8. Muestre que no es adecuado emplear una pila en la implementación de la operación `combinaHermanos` de la clase de los montículos de emparejamientos; para ello, construya una secuencia de coste amortizado lineal por operación.

22.9. Describa cómo implementar `aumentarClave` en la clase de los montículos de emparejamientos.

Problemas prácticos

22.10. Incluya un método público `mezclar` en la clase de los montículos de emparejamientos. Asegúrese de que cada nodo aparece en un único árbol.

22.11. Use un vector que se expanda siempre que sea necesario, para almacenar `vectorArboles` en `combinaHermanos`.

Prácticas de programación

22.12. Implemente una versión no recursiva del montículo sesgado.

22.13. Implemente los montículos de emparejamientos empleando un centinela `nodoNulo`.

22.14. Implemente un algoritmo que utilice colas para `combinaHermanos` y compare su eficiencia con el algoritmo de dos pasadas codificado en la Figura 22.16.

22.15. Si la operación `reducirClave` no está permitida, las referencias a los padres no son necesarias. Implemente el montículo de emparejamientos sin dichas referencias y compare su eficiencia con la del montículo binario y/o el montículo sesgado y/o el árbol de ensanchamiento.

22.16. Diseñe un applet que muestre el comportamiento de los montículos sesgados o de los montículos de emparejamientos.

Bibliografía

El *montículo izquierdista* [1] fue la primera cola de prioridad combinable eficientemente. Es la variación sugerida en el Ejercicio 22.7, pensada para el caso peor de los montículos sesgados. Los montículos sesgados se describen en [5]. Dicho artículo incluye las soluciones de los Ejercicios 22.4 y 22.5.

[2] describe el montículo de emparejamientos y demuestra que cuando se emplea la mezcla en dos pasadas, el coste amortizado de todas las operaciones es a lo

sumo logarítmico, aunque aún no se sabe si esta cota es lo suficientemente ajustada. En particular, en [6] puede encontrarse alguna evidencia en el sentido de que el coste amortizado de todas las operaciones, excepto `eliminarMin`, podría ser constante, mientras que el coste de `eliminarMin` sí parece ser logarítmico. De este modo, cualquier secuencia de D `eliminarMin` y otras I operaciones es $O(I + D \log N)$. Sin embargo, en [4] aparecen otras evidencias que parecen sugerir que esto no es así. Una estructura de datos que sí consigue con seguridad las cotas deseadas, aunque es demasiado complicada para ser usada en la práctica, es el *montículo de Fibonacci* [3]. Parece que el montículo de emparejamientos podría ser la alternativa práctica al buen comportamiento teórico del montículo de Fibonacci. Los montículos izquierdistas y de Fibonacci se discuten en [7].

En [4] puede encontrarse una comparación entre distintas colas de prioridad en el contexto del problema del árbol de cobertura mínimo (discutido en la Sección 23.2.1), empleándose un método muy similar al algoritmo de Dijkstra.

1. C. A. Crane, «Linear Lists and Priority Queues as Balanced Binary Trees», *Technical Report STAN-CS-72-259*, Computer Science Department, Stanford University, Palo Alto, Calif. (1972).
2. M. L. Fredman, R. Sedgewick, D. D. Sleator, y R. E. Tarjan, «The Pairing Heap: A New Form of Self-adjusting Heap», *Algorithmica* **1** (1986), 111-129.
3. M. L. Fredman y R. E. Tarjan, «Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms», *Journal of the ACM* **34** (1987), 596-615.
4. B. M. E. Moret y H. D. Shapiro, «An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree», *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400-411.
5. D. D. Sleator y R. E. Tarjan, «Self-adjusting Heaps», *SIAM Journal on Computing* **15** (1986), 52-69.
6. J. T. Stasko y J. S. Vitter, «Pairing Heaps: Experiments and Analysis», *Communications of the ACM* **32** (1987), 234-249.
7. M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, Benjamin/Cummings Publishing Co., Redwood City, Calif. (1994).

CAPÍTULO

23 Estructura de partición

Este capítulo describe una estructura de datos eficiente para resolver el problema de la equivalencia: la estructura de partición. Esta estructura se implementa de forma sencilla; cada rutina requiere sólo unas líneas de código. Dicha implementación también es extremadamente rápida, requiriendo un tiempo constante en promedio por operación. Esta estructura de datos también resulta interesante desde un punto de vista teórico, ya que su análisis es muy difícil; la forma funcional del coste en el caso peor no se parece a ninguna de las discutidas hasta ahora en este libro.

En este capítulo veremos:

- Dos aplicaciones simples de la estructura de partición.
- Cómo podemos implementar la estructura de partición con un esfuerzo mínimo.
- Cómo mejorar la velocidad de las estructuras de partición utilizando dos observaciones simples.
- Un análisis del tiempo de ejecución de una implementación rápida de la estructura de partición.

23.1 Relaciones de equivalencia

Una *relación* R sobre un conjunto S se define como un conjunto de pares de elementos (a, b) con $a, b \in S$. Si $(a, b) \in R$ escribiremos $a\ R\ b$ y diremos que a está relacionado con b. Una *relación de equivalencia* es una relación R que satisface las siguientes tres propiedades:

1. *Reflexiva*: $a\ R\ a$ es cierto para todo $a \in S$.
2. *Simétrica*: $a\ R\ b$ si y sólo si $b\ R\ a$.
3. *Transitiva*: $a\ R\ b$ y $b\ R\ c$ implican que $a\ R\ c$.

La conectividad eléctrica, donde todas las conexiones se realizan mediante cables eléctricos, es una relación de equivalencia. Claramente, la relación es reflexiva, ya que cualquier componente está conectada a sí misma. Si a está conectado eléctricamente a b, entonces b está conectado a a, por lo que la relación es simétrica. Finalmente, si a está conectado a b y a su vez b está conectado a c, entonces a estará conectado a c.

De la misma manera, la conectividad a través de una red bidireccional forma clases de equivalencia de componentes conectadas. Sin embargo, si las conexiones de la red fueran dirigidas (es decir, una conexión de v a w no implica una de w a v), entonces no tendríamos una relación de equivalencia porque fallaría la propiedad de simetría. Un ejemplo es una relación en la cual la ciudad a está relacionada con la ciudad b si es posible viajar de a a b mediante una carretera. Esta relación es una relación de equivalencia si las carreteras son de doble sentido.

23.2 Equivalencias dinámicas y dos aplicaciones

Dada una relación de equivalencia, que denotaremos por el símbolo $\sim$, el problema inmediato que surge es decidir, dados a y b cualesquiera, si $a \sim b$. Si la relación se almacena como un vector bidimensional de valores booleanos, esto puede decidirse en tiempo constante. El problema es que las relaciones generalmente se definen de forma implícita, en vez de explícitamente.

Por ejemplo, si consideramos una relación de equivalencia definida sobre el conjunto con cinco elementos $\{a_1, a_2, a_3, a_4, a_5\}$, hay en total 25 pares de elementos, cada uno de los cuales puede, en principio, pertenecer a la relación o no. Sin embargo, de la información $a_1 \sim a_2$, $a_3 \sim a_4$, $a_1 \sim a_5$ y $a_4 \sim a_2$ se sigue que todos los elementos han de estar relacionados entre sí. Nos gustaría ser capaces de inferir rápidamente algo como esto.

La *clase de equivalencia* de un elemento $x \in S$ es el subconjunto de S que contiene a todos los elementos relacionados con x. Observemos que las clases de equivalencia forman una partición de S: todo miembro de S aparece exactamente en una clase de equivalencia. Para decidir si $a \sim b$, sólo necesitamos comprobar si a y b están en la misma clase de equivalencia. Esto proporciona la estrategia para resolver el problema de la equivalencia.

Consideremos un tipo partición cuyos elementos se construirán a partir de la partición trivial que viene dada por una colección de N conjuntos, cada uno con un elemento. Esta partición inicial corresponde a la relación de equivalencia trivial. Cada conjunto tiene un elemento distinto, por lo que $S_i \cap S_j = \emptyset$ obteniéndose, por tanto, una partición.

Hay dos operaciones permitidas. La primera es `buscar`, que devuelve el nombre del conjunto (es decir, la clase de equivalencia) que contiene a un elemento dado. La segunda operación añade pares a la relación representada. Si queremos añadir el par (a, b) a la relación, vemos primero si a y b ya están relacionados. Esto se consigue llamando a `buscar` con a y b, comprobando si están en la misma clase de equivalencia. Si no es así, llamamos a `unir`. Esta operación mezcla las clases de equivalencia que contienen a a y a b, en una nueva clase de equivalencia. Desde el punto de vista conjuntista, el efecto es la creación de un nuevo conjunto $S_k = S_i \cup S_j$, destruyendo los originales, con lo que se preserva el hecho de que todos los conjuntos de la partición sean disjuntos. La estructura de datos que contiene estas operaciones se denomina en ocasiones *estructura de partición unir/buscar*. El término *algoritmo* `unir/buscar` se refiere al proceso de peticiones de unión y búsqueda utilizando una estructura de partición.

El algoritmo es *dinámico* en el sentido de que, durante su ejecución, los conjuntos de la partición pueden cambiar vía la operación `unir`. El algoritmo debe también ir produciendo sus respuestas en tiempo de ejecución, en el sentido de que cuando se realiza una búsqueda, debe mostrarse la respuesta antes de proseguir. Otra posibilidad sería permitir que la secuencia completa de resultados se mostrara tras la ejecución de todas las llamadas a `buscar` y `unir`. En tal caso, la respuesta que proporcionará cada `buscar` debería ser consistente con las uniones realizadas antes de la búsqueda, aunque el algoritmo procese todas las peticiones antes de producir la primera respuesta. Esta diferencia es parecida a la existente entre un examen escrito (donde generalmente sólo se tiene que dar la respuesta antes de un tiempo límite) y un examen oral (donde hay que dar la respuesta a cada pregunta antes de conocer la siguiente).

Observemos que no se realiza ninguna operación comparando los valores de los elementos sino que sólo se necesita su localización. Por esta razón, podemos asumir que todos los elementos han sido numerados de forma secuencial partiendo de 0 y que la numeración puede determinarse fácilmente utilizando un esquema hash.

Antes de describir cómo implementar las operaciones `buscar` y `unir`, esta sección proporciona dos aplicaciones de la estructura de datos.

23.2.1 Aplicación #1: árboles de recubrimiento mínimo

Un *árbol de recubrimiento* de un grafo no dirigido G es un árbol formado a partir de las aristas del grafo que conecta todos los vértices de G. Observe que, al contrario de lo que sucedía en los grafos del Capítulo 14, ahora una arista (u, v) de G es idéntica a la arista (v, u). El coste de un árbol de recubrimiento es la suma de los costes de las aristas en el árbol; el problema del *árbol de recubrimiento mínimo* consiste en buscar el árbol de recubrimiento de menor coste. Como mostraremos en breve, la conectividad de un grafo puede comprobarse como parte del cálculo del árbol de recubrimiento mínimo.

En la Figura 23.1, el grafo de la derecha es un árbol de recubrimiento mínimo del grafo de la izquierda (en este caso es el único, pero esto no tiene por qué ser cierto en general, en particular si el grafo tiene muchas aristas con el mismo coste). Observe que el número de aristas del árbol de recubrimiento mínimo es $|V| - 1$. El árbol de recubrimiento mínimo es un *árbol* porque es acíclico; es *de recubrimiento* porque cubre todos los vértices; y es *mínimo*, por razones obvias. Supongamos que tenemos que conectar varias ciudades mediante carreteras, minimizando el coste total de la construcción, con la restricción de que sólo se puede cambiar de carretera en una ciudad (en otras palabras, no se permiten cruces adicionales). En tal caso necesitamos resolver un problema de árbol de recubrimiento mínimo, en el que cada vértice es una ciudad y cada arista representa el coste de construir una carretera entre las ciudades que conecta.

Añadir cruces en lugares arbitrarios daría lugar al *problema del árbol de Steiner*, el cual es mucho más difícil de resolver. Por otra parte, entre otros resultados, se puede demostrar que si el coste de una conexión es proporcional a la distancia euclídea, el árbol de recubrimiento mínimo es, como mucho, un 15 por cierto más costoso que el árbol mínimo de Steiner. Esto significa que el árbol de recubrimiento mínimo, que es fácil de calcular, proporciona una buena aproximación al árbol mínimo de Steiner, que es difícil de calcular.

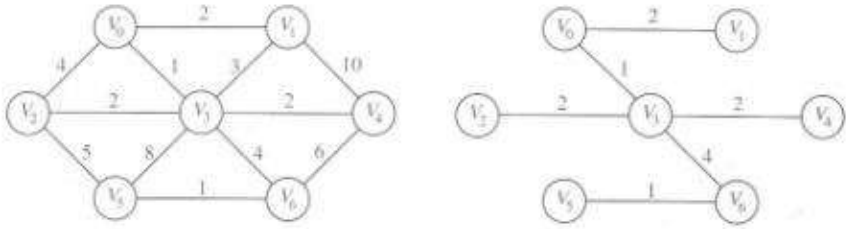


Figura 23.1 Un grafo G (a la izquierda) y su árbol de recubrimiento mínimo.

Un algoritmo sencillo, comúnmente denominado *algoritmo de Kruskal*, consiste en seleccionar las aristas en orden creciente de coste, incorporando una arista si con ello no se introduce un ciclo. Formalmente, el algoritmo de Kruskal mantiene un bosque, esto es, una colección de árboles. Inicialmente, hay $|V|$ árboles cada uno con un único nodo. Al añadir una arista se unen dos árboles en uno. Cuando el algoritmo termina, hay sólo un árbol, y éste es el árbol de recubrimiento mínimo[1]. Contando el número de aristas seleccionadas, podemos determinar cuándo debe darse por concluido el algoritmo.

La Figura 23.2 muestra la acción del algoritmo de Kruskal sobre el grafo de la Figura 23.1. Las cinco primeras aristas son todas incorporadas porque no crean ciclos. Las dos aristas siguientes (v_1, v_3) (de coste 3) y (v_0, v_2) (de coste 4), son rechazadas porque crearían un ciclo en el grafo. La arista considerada a continuación es incorporada. Ya que es la sexta arista y el grafo tiene siete vértices, podemos dar por concluida la aplicación del algoritmo.

Para generar el orden en que serán examinadas las aristas sería suficiente con ordenarlas. Podemos hacerlo con un coste $|E| \log |E|$, tras lo que iríamos recorriendo el vector ordenado de aristas. De forma alternativa, se podría construir una cola de prioridad con las $|E|$ aristas y obtener repetidamente las aristas llamando a `borrarMin`. Aunque el coste en el caso peor no cambia, utilizar una cola de prioridad es a veces mejor porque cuando los grafos son aleatorios el algoritmo de Kruskal tiende a utilizar sólo una pequeña fracción del total de aristas. Por supuesto, en el caso peor, siempre es posible que haga falta probar con todas las aristas. Por ejemplo, si añadieramos un vértice extra v_8 y una arista (v_5, v_8) de coste 100, tendríamos que examinar todas las aristas. En este caso, utilizar directamente quicksort sería más rápido. En consecuencia, la elección entre la cola de prioridad y una ordenación inicial debe hacerse en base a una especulación sobre cuántas aristas serán examinadas a la postre.

Más interesante es la cuestión de cómo decidimos si una arista (u, v) debe aceptarse o rechazarse. Claramente, añadir la arista (u, v) provocaría un ciclo si (y sólo si) u y v estuvieran ya conectados en el bosque de recubrimiento actual. Por tanto, mantendremos las componentes conexas del bosque de recubrimiento en una estructura de partición. Inicialmente, cada vértice está en una clase distinta. Si u y v están en la misma componente, la arista se rechaza porque u y v ya están conectados. En otro caso, se selecciona la arista y se realiza una operación de `unir` con los dos conjuntos disjuntos que contienen a u y a v. Esto es exactamente lo que necesitamos hacer porque una vez se ha añadido la arista (u, v) al bosque de

conexo, el algoritmo terminará con más de un árbol: cada árbol representará un árbol de recubrimiento mínimo nexa del grafo.

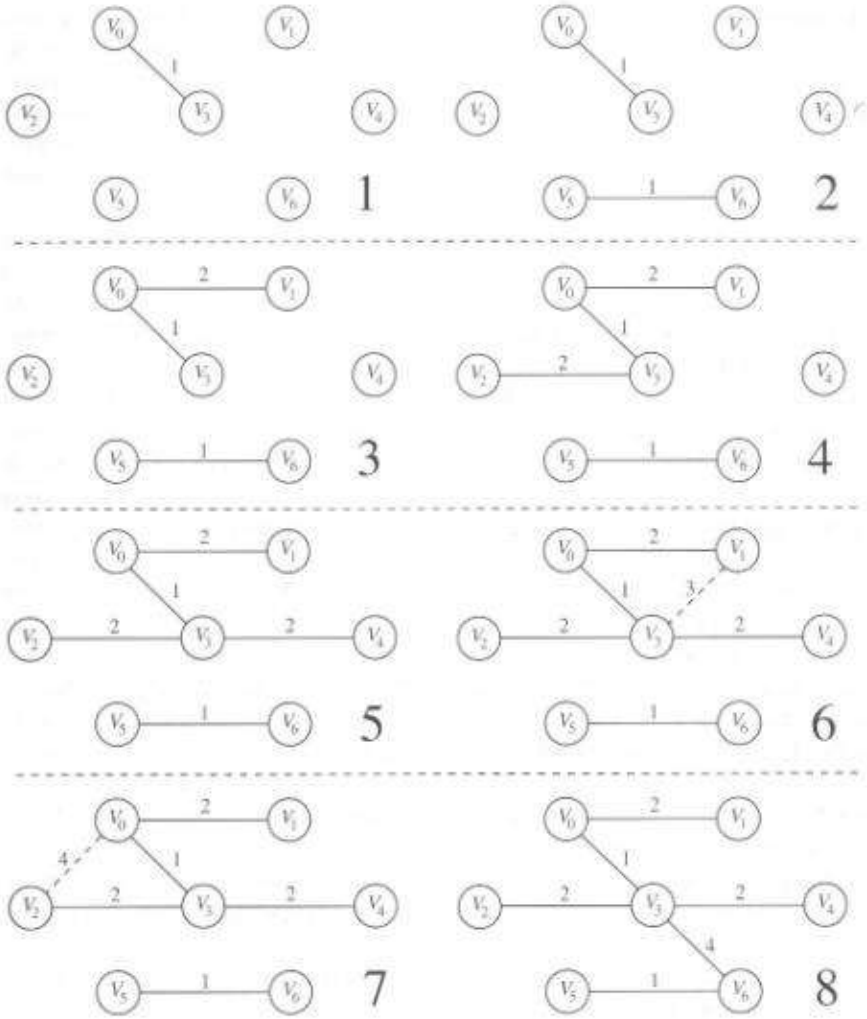


Figura 23.2 El algoritmo de Kruskal después de considerar cada arista; los pasos se presentan de arriba abajo y de izquierda a derecha, siguiendo la numeración.

recubrimiento, si w estaba conectado a u y x lo estaba a v, x y w quedan conectados y por tanto pasan a pertenecer al mismo conjunto.

23.2.2 Aplicación #2: el problema del antecesor común más próximo

Otro ejemplo de aplicación de la estructura de partición es el problema del antecesor común más próximo (ACP).

PROBLEMA DEL ANTECESOR COMÚN MÁS PRÓXIMO

Dado un árbol y una lista de pares de nodos en el árbol, buscar el antecesor común más próximo de cada par de nodos.

Como ejemplo, la Figura 23.3 muestra un árbol con una lista de pares que contiene cinco peticiones. Para el par de nodos u y z, el nodo C es el antecesor de ambos más cercano. (A y B son también antecesores, pero no son los más cercanos.) Disponemos de la secuencia completa de peticiones antes de tener que proporcionar la primera respuesta. Éste es un problema importante en la teoría de grafos que también tiene aplicaciones en el campo de la biología computacional (donde el árbol representa la evolución).

El algoritmo funciona realizando un recorrido en postorden del árbol. Cuando estamos a punto de acabar de procesar un nodo, examinamos la lista de pares para ver si hay algún calculo de antecesor que realizar. Si u es el nodo actual, (u, v) está en la lista de peticiones y ya hemos terminado la llamada recursiva con v, entonces tenemos información suficiente para determinar $ACP(u, v)$.

La Figura 23.4 ayuda a comprender cómo funciona el algoritmo. Estamos a punto de terminar la llamada recursiva correspondiente a D. Todos los nodos sombreados han sido visitados por una llamada recursiva, y excepto por los nodos en el camino a D, todas las demás llamadas recursivas ya han terminado. Marcamos un nodo después de que su llamada recursiva haya terminado. Si v está marcado, $ACP(D, v)$ es algún nodo en el camino a D. La *sujeción* de un nodo v visitado (pero no necesariamente marcado) se define como el nodo en el camino de acceso actual que está más cerca de v. En la Figura 23.4, la sujeción de p es A, la sujeción de q es B y r no está sujeto, pues aún tiene que ser visitado; podemos considerar que la sujeción de r es el mismo r la primera vez que se visita. Como muestran los dibujos, cada nodo en el camino de acceso actual es una sujeción (al menos de él mismo). Es más, los nodos visitados forman clases de equivalencia: dos nodos están relacionados si tienen la misma sujeción, y podemos considerar que cada nodo visitado está en su propia clase. Supongamos ahora que (D, v) está en la lista de pares. Tenemos tres casos:

1. v no está marcado, por lo que no tenemos información de cómo calcular $ACP(D, v)$. Sin embargo, cuando v se marque, podremos determinar $ACP(v, D)$.
2. v está marcado pero no está en el subárbol de D, por lo que $ACP(v, D)$ es la sujeción de v.

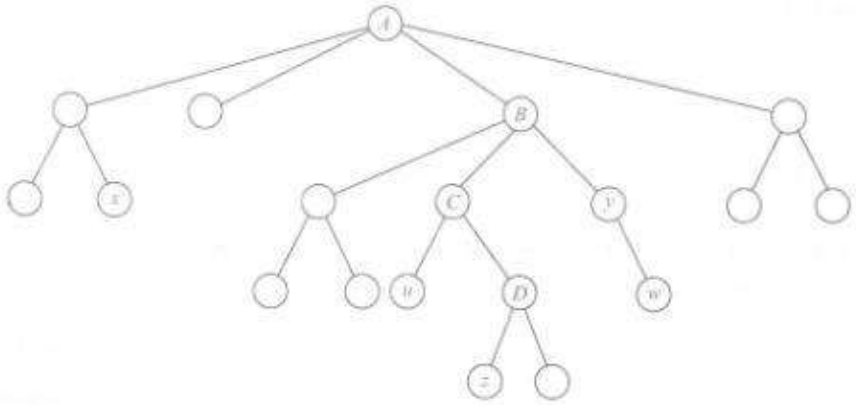


Figura 23.3 El antecesor común más próximo de cada petición en la secuencia de pares (x, y), (u, z), (w, x) (z, w) y (w, y) es A, C, A, B e y, respectivamente.

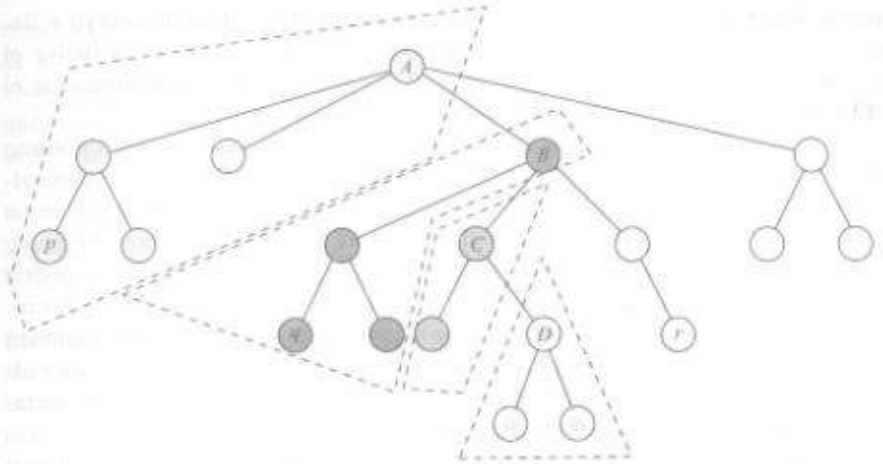


Figura 23.4 Los conjuntos justo antes de terminar la llamada recursiva correspondiente a D; D se marca como visitado y $ACP(D, v)$ es la sujeción de v al camino actual.

3. v está en el subárbol de D, por lo que $ACP(v, D) = D$. Observe que éste no es un caso especial pues la sujeción de v es D.

Todo lo que falta es asegurarnos de que en todo instante podemos determinar la sujeción de un nodo visitado. Esto se logra fácilmente utilizando el algoritmo de `unir/buscar`. Después de terminar una llamada recursiva, llamamos a `unir`. Por ejemplo, después de que termina la llamada recursiva con D en la Figura 23.4, se cambia a C la sujeción de todos los nodos en D. La nueva situación se muestra en la Figura 23.5. En consecuencia, necesitamos unir las dos clases de equivalencia

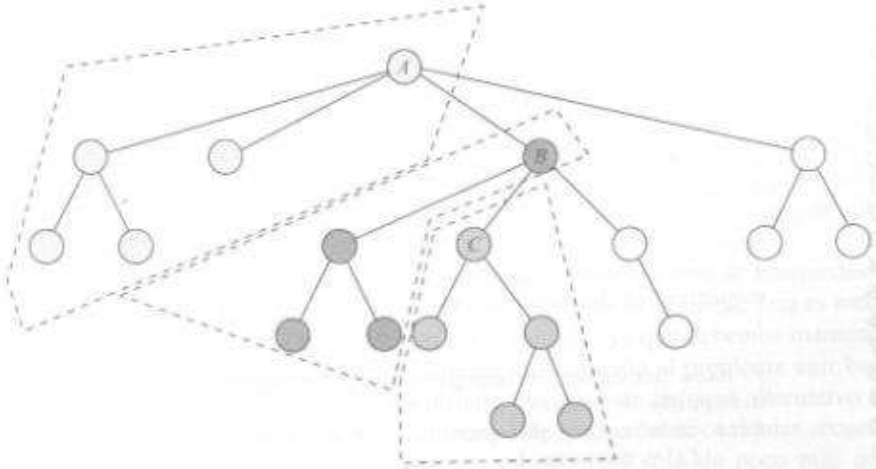


Figura 23.5 Después de que termina la llamada correspondiente a D, mezclamos el conjunto con sujeción en D con el conjunto con sujeción en C y calculamos $ACP(C, v)$ para todos los nodos v que están marcados, antes de completar la llamada recursiva correspondiente a C.

en una. En cualquier momento, podemos obtener la sujeción de un vértice v llamando a `buscar` sobre una estructura de partición. Ya que `buscar` devuelve el número de un conjunto, podemos utilizar un vector `sujecion` para almacenar el nodo sujeción correspondiente a cada conjunto.

La Figura 23.6 muestra una implementación en pseudocódigo del algoritmo ACP. Como hemos mencionado anteriormente, la operación `buscar` generalmente asume que los elementos del conjunto son 0, 1, ..., $N-1$, por lo que en un paso de preprocesamiento que calcula el tamaño del árbol almacenamos el número en preorden de cada nodo del árbol. Aunque un enfoque orientado a objetos podría intentar incorporar una traducción dentro de `buscar`, utilizando quizás un diccionario, en este caso ello sería computacionalmente ineficiente. Asumimos también que tenemos un vector de listas para almacenar las peticiones ACP. La lista i almacena las peticiones correspondientes al nodo i. Teniendo cuidado de estos detalles, el código es extraordinariamente breve.

Cuando se visita por primera vez un nodo u, se genera la sujeción a sí mismo en la línea 18 de la Figura 23.6. Entonces se procesan recursivamente sus hijos v, mediante la llamada en la línea 23. Después de que acaba cada llamada recursiva,

```
1  // Algoritmo del antecesor común más cercano
2  //
3  // Precondiciones (y objetos globales):
4  // 1. La estructura de partición está inicializada
5  // 2. Todos los nodos están inicialmente sin marcar
6  // 3. Los números en preorden ya están asignados en el campo num
7  // 4. Cada nodo puede almacenar su estado de marcaje
8  // 5. La lista de pares está disponible globalmente
9
10 Particion s = new Particion( tamanyoArbol );
11 Nodo [ ] sujecion = new Nodo[ tamanyoArbol ];
12
13 // main realiza la llamada acp( raiz )
14 // después de las inicializaciones necesarias
15
16 void acp( Nodo u )
17 {
18    sujecion[ s.buscar( u.num ) ]=u;
19
20    // Hace las llamadas en postorden
21    for( cada hijo v de u )
22    {
23       acp( v );
24       s.unir( s.buscar( u.num ), s.buscar( v.num ) );
25       sujecion[ s.buscar( u.num ) ]=u;
26       }
27
28       // Hace los cálculos acp para los pares que involucren a u
29       u.marcado = true;
30       for( cada v tal que acp( u, v ) ha sido solicitado )
31          if( v.marcado )
32             System.out.println("acp( " + u + ", " + v +
33                                " ) es " + sujecion[ s.buscar( v.num ) ] );
34 }
```

Figura 23.6 Pseudocódigo del problema ACP.

el subárbol se combina con la clase de equivalencia actual de u y nos aseguramos de actualizar la sujeción, en las líneas 24 y 25. Cuando todos los hijos han sido visitados recursivamente, podemos marcar u como procesado en la línea 29 y terminar comprobando todas las peticiones ACP que involucren a u en las líneas de la 30 a la 33[2].

23.3 El algoritmo de búsqueda rápida

Esta sección presenta los preparativos para la implementación eficiente de la estructura de partición. Hay dos estrategias básicas para resolver el problema unir/buscar. Una consigue que la instrucción `buscar` pueda ejecutarse en tiempo constante en el caso peor, y la otra logra lo mismo para la operación `unir`. Recientemente se ha demostrado que ambas cosas no pueden lograrse simultáneamente en tiempo constante (incluso amortizado) en el caso peor.

El primer enfoque se denomina *algoritmo de búsqueda rápida*. Para que la operación `buscar` sea rápida, podríamos mantener en un vector el nombre de la clase de equivalencia de cada elemento. `buscar` se reduce entonces a un simple acceso en tiempo constante. Supongamos que queremos realizar `unir`(a, b). Supongamos, también, que a está en la clase de equivalencia i y b está en j. Podemos recorrer el vector, intercambiando todas las i por j. Desgraciadamente, este recorrido cuesta un tiempo lineal. Por tanto una secuencia de $N-1$ operaciones `unir` (el máximo número posible, ya que tras ellas todos los elementos estarían en la misma clase) tardaría un tiempo cuadrático. En el caso típico, en el cual el número de búsquedas es subcuadrático, esto es claramente inaceptable.

Una idea consiste en mantener todos los elementos que están en la misma clase de equivalencia en una lista enlazada. Esto ahorra tiempo al actualizar, pues no tenemos que buscar en el vector completo. Sin embargo, esto no reduce por sí mismo el tiempo de ejecución asintótico, ya que todavía es posible realizar $\Theta(N^2)$ actualizaciones de las clases de equivalencia durante el transcurso del algoritmo.

Si mantenemos también el tamaño de las clases de equivalencia (y cuando se realiza una unión, cambiamos el nombre de la clase pequeña por el de la mayor) entonces el tiempo total invertido en N llamadas a unir es $O(N \log N)$. Esto se debe a que cada elemento puede cambiar de clase de equivalencia como mucho $\log N$ veces, ya que cada vez que cambia de clase, su nueva clase de equivalencia es al menos el doble de grande de lo que era la antigua (por lo que se puede aplicar el principio de duplicaciones sucesivas).

Esta estrategia asegura que cualquier secuencia de a lo sumo M búsquedas y $N-1$ uniones tardará, como mucho, $O(M + N \log N)$. Si M es lineal, ésta es todavía una solución costosa. También es un poco confusa, ya que debemos mantener listas ordenadas. La próxima sección examina una solución al problema unir/buscar que hace más fácil `unir`, pero más difícil `buscar`. Este enfoque alternativo es el algoritmo de unión rápida. Aun así, el tiempo de ejecución de cualquier secuencia de como mucho M búsquedas y $N-1$ uniones será sólo un poco más que $O(M + N)$, y sólo se usará un vector de enteros.

[2] Estrictamente hablando, u debería ser marcado en la última instrucción, pero haciéndolo antes tratamos las n
ACP(u, u).

Figura 23.7 Un bosque correspondiente a ocho elementos, inicialmente en conjuntos diferentes.

23.4 El algoritmo de unión rápida

Recordemos que el problema unir/buscar no requiere que la operación `buscar` devuelva un nombre específico, sólo que las búsquedas de dos elementos devuelvan la misma respuesta si y sólo si están en el mismo conjunto. Una idea puede ser utilizar un árbol para representar un conjunto, ya que cada elemento en el árbol tiene la misma raíz, por lo que la raíz puede usarse como nombre del conjunto.

Cada conjunto se representa por un árbol, con lo que en suma tenemos un *bosque*. El nombre de cada conjunto viene dado por el nodo en la raíz. Nuestros árboles no son necesariamente binarios, pero su representación es sencilla porque la única información necesaria es el padre de cada nodo. Por tanto, sólo necesitamos un vector de enteros: cada entrada `p[i]` en el vector representa el padre del elemento `i`, y para indicar que estamos en la raíz podemos utilizar -1 como padre. La Figura 23.7 muestra un bosque de árboles triviales y el vector que lo representa.

Para realizar la unión de dos conjuntos, mezclamos los dos árboles haciendo que la raíz de uno de ellos apunte a la raíz del otro. Debería estar claro que esta operación tiene un coste constante. Las Figuras 23.8, 23.9 y 23.10 representa el bosque después de `unir`(4,5), `unir`(6,7) y `unir`(4,6), donde hemos adoptado el convenio de que la nueva raíz después de `unir`(x,y) es x.

La búsqueda del elemento x se realiza devolviendo la raíz del árbol que contiene x. El tiempo necesario para realizar esta operación es proporcional al número de nodos en el camino de x a la raíz. La estrategia de unión descrita anteriormente nos permite crear un árbol en el que todo nodo esté en el camino a x, obteniéndose

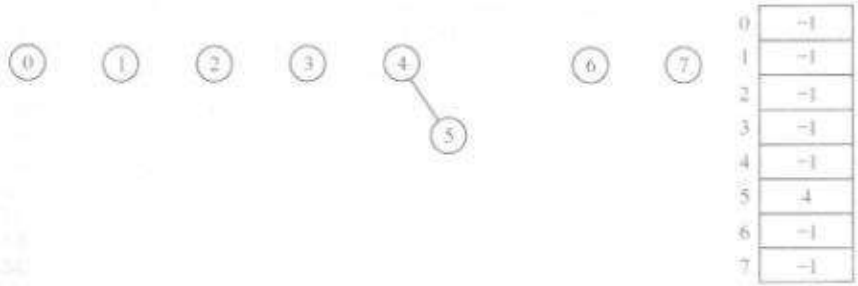


Figura 23.8 El bosque después de la unión de los árboles con raíces 4 y 5.

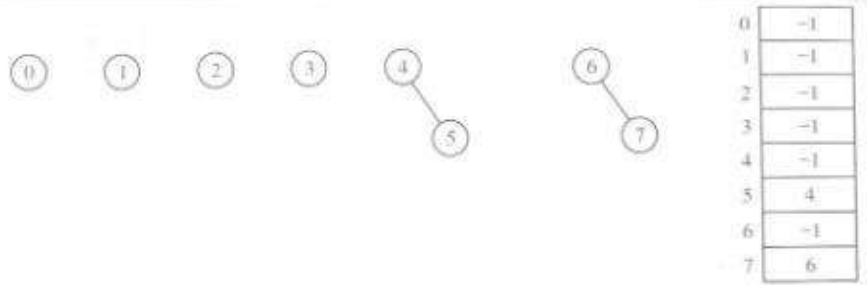


Figura 23.9 El bosque después de la unión de los árboles con raíces 6 y 7.

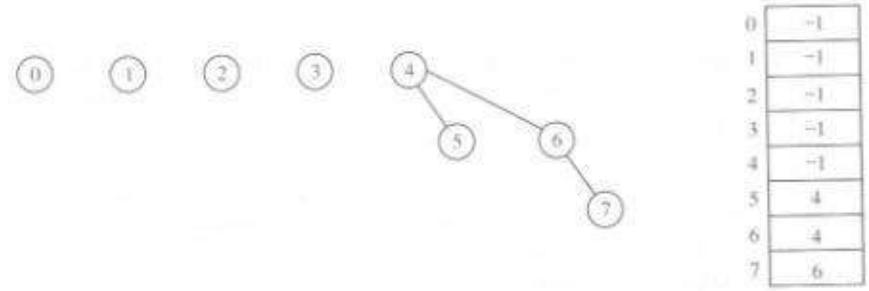


Figura 23.10 El bosque después de la unión de los árboles con raíces 4 y 6.

un tiempo de ejecución en el caso peor $\Theta(N)$ para cada operación `buscar`. Típicamente (como muestran las dos aplicaciones previas), el tiempo de ejecución se calcula para una secuencia de M instrucciones entremezcladas. En el caso peor, M operaciones consecutivas podrían tardar un tiempo $\Theta(MN)$.

En general, un tiempo de ejecución cuadrático para una secuencia de operaciones no es aceptable. Afortunadamente, hay varias formas sencillas de asegurar que este tiempo de ejecución no se dé nunca.

23.4.1 Algoritmos de unión inteligentes

Las uniones anteriores se hicieron de una forma bastante arbitraria, haciendo que el segundo árbol pasara a ser un subárbol del primero. Una mejora sencilla es exigir que siempre el árbol más pequeño pase a ser subárbol del árbol mayor, rompiendo los empates con un criterio cualquiera; este método se denomina *unión por tamaño*. Las tres uniones de la sección anterior corresponden a sendos empates, por lo que podemos considerar que se realizaron por tamaño. Si la siguiente operación fuera `unir`(3,4), se formaría el árbol de la Figura 23.11. Si no hubiéramos utilizado la heurística del tamaño, se habría generado un bosque más profundo (tres nodos, en lugar de uno sólo, verían incrementada su profundidad en un nivel).

Podemos demostrar que si las uniones se hacen por tamaño, la profundidad de cualquier nodo nunca es mejor que $\log N$. Para ver esto, observamos que inicialmente todo nodo está a profundidad 0. Cuando su profundidad aumenta como resultado de una unión, se coloca en un árbol cuyo tamaño es al menos el doble del anterior. En consecuencia su profundidad puede incrementarse como mucho

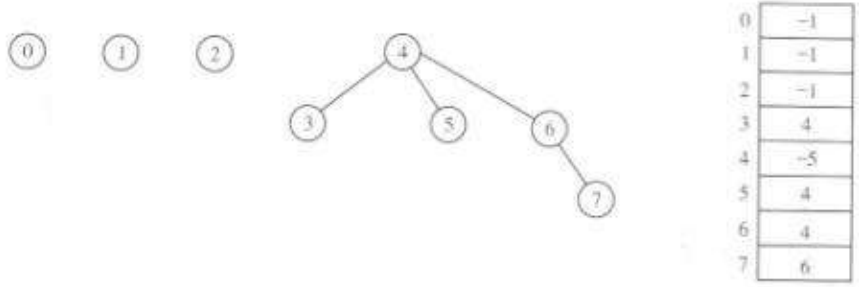


Figura 23.11 El bosque generado por uniones por tamaño, con los tamaños representados por medio de sus inversos negativos.

log N veces. (Utilizamos este mismo argumento en el algoritmo de búsqueda rápida de la Sección 23.3.) Esto implica que el tiempo de ejecución de una operación `buscar` es $O(\log N)$ y que una secuencia de M operaciones tarda como mucho $O(M \log N)$. El árbol de la Figura 23.12 muestra el peor caso posible después de 15 uniones, que se obtiene cuando todas las uniones son entre árboles de igual tamaño. Dicho caso corresponde a los denominados *árboles binomiales*. Los árboles binomiales tienen otras aplicaciones en estructuras de datos avanzadas.

Para implementar esta estrategia, necesitamos mantener información sobre el tamaño de cada árbol. Ya que sólo estamos utilizando un vector, podemos hacer que la entrada del vector correspondiente a la raíz contenga el inverso *negativo* del tamaño del árbol, como muestra la Figura 23.11. En consecuencia, la representación inicial será un vector con todas las entradas iguales a -1. Cuando se realiza una unión, comparamos los tamaños; el nuevo tamaño es la suma de los viejos. Por tanto, la unión por tamaño no es en absoluto complicada y no requiere ningún espacio extra. También es rápida en promedio. Esto se debe a que cuando se realizan uniones arbitrarias, habitualmente se mezclan conjuntos muy pequeños (usualmente de un solo elemento) con conjuntos grandes. El análisis matemático de esto es bastante complejo; la bibliografía al final del capítulo proporciona algunos punteros a la literatura.

Una implementación que también garantiza una profundidad logarítmica es la *unión por altura*. Mantenemos la altura de los árboles en vez de su tamaño y realizamos las uniones haciendo que el árbol menos profundo pase a ser subárbol del más profundo. Éste es un algoritmo sencillo, ya que la altura del árbol sólo aumenta en una unidad cuando los dos árboles unidos tienen la misma altura. En consecuencia,

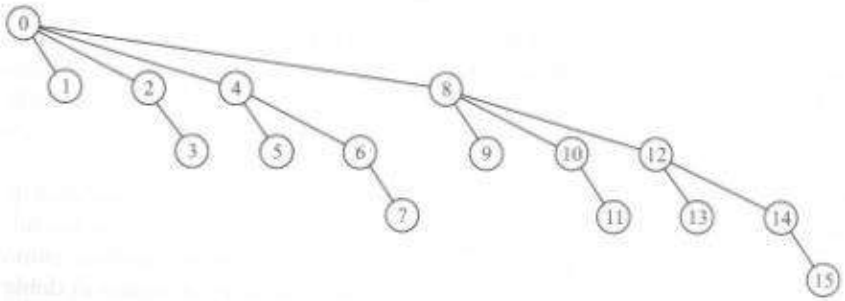


Figura 23.12 Árbol del caso peor con $N = 16$.

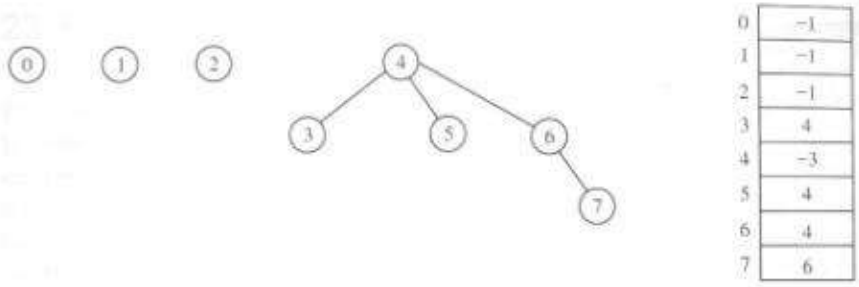


Figura 23.13 Un bosque generado por la unión por altura, con ésta codificada por medio de valores negativos.

la unión por altura es una variante trivial de la unión por tamaño. Ya que las alturas empiezan en 0, almacenamos el inverso negativo del número de nodos en el camino más profundo, en vez de la altura. Esto se muestra en la Figura 23.13.

23.4.2 Compresión de caminos

El algoritmo unir/buscar, tal y como se ha descrito hasta el momento, es bastante aceptable en la mayoría de las ocasiones. Es muy simple y lineal en promedio para una secuencia de M instrucciones. Sin embargo, el caso peor todavía es poco atractivo. Esto es así, porque no es obvio que una secuencia de operaciones `unir` de una aplicación particular (como el problema ACP) sea aleatoria (de hecho, para árboles concretos, se aleja mucho de serlo. En consecuencia, resulta deseable que intentemos cambiar las cosas de alguna forma, de manera que podamos garantizar una cota mejor para el caso peor de una secuencia de M operaciones. Parece probable que no haya más mejoras posibles del algoritmo unir, ya que el caso peor se alcanza cuando se mezclan árboles de igual tamaño. La única forma de acelerar el algoritmo, sin alterar completamente la estructura de datos, es hacer algo inteligente en la operación `buscar`.

Esta operación inteligente es la *compresión de caminos*. Claramente, después de realizar una búsqueda del elemento x, tendría sentido cambiar el padre de x tomando como tal la raíz del árbol. De esta forma, una segunda búsqueda de x, o de cualquier elemento en el subárbol de x, sería más rápida. Sin embargo, no hay necesidad de detenernos ahí. Podríamos también cambiar el padre de todos los nodos en el camino de acceso hasta x. El efecto de la compresión de caminos es que se cambia el padre de *todos* los nodos en el camino desde x a la raíz. La Figura 23.14 muestra el efecto de la compresión de caminos después de `buscar`(14), en el árbol del caso peor de la Figura 23.12. Con el cambio extra de dos padres que hemos realizado, los nodos 12 y 13 están ahora una posición más cerca de la raíz y los nodos 14 y 15 están dos posiciones más cerca. Por tanto los accesos futuros se beneficiarán (eso esperamos) del trabajo extra de la compresión de caminos. Observe que las uniones subsiguientes harán que los nodos levantados vuelvan a ser llevados a mayor profundidad.

Cuando las uniones se producen de forma arbitraria, la compresión de caminos es una buena idea, porque hay gran número de nodos profundos; éstos son acercados a la raíz por la compresión de caminos. Se ha demostrado en este caso que,

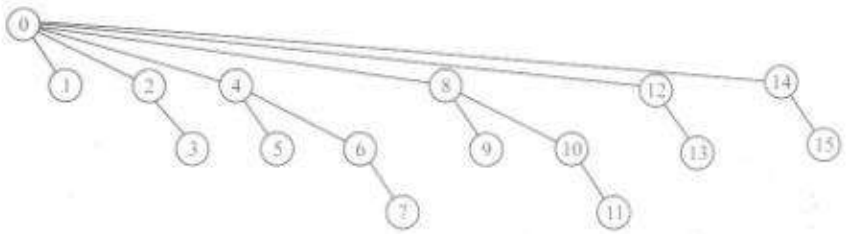


Figura 23.14 Compresión de caminos que resulta al ejecutar `buscar(14)` en el árbol de la Figura 23.12.

con la compresión de caminos, una secuencia de M operaciones requiere como mucho un tiempo $O(M \log N)$, por lo que la compresión de caminos por sí sola garantiza un coste amortizado logarítmico para la operación `buscar`.

La compresión de caminos es perfectamente compatible con la unión por tamaño, por lo que ambas rutinas pueden implementarse a la vez. Sin embargo, la compresión de caminos no es completamente compatible con la unión por alturas, porque la compresión de caminos puede cambiar la altura de los árboles. No está claro cómo calcularla de nuevo eficientemente, por lo que no intentamos seguir este camino. En consecuencia, los valores almacenados en cada árbol se convierten en estimaciones de la altura (las cuales en lo sucesivo denominaremos *rangos*), pero esto no es un problema. El algoritmo resultante recibe el nombre de *unión por rango*. Como veremos en la Sección 23.6, la combinación de una unión inteligente y la compresión de caminos garantiza un tiempo de ejecución casi lineal para una secuencia de M operaciones.

```
1  package EstructurasDatos;
2
3  // Clase EstrPartición
4  //
5  // CONSTRUCCIÓN: indicando el número inicial de conjuntos
6  //
7  // ******************OPERACIONES PÚBLICAS*********************
8  // void unir( raiz1, raiz2 ) --> Mezcla los dos conjuntos
9  // int buscar( x )           --> Devuelve el conjunto que contiene a x
10 // ******************************ERRORES**********************************
11 // Sin comprobación de errores
12
13 /**
14  * Estructura de partición, utilizando unión por rango
15  * y compresión de caminos.
16  */
17 public class EstrParticion
18 {
19     public EstrParticion( int numElementos )
20       { /* Figura 23.16 */ }
21     public void unir( int raiz1, int raiz2 )
22       { /* Figura 23.16 */ }
23     public int buscar( int x )
24       { /* Figura 23.16 */ }
25
26     private int [ ] vector;
27 }
```

Figura 23.15 Esqueleto de la clase partición

23.5 Implementación en Java

El esqueleto de la clase de la estructura de partición se muestra en la Figura 23.15; la implementación se muestra en la Figura 23.16. Se ha omitido la comprobación de errores, para evitar oscurecer los detalles del algoritmo. Por supuesto, un programa robusto debería comprobar estos errores. Sin la comprobación de errores, el algoritmo completo es sorprendentemente breve.

En la rutina, la unión se realiza sobre las raíces de los árboles. Algunas veces la operación se implementa pasándole dos elementos cualesquiera y teniendo que utilizar `buscar` para determinar las raíces.

```
1  /**
2   * Constructor de la estructura de partición.
3   * @param numElementos el número inicial de conjuntos.
4   */
5  public EstrParticion( int numElementos )
6  {
7      vector = new int [ numElementos ];
8      for( int i = 0; i < vector.length; i++ )
9          vector[ i ] = -1;
10 }
11
12 /**
13  * unir une dos conjuntos disjuntos utilizando la altura.
14  * Asumimos que raiz1 y raiz2 son distintos.
15  * @param raiz1 la raiz del primer conjunto.
16  * @param raiz2 la raiz del segundo conjunto.
17  */
18 public void unir( int raiz1, int raiz2 )
19 {
20     if( vector[ raiz2 ] < vector[ raiz1 ] ) // raiz2 más profunda
21         vector[ raiz1 ] = raiz2; // raiz2 nueva raiz
22     else
23     {
24         if( vector[ raiz1 ] == vector[ raiz2 ] )
25             vector[ raiz1 ]--;   // Actualizar altura
26         vector[ raiz2 ] = raiz1; // raiz1 nueva raiz
27     }
28 }
29
30 /**
31  * Búsqueda con compresión de caminos.
32  * Se omite la detección de errores.
33  * @param x el elemento buscado.
34  * @return el conjunto que contiene a x.
35  */
36 public int buscar( int x )
37 {
38     if( vector[ x ] < 0 )
39         return x;
40     else
41         return vector[ x ] = buscar( vector[ x ] );
42 }
```

Figura 23.16 Implementación de la estructura de partición sin detección de errores.

El procedimiento interesante es `buscar`. Una vez la búsqueda se ha realizado recursivamente, se asigna a `vector[x]` la raíz que después es devuelta. Ya que esto ocurre de forma recursiva, todos los nodos en el camino pasarán a apuntar (como padre) a la raíz original.

23.6 Caso peor de la unión por rango y la compresión de caminos

Cuando se utilizan ambas heurísticas, el algoritmo es casi lineal en el caso peor. Concretamente, el tiempo necesario en el caso peor para procesar una secuencia de como mucho $N - 1$ operaciones de unión y M de búsqueda es $\Theta(M\alpha(M, N))$ (supuesto $M \geq N$), donde $\alpha(M, N)$ es la función inversa de la *función de Ackermann*, que se define de la siguiente manera[3]:

$$\begin{aligned} A(1, j) &= 2^j && j \geq 1 \\ A(i, 1) &= A(i-1, 2) && i \geq 2 \\ A(i, j) &= A(i-1, A(i, j-1)) && i, j \geq 2 \end{aligned}$$

A partir de esto definimos

$$\alpha(M, N) = \min\{i \geq 1 \mid A(i, \lfloor M/N \rfloor) > \log N\}.$$

Podríamos querer calcular algunos valores, pero para todos los propósitos prácticos, $\alpha(M, N) \leq 4$, que es lo único importante aquí. Por ejemplo, para cualquier $j > 1$, tenemos

$$\begin{aligned} A(2, j) &= A(1, A(2, j-1)) \\ &= 2^{A(2, j-1)} \\ &= 2^{2^{2^{\cdot^{\cdot^{\cdot}}}}} \end{aligned}$$

donde el número de veces que aparece el 2 en el exponente es j. $F(N) = A(2, N)$ se denomina habitualmente la función de Ackermann de una sola variable. El inverso de la función de Ackermann de una sola variable, representado a veces como $\log^* N$, es el número de veces que se tiene que aplicar el logaritmo a N hasta que $N \leq 1$. Por tanto, $\log^* 65.536 = 4$ porque $\log\log\log\log 65.536 = 1$. $\log^* 2^{65.536} = 5$. Sin embargo, tenga en cuenta que $2^{65.536}$ tiene más de 20.000 dígitos. $\alpha(M, N)$ crece aún más lentamente que $\log^* N$. Por ejemplo, $A(3, 1) = A(2, 2) = 2^{2^2} = 16$. Por tanto, para $N < 2^{16}$, $\alpha(M, N) \leq 3$. Es más, ya que $A(4, 1) = A(3, 2) = A(3, A(2, 1)) = A(2, 16)$, el cual es 2 elevado a una pila de dieciséis 2, en la práctica, $\alpha(M, N) \leq 4$. Sin embargo, $\alpha(M, N)$ no es constante cuando M es algo mayor que N, por lo que el tiempo de ejecución no es estrictamente lineal[4].

kermann se define frecuentemente como $A(i, j) = j + 1$, para $j \geq 1$. La forma utilizada en este texto crece aún nto, el inverso crece más lentamente.

argo, que si $M = N \log^* N$, $\alpha(M, N)$ es como mucho 2. Por tanto, siempre que M sea sólo algo más que lineal, sí será lineal en M.

En el resto de esta sección, probaremos un resultado algo más débil. Mostraremos que cualquier secuencia de $M = \Omega(N)$ operaciones de unión o búsqueda requieren un tiempo total $O(M \log^* N)$. La misma cota sigue siendo cierta si utilizamos la unión por tamaño en vez de la unión por rango. Este análisis es, probablemente, el más complicado del libro y uno de los primeros análisis realmente complejos jamás realizados para un algoritmo trivial de implementar. Extendiendo esta técnica, podríamos probar también la cota más fuerte afirmada previamente.

23.6.1 Análisis del algoritmo unir/buscar

En esta sección establecemos una cota bastante ajustada del tiempo de ejecución de una secuencia de $M = \Omega(N)$ operaciones de unión y búsqueda. Las operaciones `unir` y `buscar` pueden producirse en cualquier orden, pero `unir` se hace por rango y `buscar` se hace con compresión de caminos.

Empezamos con algunos teoremas que tienen que ver con el número de nodos de rango r. Intuitivamente, debido a la regla de la unión por rango, hay muchos más nodos de rango pequeño que de rango grande. En particular, puede haber como mucho un nodo de rango $\log N$. Lo que queremos hacer es calcular una cota tan precisa como sea posible del número de nodos de cierto rango r. Ya que los rangos sólo cambian cuando se realiza la operación `unir` (y sólo cuando los dos árboles tienen el mismo rango), podemos hacer el cálculo ignorando la compresión de caminos. Esto se hace en el Teorema 23.1

> *En ausencia de la compresión de caminos, cuando una secuencia de instrucciones* `unir` *está siendo ejecutada, un nodo de rango r debe tener al menos 2^r descendientes, incluyéndole a él mismo.*
>
> *La demostración es por inducción. La base, cuando $r = 0$, es claramente cierta. Sea T el árbol de rango r con el menor número de descendientes y sea x la raíz de T. Supongamos que la última unión que involucró a x fue entre T_1 y T_2. Supongamos que la raíz de T_1 era x. Si T_1 tuviera rango r, entonces T_1 sería un árbol de rango r con menos descendientes que T. Esto contradice la hipótesis de que T tiene el menor número de descendientes posible. Por tanto, el rango de T_1 es como mucho $r - 1$. El rango de T_2 es como mucho el rango de T_1, pues la unión se hace por rango. Ya que T tiene rango r y el rango sólo podría incrementarse debido a T_2, se deduce que el rango de T_2 es $r - 1$. En consecuencia, el rango de T_1 es también $r - 1$. Por la hipótesis de inducción, cada uno de estos árboles tiene como mucho 2^{r-1} descendientes, lo que da un total de 2^r, lo que demuestra el teorema.*

El Teorema 23.1 afirma que si no se hace compresión de caminos, cualquier nodo de rango r tiene al menos 2^r descendientes. La compresión de caminos puede cambiar esto, ya que elimina descendientes de un nodo. Sin embargo, cuando se realizan uniones, aun con compresión de caminos, utilizamos los rangos, que son estimaciones de la altura. Estos rangos se comportan como si no hubiera compresión de caminos. Por tanto, cuando se está acotando el número de nodos de rango r, la compresión de caminos puede ser ignorada. Esto se hace en el Teorema 23.2.

El número de nodos de rango r es como mucho $N/2^r$.

Sin la compresión de caminos, cada nodo de rango r es la raíz de un subárbol de al menos 2^r nodos. Ningún otro nodo en el árbol puede tener rango r. Por tanto todos los subárboles de rango r son disjuntos. En consecuencia, hay como mucho $N/2^r$ subárboles disjuntos y por tanto $N/2^r$ nodos de rango r.

El Teorema 23.3 puede parecer bastante obvio, pero es crucial en el análisis.

En cualquier punto del algoritmo unir/buscar, los rangos de los nodos de un camino desde una hoja a la raíz crecen de forma monótona.

El teorema es obvio si no hay compresión de caminos. Si después de la compresión de caminos, un nodo v es descendiente de otro w, claramente v tiene que ser descendiente de w cuando sólo se realizan las operaciones de unión. Por tanto, el rango de v es estrictamente menor que el rango de w.

Presentemos ahora un resumen de nuestros resultados preliminares. El Teorema 23.2 nos indica a cuántos nodos se les puede asignar rango r. Ya que los rangos sólo se asignan mediante uniones, las cuales no tienen idea de la compresión de caminos, el Teorema 23.2 es válido en cualquier paso del algoritmo unir/buscar, aun en medio de compresiones. El Teorema 23.2 es ajustado, en el sentido de que para cada rango r es posible llegar a tener $N/2^r$ nodos. Sin embargo, es ligeramente holgado, ya que lo que no es posible es alcanzar simultáneamente todas las cotas de los distintos rangos. Mientras que el Teorema 23.2 describe el número de nodos de rango r, el Teorema 23.3 nos indica su distribución. Como era de esperar, el rango de los nodos crece estrictamente a lo largo de cada camino desde una hoja a la raíz.

Ahora estamos preparados para demostrar el teorema principal. El plan básico es el siguiente: una búsqueda de cualquier nodo v requiere un tiempo proporcional al número de nodos en el camino de v a la raíz. Durante cada búsqueda cargaremos una unidad de coste por cada nodo en el camino de v a la raíz. Para ayudar a contabilizar los cargos, colocaremos una moneda imaginaria en cada nodo del camino. Éste es un truco de contabilidad que no es parte del programa. Representa una alternativa, aunque similar en el fondo, al uso de la función de potencial en el análisis amortizado de los árboles de ensanchamiento y los montículos sesgados. Cuando termina el algoritmo *recogeremos* todas las monedas depositadas para determinar el tiempo total invertido en su ejecución.

Como truco de contabilidad adicional, depositaremos dos tipos de monedas diferentes, digamos euros y dólares. Mostraremos que durante la ejecución del algoritmo, sólo podremos depositar cierto número de euros durante cada búsqueda (independientemente de cuántos nodos haya). Mostraremos también que sólo podemos depositar cierta cantidad de dólares en cada nodo (independientemente del número de búsquedas). Sumando estos dos totales obtendremos una cota del número total de monedas que pueden ser depositadas.

Ahora esbozamos con más detalle nuestro método de contabilidad. Dividiremos los nodos por su rango. Luego dividiremos los rangos en grupos de rangos. En cada búsqueda, depositaremos euros en un fondo común general y algunos dólares en nodos específicos. Para calcular el número total de dólares depositados, comenzamos por calcular lo depositado en cada nodo. Sumando lo depositado en cada nodo de rango r, obtendremos el depósito total por rango. Luego, sumaremos todos los depósitos de cada rango r en un mismo grupo g, obteniendo el total depositado en cada grupo de rangos g. Finalmente, sumaremos todos los depósitos en cada grupo de rangos g para obtener el número total de dólares depositados. Sumando esto al número de euros en el fondo común obtenemos la respuesta final.

Como hemos mencionado anteriormente, repartiremos los rangos en grupos. Cada rango r irá al grupo $G(r)$, siendo G determinada más adelante (para equilibrar los cargos en euros y doláres). El rango mayor en cada grupo de rangos se denotará $F(g)$, de modo que $F = G^{-1}$ es una *inversa* de G. El número de rangos en un grupo de rangos, $g > 0$, es por tanto $F(g) - F(g-1)$. Claramente, $G(N)$ es una cota superior muy relajada del mayor grupo de rangos. Como ejemplo, supongamos que dividimos los rangos como en la Figura 23.17. En este caso, $G(r) = \lceil \sqrt{r} \rceil$. El mayor rango en el grupo g es $F(g) = g^2$. Observemos también que el grupo g contiene los rangos del $F(g-1)+1$ al $F(g)$, inclusive. Esta fórmula no se puede aplicar al grupo de rangos 0, por lo que por conveniencia haremos que el grupo de rangos 0 sólo contenga a los elementos de rango 0. Observe que los grupos se forman con rangos consecutivos.

Como mencionamos anteriormente en el capítulo, cada instrucción `unir` requiere un tiempo constante, con tal de que cada raíz mantenga su rango. Por tanto, las uniones son esencialmente gratis, en lo que se refiere a esta demostración.

Cada búsqueda requiere un tiempo proporcional al número de nodos en el camino desde el nodo que representa al elemento i accedido hasta la raíz. Por tanto, depositaremos un euro en cada vértice del camino. Sin embargo, si esto fuera todo lo que hiciésemos, no podríamos esperar mucho de la cota, ya que no estamos sacando ventaja de la compresión de caminos. En consecuencia, debemos utilizar en nuestro análisis alguna propiedad de la compresión de caminos. La observación clave es que, como resultado de la compresión de caminos, un nodo obtiene un nuevo padre que tiene un mayor rango que el viejo.

Para incorporar este hecho en la demostración, utilizaremos el siguiente método imaginativo de contabilidad: para cada nodo v en el camino desde el nodo i accedido a la raíz, depositamos una moneda de uno de los dos tipos posibles:

Grupo	Rango
0	0
1	1
2	2,3,4
3	de 5 a 9
4	de 10 a 16
i	de $(i-1)^2+1$ a i^2

Figura 23.17 Posible división de los rangos en grupos.

1. Si v es la raíz, o si el padre de v es la raíz o si el padre de v está en un grupo de rangos diferente al de v, cargamos una unidad en concepto de aplicación de esta regla, depositando un euro en el fondo común.
2. En otro caso, depositamos un dólar en el nodo.

El Teorema 23.4 afirma que la contabilidad es acertada.

Para cualquier operación de búsqueda, el número total de monedas depositadas, en el fondo común o en los nodos, es exactamente igual al número de nodos accedido durante la búsqueda.

Obvia.

Por tanto, sólo necesitamos sumar todos los euros depositados bajo la regla 1 con todos los dólares depositados bajo la regla 2. Antes de seguir con la demostración, esbocemos las ideas intuitivas. Los dólares son depositados en un nodo cuando es comprimido y su padre está en el mismo grupo de rangos que el nodo. Ya que después de cada compresión de caminos el nodo obtiene un padre de grado mayor y dado que el tamaño de un grupo de rangos es finito, ocasionalmente el nodo obtendrá un padre que no está en el mismo grupo. En consecuencia, el número de dólares que pueden colocarse en un nodo está limitado. Este número es aproximadamente igual al tamaño del grupo de rangos del nodo. Por otro lado, los cargos en euros también están limitados, esencialmente por el número de grupos de rangos. En consecuencia, queremos elegir los grupos de rangos de forma que sean pequeños (para limitar los cargos en dólares), pero que no haya muchos (para limitar los cargos en euros). Ahora estamos preparados para razonar con detalle por medio de una serie de teoremas, del 23.5 al 23.10.

Durante todo el algoritmo, los depósitos totales de euros bajo la regla 1 totalizan $M(G(N)+2)$.

En cada búsqueda, se depositan como mucho dos euros, debido a la raíz y a su hijo. Por el Teorema 23.3, los vértices en el camino hacia la raíz crecen monótonamente en rango, y por tanto, el grupo de rangos nunca decrece al ascender por el camino. Ya que como mucho hay $G(N)$ grupos de rangos (además del grupo 0), sólo otros $G(N)$ vértices pueden ser utilizados para generar un depósito por aplicación de la regla 1 dentro de una misma búsqueda. Por tanto, en cada búsqueda pueden colocarse a lo sumo $G(N)+2$ euros en el fondo común. En consecuencia, a lo largo de una secuencia de M búsquedas como mucho $M(G(N)+2)$ euros pueden ser depositados por aplicación de la regla 1.

En cada nodo individual en el grupo de rangos g, el número total de euros depositados es $F(g)$.

Si en aplicación de la regla 2 se deposita un dólar en un vértice v, v será movido por la compresión de caminos y pasará a tener un padre de rango mayor que su padre antiguo. Ya que el rango mayor de su grupo es $F(g)$, tenemos la garantía de que después de depositar $F(g)$ monedas, el padre de v no estará más en el grupo de rangos de v.

La cota del Teorema 23.6 puede mejorarse utilizando el tamaño del grupo de rangos en vez de su mayor miembro. Sin embargo, esto no mejoraría la cota obtenida para el algoritmo unir/buscar.

El número de nodos, $N(g)$ en el grupo de rangos $g>0$ es como mucho $N/2^{F(g-1)}$.

Por el Teorema 23.2, hay como mucho $N/2^r$ nodos de rango r. Al sumar sobre el grupo de rangos g, obtenemos

$$\begin{aligned}
N(g) &\leqslant \sum_{r=F(g-1)+1}^{F(g)} \frac{N}{2^r} \\
&\leqslant \sum_{r=F(g-1)+1}^{\infty} \frac{N}{2^r} \\
&\leqslant N \sum_{r=F(g-1)+1}^{\infty} \frac{1}{2^r} \\
&\leqslant \frac{N}{2^{F(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \\
&\leqslant \frac{2N}{2^{F(g-1)+1}} \\
&\leqslant \frac{N}{2^{F(g-1)}}.
\end{aligned}$$

El máximo número de dólares depositados en todos los vértices en el grupo de rangos g es como mucho $NF(g)/2^{F(g-1)}$.

El resultado se obtiene mediante una simple multiplicación de las cantidades obtenidas en los Teoremas 23.6 y 23.7.

La cantidad total depositada por aplicación de la regla 2 es, a lo sumo, de $N\Sigma_{g=1}^{G(N)} F(g)/2^{F(g-1)}$ dólares.

Ya que el grupo 0 contiene sólo elementos de rango 0, no puede contribuir a los cargos por aplicación de la regla 2 (no puede tener un padre en el mismo grupo de rangos). Entonces, la cota se obtiene sumando las contribuciones de los otros grupos de rangos.

Grupo	Rango
0	0
1	1
2	2
3	3,4
4	de 5 a 16
5	de 17 a 65.536
6	de 65.537 a $2^{65.536}$
7	Rangos realmente grandes

Figura 23.18 División concreta de los rangos en grupos utilizada en la demostración.

Tenemos, pues, los depósitos bajo las reglas 1 y 2. El total es

$$M(G(N) + 2) + N\Sigma_{g=1}^{G(N)} F(g)/2^{F(g-1)}. \qquad (23.1)$$

Todavía no hemos especificado $G(N)$ o su inversa $F(N)$. Obviamente, somos libres de elegir virtualmente cualquier cosa, pero parece razonable elegir $G(N)$ de forma que su participación en la Ecuación 23.1 nos conduzca a minimizar ésta. Sin embargo, si $G(N)$ es demasiado pequeña, $F(N)$ será grande, perjudicando la cota obtenida. Aparentemente una buena elección es que $F(i)$ sea la función definida recursivamente como $F(0) = 0$ y $F(i) = 2^{F(i-1)}$. Esto nos conduce a $G(N) = 1 + \lfloor \log^* N \rfloor$. La Figura 23.18 muestra cómo se clasifican los rangos. Observe que el grupo 0 contiene sólo el rango 0, lo que requeríamos en la demostración del Teorema 23.9. F es muy parecida a la función de Ackermann de una variable, diferenciándose sólo en el caso básico. Con esta elección de F y G, podemos completar el análisis.

El tiempo de ejecución del algoritmo unir/buscar con $M = \Omega(N)$ operaciones de `buscar` es $O(M \log^* N)$.

Sustituimos las definiciones de F y G en la Ecuación 23.1. El número total de euros es $O(MG(N)) = O(M \log^* N)$. Ya que $F(g) = 2^{F(g-1)}$, el número total de dólares es $NG(N) = O(N \log^* N)$. Del hecho de que $M = \Omega(N)$, se deduce la cota.

Observe que tenemos más euros que dólares. La función $\alpha(M, N)$ equilibra las cosas; ésta es la razón por la que se genera una buena cota.

Resumen

En este capítulo hemos estudiado una estructura de datos muy sencilla para mantener una estructura de partición. Cuando se ejecuta la operación `unir`, no importa, en lo que se refiere a la corrección, qué conjunto mantiene su nombre. Una valiosa lección que deberíamos aprender aquí es que es muy importante considerar alternativas cuando un paso particular no está totalmente especificado. El paso de unión es flexible. Sacando ventaja de este hecho, podemos conseguir un algoritmo más eficiente.

La comprensión de caminos fue históricamente una de las primeras formas de auto-ajustamiento, que nosotros hemos visto ya en otros lugares (árboles de ensanchamiento, montículos sesgados). Su uso aquí es extremadamente interesante desde un punto de vista teórico, ya que fue uno de los primeros ejemplos de un algoritmo sencillo con un análisis en el caso peor no tan simple.

Elementos del juego

algoritmo de búsqueda rápida Implementación de unir/buscar donde `buscar` es una operación constante en tiempo.

algoritmo de Kruskal Algoritmo que selecciona aristas en orden creciente de costes y añade una arista al árbol si con ello no se crea un ciclo.

algoritmo de unión rápida Implementación de unir/buscar donde `unir` es una operación constante en tiempo.

algoritmo unir/buscar Algoritmo que se ejecuta procesando operaciones `unir` y `buscar` utilizando una estructura de partición.

árbol de recubrimiento Árbol formado por aristas de un grafo no dirigido que conectan todos los vértices.

árbol de recubrimiento mínimo Subgrafo conexo de G que cubre todos los vértices con coste total mínimo. Es un problema fundamental en la teoría de grafos.

bosque Colección de árboles.

clase de equivalencia La clase de equivalencia de un elemento x en un conjunto S es el subconjunto de S que contiene todos los elementos relacionados con x.

compresión de caminos Hace que cada nodo accedido pase a ser hijo de la raíz hasta que ocurra otra unión.

estructura de partición Estructura de datos utilizada para manipular conjuntos disjuntos.

función de Ackermann Función que crece muy rápidamente. Su inversa está acotada en la práctica por 4.

operaciones de la estructura de partición Operaciones básicas para manipular la estructura de partición. Son `unir` y `buscar`.

problema del antecesor común más próximo (ACP) Dados un árbol y una lista de pares de nodos del árbol, encontrar el antecesor común más próximo de cada par de nodos. ACP es importante en ciertos algoritmos de grafos y en biología computacional.

rango En los algoritmos de la estructura de partición, la estimación de la altura de un nodo.

relación Definida sobre un conjunto nos dice si cada par de elementos están o no relacionados.

relación de equivalencia Relación reflexiva, simétrica y transitiva.

unión por altura Hace que el árbol de menor altura sea hijo del árbol con mayor altura durante la unión.

unión por rango Unión según la altura cuando se hace compresión de caminos.
unión por tamaño Hace que el árbol más pequeño sea hijo del árbol mayor durante la unión.

Errores comunes

1. `unir` asume que sus parámetros son raíces de árboles. Se producirían estragos si en efecto no lo fueran. Una implementación más cuidadosa realizaría dicha comprobación.

En Internet

La clase partición está disponible en Internet en el directorio **DataStructures**. El nombre del fichero es el siguiente:

DisjSet.java Traducido como `EstrParticion.java`, contiene la implementación de la clase de la estructura de partición.

Ejercicios

Cuestiones breves

23.1. Muestre el resultado de la siguiente secuencia de instrucciones: `unir(1,2)`, `unir(3,4)`, `unir(3,5)`, `unir(1,7)`, `unir(3,6)`, `unir(8,9)`, `unir(1,8)`, `unir(3,10)`, `unir(3,11)`, `unir(3,12)`, `unir(3,13)`, `unir(14,15)`, `unir(16,17)`, `unir(14,16)`, `unir(1,3)`, `unir(1,14)`, cuando se utilizan las siguientes operaciones de unión:

a) Arbitraria.
b) Por altura.
c) Por tamaño.

23.2. Para cada uno de los árboles del ejercicio anterior, realice una búsqueda con compresión de caminos del nodo más profundo.
23.3. Calcule el árbol de recubrimiento mínimo del grafo de la Figura 23.19.
23.4. Muestre el funcionamiento del algoritmo ACP con los datos de la Figura 23.3.

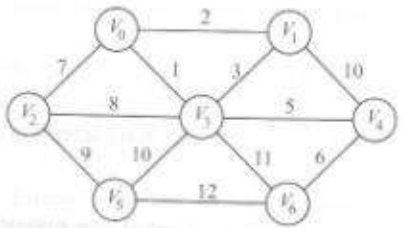


Figura 23.19 Un grafo *G* para el Ejercicio 23.3.

Problemas teóricos

23.5. Demuestre que el algoritmo de Kruskal es correcto. ¿Ha asumido que los costes de las aristas son no negativos?
23.6. Muestre que si la unión se realiza por altura, la profundidad de cualquier árbol es logarítmica.
23.7. Muestre que si todas las uniones preceden a las búsquedas, el algoritmo de la estructura de partición con compresión de caminos es lineal, aun si las uniones se realizan arbitrariamente. Observe que el algoritmo no cambia; sólo cambia su rendimiento.
23.8. Suponga que quiere añadir una nueva operación, `eliminar(x)`, que elimina `x` de su conjunto actual y lo coloca aislado. Muestre cómo modificar el algoritmo unir/buscar, de forma que el tiempo de ejecución de una secuencia de M operaciones siga siendo $O(M\alpha(M,N))$.
23.9. Demuestre que si las uniones se hacen por tamaño y se realiza la compresión de caminos, el tiempo de ejecución en el caso peor es aún $O(M \log^* N)$.
23.10. Suponga que implementa una compresión de caminos parcial en `buscar(i)` cambiando cada padre de los nodos en el camino desde `i` a la raíz por su abuelo (donde ello tenga sentido). Esto se denomina *división del camino por la mitad*. Demuestre que si se hace esto y se utiliza cualquier unión heurística, el tiempo de ejecución en el caso peor sigue siendo $O(M \log^* N)$.

Problemas prácticos

23.11. Implemente de forma no recursiva la operación `buscar`. ¿Hay una diferencia notable en el tiempo de ejecución?
23.12. Suponga que quiere añadir una operación extra, `desunir`, que deshace la última unión aún no deshecha. Una forma de hacer esto es utilizar la unión por rango, pero con un `buscar` sin compresión de caminos, y utilizar una pila para almacenar el estado antiguo anterior a la unión. Una `desunión` puede implementarse desapilando de la pila el estado antiguo.

a) ¿Por qué no podemos utilizar la compresión de caminos?
b) Implemente el algoritmo unir/buscar/desunir.

Prácticas de programación

23.13. Añada comprobación de errores a la implementación de la estructura de partición de la Figura 23.16.
23.14. Escriba un programa que determine los efectos de la compresión de caminos y de diversas estrategias de unión. Su programa debería procesar una larga secuencia de operaciones de equivalencia utilizando todas las estrategias estudiadas (incluyendo la división del camino por la mitad del Ejercicio 23.10).
23.15. Implemente el algoritmo de Kruskal.
23.16. Una alternativa al algoritmo del árbol de recubrimiento mínimo se debe a Prim [12]. Trabaja expandiendo un solo árbol en sucesivos pasos. Empie-

za eligiendo cualquier nodo como la raíz. Al comienzo de un paso, algunos nodos son parte del árbol y el resto no lo son. En cada paso, añadimos la arista de coste mínimo que conecta un nodo del árbol con uno de fuera. Una implementación del algoritmo de Prim es esencialmente idéntica al algoritmo de caminos mínimos de Dijkstra visto en la Sección 14.3, con la siguiente regla de actualización: $d_w = \min(d_w, c_{v,w})$ (en vez de $d_w = \min(d_w, d_v + c_{v,w})$). Además, ya que el grafo no es dirigido, cada arista aparece en dos listas de adyacencia. Implemente el algoritmo de Prim y compare su rendimiento con el de Kruskal.

23.17. Escriba un programa para resolver el problema ACP para árboles binarios. Compruebe su eficiencia construyendo un árbol binario aleatorio con 10.000 elementos y realizando 10.000 preguntas sobre el antecesor.

Bibliografía

La representación de los conjuntos de una partición mediante árboles fue propuesta en [8]. [1] atribuye la compresión de caminos a McIlroy y Morris y contiene varias aplicaciones de la estructura de partición. El algoritmo de Kruskal aparece en [11], mientras que la alternativa estudiada en el Ejercicio 23.16 es de [12]. El algoritmo ACP se estudia en [2]. Se pueden encontrar otras aplicaciones en [15].

La cota $O(M \log^* N)$ para el problema unir/buscar es de [9]. Tarjan [13] obtuvo la cota $O(M\alpha(M, N))$ y demostró que la misma es ajustada. Es más, la cota es intrínseca al problema y no puede ser mejorada por un algoritmo alternativo [16]. Otras estrategias para la compresión de caminos y la unión consiguen las mismas cotas; véase [16] para más detalles. Si la secuencia de uniones se conoce de antemano, es posible resolver el problema en tiempo lineal [7]. Este resultado puede utilizarse para probar que el problema ACP puede resolverse en tiempo lineal.

Diversos resultados sobre la complejidad en promedio del problema unir/buscar aparecen en [6], [10], [17] y [4]. Otros resultados que acotan el tiempo de ejecución de cada operación por separado (en oposición a la consideración de secuencias enteras) aparecen en [5].

1. A. V. Aho, J. E. Hopcroft, y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

2. A. V. Aho, J. E. Hopcroft, y J. D. Ullman, «On Finding Lowest Common Ancestors in Trees», *SIAM Journal on Computing* **5** (1976), 115-132.

3. L. Banachowski, «A Complement to Tarjan's Result about the Lower Bound on the Set Union Problem», *Information Processing Letters* **11** (1980), 59-65.

4. B. Bollobas y I. Simon, «Probabilistic Analysis of Disjoint Set Union Algorithms», *SIAM Journal on Computing* **22** (1993), 1053-1086.

5. N. Blum, «On the Single-operation Worst-case Time Complexity of the Disjoint Set Union Problem», *SIAM Journal on Computing* **15** (1986), 1021-1024.

6. J. Doyle y R. L. Rivest, «Linear Expected Time of a Simple Union Find Algorithm», *Information Processing Letters* **5** (1976), 146-148.

7. H. N. Gabow y R. E. Tarjan, «A Linear-time Algorithm for a Special Case of Disjoint Set Union», *Journal of Computer and System Sciences* **30** (1985), 209-221.

8. B. A. Galler y M. J. Fischer, «An Improved Equivalence Algorithm», *Communications of the ACM* **7** (1964), 301-303.

9. J. E. Hopcroft y J. D. Ullman, «Set Merging Algorithms», *SIAM Journal on Computing* **2** (1973), 294-303.

10. D. E. Knuth y A. Schonage, «The Expected Linearity of a Simple Equivalence Algorithm», *Theoretical Computer Science* **6** (1978), 281-315.

11. J. B. Kruskal, Jr., «On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem», *Proceedings of the American Mathematical Society* **7** (1956), 48-50.

12. R. C. Prim, «Shortest Connection Networks and Some Generalizations», *Bell System Technical Journal* **36** (1957), 1389-1401.

13. R. E. Tarjan, «Efficiency of a Good but Not Linear Set Union Algorithm», *Journal of the ACM* **22** (1975), 215-225.

14. R. E. Tarjan, «A Class of Algorithms Which Require Nonlinear Time to Maintain Disjoint Sets», *Journal of Computer and System Sciences* **18** (1979), 110-127.

15. R. E. Tarjan, «Applications of Path Compression on Balanced Trees», *Journal of the ACM* **26** (1979), 690-715.

16. R. E. Tarjan y J. van Leeuwen, «Worst Case Analysis of Set Union Algorithms», *Journal of the ACM* **31** (1984), 245-281.

17. A. C. Yao, «On the Average Behavior of Set Merging Algorithms», *Proceedings of the Eighth Annual ACM Symposium on the Theory of Computation* (1976), 192-195.

APÉNDICES

APÉNDICE

A Plataformas para Java

Este apéndice ilustra los pasos requeridos para compilar un programa escrito en Java usando tres entornos diferentes:

- *JDK* de Sun.
- *Symantec Café*.
- *Microsoft Visual J++*.

Supondremos que ya se ha realizado una instalación básica.

A.1 Estableciendo el entorno

Independientemente del sistema de desarrollo que utilicemos, necesitamos asegurarnos de que se les ha asignado un valor adecuado a las siguientes dos variables de entorno:

- `PATH`: especifica un conjunto de directorios donde se buscan los ejecutables (como *javac*, *java* y *javadoc*).
- `CLASSPATH`: especifica un conjunto de directorios donde se buscan los paquetes (como `java.lang` y `EstructurasDatos`).

Si se asigna a `PATH` un valor inadecuado, podría no encontrarse alguno de los ejecutables, como *javac*. Si se asigna una valor inadecuado a `CLASSPATH`, entonces fallarán las directivas `import` y otros usos de los paquetes.

Algunos de los productos descritos en esta sección asignan automáticamente valores a las variables de entorno al ser instalados. Aun así, para usar los programas del libro, necesitará modificar `CLASSPATH` para incluir los paquetes aquí descritos.

Las instrucciones para Unix y Windows95 son básicamente idénticas, excepto en lo que se refiere a su sintaxis concreta.

A.1.1 Instrucciones para Unix

Las siguientes instrucciones Unix suponen que estamos usando una C-shell.

1. Para comprobar si la variable `PATH` tiene un valor adecuado se usa el mandato

   ```
   which javac
   ```

   Si el sistema responde con un nombre (por ejemplo, `/bin/javac`), entonces tiene un valor adecuado, pero si responde con un mensaje diciendo

que no lo encuentra (proporcionando después la lista de lugares donde lo buscó), entonces debe ser actualizada.

2. Pregunte a su administrador de sistema dónde se encuentra situado el compilador de Java. Supongamos por ejemplo que se encuentra en el directorio `/home/jdk/bin`. Editamos su fichero `.cshrc` y añadimos una línea para actualizar la variable. En nuestro ejemplo, escribiríamos

```
set path= ( $path /home/jdk/bin )
```

3. A continuación actualizamos la variable `CLASSPATH`. Si ya tenemos una variable `CLASSPATH`, probablemente tendrá correctamente asignado el valor necesario para acceder a los paquetes del sistema. Se puede ver el valor actual escribiendo

```
echo $CLASSPATH
```

Incluso aunque tenga un valor adecuado, tendremos que actualizarla para usar los paquetes del libro. Si no posee una variable `CLASSPATH`, pregunte al administrador del sistema dónde se encuentran situados los ficheros de librerías del sistema. Entonces podríamos escribir por ejemplo

```
setenv CLASSPATH .:/jdk/lib/classes.zip
```

Observe que ordenamos buscar en dos sitios: el directorio actual (.) y en un fichero comprimido que contiene las clases de Java.

4. Añadimos al final de `CLASSPATH` el directorio que contiene los paquetes vistos en este libro. Por ejemplo

```
setenv CLASSPATH .:/jdk/lib/classes.zip:$HOME/libro
```

El nombre del directorio real variará en cada instalación. Observe que los cambios realizados en el fichero `.cshrc` no se hacen efectivos hasta la siguiente vez que entramos en el sistema. Sin embargo, podemos hacer que los cambios se hagan efectivos inmediatamente ejecutando el mandato

```
source .cshrc
```

A.1.2 Instrucciones para Windows95/NT

WindowsNT se comporta de la misma forma que Windows95, excepto por el hecho de que el proceso se puede simplificar editando las variables de entorno en el Panel de Control. (Seleccione *Panel de Control* y luego *Sistema*.) Las instrucciones para Windows95 se muestran a continuación.

1. Comprobamos que la variable `PATH` tiene un valor correcto. El mandato

```
javac
```

intenta invocar al compilador de Java. Si el sistema responde con un mensaje de error que detalla la sintaxis correcta del mandato *javac* (por ejemplo, especificando una lista de opciones), entonces la variable tiene un valor correcto. Si el sistema responde con un mensaje diciendo que no encuentra *javac*, se debe actualizar la variable `PATH`.

2. Para actualizarla, necesitamos saber dónde se encuentra situado el compilador de Java (puede usar las utilidades de búsqueda para saberlo). Supongamos, por ejemplo, que se encuentra en el directorio `C:\jdk\bin`. Entonces, editamos el fichero `autoexec.bat` (usando el *Bloc de Notas*) y añadimos una línea para actualizar la variable, como la siguiente

```
SET PATH = %PATH%;C:\jdk\bin
```

3. A continuación actualizamos la variable `CLASSPATH`. Si ya disponemos de una, probablemente tendrá el valor adecuado para encontrar los paquetes del sistema. Podemos ver los valores de todas las variables de entorno ejecutando el mandato `set` sin parámetros.

Incluso aunque la variable `CLASSPATH` tenga el valor correcto, tendremos que actualizarla para poder usar los paquetes de este libro. Si no disponemos de una variable `CLASSPATH`, necesitamos saber dónde están situados los ficheros de las clases de las librerías. Una asignación típica sería

```
set CLASSPATH = .:C:\jdk\lib\classes.zip
```

Observe que con ello mandamos buscar en dos lugares: el directorio actual (.) y un fichero comprimido que contiene las clases de Java.

4. Añadimos al final de `CLASSPATH` el directorio que contenga los paquetes de este libro. Por ejemplo, añadiendo una segunda línea

```
set CLASSPATH = %CLASSPATH%;C:\codigolibro
```

El directorio real variará en cada instalación. Observe que los cambios realizados en el fichero `autoexec.bat` no se hacen efectivos hasta la siguiente inicialización del sistema (o en el caso de WindowsNT hasta la siguiente vez que se entra en el sistema).

A.2 JDK de Sun

JDK es un sistema sin florituras. Hemos de introducir nuestro código en ficheros Java usando un editor estándar del sistema. Algunos editores de Unix son *vi*, *pico* y *emacs*; y de Windows, *Bloc de Notas* y *WordPad*.

Para compilar un fichero fuente de Java, usamos el mandato *javac*. Para ejecutar la función `main` de la clase, usamos el mandato *java*. Por ejemplo, supongamos que `main` está en la clase `Ejemplo`. Entonces los mandatos son

```
javac Ejemplo.java
java Ejemplo
```

Para cada clase definida en cada fichero fuente compilado por *javac*, el compilador almacena el código-j resultante en el fichero de clase correspondiente (con el sufijo `.class`). Para cada clase referenciada en los ficheros fuente, el compilador busca en la variable `CLASSPATH` tanto el fichero fuente como el fichero compilado, y recompila el fichero fuente (regenerando por tanto el fichero `.class`) si éste se ha modificado. En otras palabras, *javac* calcula cuál es el menor conjunto de ficheros que hace falta recompilar y lo hace de forma automática.

El mandato *javac* tiene varias opciones. En la documentación para JDK se listan con detalle. En la Figura A.1 se muestran algunas de ellas.

Opción	Resultado
`-O`	Optimiza el código compilado desplegando los métodos estáticos, finales y privados.
`-verbose`	Hace que el compilador y el enlazador impriman mensajes sobre los ficheros que se están compilando y los ficheros ya compilados (o ficheros de clase) que se están cargando.
`-depend`	Hace que el compilador recompile ficheros de clases referenciados desde otros ficheros de clase. Normalmente sólo recompila ficheros de clase obsoletos a los que se hace referencia desde el código fuente.

Figura A.1 Opciones de *javac*.

El mandato *java* también tiene opciones. Se proporciona una lista completa de ellas en la documentación para JDK.

A.3 Entornos de desarrollo visual

Symantec Café se distinguió por ser el primer sistema del mercado con un entorno de desarrollo moderno. *Microsoft Visual J++* se distingue por ser un producto Microsoft. Su apariencia y fundamentos son idénticos al popular Visual C++ de Microsoft.

Ambos sistemas incluyen, entre otras cosas,

- un editor visual que ilumina en diferentes colores las palabras clave, los comentarios y los tokens;
- botones para compilar y ejecutar programas;
- ventanas que proporcionan simultáneamente varias vistas del proyecto, como la salida del compilador, el código fuente y la lista de ficheros del proyecto;
- herramientas de depuración; y
- ayuda durante su uso.

La integración de estas vistas en un único producto implica, por ejemplo, que si se pulsa dos veces sobre un mensaje de error del compilador, aparece iluminada la línea de código donde se ha detectado el error. Una vez haya disfrutado de un sistema visual, le resultará difícil volver a un sistema basado en texto. Una característica importante es el editor de recursos, que permite al programador diseñar un entorno gráfico usando un sistema CAD. El editor de recursos generará código Java que describe el entorno. Esto es útil para el diseño de Interfaces Gráficas de Usuario.

Los sistemas visuales tienen algunas desventajas. En primer lugar, a diferencia del JDK de Sun, no son gratuitos. En segundo lugar, se debe hacer el esfuerzo de crear un proyecto para cada programa, lo cual puede resultar tedioso para programas pequeños que solamente utilizan un fichero fuente. En tercer lugar, la versión más actualizada de Java siempre aparecerá disponible antes para el sistema JDK de Sun. Las actualizaciones aparecerán en los sistemas comerciales poco después, y no serán gratuitas. Por ejemplo, cuando se escribió la versión original de este libro, el único compilador que soportaba Java 1.1 era el JDK 1.1[1].

se habla de los sistemas que estaban disponibles en el momento en que se escribió la versión original de este
ones serán probablemente similares, pero no completamente idénticas.

Este apéndice describe, para ambos entornos, cómo crear un proyecto para la clase `TestPila`. El código fuente de la versión original, `TestStack.java`, se encuentra en Internet en el directorio **Chapter06**[2]. Se supone que ya hemos cargado el código de los programas de este libro y que se ha asignado el valor adecuado a la variable `CLASSPATH`, como se ha descrito en la Sección A.1.2. Necesitaremos crear un directorio para el proyecto, y copiar la versión original, `TestStack.java`, en dicho directorio.

A.3.1 Symantec Café

1. Comenzamos por inicializar Café. Veremos brevemente un logotipo coloreado. A continuación, nos encontraremos con la peor característica de Café: la pantalla es prácticamente transparente, por lo que su entorno de sistema será aún visible. Si es la primera vez que utilizamos Café, nos encontraremos con algunos botones esparcidos por la pantalla. Se pueden mover a la parte superior para obtener algo similar a la Figura A.2. Nótese que en la figura solamente se muestra la parte superior de la pantalla.
2. Ahora crearemos un nuevo proyecto para `TestStack`, al que por simplicidad le llamaremos `TestStack`. Pulse el menú *Project* y seleccione *New*, como se muestra en la Figura A.3.
3. En este punto, inicializamos la herramienta *ProjectExpress*, que hace aparecer una secuencia de cuatro cuadros de diálogo. Es probable que sólo necesitemos los tres primeros.

Figura A.2

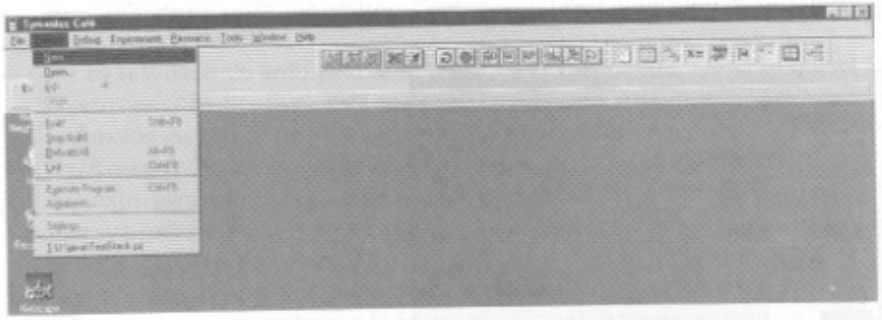


Figura A.3

[2] *N. del T.:* En el Capítulo 6 definimos una clase `TestPila` cuya versión original (en inglés) se llama `Test`
este apéndice reflejan la creación de un proyecto para la versión original de la clase, por lo que, de ahora en adel
cias se harán a la versión original, para mantener la coherencia con las figuras.

a) Seleccionamos en el primer cuadro un dispositivo de disco. Seleccionamos también el directorio en el que deseamos guardar el proyecto. Entonces escribimos el nombre del proyecto. Asegúrese de que termina con el sufijo `.prj`. Pulse *Next*. *Véase* la Figura A.4.

b) En el segundo cuadro, seleccionamos *Application* en *Target Type*. Tras ello escribimos `TestStack` como nombre de la clase principal. Pulse *Next*. *Véase* la Figura A.5.

c) En el tercer cuadro de diálogo, añadimos los ficheros al proyecto. El fichero de la clase principal debe estar ya en el mismo directorio que el fichero del proyecto. Si no es así, no se preocupe, podemos añadir ficheros al proyecto más adelante. Alternativamente, podemos copiar el fichero en este momento (desde fuera de Café) y teclear su nombre. Esto se muestra en la Figura A.6. Si `TestStack.java` estuviera ya presente en el directorio, habría aparecido en la lista de posibles ficheros a añadir. Si hay varios ficheros a añadir, seguimos pulsando *Add* o pulsando dos veces sobre el nombre del fichero en la lista de ficheros. Cuando hayamos añadido los ficheros, pulsamos *Finish*. No necesitamos ir al paso 4 ofrecido por *ProjectExpress*.

4. Una vez que se ha creado el proyecto, establecemos algunas vistas. En el menú *Window*, seleccionamos *Goto View*. Tras ello añadimos las vistas *Output* y *Project*. *Véase* la Figura A.7.

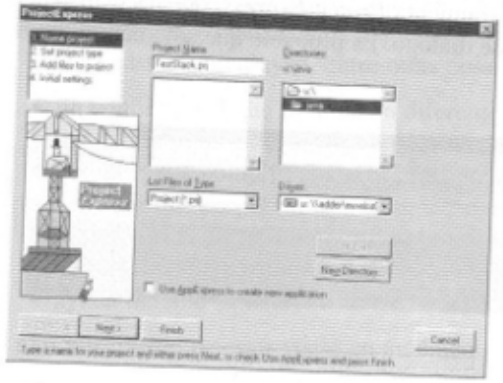


Figura A.4

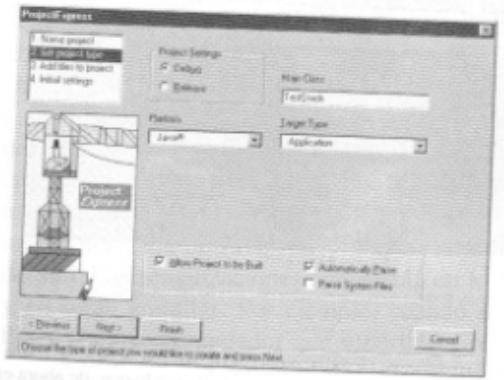


Figura A.5

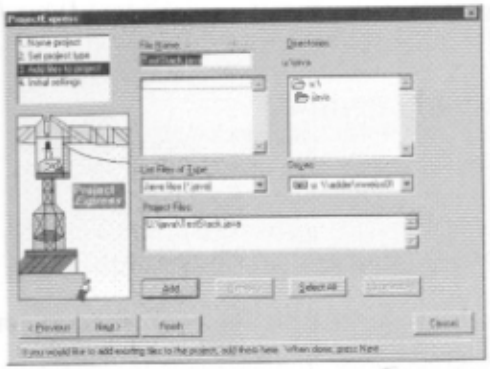


Figura A.6

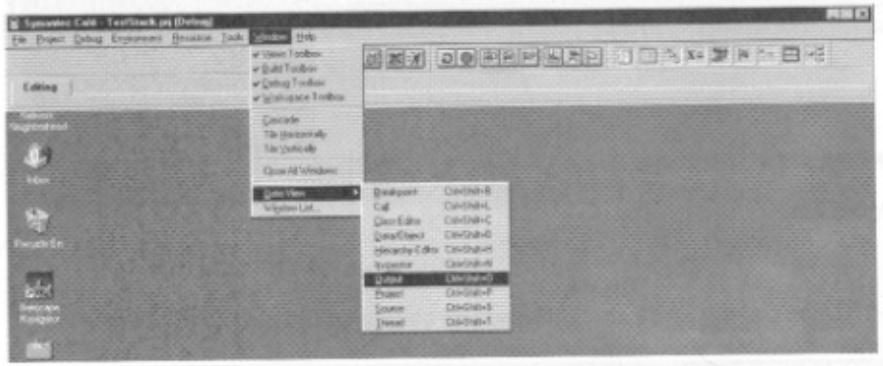


Figura A.7

Estas vistas pueden aparecer con un tamaño o en lugares que no nos gusten, pero podemos cambiar el tamaño de las ventanas, y moverlas a otros sitios. Desde la vista del proyecto, podemos pulsar dos veces sobre un fichero para obtener una ventana *Source*. Podemos cambiar el tamaño de dicha ventana y moverla de sitio.

El resultado de todo esto se muestra en la Figura A.8. Observe que el fichero `TestStack.java` está marcado en amarillo para indicar que contiene la función `main` que se va a ejecutar (esto se especificó en el paso 2 de *ProjectExpress*). Podemos pulsar con el botón derecho del ratón sobre otro objeto de texto para especificar otra clase como principal. La vista del proyecto muestra todas las clases necesarias que no forman parte del sistema, lo cual se deduce automáticamente.

5. Compilamos el proyecto pulsando el botón *Build*. (En la Figura A.9, el botón *Build* se encuentra arriba y a la izquierda de la etiqueta *Build*.)

Solamente se compilan aquellos ficheros obsoletos. (Alternativamente podemos solicitar una compilación completa.) Cualquier error aparecerá en la ventana de salida. Si no hay errores, se presentará un mensaje al respecto. En este momento, podemos ejecutar el programa pulsando el botón *Run*. (Este botón muestra una persona corriendo, así que no podemos equivocarnos.)

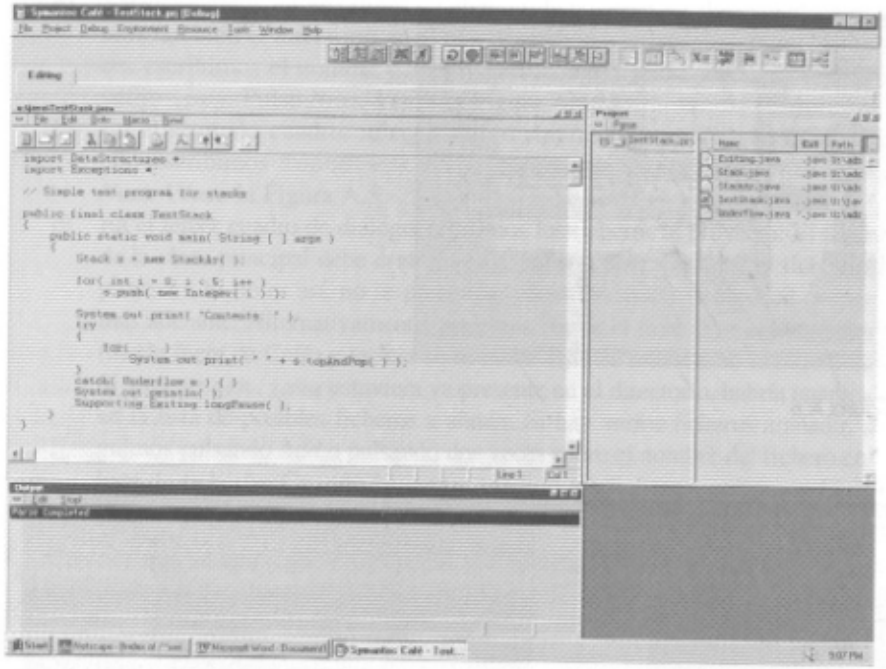


Figura A.8

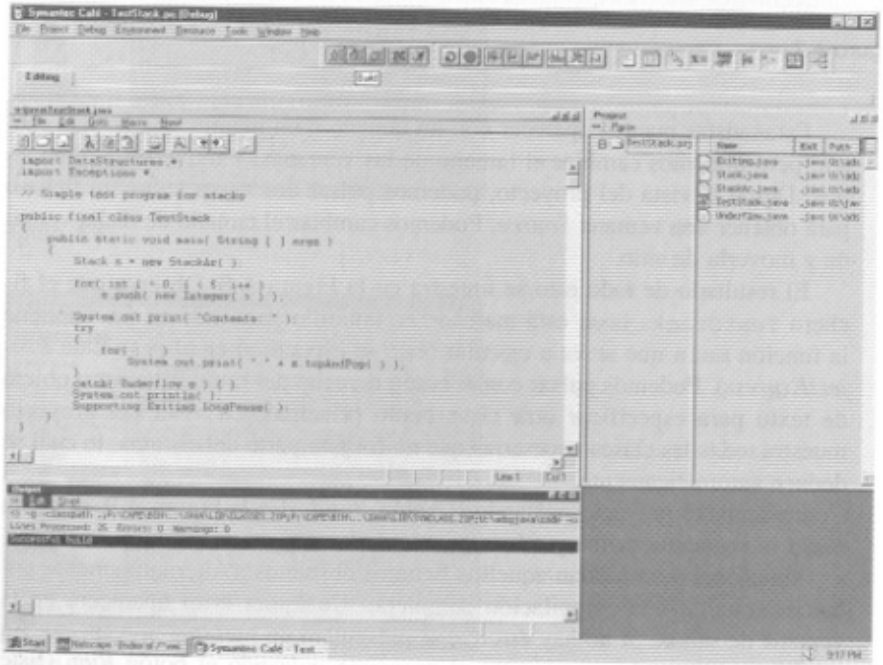


Figura A.9

Figura A.10

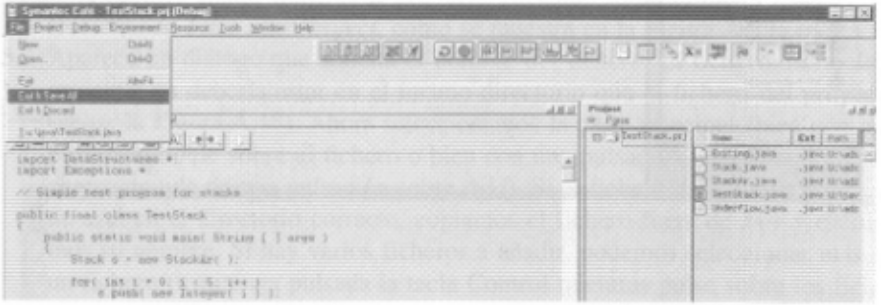


Figura A.11

Cuando se ejecuta una aplicación de consola, aparece una ventana MS-DOS. Es interactiva, con lo que se aceptan mandatos de entrada. Desafortunadamente, la ventana desaparece cuando el programa termina, lo que hace difícil ver la salida. Para evitar esto, podemos hacer una de las siguientes cosas:

I. Añadir una llamada a `Soporte.Saliendo.pausaLarga` o a `Soporte.Salir.apuntoDeSalir`, para evitar con ello que el programa termine.
II. Ejecutar el mandato *java* desde una ventana MS-DOS separada de Café.

Observemos que la opción I es inaceptable si el programa aborta por culpa de una excepción sin tratar. Sin embargo, en este caso, podemos ejecutar el depurador (mirar el menú *Debug*) para ver que es lo que ha ido mal. La opción I es también inaceptable si se está leyendo hasta el final de la entrada. La Figura A.10 muestra la ventana MS-DOS mencionada en la opción II.

6. Cuando hayamos terminado, salimos seleccionando el menú *Fichero*. Las opciones incluyen (*véase* la Figura A.11) salvar todo el trabajo (normalmente aconsejable) o salir sin salvar (aconsejable más a menudo de lo que nos gustaría).

A.3.2 Microsoft Visual J++

1. Comenzamos inicializando J++. Veremos brevemente un logotipo coloreado para Microsoft Studio, tras el cual aparecerá una pantalla. Si es la primera vez que usamos J++, nos encontraremos con varios objetos esparcidos, como una

barra de consejos y una ayuda (que contiene *Microsoft Visual J++ Books Online*). La barra de consejos se puede hacer desaparecer y la ayuda se puede colocar en la parte superior para obtener algo similar a la Figura A.12.

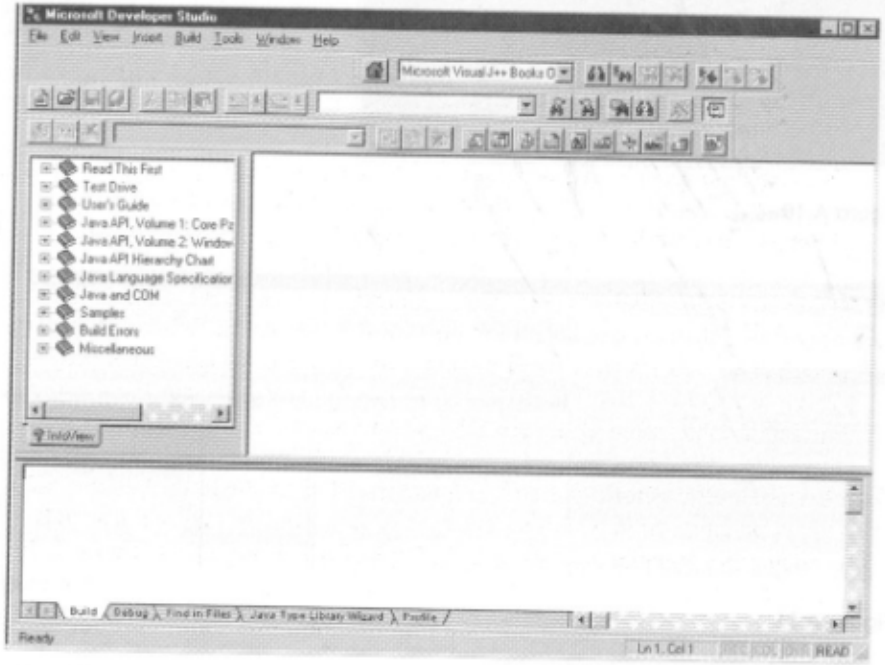


Figura A.12

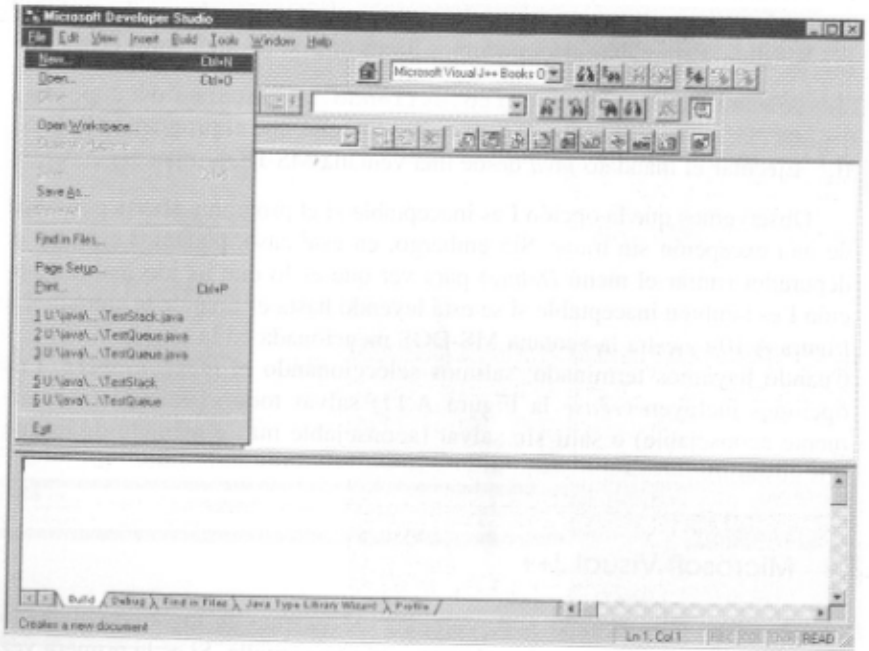


Figura A.13

2. Creamos un nuevo proyecto para `TestStack`. Pulsamos sobre el menú *File* y seleccionamos *New*, como se muestra en la Figura A.13. Esto genera un cuadro de diálogo que nos pide que especifiquemos lo que deseamos hacer. Pulsamos dos veces sobre *Project Workspace*, como se muestra en la Figura A.14.
3. A continuación aparece un cuadro de diálogo que nos solicita datos sobre el proyecto. Seleccionamos *Java Workspace* en *Type*; en *Location* indicamos el directorio donde se va a guardar el proyecto, y escribimos como nombre de proyecto `TestStack`. Todo ello se muestra en la Figura A.15. Después pulsamos *Create*.
4. En este momento, el cuadro de ayuda se sustituye por una vista de la clase, como se muestra en la Figura A.16. Si abrimos la carpeta de *TestStack classes*, veremos el fichero que constituye el proyecto. De momento el proyecto está vacío.

   Podemos añadir ficheros al proyecto pulsando sobre el menú *Insert* y seleccionando *Files into Project*, como se muestra en la Figura A.17.
5. Aparece un diálogo que solicita el nombre del fichero. El fichero de la clase principal ya debería estar en el mismo directorio que el fichero del proyecto (*véase* la Figura A.18). Ahora seleccionamos la clase principal (bien con una pulsación doble sobre el fichero o bien con una pulsación simple sobre el fichero seguida de una pulsación sobre *Add*). Si el fichero de la clase principal no está en el directorio correcto, copiamos el fichero fuera de J++ y escribimos su nombre. Si hay varios ficheros a añadir, podemos seleccionar más de un fichero (mantenga pulsada la tecla Control mientras pulse sobre los ficheros de la lista).
6. Pulse dos veces sobre *TestStack files* en la vista de la clase para ver que se ha añadido `TestStack.java` al proyecto. Pulse dos veces sobre *TestStack.java* para abrir el fichero en la ventana de código fuente.

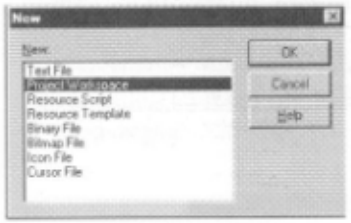


Figura A.14

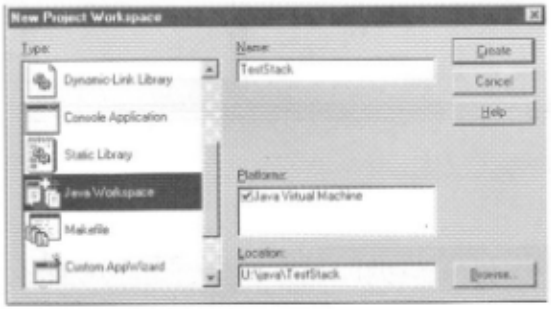


Figura A.15

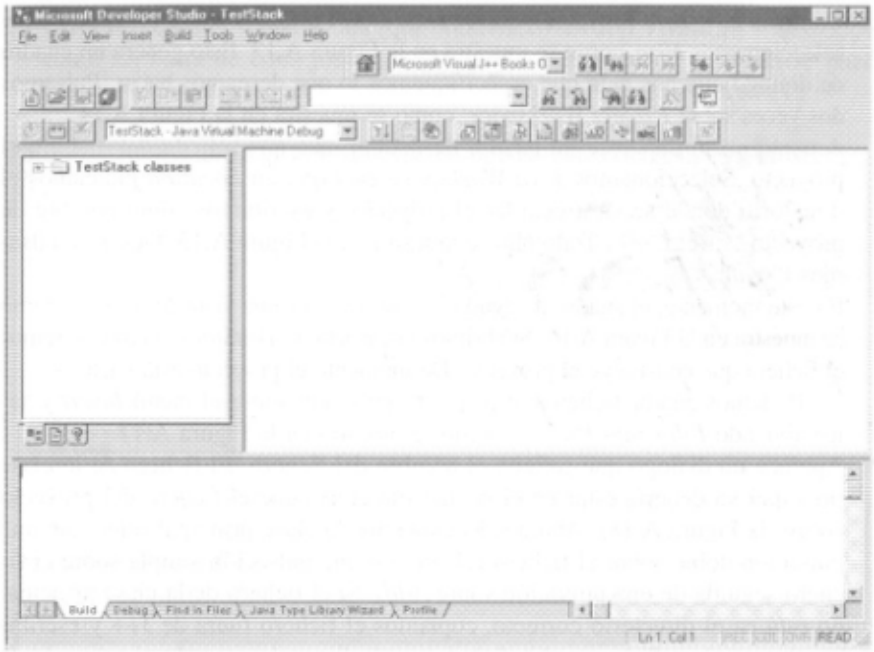


Figura A.16

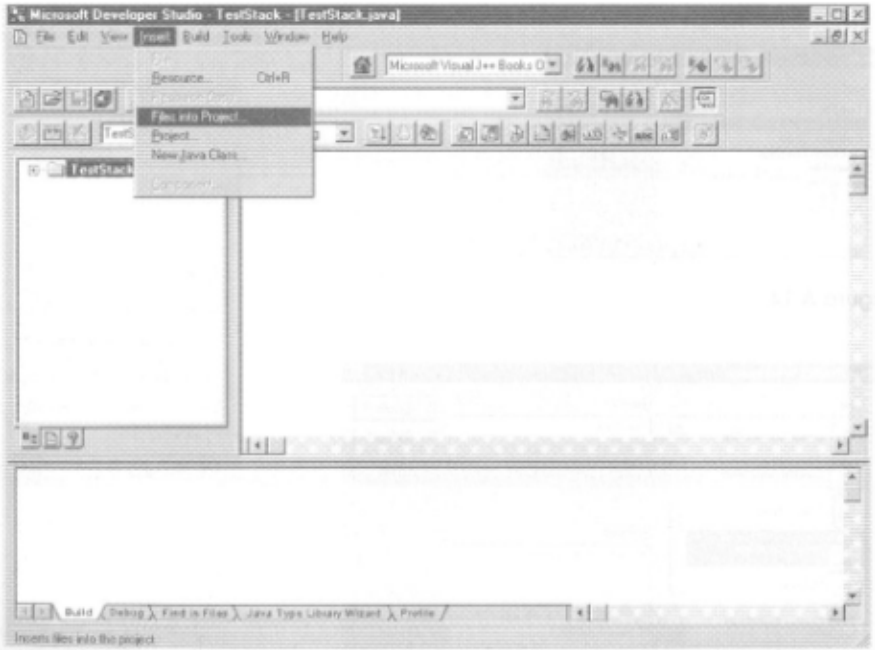


Figura A.17

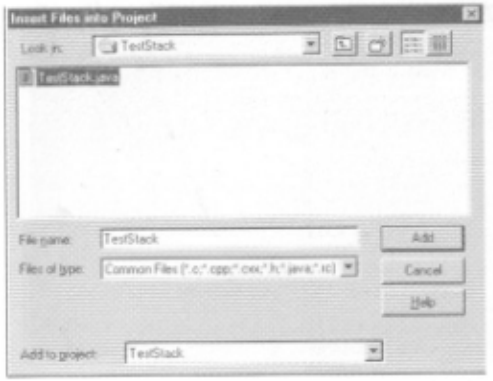


Figura A.18

7. A continuación, podemos cambiar algunas características del proyecto. En el menú *Build*, seleccionamos *Settings*, como se muestra en la Figura A.19.
8. Aparece entonces el cuadro de diálogo *Project Settings*. En él seleccionamos *Debug*, escribimos el nombre de la clase a depurar/ejecutar en *Class for debugging/executing*, y seleccionamos la opción *Stand-alone interpreter*, como se muestra en la Figura A.20.
9. Ahora está todo listo para compilar el proyecto y ejecutarlo. Los botones para hacer esto son los siguientes: el botón para compilar es el botón del medio situado directamente sobre la vista de la clase y el botón para ejecutar es el primero sobre la vista del código fuente. Estas posiciones dependen de la confi-

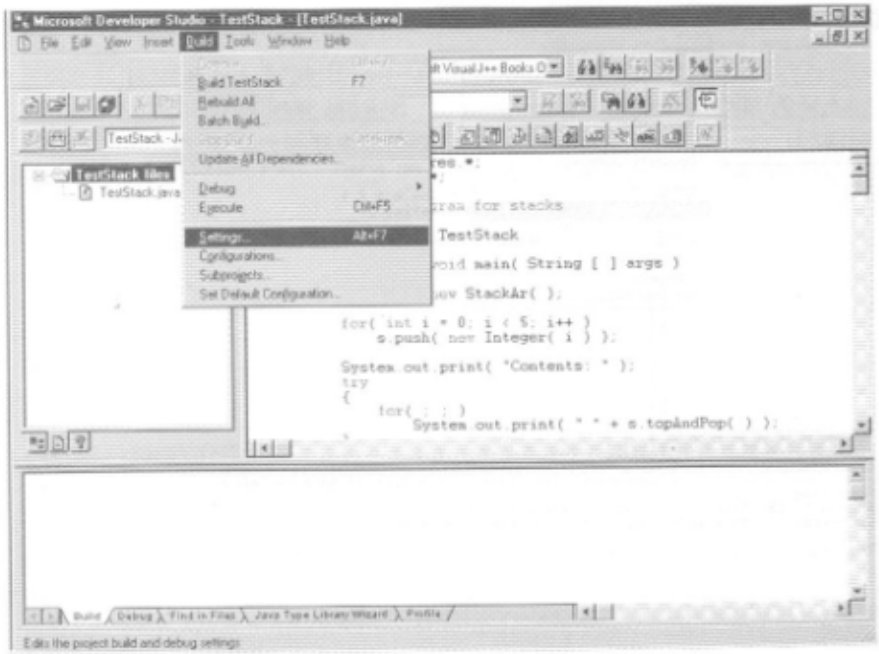


Figura A.19

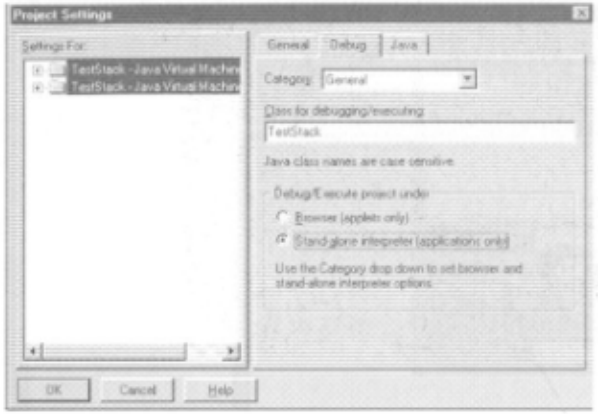


Figura A.20

guración local. Los botones podrían no ser visibles; en tal caso, se debe usar el menú *Build*. La Figura A.21 muestra el menú *Build*. Tiene una opción para compilar y otra para ejecutar. La ejecución compila automáticamente si el fichero de clase no existe o está obsoleto. La opción de compilación solamente compila los ficheros obsoletos. (Se puede solicitar también una compilación completa). Cualquier error aparecerá en la ventana de salida. Si no hay errores, aparecerá un mensaje al respecto.

Cuando ejecutamos una aplicación de consola, aparece una ventana MS-DOS, como en la Figura A.22. La ventana es interactiva, por lo que se

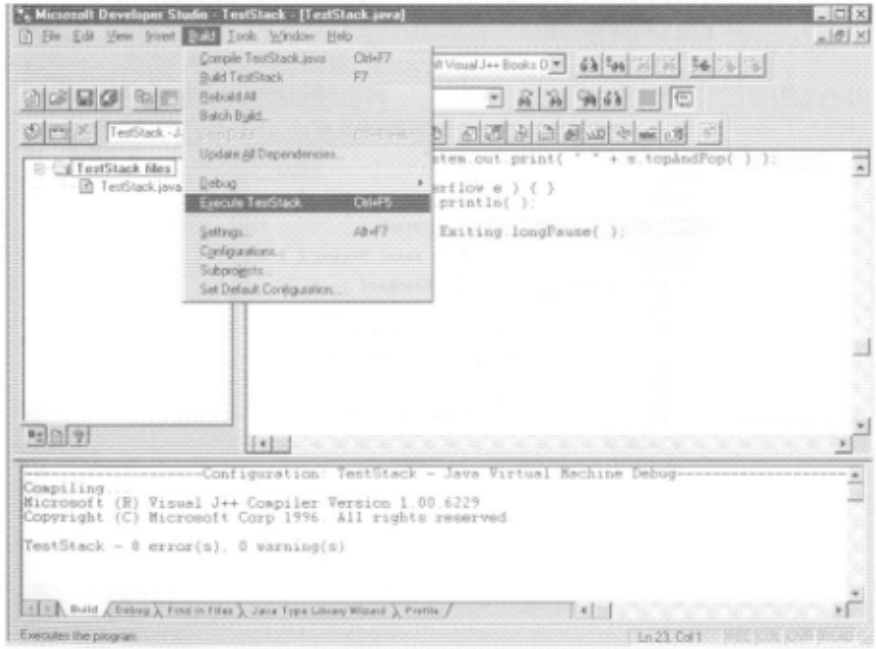


Figura A.21

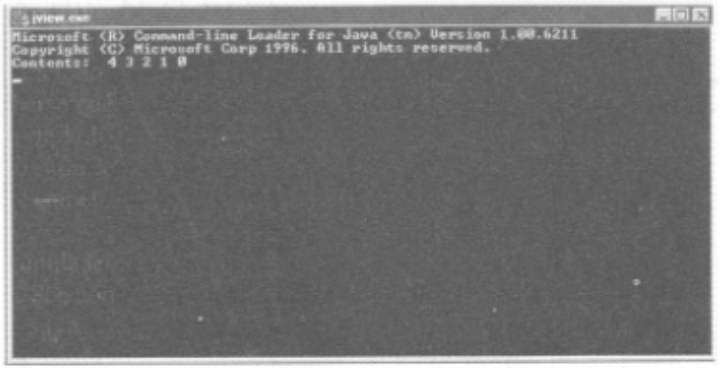


Figura A.22

pueden usar mandatos de entrada. En J++ 1.0, la ventana desaparece cuando el programa termina. Esto hace difícil ver la salida. Confiamos en que este problema desaparezca cuando esté disponible la versión de J++ acorde con Java 1.1. Hasta entonces, deberíamos utilizar las ideas del punto 5 de la Sección A.3.1 para evitar que el programa termine, impidiendo así que la ventana desaparezca.

APÉNDICE

B Operadores

La Figura B.1 muestra la precedencia y asociatividad de los operadores Java más comunes que hemos estudiado. En este libro no se han utilizado los operadores bit a bit.

Categoría	Ejemplos	Asociatividad
Operaciones sobre referencias	`. []`	A la izquierda
Unarios	`++ -- ! -` (tipo)	A la derecha
Multiplicativos	`* / %`	A la izquierda
Aditivos	`+ -`	A la izquierda
De desplazamiento (bit a bit)	`<< >> >>>`	A la izquierda
Relacionales	`< <= > >= instanceof`	A la izquierda
De igualdad	`== !=`	A la izquierda
Y booleano (o bit a bit)	`&`	A la izquierda
O exclusivo booleano (o bit a bit)	`^`	A la izquierda
O booleano (o bit a bit)	`|`	A la izquierda
Y lógico	`&&`	A la izquierda
O lógico	`||`	A la izquierda
Condicional	`? :`	A la derecha
Asignación	`= *= /= %= += -=`	A la derecha

Figura B.1 Operadores Java, listados de mayor a menor precedencia.

APÉNDICE

C Algunas rutinas de librerías

Este apéndice enumera algunas de las clases de la librería de Java utilizadas en este libro. Se da una breve descripción de cada clase, así como algunas de las componentes más utilizadas. Una descripción completa de estas clases se puede encontrar en la *JDK API Documentation*.

C.1 Clases del paquete `java.lang`

Éste es el paquete estándar que importan automáticamente todos los programas Java.

C.1.1 `Character`

La clase `Character` oculta en un objeto un valor del tipo primitivo `char`. Un objeto de tipo `Character` tiene un único campo cuyo tipo es `char`. Esta clase también proporciona varios métodos para determinar el tipo de un carácter y la conversión de caracteres entre mayúsculas y minúsculas, y viceversa. Es una clase final.

Métodos importantes

`Character( char value )`

Construye un objeto `Character` y lo inicializa de tal forma que represente el valor primitivo dado como argumento.

`char charValue( )`

Devuelve el valor del objeto `Character`.

`boolean equals( Object obj )`

Compara el objeto `Character` con el objeto especificado. El resultado es `true`, si y sólo si el argumento es distinto de `null` y es un objeto `Character` que representa el mismo valor `char` que el objeto actual.

`String toString( )`

Devuelve un objeto `String` que representa el valor del objeto `Character`. El resultado es un valor de tipo `String` cuya longitud es 1. La única componente del `String` es el valor primitivo `char` que representa el objeto.

static boolean isDigit(char ch)
Devuelve `true` si y sólo si el carácter especificado es un dígito.

static boolean isDefined(char ch)
Devuelve `true` si y sólo si el carácter especificado tiene sentido en Unicode.

static boolean isLetter(char ch)
Devuelve `true` si y sólo si el carácter especificado es una letra.

static boolean isLetterOrDigit(char ch)
Devuelve `true` si y sólo si el carácter especificado es una letra o un dígito.

static boolean isLowerCase(char ch)
Devuelve `true` si y sólo si el carácter especificado es una letra en minúscula.

static boolean isUpperCase(char ch)
Devuelve `true` si y sólo si el carácter especificado es una letra en mayúscula.

static boolean isWhiteSpace(char ch)
Devuelve `true` si y sólo si el carácter especificado es un carácter de espacio en blanco en Java.

static char toLowerCase(char ch)
Devuelve, si existe, el carácter en minúsculas equivalente al carácter dado; si no, devuelve el mismo carácter.

static char toUpperCase(char ch)
Devuelve, si existe, el carácter en mayúsculas equivalente al carácter dado; si no, devuelve el mismo carácter.

static boolean isJavaIdentifierStart(char ch)
Devuelve `true` si y sólo si el carácter especificado puede ser el primer carácter de un identificador Java. (Incorporado en Java 1.1.)

static boolean isJavaIdentifierPart(char ch)
Devuelve `true` si y sólo si el carácter especificado puede ser parte de un identificador Java. (Incorporado en Java 1.1.)

C.1.2 Integer

La clase `Integer` oculta un valor del tipo primitivo `int` en un objeto. Un objeto de tipo `Integer` contiene un único campo cuyo tipo es `int`. Esta clase tiene varios métodos para convertir un valor de tipo `int` en un valor de tipo `String` y viceversa, así como constantes y métodos útiles cuando se trabaja con enteros. Es una clase final.

Constantes importantes

static final const int MAX_VALUE
El mayor valor de tipo `int`.

static final const int MIN_VALUE
El menor valor de tipo `int`.

Métodos importantes

Integer(int val)
Construye un nuevo objeto `Integer` que representa el valor de tipo primitivo `int` del argumento.

int intValue( )
Devuelve el valor representado por el objeto `Integer` al que se aplica.

boolean equals(Object obj)
Devuelve `true`, si y sólo si el argumento es distinto de `null` y es un objeto `Integer` que representa el mismo valor `int` que el objeto.

static int parseInt(String str)
Analiza el valor de tipo `String` dado como argumento devolviendo un entero decimal con signo. Puede lanzar una excepción `NumberFormatException`.

static int parseInt(String str, int radix)
Analiza el valor de tipo `String` dado como argumento devolviendo un entero con signo en la base especificada como segundo argumento. Puede lanzar una excepción `NumberFormatException`.

String toString( )
Devuelve un valor de tipo `String` que representa el valor de tipo `Integer` al que se aplica.

static String toString(int theInt)
Devuelve un valor de tipo `String` que representa el entero especificado.

static String toString(int theInt, int radix)
Crea un objeto `String` que representa el primer argumento en la base especificada en el segundo argumento.

C.1.3 Object

La clase `Object` es la raíz de la jerarquía de clases, es decir, todas las clases tienen a `Object` como superclase. Todos los objetos, incluyendo los vectores, implementan los métodos de esta clase.

Métodos importantes

Object( )
Construye un objeto; rara vez se invoca directamente.

boolean equals (Object obj)
Compara dos objetos de tipo `Object`. Este método implementa el test más discriminante posible. Es decir, para cualquier par de referencias `x` e `y`, `x.equals(y)` devuelve `true` si y sólo si `x` e `y` son referencias al mismo objeto (`x == y` vale `true`).

String toString( )

Devuelve una cadena que representa el objeto. El resultado debería ser una representación concisa pero informativa, que sea fácil de leer.

C.1.4 String

La clase `String` representa cadenas de caracteres. Todas las constantes literales de cadenas en programas Java, como `"abc"`, se implementan como instancias de esta clase. Los valores de tipo `String` son constantes, es decir, sus valores no pueden cambiar después de su creación. En la Sección C.1.5 veremos cadenas que pueden cambiar. La clase `String` incluye multitud de métodos, algunos de los cuales describiremos a continuación.

El lenguaje Java proporciona un soporte especial para la operación de concatenación de cadenas (+) y para conversión de otros objetos en cadenas. Las conversiones a cadenas se realizan a través del método `toString`, definido en la clase `Object`, y heredado por todas las clases Java.

Métodos importantes

String( )

Crea una cadena vacía.

String(String anotherStr)

Crea una nueva cadena que contiene la misma secuencia de caracteres que la cadena dada como argumento.

char charAt(int index)

Devuelve el carácter en la posición especificada. El índice varía desde 0 hasta `length( )-1`. Se lanza `StringIndexOutOfBoundsException` si la posición está fuera de rango.

boolean equals(Object obj)

Compara la cadena con el objeto especificado. El resultado es `true`, si y sólo si el argumento es distinto de `null` y es un objeto `String` que representa la misma secuencia de caracteres que el objeto.

int compareTo(String str)

Compara dos cadenas respecto al orden lexicográfico. La comparación se basa en el valor Unicode de cada carácter en las cadenas. Devuelve el valor 0 si las cadenas son iguales; un valor menor que 0 si la cadena que recibe el mensaje es menor lexicográficamente que la cadena argumento; y un valor mayor que 0, en otro caso.

int length( )

Devuelve la longitud de la cadena.

boolean startsWith(String prefix)

Devuelve `true`, si y sólo si la cadena comienza con el prefijo especificado como argumento.

boolean endsWith(String prefix)

Devuelve `true`, si y sólo si la cadena termina con el sufijo especificado como argumento.

String subsString(int beginIdx, int endIdx)

Devuelve una nueva cadena que es subcadena de aquello sobre lo que se aplica. La subcadena empieza en el valor especificado `beginIdx` y termina en la posición `endIdx-1`. Se lanza `StringIndexOutOfBoundsException` si `beginIdx` o `endIdx` están fuera de rango.

C.1.5 StringBuffer

Ya que las cadenas no se pueden modificar, pequeños cambios en una cadena pueden resultar costosos. Concretamente, el operador `+=` es muy ineficiente. Por ejemplo, `str += 'A'` se implementa como `str = str + 'A'`. Esto implica que se crea una nueva cadena cuyo valor es el resultado de `str + 'A'`, y `str` referencia a esta nueva cadena. Como resultado, el coste de la concatenación de un solo carácter no es constante (como parece lógico que sea). Por el contrario, es proporcional a la longitud de la cadena sobre la que se trabaja. Si hay concatenaciones repetidas a la misma cadena, el coste puede ser prohibitivo.

Un ejemplo es el método trivial de la Figura C.1 que genera una cadena con N caracteres A. Añadir el i-ésimo carácter requiere un coste proporcional a i, ya que, como hemos visto, se crea una nueva cadena de longitud i. El coste total del método viene dado por $1 + 2 + 3 + ... + N$, lo cual es cuadrático. Como caso extremo, para ejecutar este método con $N = 64.000$ en un Pentium 100 se necesitan unos 5 minutos.

Para que las operaciones sean eficientes, los caracteres se deberían manipular directamente, construyendo finalmente una sola cadena conteniendo el resultado deseado, tras completar las operaciones. Esto es lo que hace la clase `StringBuffer`.

Un valor del tipo `StringBuffer` puede construirse a partir de una cadena cualquiera o bien representar la cadena vacía. Sobre ellos se pueden aplicar diversas operaciones que manipulan el `StringBuffer`, como `append`, que añade por el final; `setChar`, que cambia un carácter concreto; e `insert`, que añade por el medio desplazando caracteres si ello es necesario. El compilador utiliza los `StringBuffer` para generar código eficiente para las concatenaciones triviales. Para operaciones no triviales, tales como el ejemplo de la Figura C.1, lo tiene que hacer uno mismo. La Figura C.2 muestra cómo se puede utilizar un `StringBuffer` para generar de forma eficiente un `String` de N caracteres A. Este método es lineal y requiere 1,5 segundos para $N = 64.000$.

```
1    // Método cuadrático que genera una cadena con n Aes
2  static String cadenaLargaMala( int n )
3  {
4      String resultado = "";
5      for( int i = 0; i < n; i++ )
6          resultado += 'A';
7      return resultado;
8  }
```

Figura C.1 Método ineficiente para generar un `String` de N caracteres A.

```
1    // Método lineal que genera una cadena con n Aes
2  static String cadenaLargaBuena( int n )
3  {
4      StringBuffer resultado = new StringBuffer( );
5
6      for( int i = 0; i < n; i++ )
7          resultado.append( 'A' );
8
9      return new String( resultado );
10 }
```

Figura C.2 Método eficiente para generar un `String` de *N* caracteres *A*.

`StringBuffer` utiliza duplicación de vectores, para así asegurar que su capacidad puede expandirse, si ello es necesario. El tamaño inicial es de 16 caracteres.

Métodos importantes

`StringBuffer( )`

Construye un valor `StringBuffer` sin caracteres y con una capacidad inicial de 16 caracteres.

`StringBuffer( String str )`

Construye un valor `StringBuffer` que representa la misma cadena de caracteres que la cadena dada como argumento. La capacidad inicial es de 16 caracteres más que la longitud del argumento.

`int length( )`

Devuelve la longitud (número de caracteres) del `StringBuffer`.

`char charAt( int index )`

Devuelve el carácter en la posición indicada. El primer carácter se encuentra en la posición 0, el siguiente en la posición 1, y así sucesivamente. Se lanza la excepción `StringIndexOutOfBoundsException` si la posición no es válida.

`void setCharAt( int index, char ch )`

Asigna `ch` al carácter en la posición especificada. Lanza la excepción `StringIndexOutOfBoundsException` si la posición no es válida.

`StringBuffer append( Object obj )`

Añade al buffer la representación como `String` del argumento `Object`. El argumento se convierte en una cadena, y los caracteres de esta cadena se añaden al buffer. Esto es lo que se devuelve, por lo que puede ser usado en una ristra de operaciones. Hay otras operaciones `append` disponibles para todos los tipos primitivos, así como para vectores de caracteres.

`StringBuffer insert( int offset, Object obj )`

Añade al buffer la representación como `String` del argumento `Object`. El segundo argumento se convierte en una cadena, y los caracteres de esta cadena se inserta en la posición indicada. Esto es lo que se devuelve, por lo que puede ser usado en una ristra de operaciones. Hay otras operaciones `insert` disponibles para los tipos primitivos, así como para vectores de caracteres. Se lanza la excepción `StringIndexOutOfBoundsException` si la posición no es válida.

`StringBuffer reverse( )`

La secuencia de caracteres que contiene el buffer se reemplaza por su inversa. Esto es lo que se devuelve, por lo que puede ser usado en una ristra de operaciones.

`String toString( )`

Convierte el contenido en un valor `String`. Se crea un nuevo objeto de tipo `String` y se inicializa con la secuencia de caracteres representada en ese momento por el buffer. Cambios posteriores del buffer no afectan al contenido de la cadena.

C.1.6 `System`

La clase `System` contiene varios atributos y métodos útiles. No puede ser instanciada. Entre las facilidades proporcionadas por la clase `System` se encuentran los canales de entrada y salida estándar, el canal de errores, un método `exit`, y muchos métodos de sistema que no se utilizan en este texto.

Constantes importantes

`static final InputStream in`

El canal de entrada estándar. El canal está ya abierto y dispuesto para ser leído. Típicamente, este canal corresponde a la entrada por teclado u otra fuente de entrada especificada por el entorno o el usuario.

`static final PrintStream out`

El canal de salida estándar. El canal está ya abierto y dispuesto para ser leído. Típicamente, este canal corresponde a la salida por pantalla u otro destino de salida especificado por el entorno o el usuario.

`static final PrintStream err`

El canal de salida estándar de errores. El canal está ya abierto y dispuesto para ser leído. Típicamente, este canal corresponde a la salida por pantalla u otro destino de salida especificado por el entorno o el usuario. Por convenio, este canal de salida se utiliza para mostrar mensajes de error u otra información que debería llamar la atención inmediata del usuario incluso si el canal de la salida estándar, esto es, el valor de la variable `out`, se ha direccionado a un fichero u otro destino que no se muestra continuamente.

Métodos importantes

`static long currentTimeMillis( )`

Devuelve la hora actual en milisegundos.

`static String getProperty( String key )`

Devuelve la propiedad del sistema indicada por la clave especificada.

`static void exit( int status )`

Concluye la ejecución del programa. El argumento sirve para indicar el estado de terminación; por convenio, un código distinto de cero indica una terminación anormal. Este método no termina de forma normal, y no puede ser llamado desde un applet.

`static void gc( )`

Ejecuta el recolector de basura. La llamada a este método solicita al sistema de ejecución que invierta esfuerzos en reciclar objetos que ya no pueden ser utilizados, para hacer que la memoria que ocupan vuelva a estar disponible. Cuando el método termina, el sistema ha hecho todo lo posible por liberar el espacio de los objetos no utilizados.

C.1.7 `Thread`

Una hebra es un flujo de ejecución. En particular, un programa. Una aplicación puede tener simultáneamente diversas hebras en ejecución. Las hebras se necesitan en las animaciones. Este uso de las hebras se discute en las Secciones D.4 y D.5.5. Aquí sólo se enumeran los métodos que están relacionados con aplicaciones de animación o que se han utilizado en otro punto de este libro.

Métodos importantes

`Thread ( Runnable target )`

Crea un nuevo objeto `Thread`. `target` es el objeto cuyo método `run` es ejecutado.

`static void sleep( long milliseconds )`

Hace que se duerma (cese su ejecución) la hebra que está actualmente en ejecución, durante el número especificado de milisegundos. Lanza la excepción `InterruptedException` si otra hebra ya ha interrumpido a ésta. Esta excepción debería ser capturada.

`void start( )`

Hace que comience la ejecución de la hebra, lo que se consigue llamando al método `run` de la hebra. Lanza la excepción `IllegalThreadStateException` si la hebra ya se está ejecutando. Esta excepción no tiene por qué ser capturada.

`void stop( )`

Detiene la ejecución de la hebra. Se puede parar una hebra que todavía no ha empezado a ejecutarse. Si la hebra empieza a ejecutarse más tarde, termina inmediatamente.

`boolean isAlive( )`

Comprueba si la hebra está viva. Una hebra está viva si ha empezado a ejecutarse y todavía no ha terminado. Devuelve `true` si y sólo si la hebra está viva.

`void suspend( )`

Suspende la hebra. Si la hebra está viva, es suspendida y su ejecución no progresa hasta que se reanude explícitamente.

`void resume( )`

Reanuda la ejecución de una hebra suspendida. Si la hebra está viva pero suspendida, la ejecución es reanudada permitiéndose que haga progresos en ella.

C.1.8 `Throwable`

La clase `Throwable` es la superclase de todos los errores y excepciones en el lenguaje Java. Sólo los objetos que son instancias de esta clase (o de alguna de sus subclases) pueden ser lanzados. De forma similar, sólo esta clase o alguna de sus subclases puede ser el tipo del argumento de la cláusula `catch`. Un objeto `Throwable` contiene una instantánea de la pila de ejecución de su hebra en el momento en que ésta fue creada. También puede contener un mensaje que dé más información sobre el error detectado.

Métodos importantes

`Throwable( )`

Construye un nuevo objeto `Throwable` sin mensaje. La traza de la pila se rellena automáticamente.

`Throwable( String message )`

Construye un nuevo objeto `Throwable` con el mensaje especificado. La traza de la pila se rellena automáticamente.

`String getMessage( )`

Devuelve el mensaje del objeto.

`String getLocalizedMessage( )`

Devuelve una descripción localizada del objeto. Las subclases pueden sobrescribir este método para producir un mensaje específico. Para las subclases que no lo sobrescriban, la implementación por defecto devuelve el mismo resultado que `getMessage`.

`String toString( )`

Devuelve una breve descripción del objeto.

`void printStackTrace( )`

Imprime el objeto y su traza en el canal de errores estándar.

C.2 Clases del paquete `java.io`

Este paquete contiene clases para manipular ficheros y canales de entrada/salida. Las clases estudiadas aquí son nuevas en Java 1.1, excepto la clase `File`.

C.2.1 `BufferedReader`

Lee texto desde un canal de entrada de caracteres, almacenando los caracteres en un buffer de tal forma que la lectura de caracteres, vectores y líneas sea eficiente. Se puede especificar el tamaño del buffer, o se puede utilizar un tamaño por defecto. En la mayoría de las ocasiones el valor por defecto es suficiente. En general, cada petición de lectura sobre un `Reader` provoca la correspondiente lectura

en el canal de entrada. Por tanto, cuando la operación de lectura pueda ser costosa, es aconsejable envolver el `Reader` con un `BufferedReader`, como un `FileReader` o `InputStreamReader`.

Métodos importantes

`BufferedReader( Reader in )`
Crea un almacén para un secuencia de caracteres de entrada, utilizando el tamaño por defecto.

`int read( )`
Lee un solo carácter. Lanza la excepción `IOException` si se produce un error de E/S. El carácter se devuelve como un entero. Por tanto, un uso típico requerirá una conversión a `char`.

`String readLine( )`
Lee una línea de texto. Se considera que una línea termina con un carácter de fin de línea (`'\n'`), un salto de carro (`'\r'`) o un salto de carro seguido inmediatamente por un fin de línea. Devuelve una cadena que contiene la línea, sin incluir los caracteres de terminación, o `null` si se ha llegado al final del flujo. Lanza la excepción `IOException` si se produce un error de E/S.

`boolean ready( )`
Informa sobre si el canal está preparado para ser leído. Una secuencia de caracteres en un buffer está lista si el buffer no es vacío o si la secuencia de caracteres subyacente está lista. Lanza la excepción `IOException` si se produce un error de E/S.

`void close( )`
Cierra el canal. Lanza la excepción `IOException` si se produce un error de E/S.

C.2.2 `File`

La clase `File` proporciona una abstracción que trata de forma independiente de la máquina la mayoría de las complejidades dependientes de la máquina sobre ficheros y rutas de acceso.

Constante importante

`static final char separatorChar`
El carácter separador de directorios dependiente del sistema. Este carácter separa el directorio del nombre del fichero.

Métodos importantes

`File( String name )`
Crea un objeto `File` que representa el fichero cuyo nombre se ha dado en `name`.

`String getName( )`
Devuelve el nombre del fichero representado por el objeto. El nombre del fichero es todo aquello después del separador de directorios en el nombre completo.

`String getPath( )`
Devuelve el nombre de la ruta del fichero representado por el objeto.

`boolean exists( )`
Devuelve `true` si y sólo si el fichero especificado existe.

`boolean isDirectory( )`
Devuelve `true` si y sólo si el objeto al que se aplica es un directorio.

`long length( )`
Devuelve la longitud del fichero representado por el objeto.

`String [ ] list( )`
Devuelve un vector con los nombres de los ficheros en el directorio representado por este fichero. Esta lista no incluye ni el directorio actual ni el directorio padre («.» y «..»).

C.2.3 `FileReader`

`FileReader` es una clase cuyo uso resulta conveniente para leer ficheros de caracteres. Esta clase extiende `Reader`, pero sus métodos no deberían ser utilizados directamente. Para hacerlo adecuadamente, envuelva el objeto con uno de la clase `BufferedReader`, tanto por eficiencia como para así poder llamar a `readLine`.

Métodos importantes

`FileReader( String filename )`
Construye un objeto `FileReader` para el nombre de fichero dado. Lanza la excepción `FileNotFoundException` si el fichero no se puede abrir.

C.2.4 `InputStreamReader`

Un `InputStreamReader` es un puente entre las secuencias de bytes y las secuencias de caracteres: lee bytes y los traduce a caracteres. Su uso más importante es la construcción con `System.ini` como parámetro. Esta clase extiende la clase abstracta `Reader`, pero sus métodos no deberían usarse directamente. Para hacerlo adecuadamente, envuelva el objeto con uno de la clase `BufferedReader`, tanto por eficiencia como para así poder llamar a `readLine`.

Métodos importantes

`InputStreamReader( InputStream in )`
Crea un objeto `InputStreamReader` a partir del `InputStream` dado. Un `InputStream` típico es `System.ini`.

C.2.5 `PushbackReader`

`PushbackReader` implementa un lector de secuencias de caracteres que permite devolver caracteres a la secuencia.

Métodos importantes

`PushbackReader( Reader in )`

Crea un nuevo lector con un buffer de retroceso de un carácter.

`PushbackReader( Reader in, int bufferSize )`

Crea un nuevo lector con un buffer de retroceso cuya capacidad viene especificada por el segundo argumento.

`int read( )`

Lee un solo carácter. Devuelve el carácter leído, o -1 si se ha alcanzado el final del fichero. Lanza la excepción `IOException` si ocurre algún error de E/S. El carácter se devuelve como un entero. Este uso típico requiere una conversión de tipos.

`void unread( int ch )`

Devuelve al flujo el carácter `ch`. El parámetro que recibe es un entero. Este uso típico requiere una conversión de tipos. Lanza la excepción `IOException` si el buffer de retroceso está lleno u ocurre algún error de E/S.

`void unread( char [ ] cbuf )`

Devuelve al flujo un vector de caracteres copiandolos al principio del buffer de retroceso. Cuando el método termina, el siguiente carácter a leer tendrá el valor `cbuf[0]`, el carácter después de éste tendrá el valor `cbuf[1]`, y así sucesivamente. Lanza una excepción `IOException` si no hay suficiente espacio en el buffer o si ocurre algún error de E/S.

C.3 Clases del paquete `java.util`

Este paquete define varias clases útiles, incluyendo algunas estructuras de datos. También proporciona soporte para fechas y horas. Se describen a continuación las tres clases que se utilizan en el texto.

C.3.1 `Random`

Se utiliza una instancia de esta clase para generar una secuencia de números pseudoaleatorios. La clase utiliza una semilla de 48 bits, la cual se modifica utilizando una fórmula de congruencia lineal (dada al final de la Sección 9.2). Si se crean con la misma semilla dos instancias de la clase `Random` y se realiza sobre cada una la misma secuencia de operaciones, generarán y devolverán la misma secuencia de números.

Métodos importantes

`Random( )`

Crea un nuevo generador de números aleatorios. La semilla se inicializa con un valor basado en la hora actual.

`Random( long seed )`

Crea un nuevo generador de números aleatorios con la semilla dada.

`int nextInt( )`

Devuelve el próximo entero pseudoaleatorio y uniformemente distribuido de la secuencia del generador.

`long nextLong( )`

Devuelve el siguiente entero `long` pseudoaleatorio y uniformemente distribuido de la secuencia del generador.

`double nextDouble( )`

Devuelve el siguiente valor `double` pseudoaleatorio y uniformemente distribuido con media 0,0 y desviación típica 1,0 de la secuencia del generador.

`double nextGaussian( )`

Devuelve el próximo valor pseudoaleatorio, como valor de tipo `double` distribuido según la distribución gausiana de media 0,0 y desviación típica 1,0, a partir de la secuencia de este generador de números aleatorios.

C.3.2 `StringTokenizer`

La clase `StringTokenizer` permite que una aplicación pueda descomponer una cadena en tokens. Los separadores (los caracteres que separan tokens) pueden especificarse en el momento de la creación o de uno en uno. Una instancia de la clase `StringTokenizer` se comporta de dos posibles formas, dependiendo de si al crearse, se le dio a `returnTokens` el valor `true` o `false`. Si es `false`, los caracteres separadores sirven para separar tokens, y un token es una secuencia máxima de caracteres consecutivos que no sean separadores. Si `returnTokens` es `true`, se considera que los caracteres separadores son tokens. Un token es o bien un carácter separador o una secuencia maximal de caracteres consecutivos que no sean separadores.

Métodos importantes

`StringTokenizer( String str )`

Construyé un `StringTokenizer` para la cadena especificada. Utiliza el conjunto de separadores por defecto, el cual es `" \t\n\r"`: el espacio en blanco, el carácter tabulador, el carácter de nueva línea y el carácter de salto de línea.

`StringTokenizer( String str, String delim )`

Construye un `StringTokenizer` para la cadena especificada. Toma como separadores los caracteres en el argumento `delim`.

`StringTokenizer( String str, String delim, boolean returnTokens )`

Construye un `StringTokenizer` para la cadena especificada. Los caracteres en el argumento `delim` son los separadores. Si `returnTokens` es `true`, los caracteres separadores también se devuelven como tokens; cada separador se devuelve como una cadena de longitud uno. Si es `false`, los caracteres separadores se saltan y sirven, únicamente, para separar tokens.

`boolean hasMoreTokens( )`
Devuelve `true` si y sólo si hay más tokens disponibles en la cadena.

`String nextToken( )`
Devuelve el siguiente token. Lanza la excepción `NoSuchElementException` si no hay más tokens.

`String nextToken( String delim )`
Devuelve el siguiente token utilizando `delim` como conjunto de separadores. Este conjunto se mantiene como nuevo conjunto de separadores por defecto tras la llamada. Se lanza la excepción `NoSuchElementException` si no hay más tokens.

`int countTokens( )`
Devuelve el número de tokens que quedan en la cadena utilizando el conjunto actual de separadores.

C.3.3 `Vector`

La clase `Vector` implementa vectores de objetos de tamaño ajustable.

Métodos importantes

`Vector( )`
Construye un vector vacío.

`Vector( int initialCapacity )`
Construye un vector vacío con la capacidad inicial especificada.

`Object elementAt( int index )`
Devuelve el elemento en la posición indicada. Lanza la excepción `ArrayIndexOutOfBoundsException` si la posición no es válida.

`void setElementAt( Object obj, int index )`
Hace que la componente del vector en la posición indicada sea el objeto dado. La componente anterior en esa posición es descartada. La posición debe ser un valor mayor o igual que 0 y menor que el tamaño actual del vector. En otro caso, se lanza la excepción `ArrayIndexOutOfBoundsException`.

`void setSize( int newSize )`
Establece el tamaño del vector. Si el nuevo tamaño es mayor que el actual, se añaden nuevos elementos `null` al final del vector. Si el nuevo tamaño es menor que el actual, se descartan todas las posiciones desde `newSize`.

`int capacity( )`
Devuelve la capacidad actual del vector.

`int size( )`
Devuelve el número de componentes en el vector.

En Internet

TestString.java Contiene la versión en inglés de las Figuras C.1 y C.2 y una rutina `main` que hace las llamadas. Se encuentra en el directorio **AppendixC**.

APÉNDICE

D Interfaces gráficas de usuario[1]

Una *interfaz gráfica de usuario* (GUI[2]) es la alternativa moderna a la E/S por el terminal que permite a un programa comunicarse con su usuario. En una GUI se crea una aplicación de ventanas. Tenemos diversas alternativas para realizar la entrada: seleccionando una opción entre una lista de ellas, pulsando botones, completando formularios y empleando el ratón. La salida puede efectuarse mediante la escritura de texto o la realización de gráficos. En Java, la programación de GUI se realiza a través del *Abstract Window Toolkit* (AWT), que es un paquete estándar incluido en todos los sistemas Java. Un elemento relacionado es el applet de Java, que es un programa que puede descargarse de Internet y ejecutarse en un terminal. Los applets emplean, invariablemente, el AWT.

En este apéndice veremos:

- Los elementos básicos de las GUI en el AWT.
- Cómo estos elementos comunican información.
- Cómo pueden integrarse dichos elementos en una ventana.
- Cómo se dibujan gráficos.
- Cómo se diseñan applets en Java.

D.1 El Abstract Window Toolkit

El *Abstract Window Toolkit* (AWT) es un conjunto de utilidades GUI incluido en todos los sistemas Java. Está compuesto por las clases básicas que permiten diseñar interfaces de usuario. Dichas clases se encuentran en el paquete `java.awt`. El AWT es portable y funciona en multitud de plataformas. Su uso es sencillo si se pretende diseñar interfaces simples. Gracias a él, las GUI pueden implementarse sin recurrir a ayudas de desarrollo visual, y supone una mejora importante sobre las interfaces de terminal básicas.

Normalmente, en un programa en el que se realiza E/S por el terminal, se pregunta al usuario acerca de la entrada y después se ejecuta una instrucción que lee una línea del terminal. Cuando dicha línea es leída, se procesa. En este contexto el flujo de control puede seguirse fácilmente. Sin embargo, la programación de GUI es diferente. En ella, las componentes de la entrada se organizan en una ventana. Después de mostrar la ventana, el programa espera un evento, como que se pulse un botón, y en ese momento se invoca un manejador de eventos. Esto implica que

[1] *N. del T.:* En este apéndice los programas se muestran en su versión original para asegurar la coherencia co
[2] *N. del T.:* La abreviatura procede de su nombre en inglés: *Graphical User Interface*.

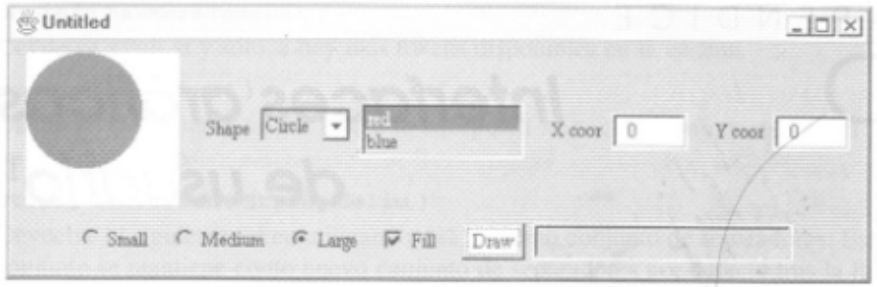


Figura D.1 Una GUI que muestra algunas de las componentes básicas.

en un programa GUI, el flujo de control es menos evidente. El programador debe suministrar el manejador de eventos empleado en la ejecución de algunos fragmentos de código.

Java 1.0 incluía un modelo de eventos que era bastante complejo de manejar. En Java 1.1 ha sido sustituido por un modelo de eventos más robusto y sencillo. Como era de esperar, los dos modelos no son completamente compatibles. Más concretamente, un compilador de Java 1.0 no puede compilar código que haga uso del nuevo modelo de eventos. Por su parte, los compiladores de Java 1.1 muestran mensajes ante el uso de las construcciones de Java 1.0. Sin embargo, el código de Java 1.0 ya compilado sí puede ejecutarse sin problemas en un intérprete de Java 1.1. En este apéndice sólo se describe el nuevo modelo de eventos.

La Figura D.1 muestra algunos de los elementos básicos incluidos en el AWT. Entre ellos están `Choice` (se ha seleccionado *Circle*), un elemento de tipo `List` (se ha seleccionado *red*), campos de texto de tipo `TextField`, campos de confirmación `Checkbox` y un botón de tipo `Button` (cuyo nombre es *Draw*). Al lado del botón puede verse un campo de texto empleado únicamente para la salida (por esta razón es más oscuro que los campos de texto que hay sobre él). En la esquina superior izquierda hay un objeto de la clase `Canvas`, empleado para dibujar y aceptar la entrada a través del ratón.

Este apéndice describe la organización básica del AWT. En primer lugar, se explican diferentes clases de objetos, cómo pueden emplearse para realizar la entrada y la salida, cómo organizarlos y cómo se manejan los distintos eventos. Tras ello, se describe el concepto relacionado de applet, por medio del cual se descarga un programa de Internet y se ejecuta en un browser como Netscape Navigator o Internet Explorer.

D.2 Elementos básicos del AWT

El AWT se organiza empleando una jerarquía de herencia de clases. En la Figura D.2 se muestra una versión reducida de esta jerarquía, ya que no se muestran algunas clases intermedias. Por ejemplo, en la jerarquía completa, `TextField` y `TextArea` extienden la clase `TextComponent`, mientras que no se muestran muchas clases de las relacionadas con fuentes de letras, colores, y otros objetos de la jerarquía de `Component`. Las clases `Font` y `Color`, definidas en el paquete `java.awt`, extienden la clase `Object`.

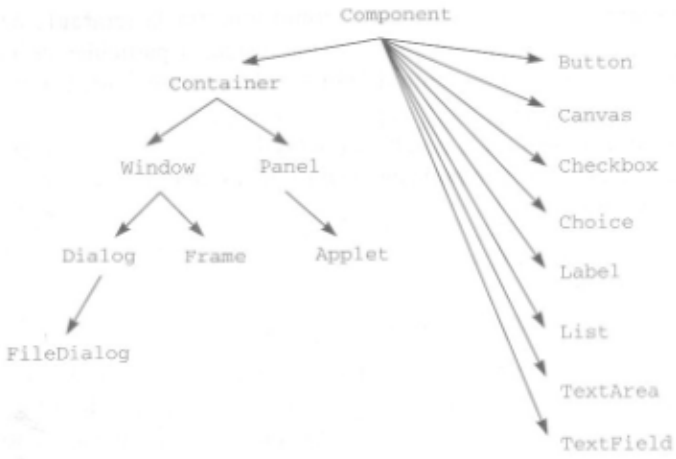


Figura D.2 Jerarquía reducida del AWT.

D.2.1 Component

La clase *`Component`* es una clase abstracta, superclase de muchos objetos del AWT. Debido a que es abstracta, no puede instanciarse. Un objeto de la clase `Component` representa un elemento que tiene una posición y un tamaño, y puede dibujarse en la pantalla, además de aceptar eventos de entrada. En la Figura D.2 se muestran algunos ejemplos de objetos de esta clase.

La clase `Component` contiene multitud de métodos. Algunos de ellos pueden emplearse para especificar el color o el tipo de fuente; otras se usan para manipular eventos. Algunos de los métodos más importantes son

```
void paint ( Graphics g );
void setSize( int width, int height );
void setBackground( Color c );
void setFont( Font f );
void show( );
```

Normalmente, el método `paint` se asocia a objetos de tipo `Canvas`. Se describe en la Sección D.3.2. El método `setSize`[3] se emplea para cambiar el tamaño de un objeto. Funciona sobre objetos de la clase `Canvas`, pero no debe invocarse sobre objetos que tengan configuración automática, como los de tipo `Button`. Los métodos `setBackground` y `setFont` se usan para modificar el color del fondo y el tipo de letra asociados a un objeto de la clase `Component`. Necesitan, respectivamente, un objeto de las clases `Color` y `Font`. Por último, el método `show` convierte una componente en visible. Su uso típico se realiza sobre un `Frame`.

D.2.2 Container

La clase *`Container`* es la superclase abstracta que representa todas las componentes que pueden contener a otras. Un ejemplo de subclase es la clase `Window`,

[3] En Java 1.1, el método `setSize` sustituye a `resize`.

que representa las ventanas de alto nivel. Tal y como muestra la jerarquía de herencia, un `Container` ES-UN(A) `Component`. Una instancia particular de la clase `Container` almacenará una colección de objetos de la clase `Component`, además de otros objetos `Container`.

Esta clase dispone de un objeto auxiliar muy útil, llamado `LayoutManager`, que es una clase que sitúa las distintas componentes dentro de un objeto `Container`. Algunos métodos destacados son

```
void setLayout( LayoutManager mgr );
void add( Component comp);
void add( Component comp, Object where );
```

Los objetos de la clase `LayoutManager` se describen en la Sección D.3.1. Un objeto de tipo `Container` debe definir primero cómo deben organizarse los objetos dentro de él. Esto se hace mediante `setLayout`. Después, se añaden uno a uno los objetos, empleando `add`.

D.2.3 Ventanas de alto nivel

Tal y como muestra la Figura D.2, existen dos tipos de objetos `Container`: las ventanas de alto nivel `Window` y los objetos de la clase `Panel`. `Window` está especializada en varias entidades de alto nivel. Existen cuatro tipos básicos de ventanas de alto nivel:

1. `Window`: ventana de alto nivel sin borde.
2. `Frame`: ventana de alto nivel con borde y una barra de menús asociada (pertenece a la clase `MenuBar`[4]).
3. `Dialog`: ventana de alto nivel empleada para crear diálogos; una subclase suya es `FileDialog`.
4. `FileDialog`: ventana de alto nivel utilizada cuando se necesita mostrar una lista de los archivos de un directorio.

Una aplicación que emplea una interfaz GUI debe tener un `Frame` (o una clase que extienda a `Frame`) como contenedor más externo. La Figura 2.11 muestra un programa que enumera el contenido de los ficheros cuyos nombres se especifican en la línea de comandos. En una aplicación GUI, el nombre del fichero debe especificarse en un objeto de tipo `FileDialog`. La Figura D.3 muestra una GUI sencilla que usa un objeto de la clase `FileDialog` para obtener el nombre del fichero de entrada. Como antes, la lista se envía a la salida estándar, aunque evidentemente, podría enviarse a una componente GUI de salida.

En la línea 1 de la Figura D.3 se importan todas las clases del paquete `java.awt`. La rutina `main` comienza creando y mostrando un nuevo `Frame f` en las líneas 9 y 10. Recordemos que en toda aplicación GUI debe crearse un `Frame`. Ya que de momento no hemos hecho nada más, el `Frame` no tiene tamaño, no contiene ninguna componente y consiste únicamente en una barra de menús en la parte superior. A continuación, en la línea 13, creamos un `FileDialog d`, asociado al `Frame f`, cuyo nombre es *List File* y su acción es *LOAD*. El resultado de la línea 14 se muestra en la Figura D.4. La llamada en la línea 17 al método `getFile`

discuten en este apéndice.

```
1  import java.awt.*;
2
3  public class FileDialogTest
4  {
5          // Muestra una ventana de diálogo, selecciona un fichero
6          // y muestra su contenido
7      public static void main( String [ ] args )
8      {
9          Frame f = new Frame( );
10         f.show( );
11
12         FileDialog d;
13         d = new FileDialog( f, "List File", FileDialog.LOAD );
14         d.show;
15
16         String fileName;
17         if ( ( fileName = d.getFile( ) ) != null )
18             ListFiles.listFile( fileName ); // Figura 2.11
19
20         System.exit( 0 );
21     }
22 }
```

Figura D.3 Programa de la Figura 2.11 adaptado para emplear una GUI.

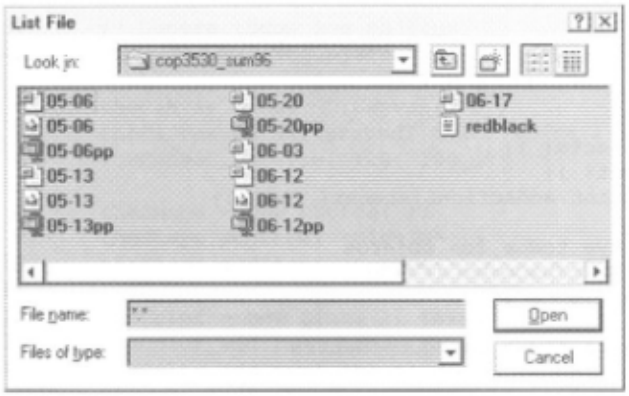


Figura D.4 Salida del programa `FileDialogTest` de la Figura D.3.

devuelve el nombre del fichero seleccionado (o `null` si la petición ha sido cancelada). En la línea 18, invocamos al método `listFile` desde la clase implementada en la Figura 2.11[5].

Este ejemplo muestra que en Java es sencillo programar cosas poco complicadas. Los objetos de tipo `Dialog` (al contrario de lo que sucede con los objetos especiales de `FileDialog`) son más difíciles de programar. Esto es debido a que el método `show` de la clase `FileDialog` no devuelve nada hasta que no se haya seleccionado un fichero. Así, en la línea 17, una acción cualquiera en `FileDialog` se ha realizado sólo cuando `getFile` termina. En la clase `Dialog`, el método `show` retorna inmediatamente. De este modo, la llamada para examinar la acción

[5] *N. del T.:* En la Figura 2.11 se tradujo `ListFiles` por `ContenidoFicheros` y `listFile` por `contenidof`

del objeto `Dialog` se realizaría antes de cualquier otra cosa, a menos que se necesite un esfuerzo adicional de programación. Además, como el `Frame f` no contiene ningún otro objeto, no debemos preocuparnos por la configuración de los objetos.

D.2.4 Panel

Otra subclase de `Container` es *`Panel`*. Los objetos de tipo `Panel` se emplean para almacenar una colección de objetos, pero no generan ningún borde. Por ello, es la más simple de las clases `Container`.

El uso principal de esta clase es la organización de objetos en unidades. Por ejemplo, considere un registro que requiere un nombre, dirección, número de la seguridad social y los números de teléfono del domicilio y del trabajo. Todos estos elementos podrían componer un `PersonalPanel`. Entonces, el formulario puede contener varias entidades de tipo `PersonalPanel` para permitir la posibilidad de realizar varios registros.

Como ejemplo, la Figura D.5 indica cómo se agrupan las componentes de la Figura D.1 en una clase `Panel` y muestra la técnica general para crear una subclase de `Panel`. Nos queda construir los objetos, inicializarlos de la forma adecuada y tratar el evento de pulsar el botón.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class GUI extends Panel implements ActionListener
5  {
6      public GUI( )
7      {
8          makeTheObjects( );
9          doTheLayout( );
10         theDrawButton.addActionListener( this );
11     }
12         // Construye todos los objetos
13     private void makeTheObjects( )
14       { /* Figura D.6 */ }
15
16         // Configura todas las componentes
17     private void doTheLayout( )
18       { /* Figura D.9 */ }
19
20         // Procesa la pulsación del botón
21     private void ActionPerformed( ActionEvent evt )
22       { /* Figura D.12 */ }
23
24     private GUICanvas theCanvas;
25     private Choice    theShape;
26     private List      theColor;
27     private TextField theXCoor;
28     private TextField theYCoor;
29     private Checkbox  smallPic;
30     private Checkbox  mediumPic;
31     private Checkbox  largePic;
32     private Checkbox  theFillBox;
33     private Button    theDrawButton;
34     private TextField theMessage;
35 }
```

Figura D.5 Clase GUI básica mostrada en la Figura D.1.

Observe que `GUI` implementa la interfaz `ActionListener`. Esto significa que sabe cómo manejar un evento (en este caso, el pulsar un botón). Para que una clase implemente la interfaz `ActionListener`, debe incluir un método `actionPerformed`. Además, cuando el botón genera el evento, debe saberse qué componente va a recibir dicho evento. En este caso, gracias a la llamada de la línea 10 (en la Figura D.5), el objeto `GUI` que contiene el objeto de tipo `Button` le comunica a este último que debe producir el evento. Los detalles del procesamiento de eventos se explican en la Sección D.3.3.

Un segundo uso de los objetos `Panel` es la agrupación de objetos en unidades con el objetivo de simplificar las configuraciones. Esto se discute en la Sección D.3.4. El tercer uso de la clase `Panel` es el `Applet`, que es una subclase suya.

D.2.5 Componentes importantes de la E/S

El AWT incluye un conjunto de componentes que pueden emplearse para realizar la entrada y salida. Estas componentes son sencillas de generar y usar. El código de la Figura D.6 muestra cómo se construye cada una de las componentes de la Figura D.1. Generalmente, esto incluye la llamada a un constructor y la aplicación de un método para adaptar la componente. El código no especifica cómo se orga-

```
1      // Genera todos los objetos
2  private void makeTheObjects( )
3  {
4      theCanvas = new GUICanvas( );
5      theCanvas.setBackground( Color.white );
6      theCanvas.setSize( 100, 100 );
7
8      theShape = new Choice( );
9      theShape.addItem( "Circle" );
10     theShape.addItem( "Square" );
11
12     theColor = new List( 2, false );
13     theColor.addItem( "red" );
14     theColor.addItem( "blue" );
15     theColor.select( 0 ); // La elección por defecto es "red"
16
17     theXCoor = new TextField( 5 );
18     theYCoor = new TextField( 5 );
19
20     CheckboxGroup theSize = new CheckboxGroup( );
21     smallPic = new Checkbox( "Small", theSize, false );
22     mediumPic = new Checkbox( "Medium", theSize, true );
23     largePic = new Checkbox( "Large", theSize, false );
24
25     theFillBox = new Checkbox( "Fill" );
26     theFillBox.setState( false );
27
28     theDrawButton = new Button( "Draw" );
29     theMessage = new TextField( 25 );
30     theMessage.setEditable( false );
31 }
```

Figura D.6 Código que genera los objetos de la Figura D.1.

nizan los elementos en el panel o cómo se examina el estado de las mismas. Recordemos que la programación de las GUI consiste en mostrar la interfaz y esperar a que se produzcan eventos. La configuración de las componentes y el tratamiento de los eventos se discuten en la Sección D.3.

Label

Label es una componente utilizada para colocar texto en un contenedor. Su uso principal es etiquetar otras componentes como Choice, List, TextField o Panel (muchas otras componentes ya muestran su nombre de algún modo). En la Figura D.1. las frases *Shape*, *X Coor* e *Y Coor* son etiquetas. Los objetos de la clase Label se construyen con una cadena de caracteres opcional que puede modificarse con el método setText. Los métodos son

```
Label( );
Label( String theLabel );
void setText( String theLabel );
```

Button

La clase *Button* se emplea para crear botones con etiqueta. La Figura D.1 contiene uno de ellos, cuya etiqueta es *Draw*. Cuando se pulsa un objeto de la clase Button, se genera un *evento*. La Sección D.3.3 describe cómo se procesan estos eventos. La interfaz Button es similar a Label. Más concretamente, un objeto de la clase Button se construye con una cadena de caracteres opcional. Esta etiqueta puede modificarse con el método setText. Los métodos son

```
Button( );
Button( String theLabel );
void setText( String theLabel );
```

Choice

La clase *Choice* se emplea para seleccionar una sola cadena de caracteres de entre una lista desplegable de ellas. Sólo podemos seleccionar cadenas en la lista y en cada momento sólo se puede realizar una única elección. En la Figura D.1 el tipo de theShape es un objeto Choice; en este momento la opción elegida es *Circle*. Algunos de los métodos de esta clase son

```
Choice( );
void addItem( String item );
String getSelectedItem( );
int getSelectedIndex( );
void select( int index );
```

Un objeto Choice se construye sin parámetros. Se pueden añadir cadenas de caracteres a la lista de alternativas Choice. Cuando se invoca al método getSelectedItem, se devuelve una cadena que representa el elemento elegido (o null si no se realiza ninguna elección). Gracias al método getSelectedIndex, puede devolverse el índice de la opción (calculado por el orden de llamadas a addItem) en lugar de devolver la cadena correspondiente. El primer elemento añadido tiene índice 0, el siguiente índice 1 y así sucesivamente. Esto puede ser muy útil ya que si un vector guarda información relativa a cada una de las elecciones, getSelectedIndex puede emplearse para indexar dicho vector. El método select se utiliza para especificar cuál es la elección por defecto.

List

La componente *List* permite la selección de una cadena de caracteres de entre una lista desplegada de ellas. En la Figura D.1 la elección de los colores se presenta en forma de lista. Los objetos de la clase List se distinguen de los objetos de la clase Choice por tres detalles:

1. Los objetos de la clase List pueden configurarse de modo que permitan seleccionar uno o varios elementos.
2. Los objetos de la clase List permiten al usuario ver varias opciones al mismo tiempo.
3. Los objetos de la clase List ocupan mayor espacio en la pantalla que los de la clase Choice.

Los métodos básicos de esta clase son

```
List( );
List( int rows, boolean multipleSelections );
void addItem( String item );
String getSelectedItem( );
String [ ] getSelectedItems( );
void select( int index );
```

Un objeto de List se construye sin parámetros o con dos de ellos. El constructor con dos parámetros especifica el número visible de filas (en otras palabras, el número de filas que se mostrarán) y un booleano que determina si se permiten selecciones múltiples. Los métodos addItem, getSelectedItem y select tienen el mismo comportamiento que los métodos correspondientes de la clase Choice. getSelectedItem devuelve null si no se selecciona ningún elemento o bien se selecciona más de uno. getSelectedItems se emplea para tratar selecciones múltiples; devuelve un vector de cadenas de caracteres correspondiente a los elementos seleccionados. Al igual que en el caso de Choice, empleando otros métodos públicos se pueden obtener índices en lugar de cadenas de caracteres.

Checkbox

Una *Checkbox* es una componente GUI que tiene dos estados posibles: *activado* y *desactivado*. El estado *activado* es true y el estado *desactivado* es false. La Figura D.1 contiene cuatro objetos de tipo Checkbox. En esta figura, la caja *Fill* es true y es diferente a las otras tres. Esto es debido a que los otros objetos Checkbox forman parte de un *CheckboxGroup*: sólo una de las tres cajas de este grupo podría ser true. Cuando se selecciona un objeto Checkbox de un grupo, todos los demás objetos del grupo dejan de estar seleccionados. Un objeto de

la clase `CheckboxGroup` se construye sin parámetros. Observe que no se trata de un elemento de tipo `Component`; es sólo un objeto lógico.

Los métodos más empleados en esta clase son

```
Checkbox( );
Checkbox( String theLabel );
Checkbox( String theLabel, CheckboxGroup group, boolean state );
boolean getState( );
void setLabel( );
void setState( );
```

Una caja `Checkbox` individual se construye con una etiqueta opcional. Si no se especifica ninguna, puede añadirse posteriormente una con `setLabel`. `setLabel` también puede emplearse para cambiar la etiqueta existente. Un objeto de la clase `Checkbox` que forma parte de un `CheckboxGroup` se construye indicando una etiqueta, el grupo y su estado inicial. Nótese que si `state` es `true`, el estado de todas las cajas de ese grupo construidas anteriormente se actualiza a `false`. `setState` es empleado con mayor asiduidad para establecer el estado por defecto de una caja `Checkbox` individual. `Checkbox.getState` devuelve el estado de un objeto de tipo `Checkbox`.

Canvas

Una componente *`Canvas`* representa un área de la pantalla, rectangular y en blanco, en la que la aplicación puede dibujar. Los gráficos más sencillos se describen en la Sección D.3.2. Un objeto de la clase `Canvas` también puede recibir datos de entrada del usuario en forma de eventos producidos por el ratón o el teclado. Los elementos de esta clase nunca se emplean directamente: el programador define una subclase de `Canvas` con la funcionalidad adecuada. La subclase diseñada debe sobrescribir el método

```
void paint( Graphics g );
```

TextField

Un objeto de la clase *`TextField`* es una componente que se muestra al usuario como una única línea de texto. Un objeto de la clase *`TextArea`* permite mostrar varias líneas y su funcionalidad es similar. Por este motivo, en esta sección sólo consideramos la clase `TextField`. Por defecto, el texto puede ser editado por el usuario, pero es posible convertirlo en no editable. En la Figura D.1 aparecen tres objetos de tipo `TextField`: dos para las coordenadas y una tercera, no editable por el usuario, que se emplea para los mensajes de error. El color de fondo de un campo de texto no editable es distinto del de un campo de texto editable. Algunos de los métodos más usuales de esta clase son

```
TextField( );
TextField( int cols );
TextField( String text, int cols );
String getText( );
void setEditable( boolean editable );
void setText( String text );
```

Un objeto de la clase `TextField` se construye sin parámetros o especificando un texto opcional inicial y el número de columnas. Una advertencia: en muchos sistemas es necesario especificar más columnas de las que se espera emplear, debido a un aparente fallo en el AWT. El método `setEditable` se emplea para impedir la escritura en el campo de texto. `setText` puede emplearse para escribir mensajes de texto en el objeto `TextField`. `getText` se utiliza para leer el texto del objeto `TextField`.

D.3 Principios básicos del AWT

Esta sección estudia tres facetas importantes de la programación AWT. En primer lugar, se estudia cómo se organizan los objetos dentro de un contenedor. Seguidamente se explica cómo se tratan los eventos tales como la pulsación de botones. Por último, describimos cómo se realizan gráficos en el interior de los objetos de tipo `Canvas`.

D.3.1 Configuradores

Un *configurador* organiza de forma automática las componentes dentro de un contenedor. Se asocia a un contenedor mediante el comando `setLayout`. Un ejemplo de su uso es la llamada

```
setLayout( new FlowLayout( ) );
```

Observe que no se necesita guardar una referencia a un configurador. El contenedor sobre el que se aplica el comando `setLayout` lo hace, almacenándolo como un atributo privado. Cuando se emplea un configurador, no funcionan muchas peticiones para cambiar el tamaño de algunas componentes, como los botones, ya que el configurador escoge sus propias dimensiones. La idea es que el configurador determinará las dimensiones ideales para satisfacer las especificaciones.

FlowLayout

El más sencillo de los configuradores es *`FlowLayout`*. Cuando un contenedor se configura empleando `FlowLayout`, sus componentes se añaden en una fila de izquierda a derecha. Cuando no queda espacio libre en una fila, se forma una nueva. Por defecto, cada fila está centrada. Esto puede modificarse especificando un parámetro adicional en el constructor con el valor `FlowLayout.LEFT` o `FlowLayout.RIGHT`.

El problema al emplear este configurador aparece cuando una fila se rompe en un lugar inadecuado. Por ejemplo, si una fila es demasiado corta, puede producirse una separación entre un objeto de tipo `Label` y otro de tipo `TextField`, aunque ambos deban aparecer juntos. Una forma de evitar estas complicaciones es crear un `Panel` con esos dos elementos y añadir después dicho `Panel` al contenedor. Otro problema que presenta `FlowLayout` es alinear verticalmente los objetos.

El configurador `FlowLayout` es el empleado por defecto en un objeto de tipo `Panel`.

BorderLayout

El configurador `BorderLayout` es el que se emplea por defecto con objetos de la jerarquía `Window`, como `Frame` o `Dialog`. Configura un contenedor situando sus componentes en cinco posiciones predeterminadas. Para que esto suceda, el método `add` debe emplearse con un segundo parámetro cuyo valor debe ser `"North"`, `"South"`, `"East"`, `"West"` o `"Center"`. La Figura D.7 muestra cinco botones añadidos en un `Frame` empleando `BorderLayout`. El código para generar esta configuración se muestra en la Figura D.8. Normalmente, alguna de las cinco posiciones queda vacante. Además, las componentes empleadas suelen ser objetos de tipo `Panel` que contienen otras componentes siguiendo configuraciones diferentes.

Como ejemplo, el código de la Figura D.9 muestra cómo se organizan los objetos de la Figura D.1. Aquí se tienen dos filas, pero deseamos asegurar que las cajas de confirmación, los botones, y las cajas de salida de texto se encuentran debajo del resto de la GUI. Para ello, la idea es crear dos objetos de la clase `Panel`, uno que contiene los elementos de la parte superior y otro que contiene los de la parte inferior. Los paneles se colocan uno encima de otro, configurándolos mediante `BorderLayout`.

Las líneas 4 y 5 crean dos objetos de tipo `Panel`, `topHalf` y `bottomHalf`. Cada uno de estos objetos se configuran independientemente usando `FlowLayout`. Observe que los métodos `setLayout` y `add` se aplican al `Panel` apropiado. Debido a que estos paneles se configuran con `FlowLayout`, pueden consumir más de una fila y no tener suficiente espacio libre disponible. Esto podría causar una separación entre los objetos de tipo `Label` y los de tipo `TextField`. Se deja como ejercicio al lector la creación de objeto de tipo `Panel` adicionales para asegurar que no se producen estas separaciones entre una etiqueta y su componente correspondiente. Una vez que los paneles han sido generados se emplea `BorderLayout` para alinearlos. Esto se hace en las líneas 28 a 30. Observe que las componentes de los dos objetos están centradas. Éste es el resultado de `FlowLayout`. Para alinear a la izquierda el contenido de los paneles, las líneas 8 y 19 podrían construir el objeto `FlowLayout` con el parámetro adicional `FlowLayout.LEFT`.

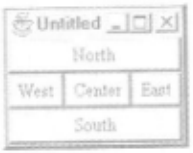


Figura D.7 Cinco botones organizados por `BorderLayout`.

```
1  import java.awt.*;
2
3      // Genera la Figura D.7
4  public class BorderTest extends Frame
5  {
6      public static void main( String [ ] args )
7      {
8          Frame f = new BorderTest( );
9
10         f.setLayout( new BorderLayout( ) );
11         f.add( new Button( "North" ), "North" );
12         f.add( new Button( "East" ), "East" );
13         f.add( new Button( "South" ), "South" );
14         f.add( new Button( "West" ), "West" );
15         f.add( new Button( "Center" ), "Center" );
16
17         f.pack( );      // Reduce el marco al tamaño mínimo
18         f.show( );      // Muestra el marco
19     }
20 }
```

Figura D.8 Código que hace uso de `BorderLayout`.

```
1      // Sitúa todos los objetos
2  private void doTheLayout( )
3  {
4      Panel topHalf = new Panel( );
5      Panel bottomHalf = new Panel( );
6
7          // Coloca la primera mitad
8      topHalf.setLayout( new FlowLayout( ) );
9      topHalf.add( theCanvas );
10     topHalf.add( new Label( "Shape" ) );
11     topHalf.add( theShape );
12     topHalf.add( theColor );
13     topHalf.add( new Label( "X coor" ) );
14     topHalf.add( theXCoor );
15     topHalf.add( new Label( "Y coor" ) x);
16     topHalf.add( theYCoor );
17
18         // Coloca la segunda mitad
19     bottomHalf.setLayout( new FlowLayout( ) );
20     bottomHalf.add( smallPic );
21     bottomHalf.add( mediumPic );
22     bottomHalf.add( largePic );
23     bottomHalf.add( theFillBox );
24     bottomHalf.add( theDrawButton );
25     bottomHalf.add( theMessage );
26
27         // Configuración de la GUI
28     setLayout( new BorderLayout( ) );
29     add( topHalf, "North" );
30     add( bottomHalf, "South" );
31 }
```

Figura D.9 Código que configura los objetos de la Figura D.1.

Cuando se emplea `BorderLayout` y los métodos `add` se utilizan sin cadenas de caracteres, la posición por defecto es `"Center"`. Si se especifica un `String`, pero no es ninguno de los cinco mencionados, se lanza una excepción[6].

El configurador `null`

El *configurador* `null` se emplea para situar los elementos en una posición exacta. En este configurador, cada objeto se incluye en el contenedor empleando `add`. Su tamaño y posición se actualizan mediante una llamada al método `setBounds`:

```
void setBounds( int x, int y, int width, int height )
```

[6] Tenga en cuenta que en Java 1.0, los argumentos de `add` están invertidos, y las cadenas de caracteres inco
ignoradas, lo que dificulta la depuración. Este estilo aún se permite, pero se desaconseja.

Aquí x e y representan la posición del extremo superior izquierdo del objeto con respecto al extremo superior izquierdo de su contenedor. width y height representan el tamaño del objeto. Las unidades empleadas son pixels.

El configurador null depende de la plataforma empleada, y esto normalmente es un gran inconveniente.

Configuradores estéticos

Java también pone a nuestra disposición CardLayout, GridLayout y GridBagLayout. El configurador CardLayout simula la estética de las aplicaciones de Windows, pero tiene una apariencia terrible en el AWT. El configurador GridLayout añade componentes en una malla, pero hace que todas las entradas a dicha malla tengan exactamente el mismo tamaño. Esto hace que en algunas ocasiones las componentes se agranden de manera extraña. Es muy útil cuando esto no puede provocar problemas, por ejemplo en el teclado de una calculadora que consiste en una malla bidimensional de botones. El configurador GridBagLayout incorpora las componentes en una malla, pero permite que un mismo elemento ocupe varias celdas. Su uso es más complicado que el del resto de configuradores.

Herramientas visuales

Algunos productos comerciales, como Symantec Café y Microsoft J++, incluyen herramientas que permiten al programador diseñar la configuración usando un sistema de tipo CAD. La herramienta produce el código Java necesario para construir los objetos y colocarlos de la forma indicada. Normalmente, genera la interfaz empleando el configurador null. Aun utilizando estos sistemas el programador debe escribir la mayoría del código, incluyendo el tratamiento de los eventos; pero, es cierto que evitan el sucio trabajo de calcular la posición precisa de los objetos.

D.3.2 Gráficos

Como ya hemos comentado en la Sección D.2.5, los gráficos se dibujan empleando un objeto de tipo Canvas. Más concretamente, para generar gráficos, el programador debe definir una nueva clase que extienda a Canvas. Esta nueva clase debe contener un constructor (el generado por defecto es inaceptable), sobrescribir el método paint y tener un método que pueda invocarse desde el contenedor del objeto Canvas. El método paint es

```
void paint( Graphics g );
```

Graphics es una clase abstracta que define una gran cantidad de métodos. Algunos de ellos son

```
void drawOval( int x, int y, int width, int height );
void drawRect( int x, int y, int width, int height );
void fillOval( int x, int y, int width, int height );
void fillRect( int x, int y, int width, int height );
void drawLine( int x1, int x2, int y1, int y2 );
void drawString( String str, int x, int y );
void setColor( Color c );
```

En Java, las coordenadas son relativas al extremo superior izquierdo de la componente. Los métodos drawOval, drawRect, fillOval y fillRect dibujan un objeto con los atributos height y width especificados, cuyo extremo superior izquierdo está indicado por los parámetros x e y. drawLine y drawString dibujan líneas y texto, respectivamente. El método setColor se emplea para cambiar el color actual; el nuevo color se emplea en todas las rutinas de dibujo, hasta que es cambiado de nuevo.

La Figura D.10 muestra cómo se implementa el objeto de tipo Canvas de la Figura D.1. La nueva clase GUICanvas extiende a Canvas. Se definen varios atributos privados que describen el estado actual del objeto Canvas. El constructor por defecto de GUICanvas es razonable, luego lo aceptamos.

```
1  import java.awt.*;
2
3  public class GUICanvas extends Canvas
4  {
5      public void setParams( String theShape, String theColor,
6                             int x, int y, int size, boolean fill )
7      {
8          this.theShape = theShape;
9          this.theColor = theColor;
10         xcoor = x; ycoor = y;
11         theSize = size;
12         fillOn = fill;
13         repaint( );
14     }
15
16     public void paint( Graphics g )
17     {
18         if( theColor.equals( "red" ) )
19             g.setColor( Color.red );
20         else if( theColor.equals( "blue" ) )
21             g.setColor( Color.blue );
22         width = 25 * ( theSize + 1 );
23
24         if( theShape.equals( "Square" ) )
25             if( fillOn )
26                 g.fillRect( xcoor, ycoor, width, width );
27             else
28                 g.drawRect( xcoor, ycoor, width, width );
29         else if( theShape.equals( "Circle" ) )
30             if( fillOn )
31                 g.fillOval( xcoor, ycoor, width, width );
32             else
33                 g.drawOval( xcoor, ycoor, width, width );
34     }
35
36     private String theShape = "";
37     private String theColor = "";
38     private int xcoor;
39     private int ycoor;
40     private int theSize; // 0 = pequeño, 1 = mediano, 2 = grande
41     private boolean fillOn;
42     private int width;
43 }
```

Figura D.10 Objeto Canvas básico mostrado en la esquina superior izquierda de la Figura D.1.

```
1  public void update( Graphics g )
2  {
3  paint( g );
4  }
```

Figura D.11 Sobrescritura de `update` para evitar el borrado en un `repaint`.

Los atributos se actualizan con el método `setParams`, definido de modo que el contenedor (esto es, la clase `GUI` que almacena el `GUICanvas`) puede comunicar al `GUICanvas` el estado de sus componentes de entrada. `setParams` se muestra en las líneas 5 a 14. Su última línea es una llamada al método `repaint`.

El método `repaint` invoca al método `update`. Por defecto, `update` vacía la componente y después llama a `paint`. Todo lo que necesitamos entonces es escribir un método `paint` que realice los dibujos tal y como se especifica en los atributos. Como puede verse en las líneas 16 a 34 de su implementación, el método `paint` se limita a llamar a los métodos de la clase `Graphics` descritos anteriormente en este mismo apéndice.

Recuerde que, por defecto, una llamada a `repaint` vacía la componente. En algunas ocasiones, preferiríamos escribir en ella en lugar de borrarla. Para hacerlo, sobrescribimos el método `update` en la nueva clase, tal y como se muestra en la Figura D.11.

Es importante observar que, a menos que se empleen hebras (Sección D.4), la llamada al método `update` no es inmediata y suele retrasarse hasta que se produzca un evento. En tal caso, el código que sigue a `repaint` se ejecuta antes de volver a pintar.

D.3.3 Eventos

Cuando el usuario utiliza el ratón o el teclado, el sistema operativo produce un evento. El sistema inicial de tratamiento de eventos de Java era bastante complejo y ha sido completamente rehecho. El nuevo modelo es mucho más sencillo de programar que el antiguo. Observe que ambos modelos son incompatibles: los eventos de Java 1.1 no son comprensibles para los compiladores de Java 1.0 y viceversa. Las reglas básicas que deben seguirse son las siguientes:

1. Cualquier clase que incluya código para el tratamiento de eventos debe implementar una interfaz *de escucha*. Ejemplos de estas interfaces son `ActionListener`, `WindowListener` y `MouseListener`. Como es habitual, implementar una interfaz supone que todos los métodos de la interfaz deben definirse en la clase.
2. Un objeto que quiere tratar un evento generado por una componente, debe declarar ese posible tratamiento mediante un mensaje de *añadir escucha* enviado a la componente que genera dicho evento. Cuando una componente genera un evento, éste se envía al objeto que ha declarado que lo recibirá. Si ningún objeto ha declarado que lo hará, entonces es ignorado.

Por ejemplo, considere el evento que se genera cuando el usuario pulsa un objeto de la clase `Button`, pulsa la tecla *Return* cuando se encuentra en una `TextField`, o realiza una selección en un objeto de la clase `List` o `MenuItem`. El modo más sencillo de tratar la pulsación de un objeto `Button` es que su contenedor implemente la interfaz `ActionListener`, codificando un método `actionPerformed` y declarando que él es el manejador del evento mencionado.

Esto se hace sobre nuestro ejemplo en la Figura D.1, como sigue. Recuerde que en la Figura D.5 ya hemos hecho dos cosas. En la línea 4, `GUI` declara que implementa la interfaz `ActionListener`, y en la línea 10 una instancia de `GUI` se declara como el manejador del evento de pulsar su objeto `Button`. En la Figura D.12 implementamos la interfaz de escucha invocando a `setParam` desde `actionPerformed` en la clase `GUICanvas`. Este ejemplo se simplifica por el hecho de que existe un único objeto `Button`, así que cuando `actionPerformed` es invocado, sabemos lo que hacer. Si `GUI` contiene varios objetos de tipo `Button` y declara ser el manejador de los eventos producidos por todos ellos, entonces `actionPerformed` debería examinar el parámetro `evt` para determinar qué evento debe ser procesado; esto probablemente suponga ejecutar una secuencia de tests `if/else`[7]. El parámetro `evt`, que en nuestro caso es una referencia de tipo `ActionEvent`, siempre se pasa a un manejador de eventos. El tipo del evento está en función del tipo de su manejador (`ActionEvent`, `WindowEvent` y así sucesivamente), pero será siempre una subclase de `AWTEvent`.

Un evento importante que necesita ser procesado es el evento de cierre de una ventana. Este evento se genera cuando se cierra una aplicación presionando ☒, situado en la esquina superior derecha de la ventana de la aplicación. Desafortunadamente, este evento se ignora por defecto, de modo que si no se indica ningún manejador para él, el mecanismo usual para cerrar una aplicación no funcionará.

El cierre de una ventana es uno de los muchos eventos asociado a `WindowListener`. Debido a que dicho interfaz requiere que implementemos muchos métodos (cuyos cuerpos serán probablemente vacíos), el modo más razonable de proceder es definir una clase que extienda a `Frame` e implemente la interfaz `WindowListener`. Esta clase, `CloseableFrame`, se muestra en la Figura D.13.

```
1      // Trata la pulsación del botón en el dibujo
2  public void actionPerformed( ActionEvent evt )
3  {
4      try
5      {
6          theCanvas.setParams(
7              theShape.getSelectedItem( ),
8              theColor.getSelectedItem( ),
9              Integer.parseInt( theXCoor.getText( ) ),
10             Integer.parseInt( theYCoor.getText( ) ),
11             smallPic.getState( ) ? 0 :
12                     mediumPic.getState( ) ? 1 : 2,
13             theFillBox.getState( ) );
14         theMessage.setText( "" );
15     }
16     catch( Exception e )
17         { theMessage.setText( "Incomplete input" ); }
18 }
```

Figura D.12 Código para tratar la pulsación del botón en el dibujo de la Figura D.1.

[7] Una forma de hacerlo es emplear `evt.getSource( )`, que devuelve una referencia al objeto que ha gener

```
1  import java.awt.*; import java.awt.event.*;
2  // Marco que se cierra cuando se genera un evento de cierre de ventana
3  public class CloseableFrame extends Frame implements WindowListener
4  {
5      public CloseableFrame( )
6        { addWindowListener( this ); }
7
8      public void windowClosing( WindowEvent event )
9        { System.exit( 0 ); }
10     public void windowClosed( WindowEvent event )
11       { }
12     public void windowDeiconified( WindowEvent event )
13       { }
14     public void windowIconified( WindowEvent event )
15       { }
16     public void windowActivated( WindowEvent event )
17       { }
18     public void windowDeactivated( WindowEvent event )
19       { }
20     public void windowOpened( WindowEvent event )
21       { }
22 }
```

Figura D.13 Clase `CloseableFrame`: es similar a `Frame` pero trata el evento de cierre de una ventana.

El manejador del evento de cierre de ventana es sencillo; sólo realiza la llamada `System.exit`. Los métodos restantes no tienen ninguna implementación especial. El constructor declara que puede recibir el evento del cierre de una ventana. En lo sucesivo podemos emplear la clase `CloseableFrame` en lugar de la clase `Frame`.

La Figura D.14 muestra un método `main` que puede emplearse para iniciar la aplicación de la Figura D.1. Lo colocamos en una clase aparte, a la que llamamos `BasicGUI`. `BasicGUI` extiende la clase `CloseableFrame`. El método `main` simplemente crea un objeto `Frame`, en el que colocamos un objeto `GUI`. Como sólo existe un objeto, puede emplearse el configurador `FlowLayout` para el `Frame f`. Después añadimos un objeto de tipo `GUI` sin nombre y aplicamos el método `pack` sobre el objeto `Frame`. La rutina `pack` comprime el `Frame` tanto como sea posible, dadas las componentes que lo forman. El método `show` muestra el `Frame`.

```
1  import java.awt.*;
2
3  public class BasicGUI extends CloseableFrame
4  {
5      public static void main( String [ ] args )
6      {
7          Frame f = new BasicGUI( );
8
9          f.setLayout( new FlowLayout( ) );
10         f.add( new GUI( ) );
11         f.pack( );
12         f.show( );
13     }
14 }
```

Figura D.14 Rutina `main` para la Figura D.1

D.3.4 Resumen: encajando las piezas

He aquí un resumen de cómo crear una aplicación GUI. La funcionalidad de la GUI debe situarse en una clase que extienda a `Panel`. Para esta clase debe hacerse lo siguiente:

- Decidir cuáles son los elementos de entrada y cuáles los de salida. Si los mismos elementos se emplean varias veces, debe construirse una clase adicional para agrupar la funcionalidad común y aplicar estos principios a dicha clase.
- Si se emplean gráficos, debemos construir una nueva clase que extienda a `Canvas`. Esta clase debe implementar un método `paint` e incluir otro método público que pueda ser empleado por el contenedor para comunicarse con ella. También se necesita codificar un método constructor.
- Escoger un configurador y emplear el comando `setLayout`.
- Añadir las componentes a la GUI usando `add`.
- Tratar los eventos. La manera más sencilla de hacerlo es utilizar un objeto de tipo `Button` y capturar su pulsación con `actionPerformed`.

Una vez que la GUI ha sido implementada, una aplicación define una clase que extiende `CloseableFrame` con una rutina `main`. La rutina `main` simplemente crea una instancia de esta clase de marco extendida, coloca el panel de la GUI dentro del marco y emplea sobre él los métodos `pack` y `show`.

D.4 Animaciones y hebras

Debido a que el AWT nos permite dibujar objetos, es natural intentar realizar animaciones. Probablemente, la habilidad de Java a la hora de animar páginas Web ha sido la responsable de su apogeo entre los lenguajes de programación.

La animación en Java puede parecer trivial; todo lo que tenemos que hacer es llamar repetidamente a `repaint` con distintas peticiones. Si separamos las llamadas a `repaint` utilizando `Thread.sleep`, podemos retardar la animación hasta que alcance una velocidad adecuada. Un ejemplo de estas ideas se muestra en la Figura D.15. Aquí, intentamos mostrar cómo un círculo de radio 50 se desplaza a lo largo de la diagonal desde (0,0) hasta (199,199), donde, como es habitual, estas coordenadas representan el extremo superior izquierdo de la caja en la que está inscrito el círculo (las posiciones son relativas al extremo superior izquierdo del marco). Invocamos reiteradamente a `repaint`, separando las llamadas 25 milisegundos.

Desgraciadamente, cuando se ejecuta este programa, se observa una espera de unos 5 segundos, tras lo cual simplemente se dibuja el último círculo. Esto no es lo que queríamos.

La solución a nuestro problema es emplear *hebras* (en inglés *threads*). Cómo emplear en general las hebras es algo complicado, pero los elementos concretos necesarios para resolver nuestro problema son simples. Comenzamos creando una hebra de ejecución independiente. Para hacerlo, realizamos los siguientes pasos:

1. Hacemos que nuestra clase implemente la interfaz `Runnable`. Dicha interfaz tiene un solo método llamado `run`.

```
1  import java.awt.*;
2
3      // Intenta dibujar una secuencia animada de círculos
4      // sobre una diagonal; no funciona
5  public class BadCircles extends CloseableFrame
6  {
7      int extremity = 0;
8
9      public BadCircles( )
10     {
11         drawCircles( );
12     }
13
14     public void drawCircles( )
15     {
16         for( extremity = 0; extremity < 200; extremity++ )
17         {
18             repaint( );
19             try
20               { Thread.sleep( 25 ); }
21             catch( InterruptedException e ) { }
22         }
23     }
24
25     public void paint( Graphics g )
26     {
27         g.fillOval( extremity, extremity, 50, 50 );
28     }
29
30     public static void main( String [ ] args )
31     {
32         Frame f = new BadCircles( );
33         f.setSize( 300, 300 );
34         f.show( );
35     }
36 }
```

Figura D.15 Programa que intenta realizar una animación del movimiento de un círculo sobre una diagonal.

2. Declaramos una hebra, u objeto de la clase `Thread`, como un atributo y lo inicializamos antes de la construcción del objeto.
3. Iniciamos la hebra o `Thread`.
4. Implementamos el método `run` de modo que él haga todo el trabajo.

La Figura D.16 muestra cómo se hace esto. Los cambios hechos en la Figura D.15 se muestran en negrita (el nombre de la clase ha sido cambiado, de modo que ambas versiones están disponibles en Internet como `BadCircles` y `GoodCircles`, respectivamente). En primer lugar, en la línea 3 declaramos que la clase implementa el interfaz `Runnable`. En segundo lugar, en la línea 6 declaramos un atributo de clase `Thread` y lo construimos en la línea 10. En tercer lugar, iniciamos la ejecución de la hebra en la línea 11. Finalmente, en las líneas 14 a 17 implementamos el método `run`, que se limita a llamar al método `drawCircles`.

Como hemos mencionado anteriormente, el tema de las hebras está relacionado con las animaciones. Son muy útiles en la implementación de programas concurrentes, en los cuales existen varias hebras de ejecución. La principal dificultad

```
1  import java.awt.*;
2  public class GoodCircles extends CloseableFrame
3                           implements Runnable
4  {
5      int extremity = 0;
6      private Thread animator = null;
7
8      public GoodCircles( )
9      {
10         animator = new Thread( this );
11         animator.start( );
12     }
13
14     public void run( )
15     {
16         drawCircles( );
17     }
18
19     public void drawCircles( )
20     {
21         for( extremity = 0; extremity < 200; extremity++ )
22         {
23             repaint( );
24             try
25               { Thread.sleep( 25 ); }
26             catch( InterruptedException e ) { }
27         }
28     }
29
30     public void paint( Graphics g )
31     {
32         g.fillOval( extremity, extremity, 50, 50 );
33     }
34
35     public static void main( String [ ] args )
36     {
37         Frame f = new BadCircles( );
38         f.setSize( 300, 300 );
39         f.show( );
40     }
41 }
```

Figura D.16 Implementación correcta de la animación del círculo.

que presentan es la comunicación de las hebras entre sí y el asegurarnos de que comparten de forma consistente los recursos comunes. Por ejemplo, en muchas ocasiones es deseable que una hebra bloquee a las restantes en lo que se refiere a la modificación o incluso al acceso a un recurso compartido, como por ejemplo una variable. Java incluye mecanismos para todo ello a través de la palabra reservada `synchronized`. Puede encontrarse más información acerca de las hebras en la mayoría de las referencias de Java.

D.5 Applets

Un *applet* es un pequeño programa inmerso en otra aplicación. El sistema Java incluye una aplicación denominada *Applet Viewer* que puede utilizarse para ejecutar un applet. Sin embargo, la situación más común es que un applet se descargue vía

Internet y se ejecute localmente por un navegador que soporte Java, tal como *Netscape Navigator* o *Microsoft Internet Explorer*.

Cuando se incluye un applet en una página Web, su código Java se descarga desde el computador servidor siendo ejecutado por el navegador. Esto tiene la ventaja de rebajar la carga en el servidor. Esto se debe a que una vez que se descarga el código Java, todo el trabajo se hace localmente.

Una de las fascinantes direcciones futuras que Java está empezando a explorar tiene que ver con el almacenamiento de software en un sitio Web. Actualmente, los usuarios de un computador personal deben instalar el software en su propia máquina. Esto requiere un cierto espacio en disco, y limita por tanto la cantidad de software diferente que puede estar simultáneamente instalado. Además, existe el problema de mantener la última versión de cada software, que habitualmente incluye la corrección de algunos errores detectados. Con los applets, todo el software se almacenaría en un sitio Web y los usuarios pagarían por cada uso del software. Esto ahorraría espacio en disco, permitiría que hubiese más software disponible y haría más fácil asegurar que siempre se utiliza la última versión disponible. El resultado sería el denominado «computador en la red» y podría ser significativamente más barato que un PC actual. En el momento en que se escribió este libro, el uso más común de Java consista en el diseño de animaciones que adornaban páginas Web que en otro caso serían tristes y estáticas.

Para usar un applet, el usuario debe conocer algo acerca del lenguaje de hipertexto HTML, el lenguaje que entienden los navegadores Web, a fin de que el navegador cargue correctamente el código del applet. Un applet también se implementa de una forma algo diferente a un programa de aplicación. En particular, no se llama a `main`. También, los applets se ejecutan con restricciones adicionales de seguridad. Sin embargo, dejando aparte los temas de la seguridad, las diferencias entre la programación de un applet y un programa de aplicación son relativamente pequeñas, lo que permite escribir programas que sirvan como aplicación y como applet. El resto de esta sección estudia estos temas.

D.5.1 Lenguaje de hipertexto

El *lenguaje de hipertexto HTML* es el lenguaje que entienden los navegadores Web. Un fichero HTML está formado por una secuencia de mandatos de formato y texto, que circula por Internet hasta el navegador. HTML es un híbrido entre los lenguajes de procesamiento de textos *troff* y *LaTeX*. Ya desde el principio, permitía el cambio de colores y fuentes, la importación de imágenes, la creación de enlaces a otras páginas Web y la generación automática de listas numeradas y con puntos. Recientemente, se ha hecho más complejo, incorporando, entre otras cosas, tablas y múltiples marcos.

La Figura D.17 muestra una breve página que contiene enlaces a las transparencias que acompañan este libro. Su código HTML se muestra en la Figura D.18. Los comandos de formato se incluyen dentro de corchetes <>: típicamente, un mandato como `<ul>`, que indica el comienzo de una lista no ordenada (o con puntos), se termina con la marca `</ul>` (observe cómo se ha añadido la barra). También se muestra el mandato `<li>`, que representa una entrada de la lista. Los ficheros HTML deberían empezar con las etiquetas `<HTML>` y `<BODY>` y terminar

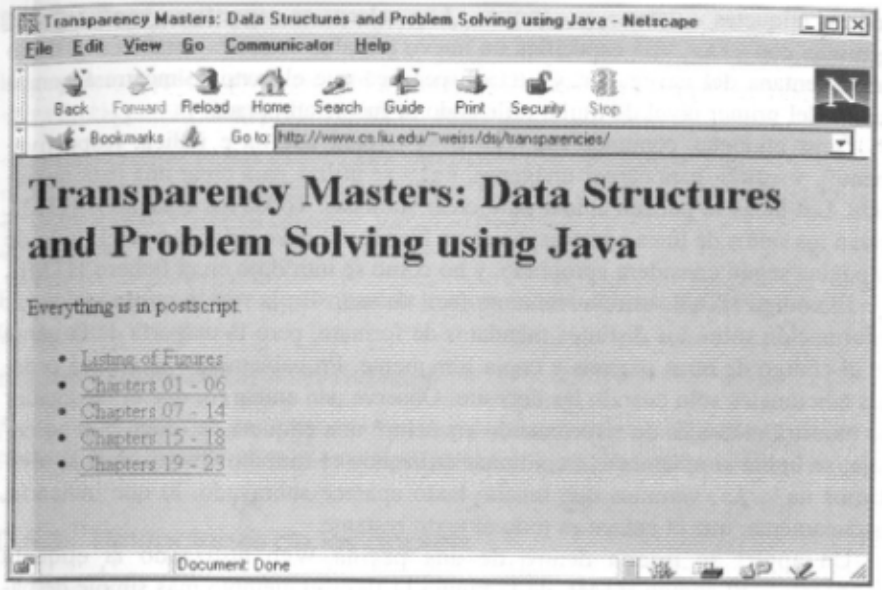


Figura D.17 Página Web, vista a través de Netscape Navigator, correspondiente al código HTML de la Figura D.18.

```
1  <HTML>
2  <BODY>
3
4  <TITLE>
5  Transparency Masters:
6  Data Structures and Problem Solving using Java
7  </TITLE>
8
9  <H1>
10 Transparency Masters:
11 Data Structures and Problem Solving using Java
12 </H1>
13
14 <p>
15 Everything is in postscript.
16
17 <ul>
18 <li><A HREF="TransparencysTOC.ps">Listing of Figures</A></li>
19 <li><A HREF="Transparencys01-06.ps">Chapters 01 - 06</A></li>
20 <li><A HREF="Transparencys07-14.ps">Chapters 07 - 14</A></li>
21 <li><A HREF="Transparencys15-18.ps">Chapters 15 - 18</A></li>
22 <li><A HREF="Transparencys19-23.ps">Chapters 19 - 23</A></li>
23 </ul>
24
25 </BODY>
26 </HTML>
```

Figura D.18 Código HTML para generar la página Web de la Figura D.17, que contiene las transparencias de este libro.

con las etiquetas </HTML> y </BODY>. Los enlaces se especifican con <A> y se terminan con </A>. <p> especifica un nuevo párrafo, <TITLE> establece el título de la ventana del navegador, y <H1> especifica que el texto se mostrará con la fuente del primer nivel de título, utilizando la fuente apropiada.

Otras etiquetas comunes son <B> e <I> (para negrita e itálica, respectivamente), y <SRC> para cargar imágenes. se utiliza para crear una lista numerada. Las listas se pueden anidar para crear sublistas. Todos los comandos HTML tratan los saltos de línea como espacios en blanco. El navegador rompe el título de la página según considera apropiado, y no como se introdujo en el fichero HTML.

El código HTML es relativamente fácil de usar. En la red se puede encontrar información sobre los distintos mandatos de formato, pero la mayoría de la gente ve el código de otras páginas y copia libremente. Probablemente aprenderá ordenes adicionales sólo cuando las necesite. Observe, sin embargo, que el navegador no muestra mensajes de error cuando encuentre una etiqueta de orden que no encaje; se limita simplemente a continuar utilizando el formato abierto. Así, si olvidamos un </A>, veremos que mucho texto aparece subrayado, lo que indicaría, erróneamente, que el enlace es todo el texto restante.

Un applet se inserta dentro de una página Web utilizando la etiqueta <APPLET>[8]. El código HTML de la Figura D.19 es el ejemplo más simple de página Web que contiene un applet. De hecho, esta página no contiene nada más. Observemos que una página Web puede contener otras cosas junto a un applet, así como varios applets. Dentro de la etiqueta <APPLET> se encuentra el nombre del fichero que contiene el código Java y las dimensiones (en pixels) del applet en el navegador.

Si el applet no se encuentra en el mismo lugar que la página Web, se debe proporcionar el atributo CODEBASE (preferentemente antes del atributo CODE). Esto permite referenciar directamente applets escritos por otros.

Observe que la mayoría de los applets estarán formados por varias clases: sólo se especifica la clase con la que comienza el applet. Sin embargo, todas las clases deben estar disponibles y deben poder ser leídas por cualquier usuario. En otro caso, se obtendrá un error de ClassLoader[9]. Si se utilizan paquetes, se debe establecer una jerarquía de directorios. Sin embargo, algunos navegadores no permiten que los applets utilicen paquetes definidos por el usuario. Esto quiere decir que si tiene un código que usa paquetes, necesitará extraer las clases utilizadas de sus paquetes.

```
1 <HTML>
2 <BODY>
3
4 <APPLET code = "BasicGUIApplet.class" width = "600" height = "150">
5 </APPLET>
6
7 </BODY>
8 </HTML>
```

Figura D.19 Fichero HTML para el applet BasicGUIApplet.

para reemplazar esta etiqueta por <EMBED>.
ror, además de escribirse en System.out, se colocan en la consola de Java. La mayoría de los navegadores rmite ver esta consola.

D.5.2 Parámetros

Los applets pueden invocarse con parámetros, utilizando la etiqueta PARAM entre <APPLET> y </APPLET>. Por ejemplo, puede tener

```
<APPLET CODE="Programa.class" WIDTH="150" HEIGHT="150">
<PARAM NAME="InitialColor" VALUE="Blue">
<PARAM NAME="InitialShape" VALUE="Oval">
</APPLET>
```

El applet puede acceder a estos parámetros por medio del método getParameter (de la clase Program):

```
String getParameter( String name );
```

Aquí, la llamada getParameter("InitialColor") devuelve "Blue". Si en la etiqueta PARAM no se proporciona el atributo NAME, se devuelve null.

D.5.3 Limitaciones de los applets

Un applet representa código que se descarga vía Internet y se ejecuta en nuestro computador. El proceso de descarga y ejecución empieza tan pronto como se visita la página Web que contiene el applet. Como resultado, es esencial que exista la garantía de que el applet no intente hacer algo malicioso en el computador local. Algo malicioso puede ser introducir un virus o acceder a información confidencial. Por ejemplo, si un applet tiene la capacidad de escribir en ficheros, sería muy fácil para un hacker elaborar un applet destructivo y borrar discos duros de visitantes confiados.

Como resultado, los applets se ejecutan con severas restricciones y pueden hacer un número significativo de cosas menos que una aplicación. Enumeramos a continuación algunas operaciones que un applet no puede realizar, explicando por qué es necesaria cada restricción. Tenga en cuenta que ésta no es una lista exhaustiva. Estas restricciones no se comprueban en tiempo de compilación. Por el contrario, si un applet intenta realizar cualquiera de estas operaciones, se producirá una excepción en tiempo de ejecución. Algunos navegadores eliminan estas restricciones cuando el applet se carga del sistema local. En consecuencia, al probar su applet, puede encontrar que funciona cuando se ejecuta localmente, pero no cuando se ejecuta en Internet.

1. *Los applets no pueden borrar ficheros del sistema local*. La razón de esta restricción se explicó anteriormente en esta sección.
2. *Los applets no pueden escribir ficheros en el sistema local*. En otro caso, todos los ficheros podrían ser reducidos a tamaño 0 o sobrescritos por un applet maligno.
3. *Los applets no pueden leer ficheros del sistema local*. En otro caso, el applet podría acceder a información potencialmente confidencial y transferirla al servidor.
4. *Los applets no pueden renombrar ficheros del sistema local*. El renombramiento permite la eliminación de ficheros importantes del sistema y el ocultamiento de ficheros por un applet maligno.
5. *Los applets no pueden crear o listar directorios, comprobar la existencia de un fichero, u obtener el tipo, tamaño o fecha de modificación de un*

fichero. Todas estas restricciones se requieren para evitar que un applet maligno obtenga información del sistema.

6. *Los applets no pueden crear una conexión por red con cualquier computador distinto de aquél del cual se obtuvo el applet. En particular, el método* `getURL` *está limitado*. En otro caso, un applet cargado por Internet podría acceder a computadores de una intranet que estén protegidos, obteniendo potencialmente información confidencial.
7. *Los applets no pueden escuchar o aceptar conexiones de otros puertos en el sistema local*. De otra forma el applet podría interpretar información local.
8. *Cuando el applet crea una ventana al nivel más alto, como una caja de diálogo, debe aparecer un mensaje de que la ventana no es fiable*. Esto previene a los programas locales confiados.
9. *Los applets no pueden invocar a programas locales (llamando a* `Runtime.exec`*)*. Esto generaría absurdos agujeros en la seguridad.
10. *Un applet no puede averiguar las propiedades del usuario, tales como el nombre o el directorio principal*. Esto permitiría la transmisión de información privada.
11. *Los applets no pueden llamar a* `System.exit`. Esto provocaría la terminación del navegador.

Aun con estas restricciones, es imposible asegurar el sistema completamente. *Ataques de negación de servicio* son todavía posibles; es decir, un applet Java que simplemente monopolice la mayoría de los recursos del sistema, tales como la memoria o los ciclos de CPU. Además los investigadores encuentran casi todos los meses nuevos agujeros en la seguridad.

Estas restricciones pueden resultar problemáticas. Por ejemplo, ¿cómo se puede conseguir el objetivo de distribuir software por Internet (tales como procesadores de textos) si estos programas no pueden acceder a los recursos fundamentales del sistema local? La respuesta es que ello es ciertamente imposible. En Java 1.1, algunas de estas restricciones pueden evitarse en aquellos applets que han probado ser parte de un código seguro. En otras palabras, el usuario local puede permitir que las applets de Microsoft se ejecuten con libertad. Cuando Microsoft diseña un applet, codifica información que prueba que el applet es de hecho de Microsoft y por tanto se puede confiar en él. Esto se denomina *firmar* un applet, y requiere la tecnología de encriptación de clave pública. Algunas de las ideas de la encriptación de clave pública se estudiaron en la Sección 7.4.4.

D.5.4 Conversión de una aplicación en un applet

En general, es bastante simple convertir una aplicación en un applet, siempre que la aplicación no intente ejecutar ninguna de las acciones prohibidas a los applets. Las modificaciones básicas a realizar son las siguientes:

1. El applet debe importar `java.applet.*`.
2. La clase que define el applet debe extender a la clase `Applet`.
3. No se utiliza la rutina `main`.
4. Se reemplaza el constructor por la rutina `public void init( )`.

Hay otras cosas a tener en cuenta si el applet utiliza hebras para lanzar varios procesos concurrentes. Cuando un usuario abandona una página Web que contiene un applet, se detiene la hebra principal del mismo hasta que el usuario regrese a la página (en cuyo momento se reinicia el applet). Sin embargo, las hebras adicionales no se detienen, por lo que se consumirán ciclos de CPU aunque el applet ya no sea visible. El usuario debe sobrescribir el método `stop` de los applets de forma que llamen al método `stop` de cualquier hebra que se haya lanzado. Se puede usar el método `start` de los applets para crear una nueva hebra y comenzar a ejecutarla, llamando al método `start` de la hebra. En la Sección D.5.5 se habla de los applets animados.

La Figura D.20 muestra un applet que reutiliza la componente GUI que se ha visto a lo largo de este capítulo. Observemos que el applet es muy similar, pero no idéntico, a la aplicación de la Figura D.1. Esto es así, porque los tamaños de las componentes y el aspecto general de cada una de ellas varían de un sistema Java a otro. En la Figura D.21 se muestra el código que genera este applet, el cual se puede usar también para generar la aplicación de la Figura D.1.

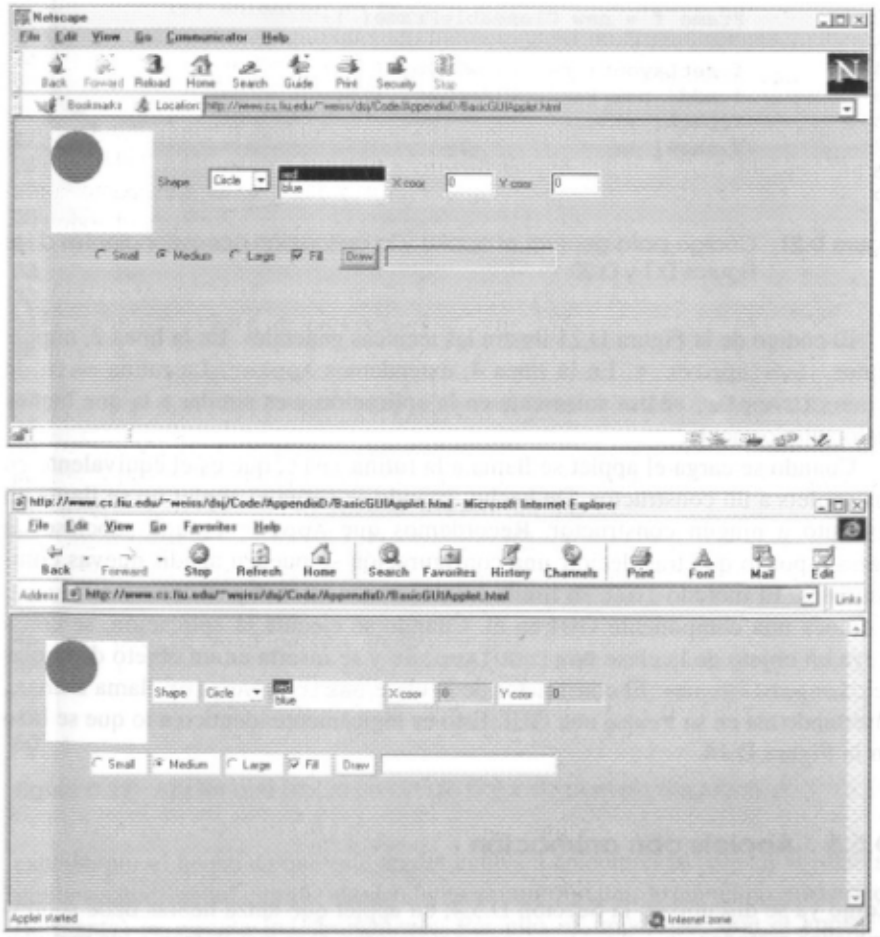


Figura D.20 Visualización del applet de la Figura D.21 en Netscape Navigator y Microsoft Internet Explorer

```
1  import java.awt.*;
2  import java.applet.*;
3
4  public class BasicGUIApplet extends Applet
5  {
6          // El applet comienza aquí
7      public void init( )
8      {
9          setLayout( new FlowLayout( ) );
10         add( new GUI( ) );
11     }
12
13         // Se llama al constructor desde la aplicación
14     public BasicGUIApplet( )
15     {
16         init( );
17     }
18
19         // método main para una aplicación
20     public static void main( String [ ] args )
21     {
22         Frame f = new CloseableFrame( );
23
24         f.setLayout( new FlowLayout( ) );
25         f.add( new BasicGUIApplet( ) );
26         f.pack( );
27         f.show( );
28     }
29 }
```

Figura D.21 Código para generar el applet y la aplicación correspondientes a las Figuras D.1 y D.20.

El código de la Figura D.21 ilustra las técnicas generales. En la línea 2, importamos `java.applet.*`. En la línea 4, extendemos `Applet`. La rutina `main` de `BasicGUIApplet` se usa solamente en la aplicación y es similar a la que hemos visto anteriormente.

Cuando se carga el applet se llama a la rutina `init`, que es el equivalente en los applets a un constructor. De hecho, cuando se carga el applet no se llama en absoluto a ningún constructor. Recordemos que `Applet` es una subclase de `Panel`, por lo que tras decidir una configuración se pueden añadir nuevas componentes. El método `init` se limita a seleccionar el configurador `FlowLayout` y coloca una componente GUI en él. Cuando se ejecuta la aplicación, se construye un objeto de la clase `BasicGUIApplet` y se inserta en un objeto de la clase `CloseableFrame`. El constructor de la clase `BasicGUIApplet` llama a `init`, insertando así en su `Frame` una GUI. Esto es lógicamente idéntico a lo que se hizo en la Figura D.14.

D.5.5 Applets con animación

Como ya se discutió en la Sección D.5.4, un applet que lanza hebras debe sobrescribir los métodos `start` y `stop` con implementaciones que llamen a los métodos `start` y `stop` de las hebras que hayan podido crearse. Ilustramos esta técnica con el applet de la Figura D.22, que es similar a la animación de la Figura D.16,

```
1  import java.awt.*;
2  import java.applet.*;
3
4  // Este applet anima un círculo moviéndolo hacia abajo a lo
5  // largo de una diagonal. Cuando llega al extremo inferior,
6  // comienza de nuevo en el extremo superior.
7
8  public class CircleApplet extends Applet implements Runnable
9  {
10     int extremity = 0;
11     private Thread animator = null;
12
13     public void stop( )
14     {
15         if( animator != null )
16             animator.stop( );
17         animator = null;
18     }
19
20     public void start( )
21     {
22         if( animator == null )
23         {
24             animator = new Thread( this );
25             animator.start( );
26         }
27     }
28
29     public void run( )
30     {
31         drawCircles( );
32     }
33
34     public void drawCircles( )
35     {
36         for( ; ; )
37             for( extremity = 0; extremity < 200; extremity++ )
38             {
39                 repaint( );
40                 try
41                   { Thread.sleep( 25 ); }
42                 catch( InterruptedException e ) { }
43             }
44     }
45
46     public void paint( Graphics g )
47     {
48         g.fillOval( extremity, extremity, 50, 50 );
49     }
50 }
```

Figura D.22 Applet que dibuja un círculo bajando por una diagonal.

excepto por el hecho de que este applet vuelve a colocar el círculo en la parte superior de la diagonal cuando alcanza la parte inferior, con lo que el programa (teóricamente) se ejecuta indefinidamente. Por ello es importante que el applet no consuma recursos una vez que el navegador abandona la página web que contiene dicho applet (recordemos que los applets no pueden llamar a `exit`).

En este applet, tenemos (a lo sumo) una hebra activa referenciada por `animator`. El método `stop` del applet termina esta hebra y asigna `null` a `animator` de forma que se puedan recolectar sus recursos. El método `start` crea una nueva hebra y la inicializa (esto tiene el efecto de llamar a `run`). No hemos sobrescrito `init`, por lo que se usa el método por defecto (que no hace nada). Esto parece correcto, ya que siempre se llama a `start` inmediatamente después de `init`.

Observe que, en este código, cuando abandonamos la página Web del applet y después volvemos, iniciamos una hebra diferente de la que se estaba ejecutando antes. Si no deseamos esto, podemos rescribir `stop` y `start` para suspender la hebra, mediante una llamada a `suspend`, y después continuar con ella, mediante una llamada a `resume` (algunos navegadores tienen errores relacionados con `suspend`).

Resumen

En este apéndice se han examinado las bases del Abstract Window Toolkit (AWT). El AWT es un paquete que permite programar GUI, lo que hace que los programas tengan una apariencia más profesional que si se utilizara una E/S simple por pantalla.

Las aplicaciones GUI difieren de las aplicaciones con E/S por pantalla en que están dirigidas por eventos. Para diseñar una GUI, escribimos una clase. Debemos decidir cuáles van a ser los elementos básicos de entrada y de salida, elegir una configuración de los mismos ejecutando un mandato `setLayout`, añadir componentes a la GUI usando `add`, y tratar los eventos, lo que en Java 1.1 se hace por medio de las escuchas de eventos. Todo ello forma parte de la citada clase.

Una vez hemos escrito esta clase, se define una clase que extiende a la clase `Frame`, que contiene una rutina `main` y un manejador de eventos que se encarga de procesar el evento de cierre de la ventana. La forma más simple de hacer esto es usar la clase `CloseableFrame` de la Figura D.13. La rutina `main` crea una instancia de esta clase `Frame` extendida, coloca una instancia de la clase GUI (cuyo constructor creará con seguridad un panel) dentro del objeto de la clase `Frame`, y ejecuta un mandato `pack` y otro `show` sobre dicho objeto.

Los applets son similares a las aplicaciones, excepto por el hecho de que son ejecutados por otros programas, como los navegadores. Estos últimos suelen ejecutarse con fuertes restricciones de seguridad. Difieren ligeramente en detalles como que no llaman a `main`, sino que usan un método `init` equivalente al constructor.

Aquí se han estudiado solamente las características básicas del AWT, pero hay libros enteros dedicados al tema.

Elementos del juego

Abstract Window Toolkit (AWT) Herramienta GUI que proporcionan todos los sistemas Java. Facilita las clases básicas utilizadas en la creación de interfaces de usuario.

`ActionEvent` Evento generado cuando un usuario presiona una componente `Button`, pulsa la tecla de retorno en una componente `TextField`, o selecciona un elemento de una componente `MenuItem` o `List`. Debería ser tratado por método `actionPerformed` en la clase que implementa la interfaz `ActionListener`.

`actionPerformed` Método usado para tratar los eventos de acción.

`Applet` Clase que debemos extender cuando queremos implementar un applet.

applet Pequeño programa dentro de otra aplicación. Normalmente se carga desde Internet y se ejecuta localmente en un navegador compatible con Java.

ataque de negación de servicio Mecanismo de un applet de Java con malas intenciones que hace fallar a un computador consumiendo la mayor parte de los recursos del sistema, como la memoria o los ciclos de CPU.

`AWTEvent` Objeto que almacena información sobre un evento.

`BorderLayout` Es el configurador gráfico por defecto para los objetos en la jerarquía `Window`. Se usa para configurar un contenedor mediante la colocación de componentes en cinco posiciones distintas (`"North"`, `"South"`, `"East"`, `"West"`, `"Center"`).

`Button` Componente usada para crear un botón etiquetado. Cuando se pulsa el botón, se genera un evento de acción.

`Canvas` Área rectangular de la pantalla sobre la que una aplicación puede dibujar y recibir entradas del usuario en forma de eventos producidos por el teclado y el ratón.

`Component` Clase abstracta que es la superclase de muchos objetos AWT. Representa algo que tiene un tamaño y una posición y que se puede dibujar en la pantalla al igual que puede recibir eventos de entrada.

configurador Objeto que coloca de forma automática las componentes en un contenedor.

configurador `null` Configurador usado para colocar las componentes de forma precisa. Permite trabajar al método `setBounds`.

`Container` Superclase abstracta que representa todas las componentes que pueden contener a otras componentes. Normalmente tiene un configurador asociado.

`Checkbox` Componente que tiene un estado que indica si está activada o no.

`CheckboxGroup` Objeto usado para agrupar una colección de objetos `Checkbox` que garantiza que solamente uno está activado a la vez.

`Choice` Componente usada para seleccionar una única cadena de caracteres de una lista de opciones.

`Dialog` Ventana de alto nivel utilizada para facilitar diálogos.

error `ClassLoader` Error generado si no está disponible una clase necesaria como fichero `.class`.

eventos Son producidos por el sistema operativo en varias situaciones, como en las operaciones de entrada, y se pasan a Java.

`FileDialog` Ventana al más alto nivel utilizada para ofrecer una elección entre todos los ficheros en un directorio.

`FlowLayout` Configurador por defecto de `Panel`. Se usa para configurar un contenedor añadiendo componentes en fila de izquierda a derecha. Cuando no queda suficiente espacio en una fila, se genera una nueva.

`Frame` Ventana de alto nivel con borde y que puede llevar asociado un objeto `MenuBar`.

`getParameter` Método usado para acceder a los parámetros de un applet especificados en la etiqueta `<PARAM>`.

`Graphics` Clase abstracta que define varios métodos que se pueden utilizar para dibujar formas.

hebra Flujo de ejecución separado. Se pueden ejecutar varias hebras simultáneamente, generando así un programa concurrente. Las hebras son necesarias para ejecutar animaciones.

`init` Equivalente en los applets a un constructor. Se le llama cuando se carga un applet.

interfaz `ActionListener` Interfaz usada para tratar eventos de acción. Contiene el método abstracto `actionPerformed`.

interfaz gráfica de usuario (GUI) Alternativa moderna a la E/S por pantalla que permite a un programa comunicarse con el usuario mediante botones, cuadros de selección, líneas de edición, listas de selección, menús y el ratón.

interfaz `WindowListener` Interfaz usada para especificar el tratamiento de los eventos de ventana, como el cierre de una ventana.

`Label` Componente utilizada para etiquetar otras componentes como `Choice`, `List` , `TextField` o `Panel`.

lenguaje de hipertexto (HTML) Lenguaje que comprenden los navegadores Web. Consta de mandatos de texto y formato.

limitaciones de los applets Restricciones que evitan la ejecución de applets con malas intenciones. Entre otras, los applets no pueden acceder a los ficheros locales o a otros computadores distintos de aquél desde el que se ha cargado el applet.

`List` Componente que permite la selección de una lista desplegada de cadenas de caracteres. Puede permitir la selección de uno o varios elementos, pero utiliza más espacio en la pantalla que `Choice`.

`pack` Método usado para compactar un objeto `Frame` hasta conseguir que su tamaño sea lo menor posible, en función de sus componentes.

`paint` Método usado para dibujar en una componente. Normalmente lo sobrescriben las clases que extienden a `Canvas`.

`Panel` Contenedor usado para almacenar una colección de objetos, pero sin bordes.

`repaint` Método usado para volver a dibujar una componente. Por defecto llama a `update`.

`setLayout` Método que asocia un configurador a un contenedor.

`show` Método que hace visible una componente.

`TextArea` Componente que presenta al usuario varias líneas de texto.

`TextField` Componente que presenta al usuario una sola línea de texto.

`update` Método llamado por `repaint`. Por defecto, borra la componente y llama después a `paint`.

`Window` Ventana de alto nivel sin borde.

Errores comunes

1. Para mostrar un objeto `String` de *N* caracteres, necesitamos una componente `TextField` de más de *N* columnas. Esto es consecuencia de un error del AWT.
2. Es un error habitual olvidarse de definir un configurador. Si es así, se utilizará uno por defecto, aunque puede que éste no sea el que deseábamos.
3. El configurador debe aparecer antes de las llamadas a `add`.
4. Es un error habitual aplicar `add` o asociar un configurador al contenedor erróneo. Por ejemplo, en un contenedor que contiene paneles, aplicar el método `add` sin especificar el panel, hará que se aplique al contenedor principal.
5. Si le falta un argumento de tipo `String` al método `add` para el configurador `BorderLayout`, se usa `"Center"` por defecto. Un error habitual es especificarlo de forma incorrecta, como por ejemplo escribiendo `"north"`. Los cinco argumentos válidos son `"North"`, `"South"`, `"East"`, `"West"` y `"Center"`. En Java 1.1, si el `String` es el segundo parámetro, una excepción en tiempo de ejecución capturará el error. Si usamos el estilo antiguo, donde el `String` aparece primero, puede que no se detecte el error.
6. La animación requiere el uso de hebras. Sin ellas, es importante que observemos que la llamada de `repaint` a `update` no es inmediata y que a veces se retrasa hasta que se produce un evento. El código a continuación de la llamada a `repaint` podría ejecutarse antes de la ejecución real de la acción de volver a pintar. Por ello, es una buena idea que `repaint` sea la última instrucción.
7. Se necesita código especial para procesar el evento de cierre de una ventana.
8. Si falta un fichero `.class` se generará un error `ClassLoader`. Este error también se puede producir si el fichero no es accesible debido a un modo de protección incorrecto.
9. Los paquetes no funcionan en algunos navegadores.
10. Los applets se ejecutan con muchas restricciones. Es importante darse cuenta de que algunas de estas restricciones se eliminan cuando se trata de applets que pertenecen al sistema local, por lo que si un applet trabaja localmente, pero falla en Internet, puede ser porque se ha topado con una restricción.
11. Los applets que se ejecutan en el Applet Viewer a veces fallan bajo Netscape, y viceversa, debido a fallos en ambos productos.
12. No se puede cambiar el tamaño de los applets; el tamaño viene determinado por la etiqueta `<APPLET>` del fichero HTML. Es obligatoria la presencia de la especificación del tamaño.
13. Un error típico en HTML es olvidar la etiqueta de cierre correspondiente a la de apertura.

En Internet

Todo el código de este apéndice está disponible en versión original en el directorio **AppendixD**. Los nombres de los ficheros son los siguientes:

BadCircles.java	Muestra la forma incorrecta de hacer animaciones; se describe en la Figura D.15.
BasicGUI.java	La clase principal, mostrada en la Figura D.14, para la aplicación GUI usada en este capítulo.

BasicGUIApplet.java	La clase, mostrada en la Figura D.21, que ejecuta tanto el applet como la aplicación vistos en la Sección D.5.4.
BasicGUIApplet.html	Código HTML, mostrado en la Figura D.19, para el applet de la Sección D.5.4.
BorderTest.java	Ilustración simple del uso de `BorderLayout`, mostrado en la Figura D.8. Usa un `CloseableFrame`.
CircleApplet.java	Applet animado de la Figura D.22. También se proporciona un fichero HTML llamado**CircleApplet.html**.
CloseableFrame.java	Implementa la interfaz `WindowListener`, como se muestra en la Figura D.13.
FileDialogTest.java	Código para mostrar el uso de `FileDialog`, mostrado en la Figura D.3. (El fichero `ListFiles.java` de la Figura 2.11 está duplicado en este directorio.)
GoodCircles.java	Muestra el uso correcto de hacer animaciones, como se describe en la Figura D.16.
GUI.java	La clase GUI usada a lo largo de todo el capítulo y mostrada en las Figuras D.5, D.6, D.9 y D.12.
GUICanvas.java	Extensión de `Canvas` usada en el ejemplo de la GUI y mostrada en la Figura D.10.

Ejercicios

Cuestiones breves

D.1. ¿Qué es una GUI?
D.2. Escriba una lista de varias componentes que puedan utilizarse para introducir datos en una GUI.
D.3. Describa las cuatro ventanas básicas de alto nivel.
D.4. ¿Cuáles son las diferencias entre las componentes `List` y `Choice`?
D.5. ¿Para qué se usa una componente `CheckBox`?
D.6. Explique los pasos necesarios para diseñar una GUI.
D.7. Explique cómo colocan las componentes los configuradores `FlowLayout`, `BorderLayout` y `null`.
D.8. Describa los pasos necesarios para incluir una componente gráfica dentro de un `Panel`.
D.9. ¿Cuál es el comportamiento por defecto cuando se produce un evento? ¿Cómo se puede cambiar dicho comportamiento?
D.10. ¿Qué eventos genera un `ActionEvent`?
D.11. ¿Cómo se trata el evento de cierre de una ventana?
D.12. Explique las diferencias entre escribir un applet y una aplicación.
D.13. Escriba una lista de algunas de las restricciones que se aplican a los applets.

Problemas prácticos

D.14. Escriba un programa de copia de ficheros (Ejercicio 2.10) que use dos objetos `FileDialog` para obtener los nombres de los ficheros.
D.15. Añada una componente `CheckBox` a la clase `GUI` para evitar borrar los dibujos entre operaciones. Necesitará modificar `GUICanvas` en concordancia e implementar un método `update`.
D.16. Se puede escribir el método `paint` para cada componente. Muestre lo que le sucede al applet cuando se pinta un círculo en la clase `GUI`.
D.17. Trate el evento producido por la pulsación de la tecla Enter en el campo de texto correspondiente a la coordenada `y` de la clase `GUI`. Necesitará modificar `actionPerformed` y registrar la existencia de un segundo manejador de eventos.
D.18. Añada (0,0) como valor por defecto para las coordenadas de una figura en la clase `GUI`.
D.19. Añada parámetros a `BasicGUIApplet` para especificar el tamaño del dibujo. Actualmente, es de 100 × 100 pixels.
D.20. La animación del círculo ilustra un problema conocido como *parpadeo*: cuando ejecuta el programa observará un retraso resultante de la llamada a `repaint`, y la animación perderá fluidez. Una forma de reducir el parpadeo consiste en redibujar solamente la parte del dibujo que ha cambiado. (Alternativamente, `repaint` aceptaría cuatro parámetros acotando la zona en que hay que volver a dibujar.) Modifique una de las animaciones del círculo para conseguir esto.

Prácticas de programación

D.21. Escriba un programa que se pueda usar para introducir dos fechas y obtener el número de días entre ellas. Use la clase `Fecha` del Ejercicio 3.12. Su programa debe funcionar como una aplicación y como un applet.
D.22. Escriba un programa que le permita dibujar líneas en un lienzo (`canvas`) usando un ratón. Una pulsación comienza la línea; una segunda pulsación la termina. Se pueden dibujar varias líneas en el lienzo. Para ello debemos extender la clase `Canvas` y tratar los eventos del ratón implementando `MouseListener`. También necesitaremos sobrescribir `update` para evitar borrar el lienzo entre los dibujos de las líneas. Añada finalmente un botón para borrar el lienzo.
D.23. Escriba una aplicación que contenga dos objetos `GUI`. Cuando se produce una acción en uno de los objetos `GUI`, el otro guarda su estado. Al efecto, necesitamos añadir un método `copiarEstado` a la clase `GUI` que copiará los estados de todos los campos de la `GUI` y volverá a dibujar el lienzo.
D.24. Escriba un programa que tenga un único contenedor y un conjunto de diez componentes de entrada cada una de las cuales especifique una forma, un color, unas coordenadas, un tamaño y un objeto `CheckBox` que indique si la componente está activa. Tras ello dibuje la unión de las componentes de entrada en un contenedor. Represente la componente `GUI` de entrada usando una clase con funciones de acceso. El programa principal debería disponer de un vector con esas componentes de entrada más el contenedor.