

# Estructuras de datos en Java

4.<sup>a</sup> edición

Mark Allen Weiss



<http://www.ingelectronico.blogspot.com>

<http://www.ingelectronico.blogspot.com>

# **Estructuras de datos en Java**

Cuarta edición

<http://www.ingelectronico.blogspot.com>

# Estructuras de datos en Java

Cuarta edición

Mark Allen Weiss

*Florida International University*

Traducción

Vuelapluma

PEARSON

<http://www.ingelectronico.blogspot.com>

**Datos de catalogación bibliográfica**

**Estructuras de datos en Java. Cuarta edición**

*Mark Allen Weiss*

PEARSON EDUCACIÓN, S. A. 2013

ISBN: 978-84-155-223-9

Materia: Informática, 004

Formato: 195 x 250 mm Páginas: 1.000

Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra solo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. Díjase a CEDRO (Centro Español de Derechos Reprográficos) si necesita fotocopiar o escanear algún fragmento de esta obra ([www.conflicencia.com](http://www.conflicencia.com); 91 702 19 70 / 93 272 04 47).

Todos los derechos reservados.

© 2013, PEARSON EDUCACIÓN S. A.

Ribera del Loira, 28

28042 Madrid (España)

[www.pearson.es](http://www.pearson.es)

*Authorized translation from the English language edition, entitled DATA STRUCTURES & PROBLEM SOLVING USING JAVA, Fourth Edition by MARK ALLEN WEISS, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2012.*

*All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.*

*SPANISH language edition published by PEARSON EDUCACION S. A., Copyright © 2013.*

ISBN: 978-84-1555-223-9

Depósito Legal: M-2037-2013

**Equipo de edición**

Editor: Miguel Martín-Romo

**Equipo de diseño**

Diseñadora Senior: Elena Jaramillo

Técnico de diseño: Pablo Hoces de la Guardia

**Equipo de producción**

Directora: Marta Illescas

Coordinadora: Tini Cardoso

Diseño de cubierta: Copibook, S. L.

Composición: Vuelapluma

Impreso por:

IMPRESO EN ESPAÑA-PRINTED IN SPAIN

**Nota sobre enlaces a páginas web ajena:** este libro incluye enlaces a sitios web cuya gestión, mantenimiento y control son responsabilidad única y exclusiva de terceros ajenos a PEARSON EDUCACIÓN, S. A. Los enlaces u otras referencias a sitios web se incluyen con finalidad estrictamente informativa y se proporcionan en el estado en que se encuentran en el momento de publicación sin garantías, expresas o implícitas, sobre la información que se proporcione en ellas. Los enlaces no implican el aval de PEARSON EDUCACIÓN, S. A. a tales sitios, páginas web, funcionalidades y sus respectivos contenidos o cualquier asociación con sus administradores. En consecuencia, PEARSON EDUCACIÓN, S. A., no asume responsabilidad alguna por los daños que se puedan derivar de hipotéticas infracciones de los derechos de propiedad intelectual y/o industrial que puedan contener dichos sitios web ni por las pérdidas, delitos o los daños y perjuicios derivados, directa o indirectamente, del uso de tales sitios web y de su información. Al acceder a tales enlaces externos de los sitios web, el usuario estará bajo la protección de datos y políticas de privacidad o prácticas y otros contenidos de tales sitios web y no de PEARSON EDUCACIÓN, S. A.

Este libro ha sido impreso con papel y tintas ecológicos

A David y David



# PREFACIO

Este libro está diseñado para una secuencia de dos semestres académicos dentro de una carrera en Ciencias de la Computación, comenzando por lo que típicamente se conoce como Estructuras de datos y continuando con estructuras de datos avanzadas y análisis de algoritmos. Es apropiado para las secuencias tanto de dos cursos como de tres cursos relativos a temas introductorios, tal como se esbozan en el informe final del proyecto 2001 sobre Curricula en Ciencias de la Computación (CC2001), un proyecto conjunto de la ACM y el IEEE.

El contenido del curso sobre Estructuras de datos ha estado evolucionando durante algún tiempo. Aunque existe un cierto consenso general en lo que respecta a la cobertura de temas, sigue habiendo considerables desacuerdos acerca de los detalles. Uno de los temas generalmente aceptados son los principios de desarrollo software, y especialmente los conceptos de encapsulación y ocultamiento de información. Algorítmicamente, todos los cursos sobre 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. Muchas universidades ofrecen un curso avanzado en el que se cubren temas sobre estructuras de datos, algoritmos y análisis de tiempo de ejecución a un nivel más alto. El material de este texto está diseñado para utilizarlo en ambos tipos de cursos, eliminando así la necesidad de adquirir un segundo libro de texto.

Aunque los debates más apasionados en el campo de las Estructuras de datos se refieren al tema de la elección de un lenguaje de programación, es necesario también realizar algunas otras elecciones fundamentales:

- Si introducir en una fase temprana el diseño orientado a objetos o el diseño basado en objetos.
- El nivel de rigor matemático.
- El equilibrio apropiado entre la implementación de las estructuras de datos y su uso.
- Detalles de programación relacionados con el lenguaje elegido (por ejemplo, si las interfaces GUI deberían emplearse en una fase temprana).

Mi objetivo al escribir este texto era proporcionar una introducción práctica a las estructuras de datos y algoritmos, desde el punto de vista de pensamiento abstracto y de las técnicas de resolución de problemas. He tratado de cubrir todos los detalles importantes concernientes a las estructuras de datos, sus análisis y sus implementaciones Java, al mismo tiempo que evitaba las estructuras de datos que son interesantes desde el punto de vista teórico, pero que no se utilizan ampliamente. Es imposible cubrir en un solo curso todas las distintas estructuras de datos descritas en este texto, incluyendo sus usos y el análisis. Por ello, he diseñado el libro de texto para proporcionar a los profesores un cierto grado de flexibilidad en cuanto a la cobertura de temas. El profesor tendrá que decidir el equilibrio apropiado entre teoría y práctica, y luego seleccionar los temas que mejor encajen con el curso. Como se explica posteriormente en este Prefacio he organizado el texto con el fin de minimizar las dependencias entre los distintos capítulos.

## Resumen de los cambios en la cuarta edición

1. Esta edición proporciona explicaciones adicionales sobre la utilización de las clases (Capítulo 2), la escritura de clases (Capítulo 3) y las interfaces (Capítulo 4).
2. El Capítulo 6 contiene material adicional en el que se analiza el tiempo de ejecución de las listas, el uso de mapas y la utilización de vistas en la API de Colecciones de Java.
3. Se describe la clase `Scanner` y el código proporcionado en diversos puntos del libro hace uso de la clase `Scanner`.
4. El Capítulo 9 describe e implementa el generador lineal de congruencias de 48 bits que forma parte tanto de la librería Java como de muchas librerías C++.
5. El Capítulo 20 contiene nuevo material sobre tablas hash con encadenamiento separado y sobre el método `hashCode` de `String`.
6. Se han hecho numerosas revisiones en el texto que han permitido mejorar la prosa de la edición anterior.
7. En las Partes Uno, Dos y Cuatro se proporcionan muchos nuevos ejercicios.

## Un enfoque original

Mi premisa básica es que las herramientas de desarrollo software en todos los lenguajes incluyen librerías de gran tamaño, junto con estructuras de datos que forman parte de esas librerías. Mi previsión es que terminará habiendo un desplazamiento en el foco de interés de los cursos sobre estructuras de datos, en el que se pasará de la implementación al uso. En este libro, he decidido adoptar un enfoque original, separando las estructuras de datos en su especificación y su subsiguiente implementación, y aprovechando una librería de estructuras de datos ya existente, la API de Colecciones de Java.

Analizamos un subconjunto de la API de Colecciones, que es adecuado para la mayoría de las aplicaciones, en un único capítulo (Capítulo 6) de la Parte Dos. La Parte Dos también cubre las técnicas de análisis básico, la recursión y la ordenación. La Parte Tres contiene diversas aplicaciones que utilizan las estructuras de datos de la API de Colecciones. La implementación de la API de Colecciones no se muestra hasta la Parte Cuatro, una vez que ya se han utilizado las estructuras de datos. Puesto que la API de Colecciones forman parte de Java, los estudiantes pueden diseñar proyectos de gran envergadura casi desde el principio, utilizando componentes software ya existentes.

A pesar del papel fundamental que la API de Colecciones desempeña en este texto, este no es un libro sobre la API de Colecciones, ni tampoco es un tomo dedicado específicamente a estudiar la implementación de la API de Colecciones, sigue siendo un libro que pone el énfasis en las estructuras de datos y en las técnicas básicas de la resolución de problemas. Por supuesto, las técnicas generales empleadas en el diseño de estructuras de datos son aplicables a la implementación de la API de Colecciones, por lo que varios capítulos de la Parte Cuatro incluyen implementaciones de la API de Colecciones. Sin embargo, los profesores pueden seleccionar las implementaciones más simples de la Parte Cuatro donde no se analice el protocolo de la API de Colecciones. El Capítulo 6,

que presenta la API de Colecciones, resulta esencial para comprender el código de la Parte Tres. En el libro, he intentado utilizar únicamente las partes básicas de la API de Colecciones.

Muchos profesores preferirán un enfoque más tradicional en el que se defina, implemente y luego se utilice cada estructura de datos. Puesto que no existe ninguna dependencia entre los materiales de las Partes Tres y Cuatro, puede impartirse fácilmente un curso tradicional utilizando este libro.

## Prerrequisitos

Los estudiantes que utilicen este libro deberían tener conocimientos de un lenguaje de programación orientado a objetos o de un lenguaje de programación procedimental. Se supone un cierto conocimiento de las características básicas, incluyendo los tipos de datos primitivos, los operadores, las estructuras de control, las funciones (métodos) y la entrada y salida (aunque no necesariamente de matrices y clases).

Los estudiantes que hayan seguido un primer curso en el que se empleara C++ o Java pueden encontrarse con que la lectura de los cuatro primeros capítulos es bastante “ligera” en algunos lugares. Sin embargo, otras partes son ciertamente “avanzadas” con detalles Java que pueden no haber sido cubiertos en cursos introductorios.

Los estudiantes que hayan seguido un primer curso en algún otro lenguaje deberían comenzar por el Capítulo 1 e ir progresando lentamente. Si un estudiante quiere utilizar también algún libro de referencia sobre Java, en el Capítulo 1 se proporcionan algunas recomendaciones.

El conocimiento de las matemáticas discretas resulta útil, aunque no es un prerequisito absoluto. Se presentan diversas demostraciones matemáticas, pero las más complejas van precedidas de una breve revisión matemática. Los Capítulos 7 y 19 a 24 requieren un cierto grado de sofisticación matemática. El profesor puede elegir fácilmente saltarse los aspectos matemáticos de las demostraciones, presentando únicamente los resultados. Todas las demostraciones del texto están claramente marcadas y separadas del cuerpo principal del libro.

## Java

Este libro de texto presenta el material utilizando el lenguaje de programación Java. Java es un lenguaje que a menudo se suele examinar en comparación con C++. Java ofrece muchas ventajas, y los programadores ven Java como un lenguaje más seguro, más portable y más fácil de utilizar que C++.

El uso de Java requiere que se tomen ciertas decisiones a la hora de escribir un libro de texto. Algunas de las decisiones tomadas son las siguientes:

1. *El compilador mínimo requerido es Java 5.* Asegúrese de que está utilizando un compilador compatible con Java 5.
2. *No se pone el énfasis en las interfaces GUI.* Aunque las interfaces GUI son una característica muy interesante de Java, parecer ser más un detalle de implementación que un tema fundamental de Estructuras de datos. No utilizamos Swing en el texto, pero como muchos profesores prefieren hacerlo, en el Apéndice B se proporciona una breve introducción a Swing.

3. *No se pone el énfasis en las applets.* Las applets se utilizan en las interfaces GUI. Además, el enfoque del curso es en las estructuras de datos, más que en las características del lenguaje. Los profesores que quieran explicar las applets tendrán que complementar este texto con una referencia de Java.
4. *Se utilizan clases internas.* Las clases internas se usan principalmente en la implementación de la API de Colecciones y pueden ser obviadas por los profesores que así lo deseen.
5. *El concepto de puntero se explica cuando se presentan las variables de referencia.* Java no tiene un tipo de puntero. En lugar de ello, tiene un tipo de referencia. Sin embargo, los punteros han sido tradicionalmente un tema importante en el campo de las Estructuras de datos, por lo que es necesario presentarlo. En este libro se ilustra el concepto de punteros en otros lenguajes a la hora de explicar las variables de referencia.
6. *No se explican las hebras.* Algunos miembros de la comunidad académica informática argumentan que la computación multihebra debería convertirse en un tema central de la secuencia lectiva introductoria a la programación. Aunque es posible que esto llegue a suceder en el futuro, pocos cursos de introducción a la programación analizan este difícil tema.
7. Algunas características de Java 5 no se utilizan, incluyendo,
  - Importaciones estáticas, no utilizadas porque, en mi opinión, hacen que el código sea más difícil de leer.
  - Tipos enumerados, no se utilizan porque había pocos lugares donde declarar tipos enumerados públicos que fueran utilizables por los clientes. En los pocos lugares posibles, no parecía que los tipos enumerados ayudaran a mejorar la legibilidad del código.

## Organización del texto

En este texto, se presenta el lenguaje Java y la programación orientada a objetos (en particular la abstracción) en la Parte Uno. Se explican los tipos primitivos, los tipos de referencia y algunas de las clases y excepciones predefinidas antes de pasar al diseño de clases y a la herencia.

En la Parte Dos se explican los paradigmas O mayúscula y algorítmico, incluyendo la recursión y la aleatorización. Se dedica un capítulo completo al tema de la ordenación y un capítulo separado contiene una descripción de las estructuras de datos básicas. Se utiliza la API de Colecciones para presentar las interfaces y los tiempos de ejecución de las estructuras de datos. En este punto del texto, el profesor puede adoptar varios enfoques para presentar el material restante, incluyendo los dos siguientes:

1. Explicar las implementaciones correspondientes (bien las versiones de la API de Colecciones o las versiones más simples) de la Parte Cuatro, a medida que se describe cada estructura de datos. El profesor puede pedir a los alumnos que amplíen las clases de diversas formas, como se sugiere en los ejercicios.
2. Mostrar cómo se utiliza cada clase de la API de Colecciones y cubrir la implementación en un momento posterior del curso. Los casos de estudio de la Parte Tres pueden utilizarse como ayuda para adoptar este enfoque. Como hay disponibles implementaciones completas en todo compilador Java moderno, el profesor puede utilizar la API de Colecciones en los

proyectos de programación. Más adelante proporcionaré más detalles sobre la utilización de este enfoque.

La Parte Cinco describe estructuras de datos avanzadas tales como los árboles splay, los montículos de emparejamiento y la estructura de datos para conjuntos disjuntos, que se pueden cubrir, si el tiempo lo permite o, más probablemente, en un curso posterior.

## Organización del texto capítulo a capítulo

La Parte Uno consta de cuatro capítulos que describen los aspectos básicos de Java utilizados a lo largo del libro. El Capítulo 1 describe los tipos primitivos e ilustra cómo escribir programas básicos en Java. El Capítulo 2 habla de los tipos de referencia e ilustra el concepto general de puntero, aun cuando Java no tiene punteros, para que los estudiantes puedan aprender acerca de este importante tema sobre Estructuras de datos. Se ilustran varios de los tipos de referencia básicos (cadenas de caracteres, matrices, archivos y objetos Scanner) y se analiza también el uso de excepciones. El Capítulo 3 continúa esta explicación describiendo cómo se implementa una clase. El Capítulo 4 ilustra el uso de la herencia en el diseño de jerarquías (incluyendo clases de excepciones y de E/S) y componentes genéricos. En la Parte Uno se puede encontrar el material relativo a los patrones de diseño, incluyendo el patrón envoltorio, el adaptador y el decorador.

La Parte Dos se centra en los algoritmos y bloques componentes básicos. En el Capítulo 5 se proporciona una explicación completa del tema de la complejidad temporal y de la notación O mayúscula. También se explica y analiza la búsqueda binaria. El Capítulo 6 es crucial, porque cubre la API de Colecciones y argumenta intuitivamente cuál debería ser el tiempo de ejecución de las operaciones soportadas para cada estructura de datos. (La implementación de estas estructuras de datos, tanto en el estilo de la API de Colecciones como en versión simplificada no se proporciona hasta la Parte Cuatro). Este capítulo también introduce el patrón iterador, las clases anidadas, locales y anónimas. Las clases internas se dejan para la Parte Cuatro, donde se explican como técnica de implementación. El Capítulo 7 describe la recursión, introduciendo primero el concepto de demostración por inducción. También explica las técnicas de divide y vencerás, programación dinámica y retroceso. Una sección describe varios algoritmos numéricos recursivos que se emplean para implementar el criptosistema RSA. Para muchos estudiantes, el material de la segunda mitad del Capítulo 7 resulta más adecuado para un curso posterior. El Capítulo 8 describe, codifica y analiza varios algoritmos de ordenación básicos, incluyendo la ordenación por inserción, la ordenación Shellsort, la ordenación por mezcla y la ordenación rápida, así como la ordenación indirecta. También demuestra la cota inferior clásica para la ordenación y explica los problemas de selección relacionados. Finalmente, el Capítulo 9 es un corto capítulo en el que se explican los números aleatorios, incluyendo su generación y uso en algoritmos aleatorizados.

La Parte Tres proporciona varios casos de estudio, y cada capítulo está organizado alrededor de un tema general. El Capítulo 10 ilustra varias técnicas importantes examinando el tema de los juegos. El Capítulo 11 explica el uso de pilas en lenguajes de computadora, examinando un algoritmo para comprobar el equilibrado de símbolos y el algoritmo clásico de análisis de precedencia de operadores. Para ambos algoritmos se proporcionan implementaciones completas con su código. El Capítulo 12 analiza las utilidades básicas de compresión de archivos y de generación de referencias cruzadas y proporciona una implementación completa de ambas. El Capítulo 13 examina de

manera amplia el tema de la simulación examinando un problema que puede contemplarse como una simulación y luego echando un vistazo a las simulaciones más clásicas dirigidas por sucesos. Finalmente, el Capítulo 14 ilustra cómo se utilizan las estructuras de datos para implementar de manera eficiente varios algoritmos del camino más corto para grafos.

La Parte Cuatro presenta las implementaciones de las estructuras de datos. El Capítulo 15 explica las clases internas como una técnica de implementación e ilustra su uso en la implementación de `ArrayList`. En los restantes capítulos de la Parte Cuatro, se proporcionan implementaciones que utilizan protocolos simples (variaciones de `insert`, `find`, `remove`). En algunos casos, se presentan implementaciones de la API de Colecciones que tienden a utilizar sintaxis Java más complicada (además de ser ellas mismas complejas, debido a su conjunto tan amplio de operaciones requeridas). En esta parte se usan algunos conceptos matemáticos, especialmente en los Capítulos 19 a 21, y el profesor puede decidir si quiere saltarse dichos temas. El Capítulo 16 proporciona implementaciones tanto para pilas como para colas. En primer lugar, se implementan estas estructuras de datos utilizando una matriz ampliable y luego se implementan empleando listas enlazadas. Al final del capítulo se explican las versiones de la API de Colecciones. Las listas enlazadas de carácter general se describen en el Capítulo 17. Las listas simplemente enlazadas se ilustran con un protocolo simple, y al final del capítulo se proporciona la versión más compleja de la API de Colecciones que utiliza listas doblemente enlazadas. El Capítulo 18 describe los árboles e ilustra los esquemas de recorrido básicos. El Capítulo 19 es un capítulo detallado que proporciona varias implementaciones de árboles de búsqueda binaria. Inicialmente se muestra el árbol de búsqueda binaria básico y luego se desarrolla un árbol de búsqueda binaria que soporta estadísticas de orden. Los árboles AVL se explican pero no se implementan, aunque sí que se implementan los árboles rojo-negro y los árboles AA que son más prácticos. Después, se implementan las estructuras `TreeSet` y `TreeMap` de la API de Colecciones. Por último, se examina el árbol-B. El Capítulo 20 explica las tablas hash e implementa el esquema de sondeo cuadrático como parte de `HashSet` y `HashMap`, después de examinar una alternativa más simple. El Capítulo 21 describe el montículo binario y examina la ordenación `heapsort` y la ordenación externa.

La Parte Cinco contiene material adecuado para su uso en un curso más avanzado o como referencia general. Los algoritmos son accesibles incluso en un nivel de primer año. Sin embargo, en aras de la exhaustividad, se han incluido análisis matemáticos sofisticados que caen, casi con total seguridad, fuera del alcance de un estudiante de primer año. El Capítulo 22 describe el árbol *splay*, que es un árbol de búsqueda binaria que parece comportarse de manera extremadamente adecuada en la práctica, y que puede competir con el montículo binario en algunas aplicaciones que requieren colas con prioridad. El Capítulo 23 describe las colas con prioridad que soportan operaciones de mezcla y proporciona una implementación del montículo de emparejamiento. Por último, el Capítulo 24 examina la estructura de datos clásica para conjuntos disjuntos.

Los apéndices contienen material de referencia adicional de Java. El Apéndice A enumera los operadores y su precedencia. El Apéndice B proporciona información sobre Swing y el Apéndice C describe los operadores bit a bit utilizados en el Capítulo 12.

## Dependencias de los capítulos

Generalmente, la mayoría de los capítulos son independientes entre sí. Sin embargo, he aquí las dependencias más notables.

- *Parte Uno (Introducción a Java)*: los primeros cuatro capítulos tienen que cubrirse en su totalidad y en el orden indicado, antes de continuar con el resto del libro.
- *Capítulo 5 (Análisis de algoritmos)*: este capítulo se debe cubrir antes de los Capítulos 6 y 8. La recursión (Capítulo 7) puede verse antes de este capítulo, pero el profesor tendrá que complementar las explicaciones con algunos detalles acerca de cómo evitar una recursión ineficiente.
- *Capítulo 6 (La API de Colecciones)*: este capítulo se puede cubrir antes del material de las Partes Tres o Cuatro o en conjunción con dicho material.
- *Capítulo 7 (Recursión)*: el material de las Secciones 7.1 a 7.3 debe cubrirse antes de hablar de los algoritmos recursivos de ordenación, de los árboles, del caso de estudio del juego de tres en raya y de los algoritmos de determinación del camino más corto. Por otro lado, los materiales relacionados con el criptosistema RSA, la programación dinámica y las técnicas de retroceso (a menos que se explique el juego de tres en raya) son opcionales.
- *Capítulo 8 (Algoritmos de ordenación)*: este capítulo debe seguir a los Capítulos 5 y 7. Sin embargo, es posible tratar el tema de la ordenación Shellsort sin los Capítulos 5 y 7. Shellsort no es recursiva (por tanto no es necesario el Capítulo 7) y un análisis riguroso de su tiempo de ejecución es demasiado complejo y no se ha incluido en el libro (por lo que existe escasa necesidad del Capítulo 5).
- *Capítulo 15 (Clases internas e implementación de ArrayList)*: este material debe preceder a las explicaciones acerca de las implementaciones de la API de Colecciones.
- *Capítulos 16 y 17 (Pilas y colas/ Listas enlazadas)*: estos capítulos pueden abordarse en cualquier orden. Sin embargo, yo prefiero cubrir primero el Capítulo 16 porque creo que presenta un ejemplo más simple de lo que son las listas enlazadas.
- *Capítulos 18 y 19 (Árboles/ Árboles de búsqueda binaria)*: estos capítulos se pueden tratar en cualquier orden o simultáneamente.

## Entidades separadas

Los restantes capítulos tienen pocas o ninguna dependencia:

- *Capítulo 9 (Aleatorización)*: el material sobre números aleatorios se puede cubrir en cualquier punto, según sea necesario.
- *Parte Tres (Aplicaciones)*: los Capítulos 10 a 14 se pueden cubrir en conjunción con la API de Colecciones (en el Capítulo 6) o después de ella, y básicamente en cualquier orden que se desee. Hay pocas referencias a los capítulos anteriores. Estas incluyen la Sección 10.2 (el juego de la tres en raya), que se refiere a una explicación contenida en la Sección 7.7 y la Sección 12.2 (generación de referencias cruzadas), que hace referencia a un código similar de análisis léxico en la Sección 11.1 (comprobación del equilibrado de símbolos).
- *Capítulos 20 y 21 (Tablas hash/Una cola con prioridad: el montículo binario)*: estos capítulos se pueden abordar en cualquier punto.
- *Parte Cinco (Estructuras de datos avanzadas)*: el material de los Capítulos 22 a 24 es autocontenido y se suele cubrir en un curso posterior.

## Matemáticas

He tratado de proporcionar el necesario rigor matemático para los cursos sobre Estructuras de datos que enfatizan la teoría y para los cursos posteriores que requieren un mayor grado de análisis. Sin embargo, este material destaca del texto principal en forma de teoremas separados y, en algunos casos, secciones o subsecciones separadas. Por tanto, los profesores que imparten cursos en los que no haya tanto énfasis teórico pueden saltarse este material.

En todos los casos, la demostración de un teorema no es necesaria para comprender el significado del teorema. Esta es otra ilustración de la separación de una interfaz (el enunciado del teorema) de su implementación (la demostración). Parte del material inherentemente matemático, como la Sección 7.4 (aplicaciones numéricas de la recursión), puede ser obviado sin que eso afecte a la compresión del resto del capítulo.

## Organización del curso

Una cuestión crucial a la hora de impartir el curso es decidir cómo utilizar el material de las Partes Dos a Cuatro. El contenido de la Parte Uno debe ser cubierto en profundidad y el estudiante debería escribir uno o dos programas que ilustren el diseño, la implementación y prueba de clases y de clases genéricas y quizás también el diseño orientado a objetos utilizando la herencia. El Capítulo 5 explica la notación O mayúscula. Puede proporcionarse al estudiante un ejercicio en el que escriba un corto programa y compare el tiempo de ejecución con su correspondiente análisis, con el fin de comprobar su compresión del tema.

Cuando se emplea la técnica de separar el uso de las clases de su implementación, el concepto clave del Capítulo 6 es que las diferentes estructuras de datos soportan diferentes esquemas de acceso y con una eficiencia distinta. Se puede emplear cualquier caso de estudio (excepto el ejemplo del juego de las tres en raya, que utiliza recursión) para explicar las aplicaciones de las estructuras de datos. De esta forma, el estudiante puede ver la estructura de datos y saber cómo se utiliza, aunque no cómo se implementa de manera eficiente. Esto es una auténtica separación entre la implementación y el uso de las clases. Ver las cosas de esta forma permitirá mejorar enormemente la capacidad de los estudiantes para pensar de forma abstracta. Los estudiantes también pueden proporcionar implementaciones simples de algunos de los componentes de la API de Colecciones (se proporcionan algunas sugerencias en los ejercicios del Capítulo 6) y ver la diferencia existente entre las implementaciones eficientes de estructuras de datos en la API de Colecciones existente y las implementaciones ineficientes de esas mismas estructuras de datos que ellos desarrollarán. También se puede pedir a los estudiantes que amplíen el caso de estudio, pero de nuevo, no hace falta que conozcan nada acerca de los detalles de las estructuras de datos.

La implementación eficiente de las estructuras de datos se puede explicar posteriormente, y el tema de la recursión puede introducirse cuando el profesor crea que es apropiado, siempre y cuando lo haga antes de tocar el tema de los árboles de búsqueda binaria. Los detalles relativos a los algoritmos de ordenación se pueden explicar en cualquier momento posterior a la recursión. En este punto, el curso puede continuar empleando los mismos casos de estudio y experimentando con modificaciones en las implementaciones de las estructuras de datos. Por ejemplo, el estudiante puede experimentar con diversos tipos de árboles de búsqueda binaria equilibrados.

Los profesores que opten por un enfoque más tradicional pueden simplemente analizar un caso de estudio en la Parte Tres después de explicar una implementación de una estructura de datos en la Parte Cuatro. De nuevo, los capítulos de este libro están diseñados para ser lo más independientes posible entre sí.

## Ejercicios

Se presentan ejercicios de varios tipos; en concreto, he proporcionado cuatro variedades.

Los ejercicios básicos *En resumen* plantean una pregunta simple o requieren simulaciones a mano de un algoritmo descrito en el texto. La sección *En teoría* plantea cuestiones que requieren un análisis matemático o que piden soluciones interesantes, desde el punto vista teórico a los problemas. La sección *En la práctica* contiene cuestiones simples de programación, incluyendo cuestiones acerca de la sintaxis o acerca de líneas particularmente complejas de código. Finalmente, la sección *Proyectos de programación* contiene ideas para la asignación de trabajos de mayor envergadura.



## Características pedagógicas

- Se utilizan notas al margen para resaltar los temas importantes.
- La sección *Conceptos clave* enumera los términos importantes proporcionando su definición y las correspondientes referencias de página.
- La sección *Errores comunes* al final de cada capítulo proporciona una lista de errores que suelen cometerse.
- Al final de la mayoría de los capítulos se proporcionan referencias a lecturas adicionales.



## Suplementos

Hay disponibles diversos materiales complementarios para este texto. Los siguientes recursos están disponibles en <http://www.aw.com/cssupport> para todos los lectores de este libro de texto:

- *Archivos de código fuente del libro.* (La sección *Internet* al final de cada capítulo enumera los nombres de archivo correspondientes al código del capítulo.)



Además, los siguientes suplementos están disponibles para los profesores. Para acceder a ellos, visite <http://www.pearsonhighered.com/cs> y busque en nuestro catálogo el título *Data Structures and Problem Solving Using Java*. Una vez en la página del catálogo del libro, seleccione el enlace a Instructor Resources (recursos del profesor).

- *Diapositivas PowerPoint* de todas las figuras del libro.
- *Instructor's Guide* (guía del profesor) que ilustra diversos enfoques sobre el material. Incluye ejemplos de preguntas de examen, asignaciones de tareas y planes de estudio. También se proporcionan las respuestas a una serie de ejercicios seleccionados.

## Agradecimientos

Son muchas las personas que me han ayudado en la preparación de este libro. A muchas de ellas ya las he dado las gracias en la edición anterior y en la versión relacionada sobre C++. Otras, demasiado numerosas para mencionarlas a todas, han enviado mensajes de correo electrónico y me han señalado errores e incoherencias en las explicaciones, las cuales he tratado de corregir en esta edición.

Para esta edición me gustaría dar las gracias a mi editor Michael Hirsch, a la ayudante editorial Stephanie Sellinger, a la supervisora senior de producción Marilyn Lloyd y a la jefa de proyecto Rebecca Lazure y a su equipo en Laserwords. Gracias también a Allison Michael y Erin Davis de marketing y a Elena Sidorova y Suzanne Heiser de Night & Day Design por la maravillosa cubierta del libro.

Parte del material de este texto está adaptado de mi libro *Efficient C Programming: A Practical Approach* (Prentice Hall, 1995) y está utilizado con permiso del editor. He incluido referencias al final de los capítulos allí donde ha sido apropiado.

Mi página web, <http://www.cs.fiu.edu/~weiss>, contendrá código fuente actualizado, una lista de erratas y un enlace para recibir informes sobre errores.

*M. A. W.  
Miami, Florida*

# CONTENIDO

## parte uno    Introducción a Java

Capítulo 1	Estructura primitiva del lenguaje Java	3
1.1	El entorno general.....	3
1.2	El primer programa .....	4
1.2.1	Comentarios.....	5
1.2.2	ma i n.....	5
1.2.2	Salida a través de terminal.....	6
1.3	Tipos primitivos .....	6
1.3.1	Los tipos primitivos.....	6
1.3.2	Constantes.....	6
1.3.3	Declaración e inicialización de tipos primitivos.....	7
1.3.4	Entrada y salida a través de 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 .....	17
1.5.9	El operador condicional.....	17
1.6	Métodos .....	17
1.6.1	Nombres de métodos sobrecargados .....	19
1.6.2	Clases de almacenamiento.....	20
Resumen .....	20	
Conceptos clave .....	20	
Errores comunes .....	22	
Internet .....	23	

Ejercicios.....	23
Referencias.....	25

**Capítulo 2****Tipos de referencia****27**

<b>2.1 ¿Qué es una referencia? .....</b>	<b>27</b>
<b>2.2 Conceptos básicos sobre objetos y referencias .....</b>	<b>29</b>
2.2.1 El operador punto (.) .....	29
2.2.2 Declaración de objetos.....	30
2.2.3 Recolección de basura.....	31
2.2.4 El significado de =.....	31
2.2.5 Paso de parámetros.....	32
2.2.6 El significado de ==.....	33
2.2.7 No hay sobrecarga de operadores para los objetos .....	34
<b>2.3 Cadenas .....</b>	<b>34</b>
2.3.1 Conceptos básicos sobre manipulación de cadenas .....	34
2.3.2 Concatenación de cadenas.....	35
2.3.3 Comparación de cadenas .....	35
2.3.4 Otros métodos String.....	36
2.3.5 Conversión de otros tipos a cadenas de caracteres .....	36
<b>2.4 Matrices.....</b>	<b>37</b>
2.4.1 Declaración, asignación y métodos .....	37
2.4.2 Expansión dinámica de matrices .....	40
2.4.3 ArrayList.....	41
2.4.4 Matrices multidimensionales .....	43
2.4.5 Argumentos de la línea de comandos.....	46
2.4.6 Bucle for avanzado .....	46
<b>2.5 Tratamiento de excepciones .....</b>	<b>47</b>
2.5.1 Procesamiento de excepciones .....	47
2.5.2 La cláusula finally .....	48
2.5.3 Excepciones comunes .....	49
2.5.4 Las cláusulas throw y throws .....	50
<b>2.6 Entrada y salida.....</b>	<b>51</b>
2.6.1 Operaciones básicas de flujos .....	52
2.6.2 El tipo Scanner .....	52
2.6.3 Archivos secuenciales.....	56
Resumen .....	59
Conceptos clave .....	59
Errores comunes .....	61
Internet .....	61
Ejercicios.....	62
Referencias .....	67

**Capítulo 3      Objetos y clases** 69

3.1	¿Qué es la programación orientada a objetos? .....	69
3.2	Un ejemplo simple .....	71
3.3	javadoc.....	73
3.4	Métodos básicos .....	74
3.4.1	Constructores.....	76
3.4.2	Mutadores y accesores .....	76
3.4.3	Salida de información y el método <code>toString</code> .....	78
3.4.4	<code>equals</code> .....	78
3.4.5	<code>main</code> .....	78
3.5	Ejemplo: utilización de <code>java.math.BigInteger</code> .....	78
3.6	Constructores adicionales .....	80
3.6.1	La referencia <code>this</code> .....	81
3.6.2	La abreviatura <code>this</code> para constructores.....	82
3.6.3	El operador <code>instanceof</code> .....	82
3.6.4	Miembros de instancia y miembros estáticos .....	82
3.6.5	Campos y métodos estáticos .....	82
3.6.6	Inicializadores estáticos .....	85
3.7	Ejemplo: implementación de una clase <code>BigRational</code> .....	86
3.8	Paquetes .....	90
3.8.1	La directiva <code>import</code> .....	91
3.8.2	La instrucción <code>package</code> .....	92
3.8.3	La variable de entorno <code>CLASSPATH</code> .....	93
3.8.4	Reglas de visibilidad de paquete .....	94
3.9	Un patrón de diseño: compuesto (par) .....	94
	Resumen .....	95
	Conceptos clave .....	97
	Errores comunes .....	99
	Internet .....	100
	Ejercicios .....	100
	Referencias .....	106

**Capítulo 4      Herencia** 107

4.1	¿Qué es la herencia? .....	107
4.1.1	Creación de nuevas clases .....	108
4.1.2	Compatibilidad de tipos .....	113
4.1.3	Despacho dinámico y polimorfismo .....	114
4.1.4	Jerarquías de herencia .....	115

4.1.5	Reglas de visibilidad .....	115
4.1.6	El constructor y super .....	116
4.1.7	Clases y métodos final .....	117
4.1.8	Sustitución de un método .....	119
4.1.9	Un nuevo análisis de la compatibilidad de tipos .....	119
4.1.10	Compatibilidad de tipos matriciales .....	122
4.1.11	Tipos de retorno covariantes .....	122
<b>4.2</b>	<b>Diseño de jerarquías .....</b>	<b>123</b>
4.2.1	Clases y métodos abstractos .....	124
4.2.2	Diseño pensando en el futuro .....	128
<b>4.3</b>	<b>Herencia múltiple .....</b>	<b>129</b>
<b>4.4</b>	<b>La interfaz .....</b>	<b>132</b>
4.4.1	Especificación de una interfaz .....	132
4.4.2	Implementación de una interfaz .....	132
4.4.3	Interfaces múltiples .....	133
4.4.4	Las interfaces son clases abstractas .....	134
<b>4.5</b>	<b>Herencia fundamental en Java .....</b>	<b>134</b>
4.5.1	La clase Object .....	134
4.5.2	La jerarquía de excepciones .....	135
4.5.3	E/S: el patrón decorador .....	136
<b>4.6</b>	<b>Implementación de componentes genéricos mediante la herencia .....</b>	<b>140</b>
4.6.1	Utilización de Object para la programación genérica .....	140
4.6.2	Envoltorios para tipos primitivos .....	141
4.6.3	Autoboxing/unboxing .....	143
4.6.4	Adaptadores: modificación de una interfaz .....	144
4.6.5	Utilización de tipos de interfaz para la programación genérica .....	145
<b>4.7</b>	<b>Implementación de componentes genéricos con los componentes genéricos de Java 5 .....</b>	<b>147</b>
4.7.1	Interfaces y clases genéricas simples .....	148
4.7.2	Comodines con límites .....	148
4.7.3	Métodos estáticos genéricos .....	150
4.7.4	Límites de tipo .....	151
4.7.5	Borrado de tipos .....	152
4.7.6	Restricciones a los genéricos .....	152
<b>4.8</b>	<b>El functor (objetos función) .....</b>	<b>155</b>
4.8.1	Clases anidadas .....	158
4.8.2	Clases locales .....	160
4.8.3	Clases anónimas .....	161
4.8.4	Clases anidadas y genéricos .....	162
<b>4.9</b>	<b>Detalles sobre el mecanismo de despacho dinámico .....</b>	<b>163</b>

Resumen .....	166
Conceptos clave .....	167
Errores comunes .....	169
Internet .....	170
Ejercicios .....	171
Referencias .....	181

## parte dos      Algoritmos y bloques fundamentales

<b>Capítulo 5      Análisis de algoritmos</b>	<b>185</b>
---	------------

5.1    ¿Qué es el análisis de algoritmos? .....	185
5.2    Ejemplos de tiempos de ejecución de diversos algoritmos .....	189
5.3    El problema de la suma máxima de una subsecuencia contigua .....	191
5.3.1    El algoritmo obvio $O(N^3)$ .....	191
5.3.2    Un algoritmo $O(N^2)$ mejorado .....	194
5.3.3    Un algoritmo lineal .....	195
5.4    Reglas generales para el cálculo de cotas O mayúscula.....	198
5.5    El logaritmo .....	202
5.6    Problema de la búsqueda estática .....	204
5.6.1    Búsqueda secuencial .....	205
5.6.2    Búsqueda binaria .....	205
5.6.3    Búsqueda por interpolación .....	207
5.7    Comprobación del análisis de un algoritmo .....	209
5.8    Limitaciones del análisis O mayúscula .....	211
Resumen .....	211
Conceptos clave .....	212
Errores comunes .....	213
Internet .....	213
Ejercicios .....	214
Referencias .....	223

<b>Capítulo 6      La API de Colecciones</b>	<b>225</b>
--	------------

6.1    Introducción .....	225
6.2    El patrón iterador .....	227
6.2.1    Diseño básico de un iterador .....	228
6.2.2    Factorías e iteradores basados en herencia .....	230

6.3	<b>La API de Colecciones: contenedores e iteradores .....</b>	232
6.3.1	La interfaz Collection.....	232
6.3.2	La interfaz Iterator .....	236
6.4	<b>Algoritmos genéricos .....</b>	238
6.4.1	Objetos función Comparator.....	238
6.4.2	La clase Collections .....	239
6.4.3	Búsqueda binaria .....	242
6.4.4	Ordenación.....	242
6.5	<b>La Interfaz List.....</b>	244
6.5.1	La interfaz ListIterator .....	244
6.5.2	La clase LinkedList .....	247
6.5.3	Tiempo de ejecución para las distintas listas.....	249
6.5.4	Eliminación y adición de elementos en mitad de una colección List.....	251
6.6	<b>Pilas y colas.....</b>	254
6.6.1	Pilas .....	254
6.6.2	Pilas y lenguajes informáticos.....	254
6.6.3	Colas .....	256
6.6.4	Pilas y colas en la API de Colecciones.....	257
6.7	<b>Conjuntos .....</b>	257
6.7.1	La clase TreeSet.....	259
6.7.2	La clase HashSet .....	260
6.8	<b>Mapas .....</b>	264
6.9	<b>Colas con prioridad .....</b>	270
6.10	<b>Vistas en la API de Colecciones .....</b>	273
6.10.1	El método subList para objetos List .....	273
6.10.2	Los métodos headSet, subSet y tailSet para conjuntos SortedSet .....	273
	<b>Resumen .....</b>	274
	<b>Conceptos clave .....</b>	275
	<b>Errores comunes .....</b>	276
	<b>Internet .....</b>	277
	<b>Ejercicios .....</b>	277
	<b>Referencias .....</b>	286

**Capítulo 7 Recursión****287**

7.1	<b>¿Qué es la recursión? .....</b>	287
7.2	<b>Fundamentos: demostraciones por inducción matemática .....</b>	288
7.3	<b>Recursión básica .....</b>	291
7.3.1	Impresión de números en cualquier base .....	292
7.3.2	Por qué funciona .....	294

7.3.3	Cómo funciona .....	296
7.3.4	Demasiada recursión puede ser peligrosa.....	297
7.3.5	Previsualización de árboles .....	299
7.3.6	Ejemplos adicionales .....	300
<b>7.4</b>	<b>Aplicaciones numéricas .....</b>	<b>305</b>
7.4.1	Aritmética modular .....	305
7.4.2	Exponenciación modular.....	306
7.4.3	Máximo común divisor e inversas multiplicativas .....	307
7.4.4	El criptosistema RSA .....	309
<b>7.5</b>	<b>Algoritmos de tipo divide y vencerás .....</b>	<b>313</b>
7.5.1	El problema de la suma máxima de subsecuencia contigua .....	313
7.5.2	Análisis de una recurrencia básica de tipo divide y vencerás.....	315
7.5.3	Una cota superior general para el tiempo de ejecución de los algoritmos divide y vencerás.....	320
<b>7.6</b>	<b>Programación dinámica .....</b>	<b>322</b>
<b>7.7</b>	<b>Retroceso.....</b>	<b>326</b>
<b>Resumen .....</b>		<b>331</b>
<b>Conceptos clave .....</b>		<b>331</b>
<b>Errores comunes .....</b>		<b>332</b>
<b>Internet .....</b>		<b>333</b>
<b>Ejercicios .....</b>		<b>334</b>
<b>Referencias .....</b>		<b>340</b>

Capítulo 8 Algoritmos de ordenación

341

<b>8.1</b>	<b>¿Por qué es importante la ordenación? .....</b>	<b>341</b>
<b>8.2</b>	<b>Preliminares.....</b>	<b>343</b>
<b>8.3</b>	<b>Análisis de la ordenación por inserción y de otras ordenaciones simples .....</b>	<b>343</b>
<b>8.4</b>	<b>Ordenación Shell.....</b>	<b>347</b>
8.4.1	Rendimiento de la ordenación Shell .....	348
<b>8.5</b>	<b>Ordenación por mezcla .....</b>	<b>350</b>
8.5.1	Mezcla en tiempo lineal de matrices ordenadas .....	351
8.5.2	El algoritmo de ordenación por mezcla.....	352
<b>8.6</b>	<b>Ordenación rápida .....</b>	<b>355</b>
8.6.1	El algoritmo de ordenación rápida .....	355
8.6.2	Análisis del algoritmo de ordenación rápida .....	358
8.6.3	Selección del pivote .....	362
8.6.4	Una estrategia de particionamiento .....	363
8.6.5	Claves iguales al pivote .....	365
8.6.6	Particionamiento basado en la mediana de tres .....	366

8.6.7	Matrices de pequeño tamaño.....	367
8.6.8	Rutina de ordenación rápida en Java .....	368
8.7	<b>Selección rápida.....</b>	<b>370</b>
8.8	<b>Una cota inferior para la ordenación.....</b>	<b>372</b>
Resumen .....	.....	373
Conceptos clave .....	.....	374
Errores comunes .....	.....	375
Internet .....	.....	375
Ejercicios.....	.....	375
Referencias .....	.....	380

**Capítulo 9 Aleatorización 383**

9.1	<b>¿Por qué necesitamos números aleatorios? .....</b>	<b>383</b>
9.2	<b>Generadores de números aleatorios.....</b>	<b>384</b>
9.3	<b>Números aleatorios no uniformes.....</b>	<b>392</b>
9.4	<b>Generación de una permutación aleatoria .....</b>	<b>394</b>
9.5	<b>Algoritmos aleatorizados .....</b>	<b>395</b>
9.6	<b>Prueba aleatorizada de primalidad.....</b>	<b>398</b>
Resumen .....	.....	402
Conceptos clave .....	.....	402
Errores comunes .....	.....	403
Internet .....	.....	403
Ejercicios.....	.....	404
Referencias .....	.....	406

## parte tres Aplicaciones

**Capítulo 10 Entretenimiento y juegos 409**

10.1	<b>Sopa de letras.....</b>	<b>409</b>
10.1.1	Teoría.....	409
10.1.2	Implementación java .....	411
10.2	<b>El juego de las tres en raya.....</b>	<b>418</b>
10.2.1	Poda alfa-beta.....	418
10.2.2	Tablas de transposición .....	421
10.2.3	Ajedrez por computadora .....	425
Resumen .....	.....	426
Conceptos clave .....	.....	426
Errores comunes .....	.....	427

Internet .....	427
Ejercicios.....	427
Referencias .....	429

**Capítulo 11 Pilas y compiladores 431**

<b>11.1 Comprobador de equilibrado de los símbolos.....</b>	<b>431</b>
11.1.1 Algoritmo básico.....	431
11.1.2 Implementación.....	433
<b>11.2 Una calculadora simple .....</b>	<b>444</b>
11.2.1 Máquinas postfijas .....	445
11.2.2 Conversión de notación infija a notación postfija .....	445
11.2.3 Implementación.....	448
11.2.4 Árboles de expresión .....	457
Resumen .....	458
Conceptos clave.....	458
Errores comunes .....	459
Internet .....	459
Ejercicios.....	460
Referencias .....	461

**Capítulo 12 Utilidades 463**

<b>12.1 Compresión de archivos .....</b>	<b>463</b>
12.1.1 Códigos prefijo .....	464
12.1.2 Algoritmo de Huffman .....	466
12.1.3 Implementación.....	469
<b>12.2 Un generador de referencias cruzadas.....</b>	<b>484</b>
12.2.1 Ideas básicas .....	484
12.2.2 Implementación java .....	485
Resumen .....	489
Conceptos clave.....	489
Errores comunes .....	490
Internet .....	490
Ejercicios.....	490
Referencias .....	495

**Capítulo 13 Simulación 497**

<b>13.1 El problema de Josefo.....</b>	<b>497</b>
13.1.1 La solución simple.....	498
13.1.2 Un algoritmo más eficiente.....	500

<b>13.2 Simulación dirigida por sucesos</b>	501
13.2.1 Ideas básicas	503
13.2.2 Ejemplo: una simulación de un servicio de atención telefónica	504
<b>Resumen</b>	512
<b>Conceptos clave</b>	512
<b>Errores comunes</b>	513
<b>Internet</b>	513
<b>Ejercicios</b>	513

Capítulo 14 Grafos y rutas

5615

<b>14.1 Definiciones</b>	.....	515
14.1.1 Representación	.....	517
<b>14.2 Problema del camino más corto no ponderado</b>	.....	527
14.2.1 Teoría	.....	527
14.2.2 Implementación Java	.....	533
<b>14.3 Problema del camino más corto con ponderaciones positivas</b>	.....	533
14.3.1 Teoría: algoritmo de Dijkstra	.....	535
14.3.2 Implementación Java	.....	538
<b>14.4 Problema del camino más corto con ponderaciones negativas</b>	.....	540
14.4.1 Teoría	.....	540
14.4.2 Implementación Java	.....	541
<b>14.5 Problemas de caminos en grafos acíclicos</b>	.....	543
14.5.1 Ordenación topológica	.....	543
14.5.2 Teoría del algoritmo del camino más corto para grafos acíclicos	.....	545
14.5.3 Implementación Java	.....	545
14.5.4 Una aplicación: análisis del camino crítico	.....	548
<b>Resumen</b>	.....	551
<b>Conceptos clave</b>	.....	551
<b>Errores comunes</b>	.....	552
<b>Internet</b>	.....	553
<b>Ejercicios</b>	.....	553
<b>Referencias</b>	.....	557

## parte cuatro **Implementaciones**

Capítulo 15

## Clases internas e implementación de ArrayList

561

<b>15.1 Iteradores y clases anidadas .....</b>	<b>561</b>
<b>15.2 Iteradores y clases internas.....</b>	<b>563</b>
<b>15.3 La clase AbstractCollection .....</b>	<b>567</b>

15.4	StringBuilder .....	571
15.5	Implementación de ArrayList con un iterador .....	572
Resumen .....		579
Conceptos clave .....		579
Errores comunes .....		579
Internet .....		579
Ejercicios .....		580

**Capítulo 16 Pilas y colas 583**

16.1	Implementaciones basadas en matriz dinámica .....	583
16.1.1	Pilas .....	583
16.1.2	Colas .....	588
16.2	Implementaciones con lista enlazada .....	594
16.2.1	Pilas .....	594
16.2.2	Colas .....	597
16.3	Comparación de los dos métodos .....	601
16.4	La clase java.util.Stack .....	602
16.5	Colas de doble terminación .....	604
Resumen .....		604
Conceptos clave .....		604
Errores comunes .....		604
Internet .....		605
Ejercicios .....		605

**Capítulo 17 Listas enlazadas 607**

17.1	Ideas básicas .....	607
17.1.1	Nodos de cabecera .....	609
17.1.2	Clases iteradoras .....	610
17.2	Implementación Java .....	612
17.3	Listas doblemente enlazadas y listas circularmente enlazadas .....	618
17.4	Listas enlazadas ordenadas .....	621
17.5	Implementación de la clase LinkedList de la API de Colecciones .....	621
Resumen .....		635
Conceptos clave .....		635
Errores comunes .....		635
Internet .....		636
Ejercicios .....		636

---

<b>Capítulo 18 Árboles</b>	<b>641</b>
<b>18.1 Árboles generales .....</b>	<b>641</b>
18.1.1 Definiciones .....	641
18.1.2 Implementación .....	643
18.1.3 Una aplicación: sistemas de archivos .....	644
<b>18.2 Árboles binarios .....</b>	<b>649</b>
<b>18.3 Recursión y árboles .....</b>	<b>654</b>
<b>18.4 Recorrido del árbol: clases iteradoras.....</b>	<b>657</b>
18.4.1 Recorrido en postorden .....	659
18.4.2 Recorrido en orden .....	664
18.4.3 Recorrido en preorden .....	664
18.4.4 Recorridos por niveles .....	667
Resumen .....	671
Conceptos clave .....	671
Errores comunes .....	672
Internet .....	672
Ejercicios.....	673
<b>Capítulo 19 Árboles de búsqueda binaria</b>	<b>677</b>
<b>19.1 Ideas básicas .....</b>	<b>677</b>
19.1.1 Las operaciones .....	678
19.1.2 Implementación java .....	680
<b>19.2 Estadísticas de orden .....</b>	<b>687</b>
19.2.1 Implementación Java .....	688
<b>19.3 Análisis de las operaciones con árboles de búsqueda binaria.....</b>	<b>692</b>
<b>19.4 Árboles AVL.....</b>	<b>696</b>
19.4.1 Propiedades .....	696
19.4.2 Rotación simple.....	699
19.4.3 Rotación doble .....	701
19.4.4 Resumen de la inserción AVL .....	704
<b>19.5 Árboles rojo-negro .....</b>	<b>704</b>
19.5.1 Inserción abajo-arriba .....	705
19.5.2 Árboles rojo-negro arriba-abajo .....	707
19.5.3 Implementación Java .....	709
19.5.4 Borrado arriba-abajo .....	716
<b>19.6 Árboles AA.....</b>	<b>718</b>
19.6.1 Inserción.....	719
19.6.2 Borrado .....	722

19.6.3	Implementación Java .....	723
<b>19.7</b>	<b>Implementación de las clases TreeSet y TreeMap de la API de Colecciones.....</b>	<b>726</b>
<b>19.8</b>	<b>Árboles-B.....</b>	<b>747</b>
Resumen .....	752	
Conceptos clave .....	753	
Errores comunes .....	754	
Internet .....	755	
Ejercicios.....	755	
Referencias .....	759	

**Capítulo 20 Tablas hash 763**

<b>20.1</b>	<b>Ideas básicas.....</b>	<b>763</b>
<b>20.2</b>	<b>Función hash.....</b>	<b>765</b>
20.2.1	hashCode en java.lang.String .....	767
<b>20.3</b>	<b>Sondeo lineal.....</b>	<b>768</b>
20.3.1	Un análisis simplista del sondeo lineal.....	770
20.3.2	Lo que realmente sucede: agrupamiento primario.....	771
20.3.3	Ánalisis de la operación find.....	772
<b>20.4</b>	<b>Sondeo cuadrático.....</b>	<b>774</b>
20.4.1	Implementación Java .....	778
20.4.2	Ánalisis del sondeo cuadrático.....	783
<b>20.5</b>	<b>Hash con encadenamiento separado.....</b>	<b>788</b>
<b>20.6</b>	<b>Comparación entre las tablas hash y los árboles de búsqueda binaria.....</b>	<b>789</b>
<b>20.7</b>	<b>Aplicaciones de las tablas hash .....</b>	<b>791</b>
Resumen .....	791	
Conceptos clave .....	792	
Errores comunes .....	793	
Internet .....	793	
Ejercicios.....	793	
Referencias .....	795	

**Capítulo 21 Una cola con prioridad: el montículo binario 797**

<b>21.1</b>	<b>Ideas básicas.....</b>	<b>797</b>
21.1.1	Propiedad estructural.....	798
21.1.2	Propiedad de ordenación del montículo.....	800
21.1.3	Operaciones permitidas .....	800
<b>21.2</b>	<b>Implementación de las operaciones básicas .....</b>	<b>801</b>

21.2.1	Inserción.....	801
21.2.2	La operación deleteMin.....	805
21.3	<b>La operación buildHeap: construcción de un montículo en tiempo lineal.....</b>	807
21.4	<b>Operaciones avanzadas: decreaseKey y merge .....</b>	812
21.5	<b>Ordenación interna: heapsort.....</b>	813
21.6	<b>Ordenación externa .....</b>	816
21.6.1	Por qué necesitamos nuevos algoritmos .....	816
21.6.2	Modelo para la ordenación externa.....	816
21.6.3	El algoritmo simple.....	817
21.6.4	Mezcla multivía .....	818
21.6.5	Mezcla polifásica.....	819
21.6.6	Selección de sustitutos .....	820
	<b>Resumen .....</b>	822
	<b>Conceptos clave.....</b>	822
	<b>Errores comunes .....</b>	823
	<b>Internet .....</b>	823
	<b>Ejercicios.....</b>	824
	<b>Referencias .....</b>	827

## parte cinco **Estructuras de datos avanzadas**

<b>Capítulo 22</b>	<b>Árboles splay</b>	<b>831</b>
--------------------	----------------------	------------

22.1	<b>Autoajuste y análisis amortizado .....</b>	831
22.1.1	Cotas de tiempo amortizadas .....	832
22.1.2	Una estrategia simple de autoajuste (que no funciona).....	833
22.2	<b>El árbol splay básico de tipo abajo-arriba .....</b>	835
22.3	<b>Operaciones básicas con un árbol splay .....</b>	837
22.4	<b>Análisis del splaying abajo-arriba .....</b>	838
22.4.1	Demostración de la cota de splaying .....	841
22.5	<b>Árboles splay de tipo arriba-abajo .....</b>	843
22.6	<b>Implementación de los árboles splay de tipo abajo-arriba .....</b>	846
22.7	<b>Comparación del árbol splay con otros árboles de búsqueda.....</b>	852
	<b>Resumen .....</b>	853
	<b>Conceptos clave.....</b>	853
	<b>Errores comunes .....</b>	854
	<b>Internet .....</b>	854
	<b>Ejercicios.....</b>	854

Referencias .....	855
-------------------	-----

<b>Capítulo 23 Mezcla de colas con prioridad</b>	<b>857</b>
--	------------

<b>23.1 El montículo sesgado .....</b>	<b>857</b>
23.1.1 El mezclado es fundamental .....	857
23.1.2 Mezcla simplista de árboles con ordenación de montículo .....	858
23.1.3 El montículo sesgado: una modificación simple .....	859
23.1.4 Análisis del montículo sesgado .....	860
<b>23.2 El montículo de emparejamiento .....</b>	<b>862</b>
23.2.1 Operaciones con el montículo de emparejamiento .....	863
23.2.2 Implementación del montículo de emparejamiento .....	865
23.2.3 Aplicación: algoritmo de Dijkstra para el cálculo del camino más corto .....	870
Resumen .....	874
Conceptos clave .....	874
Errores comunes .....	874
Internet .....	875
Ejercicios .....	875
Referencias .....	876

<b>Capítulo 24 La clase conjunto disjunto</b>	<b>879</b>
---	------------

<b>24.1 Relaciones de equivalencia .....</b>	<b>879</b>
<b>24.2 Equivalencia dinámica y aplicaciones .....</b>	<b>880</b>
24.2.1 Aplicación: generación de laberintos .....	881
24.2.2 Aplicación: árboles mínimos de recubrimiento .....	883
24.2.3 Aplicación: el problema del ancestro común más próximo .....	886
<b>24.3 El algoritmo rápido de búsqueda .....</b>	<b>889</b>
<b>24.4 El algoritmo rápido de unión .....</b>	<b>890</b>
24.4.1 Algoritmos Inteligentes de unión .....	892
24.4.2 Compresión de caminos .....	894
<b>24.5 Implementación Java .....</b>	<b>895</b>
<b>24.6 Caso peor para la unión por rango con compresión de caminos .....</b>	<b>898</b>
24.6.1 Análisis del algoritmo union/find .....	899
Resumen .....	905
Conceptos clave .....	905
Errores comunes .....	906
Internet .....	906
Ejercicios .....	906
Referencias .....	908

<b>Apéndice A</b>	<b>Operadores</b>	<b>911</b>
<b>Apéndice B</b>	<b>Interfaces gráficas de usuario</b>	<b>913</b>
B.1	Abstract Window Toolkit y Swing.....	913
B.2	Objetos básicos en Swing.....	915
B.2.1	Component .....	916
B.2.2	Container .....	916
B.2.3	Contenedores de nivel superior .....	917
B.2.4	JPanel .....	917
B.2.5	Componentes de E/S importantes .....	919
B.3	Principios básicos .....	924
B.3.1	Gestores de disposición.....	924
B.3.2	Gráficos.....	928
B.3.3	Sucesos .....	929
B.3.4	Tratamiento de sucesos: clases adaptadoras y clases internas anónimas .....	933
B.3.5	Resumen: encajando la piezas.....	935
B.3.6	¿Es esto todo lo que necesito saber acerca de Swing? .....	936
	Resumen .....	936
	Conceptos clave .....	937
	Errores comunes .....	938
	Internet .....	939
	Ejercicios.....	939
	Referencias .....	940
<b>Apéndice C</b>	<b>Operadores bit a bit</b>	<b>941</b>
<b>Índice</b>		<b>945</b>

parte  
uno

# Introducción a Java



Capítulo 1 Estructura primitiva  
del lenguaje Java

Capítulo 2 Tipos de referencia

Capítulo 3 Objetos y clases

Capítulo 4 Herencia



## Estructura primitiva del lenguaje Java

El enfoque fundamental de este libro son las técnicas de resolución de problemas que permiten la construcción de programas sofisticados y con un tiempo de ejecución eficiente. Casi todo el material que se expone es aplicable a cualquier lenguaje de programación. Algunas personas podrían argumentar, en ese sentido, que una descripción amplia de estas técnicas, en términos de pseudocódigo, bastaría para ilustrar los conceptos. Sin embargo, estamos convencidos de que es de enorme importancia trabajar con código real.

Desde luego, lo que no falta son textos sobre lenguajes de programación. Este texto utiliza Java, que goza de una gran popularidad tanto académica como comercial. En los primeros cuatro capítulos, expondremos las características de Java que se utilizarán ampliamente a lo largo del libro. Las características no utilizadas y los detalles más técnicos no se tratarán. Las personas que estén buscando información más profunda sobre Java podrán encontrarla en cualquiera de los múltiples libros de Java disponibles.

Comenzaremos analizando la parte del lenguaje que más se asemeja a un lenguaje de programación de los años 1970, como por ejemplo Pascal o C. Esto incluye los tipos primitivos, las operaciones básicas, las estructuras condicionales e iterativas y el equivalente Java de las funciones.

En este capítulo, veremos:

- Algunos de los fundamentos de Java, incluyendo elementos léxicos simples.
- Los tipos primitivos en Java, incluyendo algunas de las operaciones que se pueden realizar con variables de tipo primitivo.
- Cómo se implementan en Java las instrucciones condicionales y las estructuras de bucle.
- Una introducción al *método estático*, el equivalente Java de las funciones y procedimientos que se usan en los lenguajes no orientados a objetos.

### 1.1 El entorno general

¿Cómo se introducen, compilan y ejecutan los programa de aplicación en Java? La respuesta depende, por supuesto, de la plataforma concreta en la que esté albergado el compilador de Java.

`javac` compila los archivos `.java` y genera archivos `.class` que contienen código de bytes. `java` invoca al intérprete de Java, el cual también es conocido como Máquina virtual.

El código fuente Java reside en archivos cuyos nombres terminan con el sufijo `.java`. El compilador local, `javac`, compila el programa y genera archivos `.class`, que contienen código de bytes. El *código de bytes* Java representa el lenguaje intermedio portable que luego será interpretado ejecutando el intérprete de Java, `java`. El intérprete también se conoce con el nombre de *Máquina virtual*.

Para los programas Java, la entrada puede provenir de varios puntos diferentes:

- El terminal, cuya entrada se denomina *entrada estándar*.
- Parámetros adicionales en la invocación de la Máquina virtual: *argumentos de la línea de comandos*.
- Un componente GUI.
- Un archivo.

Los argumentos de la línea de comandos son especialmente importantes para especificar las opciones del programa; hablaremos de ellos en la Sección 2.4.5. Java proporciona mecanismos que permiten leer y escribir archivos, lo que se aborda brevemente en la Sección 2.6.3 y en más detalle en la Sección 4.5.3, como ejemplo del *patrón decorador*. Muchos sistemas operativos proporcionan una alternativa que se conoce con el nombre de *redirección de archivos*, en la que el sistema operativo se encarga de leer la entrada desde (o de enviar la salida a) un archivo, de forma totalmente transparente para el programa que se está ejecutando. En Unix (y también desde una ventana MS/DOS), por ejemplo el comando

`Programa java < archivoentrada > archivosalida`

se encarga de organizar las cosas automáticamente para que las lecturas del terminal se redirijan de modo que provengan de `archivoentrada` y para que las escrituras en el terminal se redirijan y vayan a `archivosalida`.

## 1.2 El primer programa

Comenzaremos examinando el sencillo programa Java mostrado en la Figura 1.1. Este programa imprime una frase corta en el terminal. Observe que los números de línea mostrados a la izquierda del código *no forman parte del programa*. Se suministran simplemente para facilitar las referencias.

Guarde el programa en el archivo fuente `FirstProgram.java` y después compleelo y ejecútelo. Tenga en cuenta que el nombre del archivo fuente debe corresponderse con el nombre de la clase (mostrado en la línea 4), respetando el uso de mayúsculas y minúsculas. Si está empleando el JDK, los comandos serán:<sup>1</sup>

```
javac FirstProgram.java
java FirstProgram
```

<sup>1</sup> Si está empleando el JDK de Sun, `javac` y `java` se utilizan directamente. En caso contrario, en un entorno de desarrollo interactivo (IDE) típico, como Netbeans o Eclipse, estos comandos son ejecutados en nuestro nombre entre bastidores.

```
1 // Primer programa
2 // MW, 5/1/10
3
4 public class FirstProgram
5 {
6     public static void main( String [ ] args )
7     {
8         System.out.println( "Is there anybody out there?" );
9     }
10 }
```

Figura 1.1 Un primer programa sencillo.

### 1.2.1 Comentarios

Java dispone de tres formas de comentarios. La primera de ellas, heredada de C, comienza con la secuencia de símbolos `/*` y termina con `*/`. He aquí un ejemplo:

```
/* Esto es un comentario
de dos líneas */
```

Los comentarios no pueden anidarse.

La segunda forma, heredada de C++, comienza con la secuencia de símbolos `//` y no emplea ninguna secuencia de símbolos de terminación. En lugar de ello, el comentario se extiende simplemente hasta el final de la línea, como se muestra en las líneas 1 y 2 de la Figura 1.1.

Los comentarios facilitan a las personas la lectura del código. Java dispone de tres formatos de comentario.

La tercera forma comienza con los símbolos `/**` en lugar de `/*`. Esta forma se puede utilizar para proporcionar información a la utilidad `javadoc`, la cual generará la documentación del programa a partir de los comentarios. Esta forma se explica en la Sección 3.3.

La finalidad de los comentarios es facilitar la lectura del código a los seres humanos, entre los que se incluyen otros programadores que necesiten modificar o utilizar nuestro código, aunque también nos facilitarán la compresión del programa a nosotros mismos. Un programa bien comentado es un signo claro de que su autor es un buen programador.

### 1.2.2 main

Un programa Java está compuesto por una colección de clases que interactúan, las cuales contienen una serie de métodos. El equivalente Java de las funciones y procedimientos es el *método estático*, que describiremos en la Sección 1.6. Al ejecutar cualquier programa, se invoca el método estático especial `main`. La línea 6 de la Figura 1.1 muestra que el método estático `main` puede invocarse con argumentos de la línea de comandos. Los tipos de parámetros de `main` y el tipo de retorno `void` que se muestran en esa línea son obligatorios.

Cuando se ejecuta un programa, se invoca el método especial `main`.

## 1.2.2 Salida a través de terminal

`println` se utiliza para enviar la salida del programa.

El programa de la Figura 1.1 está compuesto por una única instrucción, mostrada en la línea 8. `println` es el mecanismo de salida principal en Java. Aquí, se coloca una cadena de caracteres constante en el flujo estándar de salida `System.out` aplicando un método `println`. Hablaremos con más detalle acerca de la entrada y la salida en la Sección 2.6. Por

ahora, nos limitaremos a mencionar que se utiliza esa misma sintaxis para llevar a cabo la salida, independientemente de la entidad que se quiera emplear: no importa que esa entidad sea un entero, un número en coma flotante, una cadena de caracteres o de cualquier otro tipo.

## 1.3 Tipos primitivos

Java define ocho *tipos primitivos*. También proporciona al programador una gran flexibilidad a la hora de definir nuevos tipos de objetos, denominados *clases*. Sin embargo, existen importantes diferencias en Java entre los tipos primitivos y los tipos definidos por el usuario. En esta sección vamos a examinar los tipos primitivos y las operaciones básicas que pueden definirse con ellos.

### 1.3.1 Los tipos primitivos

Los tipos primitivos en Java son los de tipo entero, de coma flotante, booleanos y de carácter.

El estándar Unicode contiene más de 30.000 caracteres codificados distintos, lo que cubre los idiomas escritos más importantes.

Java dispone de los ocho tipos primitivos mostrados en la Figura 1.2. El tipo más común es el correspondiente a los números enteros, el cual se especifica mediante la palabra clave `int`. A diferencia de muchos otros lenguajes de programación, el rango de los enteros no depende de la máquina. En lugar de ello, coincide en todas las implementaciones Java, independientemente de la arquitectura de computadora subyacente. Java también permite utilizar entidades de tipo `byte`, `short` y `long`, que son conocidas como *tipos enteros*. Los números en coma flotante se representan mediante los tipos `float` y `double`. `double` tiene más dígitos significativos, por lo que se recomienda su uso antes que el de `float`. El tipo `char` se utiliza para representar caracteres individuales. Un `char` ocupa 16 bits para representar el estándar Unicode. Este estándar contiene más de 30.000 caracteres codificados distintos, lo que cubre todos los idiomas escritos más importantes. El subconjunto de menor peso de Unicode es idéntico a ASCII. El último de los tipos primitivos es `boolean`, que puede tomar los valores `true` o `false`.

### 1.3.2 Constantes

Las constantes enteras pueden representarse en notación decimal, octal o hexadecimal.

Las *constantes enteras* pueden representarse en notación decimal, octal o hexadecimal. La notación octal se indica mediante el prefijo `0`; la notación hexadecimal se indica mediante el prefijo `0x` o `0X`. Todas las siguientes serían formas equivalentes de representar el entero 37: `37`, `045`, `0x25`. En este texto no vamos a utilizar los enteros octales; sin embargo, tenemos que ser conscientes de su existencia, con el fin de utilizar `0s` como prefijo solamente

Tipo primitivo	Lo que almacena	Rango
byte	enteros de 8 bits	-128 a 127
short	enteros de 16 bits	-32.768 a 32.767
int	enteros de 32 bits	-2.147.483.648 a 2.147.483.647
long	enteros de 64 bits	-263 a 263 - 1
float	coma flotante de 32 bits	6 dígitos significativos ( $10^{-46}$ , $10^{38}$ )
double	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.

cuando pretendamos hacerlo conscientemente así. Utilizaremos hexadecimales en un único lugar (Sección 12.1), y allí hablaremos con algo más de detalle acerca de los mismos.

Una *constante de caracteres* se encierra entre una pareja de comillas simples, como en 'a'. Internamente, esta secuencia de caracteres se interpreta como un número de pequeña magnitud. Las rutinas de salida interpretarán posteriormente ese número de pequeña magnitud como el carácter correspondiente. Una *constante de cadena* consta de una secuencia de caracteres encerrada entre comillas dobles, como en "Hola". Existen algunas secuencias especiales, conocidas con el nombre de *secuencias de escape*, que tienen usos específicos (por ejemplo, ¿cómo representaríamos una comilla simple?). En este texto utilizaremos '\n', '\\', '\'' y '\"', que representan, respectivamente, el carácter de nueva línea, el carácter de barra inclinada a la izquierda, la comilla simple y la comilla doble.

Una constante de cadena está compuesta por una secuencia de caracteres encerrados entre dobles comillas.

Las secuencias de escape se utilizan para representar ciertas constantes de carácter.

### 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 deber ser un *identificador*. Los identificadores pueden estar compuestos por cualquier combinación de letras, dígitos y caracteres de guión bajo; sin embargo, no pueden empezar por un dígito. Las palabras reservadas, como int, no están permitidas. Aunque es legal hacerlo, es recomendable no reutilizar los nombres de identificador que ya se estén empleando de forma visible (por ejemplo, no utilice main como nombre de una entidad).

Las variables se nombran mediante un identificador.

Java *diferencia* entre mayúsculas y minúsculas, lo que quiere decir que Age y age son identificadores distintos. En este texto se utiliza el siguiente convenio para denominar a las variables: todas las variables comienzan con una letra minúscula y cada palabra nueva empieza con una letra mayúscula. Un ejemplo sería el identificador minimumWage.

Java diferencia entre mayúsculas y minúsculas.

He aquí algunos ejemplos de declaraciones:

```
int num3;                      // Inicialización predeterminada
double minimumWage = 4.50;      // Inicialización estándar
int x = 0, num1 = 0;            // Se declaran dos entidades
int num2 = num1;
```

Las variables deben declararse cerca del lugar en el que se las utilice por primera vez. Como veremos, la colocación de una declaración determina su ámbito y su significado.

### 1.3.4 Entrada y salida a través de terminal

La E/S formateada básica a través de terminal se realiza mediante `nextLine` y `println`. El flujo estándar de entrada es `System.in` y el flujo estándar de salida es `System.out`.

El mecanismo básico para la E/S formateada utiliza el tipo `String`, del que hablaremos en la Sección 2.3. Para la salida, `+` permite combinar dos objetos `String`. Si el segundo argumento no es de tipo `String`, se crea para él un objeto `String` temporal, si es de tipo primitivo. Estas conversiones a `String` también pueden definirse para los objetos (Sección 3.4.3). Para la entrada, asociamos un objeto `Scanner` con `System.in`, lo que nos permitirá leer un objeto `String` o un tipo primitivo. En la Sección 2.6 se proporciona una explicación más detallada de la E/S, incluyendo el tratamiento de los archivos formateados.

## 1.4 Operadores básicos

En esta sección se describen algunos de los operadores disponibles en Java. Estos operadores se emplean para formar *expresiones*. Una constante o una entidad son, por sí mismas, una expresión, como también lo son las combinaciones de constantes y variables mediante operadores. Una expresión seguida por un punto y coma es una instrucción simple. En la Sección 1.5, examinaremos otros tipos de instrucciones, en las que presentaremos operadores adicionales.

### 1.4.1 Operadores de asignación

En la Figura 1.3 se muestra un programa Java simple que ilustra unos cuantos operadores. El operador de asignación básico es el signo igual. Por ejemplo, en la línea 16, se asigna a la variable `a` el valor de la variable `c` (que en ese punto es 6). Las modificaciones posteriores en el valor de `c` no afectan a `a`. Los operadores de asignación se pueden encadenar, como por ejemplo en `z=y=x=0`.

Otro operador de asignación es `+=`, cuyo uso se ilustra en la línea 18 del listado de la Figura 1.3. El operador `+=` suma el valor situado en el lado derecho (del propio operador) a la variable indicada en el lado izquierdo. Por tanto, en la Figura 1.3, `c` se incrementa, pasando del valor de 6 que tenía antes de la línea 18 a un valor de 14.

Java proporciona diversos operadores de asignación, incluyendo `-`, `+=`, `-=`, `*=` y `/=`.

Java proporciona varios otros operadores de asignación, como por ejemplo `--`, `*=` y `/=`, que modifican la variable indicada en el lado izquierdo del operador mediante las operaciones de resta, multiplicación y división, respectivamente.

```

1 public class OperatorTest
2 {
3     // Programa que ilustra 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.

#### 1.4.2 Operadores aritméticos binarios

La línea 20 de la Figura 1.3 ilustra uno de los *operadores aritméticos binarios* que son típicos de todos los lenguajes de programación: el operador suma (+). El operador + hace que los valores de b y c se sumen; b y c por su parte, no sufrirán ninguna modificación. El valor resultante se asigna a a. Otros operadores aritméticos típicamente utilizados en Java son −, \*, / y %, que se emplean, respectivamente, para la resta, la multiplicación, la división y la operación de cálculo del resto. La división entera solo devuelve la parte entera y descarta el resto.

Como suele ser habitual, la suma y la resta tienen la misma precedencia, y dicha precedencia es inferior a la del grupo compuesto por los operadores de multiplicación, división y módulo; por tanto,  $1+2*3$  daría como resultado 7. Todos estos operadores se asocian de izquierda a derecha (por lo que  $3-2-2$  da como resultado -1). Todos los operadores tienen precedencia y asociatividad. En el Apéndice A se proporciona una tabla completa de operadores.

Java proporciona diversos operadores aritméticos binarios, incluyendo +, −, \*, / y %.

### 1.4.3 Operadores unarios

Están definidos varios operadores unarios, incluyendo -.

El operador de autoincremento y el de autodecremento suman 1 y restan 1, respectivamente. Los operadores para esto son ++ y --. Hay dos formas de incrementar y decrementar: prefijo y postfixia.

Además de los operadores aritméticos binarios, que requieren dos operandos, Java proporciona operadores unarios, que solo requieren un único operando. El más familiar de estos operadores es el menos unario, que proporciona como resultado el negado de su operando. Así,  $-x$  devuelve el negado de  $x$ .

Java proporciona también el operador de autoincremento para sumar 1 a una variable (dicho operador se denota mediante  $++$ ) y el operador de autodeCREMENTO para restar 1 de una variable (dicho operador se denota mediante  $--$ ). El uso más benigno de esta funcionalidad se muestra en las líneas 22 y 23 de la Figura 1.3. En ambas líneas, el *operador de autoincremento*  $++$  suma 1 al valor de la variable. En Java, sin embargo, un operador aplicado a una expresión nos da una expresión que tiene un valor. Aunque se garantiza que la variable será incrementada antes de la ejecución de la siguiente instrucción, nos surge inmediatamente la pregunta: ¿cuál es el valor de la expresión de autoincremento si se utiliza dentro de una expresión de mayor tamaño?

En este caso, es crucial la colocación del operador  $++$ . La semántica de  $++x$  es que el valor de la expresión será el nuevo valor de  $x$ . Esto se denomina *incremento prefijo*. Por contraste,  $x++$  significa que el valor de la expresión es el valor original de  $x$ . Esto se denomina *incremento postfixio*. Esta característica se muestra en la línea 24 de la Figura 1.3, donde  $a$  y  $b$  se incrementan en 1, y  $c$  se calcula sumando el valor original de  $a$  al valor *incrementado* de  $b$ .

### 1.4.4 Conversiones de tipo

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

```
double quotient;
int x = 6;
int y = 10;
quotient = x / y; // ¡Probablemente incorrecto!
```

La primera operación es la división, y como  $x$  e  $y$  son ambas enteras, el resultado será una división entera, y obtendremos 0. El entero 0 será después convertido implícitamente a un valor `double` para poderlo asignar a `quotient`. Pero lo que queríamos era asignar a `quotient` el valor 0.6. La solución es generar una variable temporal para  $x$  o  $y$ , de modo que la división se lleve a cabo aplicando las reglas para `double`. Esto se haría de la siguiente forma:

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

El operador de conversión de tipo se utiliza para generar una entidad temporal de un tipo nuevo.

Observe que ni  $x$  ni  $y$  cambian. Se crea una entidad temporal sin nombre y se utiliza el valor de esa entidad temporal para la división. El operador de conversión de tipo tiene una mayor precedencia que la división, por lo que primero se hace la conversión de tipo de  $x$  y luego se lleva a cabo la división (en lugar de realizarse la conversión después de hacer la división de dos valores `int`).

## 1.5 Instrucciones condicionales

En esta sección se examinan las instrucciones que afectan al flujo de control: las instrucciones condicionales y los bucles. Como consecuencia, presentaremos nuevos operadores.

### 1.5.1 Operadores relacionales y de igualdad

La prueba básica que podemos realizar con los tipos primitivos es la comparación. Esta se lleva a cabo utilizando los operadores de igualdad y desigualdad, así como los operadores relacionales (menor que, mayor que, etc.).

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

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

```
exprIzquierda==exprDerecha
```

se evalúa como `true` si `exprIzquierda` y `exprDerecha` son iguales; en caso contrario, dará como resultado `false`. De forma similar,

```
exprIzquierda!=exprDerecha
```

se evalúa como `true` si `exprIzquierda` y `exprDerecha` no son iguales; y `false` en caso contrario.

Los *operadores relacionales* son <, <=, > y >=. Estos tienen sus significados naturales para los tipos predefinidos. Los operadores relacionales tienen una precedencia mayor que los operadores de igualdad y ambos tienen una precedencia inferior a los operadores aritméticos, pero mayor precedencia que los operadores de asignación, de modo que frecuentemente es innecesario utilizar paréntesis. Todos estos operadores se asocian de izquierda a derecha, pero este hecho es irrelevante: en la expresión `a<b<6`, por ejemplo, el primer < genera un `boolean` y el segundo es ilegal porque < no está definido para los valores de tipo `boolean`. En la siguiente sección se describe la forma correcta de realizar esta comprobación.

Los operadores relacionales son <, <=, > y >=.

### 1.5.2 Operadores lógicos

Java proporciona *operadores lógicos* que se utilizan para simular los conceptos de AND, OR y NOT propios del álgebra booleana. En ocasiones, estos operadores se designan con el nombre de *conjunción*, *disyunción* y *negación*, respectivamente, y sus operadores correspondientes son &&, || y !. La comprobación de la sección anterior se podría implementar correctamente como `a<b && b<6`. La precedencia de la conjunción y la disyunción es lo suficientemente baja como para que no sean necesarios los paréntesis. El operador && tiene una mayor precedencia que ||, mientras que ! se agrupa con otros operadores unarios (y tiene, por tanto, la mayor precedencia de los tres). Los operandos y los resultados de los operadores lógicos son de tipo `boolean`. La Figura 1.4 muestra el resultado de aplicar los operadores lógicos a todas las posibles entradas.

Java proporciona operadores lógicos que se utilizan para simular los conceptos de AND, OR y NOT propios del álgebra booleana. Los operadores correspondientes son &&, || y !

Una regla importante es que && y || son operaciones de evaluación cortocircuitables. El concepto de *evaluación cortocircuitable* quiere decir que si el resultado puede determinarse examinando la primera expresión, entonces no se evalúa la segunda expresión. Por ejemplo, en

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.

El concepto de evaluación cortocircuitable quiere decir que si el resultado de un operador lógico puede determinarse examinando la primera expresión, entonces no se evalúa la segunda expresión.

$x != 0 \&& 1/x != 3$

si  $x$  es 0, entonces la primera mitad será `false`. Automáticamente, el resultado de la operación AND deberá ser `false`, por lo que ni siquiera se llega a evaluar la segunda mitad. Esto es bueno porque la división por cero nos daría un comportamiento erróneo. La técnica de evaluación cortocircuitable nos permite no tener que preocuparnos acerca de esa división por cero concreta.<sup>2</sup>

### 1.5.3 La instrucción if

La instrucción `if` es la principal de las formas para llevar a cabo la toma de decisiones en los programas.

La instrucción `if` es la principal de las formas para llevar a cabo la toma de decisiones en los programas. Su forma básica es:

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

Si `expresión` se evalúa como `true`, entonces se ejecuta `instrucción`; en caso contrario, no se ejecuta. Cuando se completa la instrucción `if` (sin que se haya producido un error no tratado), el control pasa a la siguiente instrucción.

Opcionalmente, podemos utilizar una instrucción `if-else` de la forma siguiente:

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

En este caso, si `expresión` se evalúa como `true`, entonces se ejecuta `instrucción1`; en caso contrario, se ejecuta `instrucción2`. En cualquiera de los dos casos, el control pasa a continuación a la siguiente instrucción, como en

<sup>2</sup> Hay casos (extremadamente) raros en los que es preferible no realizar el cortocircuito. En tales casos, los operadores `&` y `|` con argumentos boolean garantizan que ambos argumentos se evalúen, incluso si el resultado de la operación puede determinarse a partir del primer argumento.

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

Recuerde que cada una de las cláusulas `if` y `else` contiene al menos una instrucción, independientemente de cómo utilicemos la sangría del texto. He aquí dos errores típicos:

```
if( x == 0 ); // ; es una instrucción nula (y hay que tenerla en cuenta)
System.out.println( "x is zero" );
else
    System.out.print( "x is " );
    System.out.println( x ); // Dos instrucciones
```

El primer error es la inclusión del carácter `;` al final del primer `if`. Este punto y coma cuenta, por sí mismo, como *instrucción nula*, en consecuencia, este fragmento no se compilará correctamente (el `else` ya no está asociado con un `if`). Una vez corregido este error, tenemos otro error lógico: la última línea no forma parte del `else`, aun cuando el sangrado del texto lo sugiera. Para corregir este problema tenemos que utilizar un *bloque*, en el que tendremos una secuencia de instrucciones encerrada entre una pareja de llaves:

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

La instrucción `if` puede ser a su vez incluida dentro de otra cláusula `if` o `else`, al igual que otras instrucciones de control de las que hablaremos más adelante en esta sección. En el caso de instrucciones `if-else` anidadas, cada `else` se corresponderá con el `if` abierto más interno. Puede ser necesario añadir llaves si ese no es el significado que pretendemos.

Un punto y coma aislado  
será, por sí mismo, una  
*instrucción nula*.

Un bloque es una  
secuencia de instrucciones  
encerrada entre llaves.

#### 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 para la instrucción `while` es

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

La instrucción `while`  
es una de las tres formas  
básicas de implementación  
de bucles.

Observe que, como en la instrucción `if`, no hay ningún punto y coma en la sintaxis. Si se incluye uno, se interpretará como instrucción nula.

Mientras que `expresión` es `true` se ejecuta `instrucción`; después, vuelve a evaluarse `expresión`. Si `expresión` es inicialmente `false`, `instrucción` no se ejecutará nunca. Generalmente, `instrucción` hace algo que puede modificar potencialmente el valor de `expresión`; en caso contrario, podría ser infinito. Cuando el bucle `while` termina (normalmente), el control se devuelve a la siguiente instrucción.

### 1.5.5 La instrucción `for`

La instrucción `while` es suficiente para expresar todo tipo de repeticiones. Aun así, Java proporciona otras dos formas de implementación de bucles: la instrucción `for` y la instrucción `do`. La instrucción `for` se utiliza principalmente para las iteraciones. Su sintaxis es

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

La instrucción `for` es una estructura de bucle que se utiliza principalmente para iteraciones simples.

Aquí, `inicialización`, `comprobación` y `actualización` son todas expresiones y son todas ellas opcionales. Si no se proporciona `comprobación`, toma como valor predeterminado `true`. No se incluye ningún punto y coma después del paréntesis de cierre.

La instrucción `for` se ejecuta realizando primero la `inicialización`. Después, mientras que `comprobación` es `true`, se llevan a cabo las dos acciones siguientes: se ejecuta `instrucción` y luego se realiza la `actualización`. Si se omiten la `inicialización` y la `actualización`, entonces la instrucción `for` se comporta exactamente como una instrucción `while`. La ventaja de una instrucción `for` es la claridad, en el sentido de que para las variables que sirven como contador (de iteraciones), la instrucción `for` hace que sea mucho más fácil ver cuál es el rango de ese contador. Por ejemplo, el siguiente fragmento imprime los primeros 100 enteros positivos:

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

Este fragmento ilustra la técnica común de declarar un contador en la parte de `inicialización` del bucle. El ámbito del contador solo abarcará el interior del bucle.

Tanto `inicialización` como `actualización` pueden utilizar una coma para incluir múltiples expresiones. El siguiente fragmento ilustra esta variante:

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

Los bucles se anidan de la misma forma que las instrucciones `if`. Por ejemplo, podemos encontrar todas las parejas de números pequeños cuya suma sea igual a su producto (como por ejemplo 2 y 2, cuya suma y producto son ambos iguales a 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 );
```

Como veremos, sin embargo, al anidar bucles podemos llegar a crear fácilmente programas cuyos tiempos de ejecución crezcan rápidamente.

Java 5 añade un bucle `for` "mejorado". Hablaremos de esta mejora en la Sección 2.4 y en el Capítulo 6.

### 1.5.6 La instrucción do

La instrucción `while` ejecuta de forma repetida una comprobación. Si la comprobación es `true`, entonces ejecuta una instrucción especificada. Sin embargo, si la comprobación inicial da como resultado `false`, la instrucción especificada no llega nunca a ejecutarse. No obstante, en algunos casos, desearemos garantizar que la instrucción especificada se ejecute al menos una vez. Esto se hace utilizando la instrucción `do`. La instrucción `do` es idéntica a la instrucción `while`, excepto porque la comprobación se realiza después de haber ejecutado la instrucción especificada. La sintaxis es:

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

La instrucción `do` es una estructura de bucle que garantiza que el bucle se ejecuta al menos una vez.

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

```
do  
{  
    Pedir datos al usuario;  
    Leer el valor;  
} while( el valor no sea correcto);
```

La instrucción `do` es, con mucho, la menos frecuentemente utilizada de las tres estructuras de bucle. Sin embargo, cuando tenemos que hacer algo al menos una vez y por alguna razón resulta inapropiado emplear un bucle `for`, entonces la instrucción `do` es el método preferido.

### 1.5.7 break y continue

Las instrucciones `for` y `while` permiten terminar el bucle antes del principio de una instrucción repetida. La instrucción `do` permite terminar el bucle después de la ejecución de una instrucción repetida. Ocasionalmente, lo que queríamos es terminar la ejecución en mitad de una instrucción (compuesta) repetida. La instrucción `break`, que está compuesta por la palabra clave `break` seguida por un punto y coma, puede emplearse para conseguir precisamente esto. Normalmente, la instrucción `break` irá precedida de una instrucción `if`, como en

```

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

```

La instrucción `break` hace que se salga del bucle o instrucción `switch` más internos. La instrucción `break` etiquetada permite salir de un bucle anidado.

La instrucción `break` solo existe en el bucle más interno (también se utiliza en conjunción con la instrucción `switch`, que se describe en la siguiente sección). Si es necesario salir de varios bucles, la instrucción `break` no funcionará, y lo más probable es que terminemos obteniendo un código con un diseño bastante pobre. Aun así, Java proporciona una instrucción `break` etiquetada. En la instrucción `break` etiquetada, se etiqueta un bucle y luego puede aplicarse una instrucción `break` al bucle, independientemente de cuántos otros bucles haya anidados. He aquí un ejemplo:

```

externo:
while( ... )
{
    while( ... )
        if( desastre )
            break externo; // Ir a la instrucción siguiente a externo
    }
    // El control pasa aquí después de salir del bucle externo
}

```

La instrucción `continue` pasa a la siguiente iteración del bucle más interno.

Ocasionalmente, lo que queremos es terminar la iteración actual de un bucle e ir directamente a la siguiente iteración. Esto puede conseguirse utilizando una instrucción `continue`. Al igual que la instrucción `break`, la instrucción `continue` incluye un punto y coma y se aplica únicamente al bucle más interno. El siguiente fragmento imprime los primeros 100 enteros, con excepción de aquellos que sean divisibles por 10:

```

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

```

Por supuesto, en este ejemplo, existen alternativas que podrían utilizarse en lugar de la instrucción `continue`. Sin embargo, se emplea `continue` de forma bastante común para evitar incluir patrones `if-else` complicados dentro de los bucles.

### 1.5.8 La instrucción switch

La instrucción `switch` se utiliza para elegir entre varios valores pequeños de tipo entero (o carácter). Está compuesta por una expresión y un bloque. El bloque contiene una secuencia de instrucciones y una colección de etiquetas, que representan los posibles valores de la expresión. Todas las etiquetas deben ser constantes de tiempo de compilación diferentes. Si está presente, una etiqueta predeterminada opcional permite hacer referencia a todas las etiquetas no representadas. Si no hay ningún caso aplicable a la expresión de la instrucción `switch`, la instrucción `switch` termina; en caso contrario, el control pasa a la etiqueta apropiada y se ejecutan todas las instrucciones que se encuentren a partir de ahí. Puede utilizarse una instrucción `break` para forzar la terminación anticipada de la instrucción `switch` y, de hecho, casi siempre se emplea una instrucción `break` para separar casos que sean lógicamente distintos. En la Figura 1.5 se muestra un ejemplo típico de esta estructura.

La instrucción `switch` se utiliza para seleccionar entre varios valores pequeños de tipo entero (o carácter).

### 1.5.9 El operador condicional

El *operador condicional* `? :` se utiliza como abreviatura para instrucciones `if-else` simples. Su formato general es:

```
comprobacionExpr ? exprSi : exprNo
```

El operador condicional `? :` se utiliza como abreviatura para instrucciones `if-else` sencillas.

En primer lugar se evalúa `comprobacionExpr`, seguida por `exprSi` o `exprNo`, generando así el resultado de la expresión completa. `exprSi` se evalúa si `comprobacionExpr` es `true`; en caso contrario, se evalúa `exprNo`. La precedencia del operador condicional está justo por encima de la de los operadores de asignación. Por esta razón, podemos evitar el uso de paréntesis al asignar el resultado del operador condicional a una variable. Como ejemplo, podemos asignar a `minVal` el mínimo de `x` e `y` de la forma siguiente:

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

## 1.6 Métodos

Lo que se conoce como función o procedimiento en otros lenguajes, en Java, se denomina *método*. En el Capítulo 3 explicaremos en detalle los métodos. En esta sección se presentan algunos de los conceptos básicos para la escritura de funciones, tales como `main`, de una forma no orientada a objetos (como la que podríamos encontrar en un lenguaje como C), con el fin de poder escribir algunos programas sencillos.

Un método es similar a una función en otros lenguajes. La cabecera del método está compuesta por el nombre, el tipo de retorno y una lista de parámetros. La declaración del método incluye el cuerpo del mismo.

Una *cabecera de método* consta de un nombre, una lista de parámetros (posiblemente vacía) y un tipo de retorno. El código concreto para implementar el método, en ocasiones denominado *cuerpo del método*, es formalmente un *bloque*. Una *declaración de método* consta de una cabecera y un cuerpo. En la Figura 1.6 se muestran un ejemplo de declaración de método y una rutina `main` que lo utiliza.

```
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 manejar el carácter de nueva línea
17        break;
18
19    default:
20        // Código para manejar otros casos
21        break;
22 }
```

Figura 1.5 Estructura de una instrucción switch.

```
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    // Declaración del método
12    public static int min( int x, int y )
13    {
14        return x < y ? x : y;
15    }
16 }
```

Figura 1.6 Ilustración de la declaración de un método y de una llamada a ese método.

Anteponiendo a cada método las palabras `public static`, podemos imitar las funciones globales de estilo C. Aunque declarar un método como `static` resulta ser una técnica útil en algunas instancias, no se debe abusar de ella, ya que, en general, no nos interesaría utilizar Java para escribir código "estilo C". En la Sección 3.6 hablaremos de la utilización más típica de `static`.

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

La instrucción `return` se utiliza para devolver un valor al llamante. Si el tipo de retorno es `void`, entonces no se devuelve ningún valor y no debe utilizarse `return;` dentro del método.

Un método `public static` es el equivalente de una función global "estilo C".

En el *paso por valor*, los argumentos reales se copian en los parámetros formales. Las variables se pasan mediante *paso por valor*.

La instrucción `return` se utiliza para devolver un valor al llamante.

### 1.6.1 Nombres de métodos sobrecargados

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

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

En algunos lenguajes, esto puede ser inaceptable si `max` ya está declarada. Por ejemplo, podríamos escribir también

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

Java permite la *sobrecarga* de los nombres de métodos. Esto significa que puede haber varios métodos con el mismo nombre y que todos ellos pueden declararse con el mismo ámbito de clase, siempre y cuando sus *signaturas* (es decir, los tipos de su lista de parámetros) sean distintas. Cuando se hace una llamada a `max`, el compilador puede deducir cuál de los significados posibles hay que aplicar, basándose en el tipo de los argumentos utilizados. Puede haber dos signaturas que tengan el mismo número de parámetros, siempre y cuando al menos uno de los tipos de esos parámetros sea distinto.

Observe que el tipo de retorno no se incluye en la signatura. Esto quiere decir que es ilegal tener dos métodos con el mismo ámbito de clase que solo se diferencien por el tipo de retorno. Los métodos con diferentes ámbitos de clase pueden tener los mismos nombres, las mismas signaturas e incluso los mismos tipos de retorno; esto se explica en el Capítulo 3.

La sobrecarga de un nombre de método quiere decir que puede haber varios métodos con el mismo nombre, siempre y cuando los tipos de su lista de parámetros sean distintos.

### 1.6.2 Clases de almacenamiento

Las entidades que se declaran dentro del cuerpo de un método son variables locales y solo se puede acceder a ellas por su nombre dentro del cuerpo del método. Estas entidades se crean al ejecutarse el cuerpo del método y desaparecen cuando el cuerpo del método termina.

Las variables static final son constantes.

Una variable declarada fuera del cuerpo de un método será global para esa clase. Es similar a las variables globales en otros lenguajes, si se utiliza la palabra `static` (que es probable que sea necesaria, para poder hacer que la entidad sea accesible por parte de los métodos estáticos). Si se utilizan tanto `static` como `final`, serán constantes simbólicas globales. Como por ejemplo,

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

Observe el uso del convenio común para denominar las constantes simbólicas, que se escriben completamente en mayúsculas. Si el nombre del identificador está formado por varias palabras, se las separa mediante un carácter de guión bajo, como en `MAX_INT_VALUE`.

Si se omite la palabra `static`, entonces la variable (o constante) tiene un significado distinto, del que hablaremos en la Sección 3.6.5.

## Resumen

---

En este capítulo hemos visto las características primitivas de Java, tales como los tipos primitivos, los operadores, las instrucciones condicionales y de bucle, y los métodos, características todas ellas que se encuentran en prácticamente todos los lenguajes.

Qualquier programa no trivial requerirá el uso de tipos no primitivos, denominados *tipos de referencia*, de los que hablaremos en el siguiente capítulo.



### Conceptos clave

**bloque** Una secuencia de instrucciones encerrada entre llaves. (13)

**break, instrucción** Una instrucción que permite salir de la instrucción de bloque o `switch` más interna. (15)

**break etiquetada, instrucción** Una instrucción `break` utilizada para salir de bucles anidados. (16)

**cabecera de método** Está compuesta por el nombre, el tipo de retorno y la lista de parámetros. (17)

**constante de cadena** Una constante compuesta por una secuencia de caracteres encerrados entre dobles comillas. (7)

**continue, instrucción** Una instrucción que hace que se salte a la siguiente iteración del bucle más interno. (15)

**código de bytes** Código intermedio portable generado por el compilador Java. (4)

**comentarios** Hacen que el código sea más fácil de leer por parte de las personas, pero no tienen ningún significado semántico. Java proporciona tres formas de comentarios. (5)

**constantes enteras octales y hexadecimales** Constantes enteras que pueden representarse en notación decimal, octal o hexadecimal. La notación octal se indica mediante un 0 prefijo; la notación hexadecimal se indica mediante el prefijo 0x o ox. (6)

**declaración de método** Esta compuesta por la cabecera y el cuerpo del método. (18)

**do, instrucción** Una estructura de bucle que garantiza que el bucle se ejecute al menos una vez. (15)

**entrada estándar** El terminal, a menos que se redirija. También hay flujos para la salida estándar y la salida estándar de error. (4)

**evaluación cortocircuitable** El proceso por el cual, si el resultado de un operador lógico puede determinarse examinando la primera expresión, entonces la segunda expresión no se evalúa. (11)

**for, instrucción** Una estructura de bucle utilizada principalmente para iteraciones simples. (14)

**identificador** Se emplea para denominar una variable o método. (7)

**if, instrucción** La instrucción fundamental para la implementación de la toma de decisiones. (12)

**instrucción nula** Una instrucción que está compuesta por solo un punto y coma. (13)

**java** El intérprete Java, que procesa código de bytes. (4)

**javac** El compilador Java; genera código de bytes. (4)

**main** El método especial que se invoca al ejecutarse el programa. (5)

**Máquina virtual** El intérprete del código de bytes. (4)

**método** El equivalente Java de una función. (18)

**operador condicional (?:)** Un operador que se utiliza en una expresión como abreviatura para instrucciones if-else sencillas. (18)

**operador de conversión de tipo** Un operador utilizado para generar una variable temporal sin nombre de un nuevo tipo. (10)

**operadores aritméticos binarios** Se utilizan para efectuar las operaciones aritméticas básicas. Java proporciona varios de estos operadores, incluyendo +, -, \*, / y %. (9)

**operadores de asignación** En Java, se utilizan para modificar el valor de una variable. Estos operadores incluyen =, +=, -=, \*= y /=. (8)

**operadores de autoincremento (++) y autodecremento (--)** Operadores que suman y restan 1, respectivamente. Existen dos formas de incremento y decremento: prefija y postfija. (10)

**operadores de igualdad** En Java, == y != se emplean para comparar dos valores; devuelven true o false (según sea apropiado). (11)

**operadores lógicos &&, || y !**, utilizados para simular los conceptos de AND, OR y NOT propios del álgebra booleana. (11)

**operadores relacionales** En Java, <, <=, > y >= se utilizan para decidir cuál de dos valores es menor o mayor. Devuelven true o false. (11)

**operadores unarios** Requieren un operando. Hay definidos varios operadores unarios, incluyendo el menos unario (-) y los operadores de autoincremento y autodecrecimiento (++ y --). (10)

**paso por valor** El mecanismo de paso de parámetros en Java mediante el cual se copia el argumento real en el parámetro formal. (19)

**return, instrucción** Una instrucción utilizada para devolver información al llamante. (19)

**secuencia de escape** Se utiliza para representar ciertas constantes de carácter. (7)

**signatura** La combinación del nombre del método y de los tipos de la lista de parámetros. El tipo de retorno no forma parte de la signatura. (19)

**sobrecarga de nombres de métodos** La acción de permitir que haya varios métodos con el mismo nombre, siempre y cuando los tipos de su lista de parámetros difieran. (19)

**static final, entidad** Una constante global. (20)

**static, método** Ocasionalmente utilizado para simular funciones estilo C; hablaremos más en detalle de este tipo de método en la Sección 3.6. (18)

**switch, instrucción** Una instrucción utilizada para seleccionar entre valores enteros pequeños. (17)

**tipos enteros** byte, char, short, int y long. (6)

**tipos primitivos** En Java, son los enteros, los de coma flotante, los booleanos y los de carácter. (6)

**Unicode** Conjunto internacional de caracteres que contiene más de 30.000 caracteres distintos, que cubren los lenguajes escritos más importantes. (6)

**while, instrucción** La forma más básica de bucle. (13)



## Errores comunes

1. Añadir caracteres innecesarios de punto y coma da lugar a errores lógicos, porque el punto y coma aislado equivale a la instrucción nula. Esto quiere decir que un punto y coma de más situado inmediatamente después de una instrucción for, while o if quedará muy probablemente sin detectar y hará que el programa falle.
2. En tiempo de compilación, el compilador Java debe detectar todos los casos en que un método que debe devolver un valor no lo hace. Ocasionalmente, proporciona una falsa alarma haciendo necesario reordenar el código.
3. Utilizar un 0 como prefijo hace que una constante entera sea interpretada como octal cuando se la analiza como símbolo sintáctico en el código fuente. Por tanto, 037 es equivalente al valor decimal 31.
4. Utilice && y || para las operaciones lógicas; & y | no son cortocircuitables.

- 5 La cláusula `else` se corresponde con el `if` no cerrado más próximo. Es un error bastante común olvidarse de incluir las llaves necesarias para que el `else` se corresponda con un `if` abierto distante.
- 6 Cuando se utiliza una instrucción `switch` es bastante común olvidarse de incluir la instrucción `break` entre los distintos casos lógicos. Si nos olvidamos de hacerlo, el control pasa al siguiente caso; generalmente, este no es el comportamiento deseado.
- 7 Las secuencias de escape comienzan con el carácter `\`, no con el carácter `/`.
- 8 Los símbolos de llaves que no se corresponden adecuadamente pueden generar resultados incorrectos. Utilice `Balance`, descrito en la Sección 11.1, para comprobar si esta es la causa de un mensaje de error del compilador.
- 9 El nombre del código fuente Java debe corresponderse con el nombre de la clase que se está compilando.



## Internet

A continuación se indican los archivos disponibles para este capítulo. Todo está autocontenido y nada de ello se utiliza posteriormente en el texto.

**FirstProgram.java** El primer programa, como se muestra en la Figura 1.1.

**OperatorTest.java** Ilustración de varios operadores, como se muestran en la Figura 1.3.

**MinTest.java** Ilustración de los métodos, como se muestra en la Figura 1.6.



## Ejercicios

### EN RESUMEN

- 1.1 ¿Cuáles son los ocho tipos primitivos en Java?
- 1.2 Describa los tres tipos de bucles existentes en Java.
- 1.3 ¿Qué extensiones se utilizan para los archivos fuente Java y para los archivos compilados?
- 1.4 ¿Qué es lo que hace la instrucción `continue`?
- 1.5 Describa el método de paso por valor.
- 1.6 Describa los tres tipos de comentarios utilizados en programas Java.
- 1.7 ¿Qué es la sobrecarga de métodos?
- 1.8 ¿Cuál es la diferencia entre los operadores `*` y `**`?
- 1.9 Describa todos los usos de una instrucción `break`. ¿Qué es una instrucción `break` etiquetada?
- 1.10 Explique la diferencia entre los operadores de incremento prefijo y postfixo.

**EN TEORÍA**

- 1.11** Para los siguientes fragmentos de código, proporcione un ejemplo en el que el bucle `for` de la izquierda no sea equivalente al bucle `while` de la derecha:

```

init;
for( inic; comprob; actualiza )
{
    instrucciones
}
while( comprob )
{
    instrucciones
    actualizacion;
}

```

- 1.12** Para el siguiente programa, ¿cuáles son las posibles salidas?

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

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

- 1.13** Supongamos que `b` tiene el valor 7 y que `c` tiene el valor 12. ¿Cuál será el valor de `a`, `b` y `c` después de cada línea del siguiente fragmento de programa:

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

- 1.14** ¿Cuál es el resultado de `true || false && true`?

**EN LA PRÁCTICA**

- 1.15** Escriba un método estático que tome un año como parámetro y devuelva `false` si el año es bisiesto y `true` en caso contrario.
- 1.16** Escriba dos métodos estáticos. El primero debe devolver el máximo de cuatro enteros y el segundo debe devolver el máximo de cinco enteros.
- 1.17** Escriba un programa para generar las tablas de suma y multiplicación para números de un único dígito (la tabla que los alumnos de la escuela elemental están acostumbrados a ver).

- 1.18** Escriba un instrucción `while` que sea equivalente al siguiente fragmento `for`. ¿Para qué podría resultar esto útil?

```
for( : : )  
    instrucción
```

## PROYECTOS DE PROGRAMACIÓN

- 1.19** Suponga que queremos imprimir números entre corchetes formateados de la manera siguiente: [1][2][3], etc. Escriba un método que admita dos parámetros: `howMany`, que indique cuántos números hay que escribir y `lineLength`, que indique la longitud de la línea. El método debe imprimir los caracteres en líneas sucesivas desde 0 hasta `howMany` en el formato anterior, pero no debe imprimir más de `lineLength` caracteres en una línea. Asimismo, el método no debe imprimir un [ en una línea a menos que también quepa en ella el correspondiente ].
- 1.20** En el siguiente puzzle aritmético decimal, a cada una de las 10 letras diferentes se le asigna un dígito. Escriba un programa que encuentre todas las posibles soluciones (una de las cuales se muestra).

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	

- 1.21** Escriba un programa para determinar todas las parejas de enteros positivos,  $(a, b)$ , tales que  $a < b < 500$  y para los que  $(a^2 + b^2 + 1) / (ab)$  sea un entero.
- 1.22** Escriba un método que imprima la representación de su parámetro entero en números romanos. Por ejemplo, si el parámetro es 1998, la salida debe ser MCMXCVIII.



## Referencias

Parte del material estilo C de este capítulo se ha tomado de [5]. Puede encontrar la especificación completa del lenguaje Java en [2]. Entre los libros de introducción a Java podemos citar [1], [3] y [4].

1. G. Cornell y C. S. Horstmann, *Core Java 2 Volumes 1 and 2*, 8<sup>a</sup> ed., Prentice Hall, Upper Saddle River, NJ, 2008.
2. J. Gosling, B. Joy, G. Steele y G. Bracha, *The Java Language Specification*, 3<sup>a</sup> ed., Addison-Wesley, Reading, MA, 2006.
3. J. Lewis y W. Loftus, *Java Software Solutions*, 6<sup>a</sup> ed., Addison-Wesley, Boston, MA, 2008.
4. W. Savitch y F. M. Carrano, *Java: An Introduction to Problem Solving & Programming*, 5<sup>a</sup> ed., Prentice Hall, Upper Saddle River, NJ, 2009.
5. M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice Hall, Upper Saddle River, NJ, 1995.



# Capítulo 2

## Tipos de referencia

En el Capítulo 1 se han examinado los tipos primitivos de Java. Todos los tipos distintos de los ocho tipos primitivos son los *tipos de referencia*, incluyendo entidades de importancia como las cadenas, las matrices y los flujos de archivos. En este capítulo, vamos a ver

- Qué es un tipo de referencia y un valor de referencia.
- En qué difieren los tipos de referencia de los tipos primitivos.
- Ejemplos de tipos de referencia, incluyendo cadenas, matrices y flujos.
- Cómo se emplean las excepciones para señalizar el comportamiento erróneo.

### 2.1 ¿Qué es una referencia?

En el Capítulo 1 se han descrito los ocho tipos primitivos, junto con algunas de las operaciones que estos tipos pueden realizar. Todos los demás tipos en Java son tipos de referencia, incluyendo las cadenas, las matrices y los flujos de archivos. ¿Entonces, qué es una referencia? Una variable de referencia (que a menudo se abrevia designándola simplemente como *referencia*) en Java es una variable que almacena de alguna manera la dirección de memoria en la que un objeto reside.

Como ejemplo, en la Figura 2.1 se muestran dos objetos de tipo `Point`. Sucede, por azar, que estos objetos están almacenados en las posiciones de memoria 1000 y 1024, respectivamente. Para estos dos objetos, hay tres referencias: `point1`, `point2` y `point3`. Tanto `point1` como `point3` hacen referencia al objeto almacenado en la posición de memoria 1000; `point2` hace referencia al objeto almacenado en la posición de memoria 1024. Tanto `point1` como `point3` almacenan el valor 1000, mientras que `point2` almacena el valor 1024. Observe que las posiciones concretas, como por ejemplo 1000 y 1024, son asignadas por el sistema de tiempo de ejecución de manera completamente discrecional (cuando encuentra memoria disponible). Por tanto, estos valores no son útiles externamente como números. Sin embargo, el hecho de que `point1` y `point3` almacenen valores idénticos sí que resulta útil: quiere decir que están haciendo referencia al mismo objeto.

Una referencia se almacenará siempre en la dirección de memoria en la que resida el objeto, a menos que no esté haciendo actualmente referencia a ningún objeto. En ese caso, almacenará la *referencia nula*, `null`. Java no permite referencias a variables primitivas.

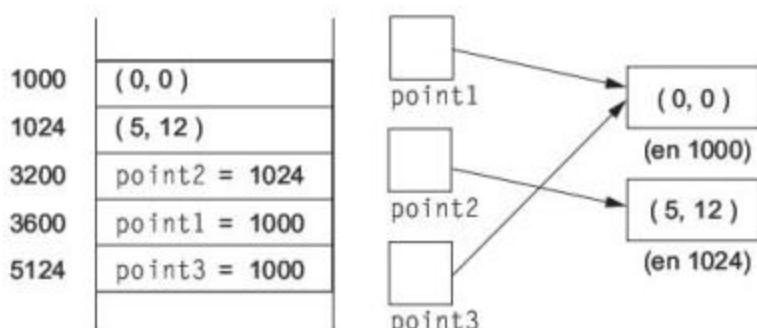


Figura 2.1 Ilustración de las referencias. El objeto Point almacenado en la posición de memoria 1000 está siendo referenciado tanto por point1 como por point3. El objeto Point almacenado en la posición de memoria 1024 está siendo referenciado por point2. Las posiciones de memoria en las que están almacenadas las variables son arbitrarias.

Hay dos categorías amplias de operaciones que se pueden aplicar a las variables de referencia. Una de ellas nos permite examinar o manipular el valor de referencia. Por ejemplo, si cambiamos el valor almacenado de point1 (que es 1000), podemos conseguir que haga referencia a otro objeto. También podemos comparar point1 y point3 y determinar si estamos haciendo referencia al mismo objeto. La otra categoría de operaciones se aplica al objeto que está siendo referenciado; quizás podamos examinar o cambiar el estado interno de uno de los objetos Point. Por ejemplo, podríamos examinar algunas de las coordenadas *x* e *y* de los objetos Point.

Antes de describir lo que podemos hacer con las referencias, veamos qué es lo que no está permitido. Considere la expresión point1\*point2. Puesto que los valores almacenados de point1 y point2 son 1000 y 1024, respectivamente, su producto sería 1024000. Sin embargo, este es un cálculo que no tiene ningún sentido y que no tendría ninguna aplicación. Las variables de referencia almacenan direcciones, y no existe ningún significado lógico que pueda asociarse con el hecho de multiplicar dos direcciones.

De forma similar, point1++ no tiene significado en Java; sugiere que point1 -1000- debería incrementarse a 1001, pero en ese caso no estaría haciendo referencia a un objeto Point válido. Muchos lenguajes (por ejemplo, C++) definen el *puntero*, que se comporta como una variable de referencia. Sin embargo, los punteros en C++ son mucho más peligrosos, porque se permiten operaciones aritméticas con las direcciones almacenadas. Por tanto, en C++, point1++ sí que tiene significado. Puesto que C++ permite punteros a tipos primitivos, es preciso tener cuidado a la hora de distinguir entre las operaciones aritméticas con las direcciones y las operaciones aritméticas con los objetos a los que se hace referencia. Esto se hace *des-referenciando* explícitamente el puntero. En la práctica, los punteros no seguros de C++ tienden a causar numerosos errores de programación.

Algunas operaciones se realizan sobre las propias referencias, mientras que otras operaciones se llevan a cabo sobre los objetos que están siendo referenciados. En Java, los únicos operadores permitidos para los tipos de referencia (con solo una excepción en el caso de String) son la asignación mediante = y la comparación de igualdad mediante == o !=.

La Figura 2.2 ilustra el operador de asignación para variables de referencia. Al asignar a point3 el valor almacenado de point2, obligamos a point3 a hacer referencia al mismo objeto que estaba siendo referenciado por point2. Ahora, point2==point3 será true, porque tanto point2 como point3 almacenan 1024 y hacen referencia por tanto al mismo objeto. point1!=point2 es también true porque point1 y point2 hacen referencia a objetos distintos.

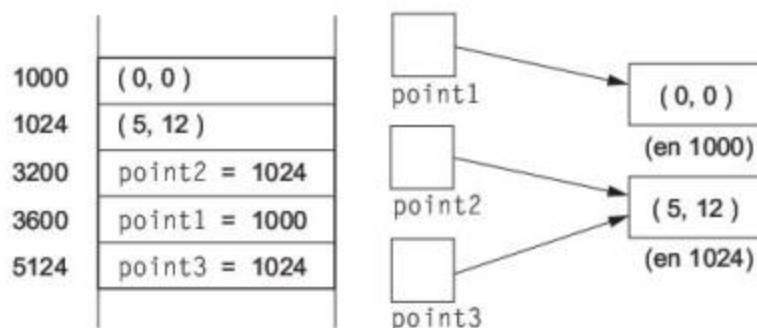


Figura 2.2 El resultado de `point3=point2`: `point3` hace ahora referencia al mismo objeto que `point2`.

La otra categoría de operaciones trata con el objeto que está siendo referenciado. Solo hay tres acciones básicas que se pueden llevar a cabo:

1. Aplicar una conversión de tipo (Sección 1.4.4).
2. Acceder a un campo interno o invocar un método mediante el operador punto (.) (Sección 2.2.1).
3. Utilizar el operador `instanceof` para verificar que el objeto almacenado es de un cierto tipo (Sección 3.6.3).

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

## 2.2 Conceptos básicos sobre objetos y referencias

En Java, un *objeto* es una instancia de cualquiera de los tipos no primitivos. Los objetos se tratan de forma distinta a los tipos primitivos. Los tipos primitivos, como ya hemos visto, se manejan mediante su *valor*, lo que quiere decir que los valores asumidos por las variables primitivas se almacenan en esas variables y se copian de una variable primitiva a otra durante las asignaciones. Como se muestra en la Sección 2.1, las variables de referencia almacenan referencias a objetos. El propio objeto está almacenado en algún otro lugar de la memoria, y la variable de referencia almacena la dirección en memoria del objeto. Por tanto, una variable de referencia representa simplemente un nombre para designar a esa parte de la memoria. Esto quiere decir que las variables primitivas y las variables de referencia se comportan de forma distinta. Esta sección examina dichas diferencias con más detalle e ilustra las operaciones permitidas para las variables de referencia.

En Java, un *objeto* es una instancia de cualquiera de los tipos no primitivos.

### 2.2.1 El operador punto (.)

El operador punto (.) se utiliza para seleccionar un método con el fin de aplicarlo a un objeto. Por ejemplo, suponga que tenemos un objeto de tipo `Circle` que define un método `area` para calcular

el área de un círculo. Si `theCircle` hace referencia a `Circle`, entonces podemos calcular el área del `Circle` referenciado (y guardarla en una variable de tipo `double`) de la forma siguiente:

```
double theArea = theCircle.area();
```

Es perfectamente posible que `theCircle` almacene la referencia `null`. En este caso, aplicar el operador punto generaría una excepción `NullPointerException` durante la ejecución del programa. Generalmente, esto provocará una terminación anormal del programa.

El operador punto también puede utilizarse para acceder a los componentes individuales de un objeto, siempre y cuando se hayan tomado medidas para permitir que los componentes internos sean visibles. En el Capítulo 3 se explica cómo conseguir esto. También se explica en el Capítulo 3 por qué es preferible, generalmente, no permitir el acceso directo a los componentes individuales.

### 2.2.2 Declaración de objetos

Ya hemos la sintaxis para declarar variables primitivas. En el caso de los objetos, hay una diferencia importante. Cuando declaramos una variable de referencia, estamos proporcionando simplemente un nombre que puede utilizarse para hacer referencia a un objeto almacenado en la memoria. Sin embargo, la declaración no proporciona por sí misma ningún objeto. Por ejemplo, supongamos que hay un objeto de tipo `Button` que queremos añadir a un `Panel` p existente utilizando el método `add` (todo esto se proporciona en la librería Java). Considere las instrucciones:

```
Button b; // b puede hacer referencia al objeto Button  
b.setLabel("No"); // Etiquetar el botón b con "No"  
p.add(b); // y añadirlo al Panel p
```

Todo parece ser correcto con estas instrucciones hasta que recordamos que `b` es el nombre de algún objeto `Button`, pero que no se ha creado todavía ningún `Button`. Como resultado, después de la declaración de `b`, el valor almacenado por la variable de referencia `b` es `null`, lo que quiere decir que `b` no está todavía haciendo referencia a un objeto `Button` válido. En consecuencia, la segunda línea es ilegal, porque estamos intentando modificar un objeto que aun no existe. En este escenario, el compilador detectaría probablemente el error y nos indicaría que “`b` no está inicializado”. En otros casos, el compilador no se dará cuenta, y algún error de tiempo de ejecución provocará la aparición del críptico mensaje de error `NullPointerException`.

Cuando se declara un tipo de referencia, no se crea ningún objeto. En ese momento, la referencia es a `null`. Para crear el objeto, utilice `new`.

La palabra clave `new` se utiliza para construir un objeto.

La forma (la única forma común) de crear un objeto es utilizar la palabra clave `new`. `new` se emplea para construir un objeto. Una forma de hacer esto sería la siguiente:

```
Button b; // b puede hacer referencia al objeto Button  
b = new Button(); // Ahora b hace referencia a un objeto ya creado  
b.setLabel("No"); // Etiquetar el botón b con "No"  
p.add(b); // y añadirlo al Panel p
```

Observe que los paréntesis son necesarios después del nombre del objeto.

También se pueden combinar la declaración y la construcción del objeto, como en

```
Button b = new Button();
b.setLabel("No");           // Etiquetar el botón b con "No"
p.add(b);                  // y añadirlo al Panel p
```

Los paréntesis son obligatorios cuando se utiliza `new`.

Muchos objetos pueden también construirse con valores iniciales. Por ejemplo, sucede que `Button` puede construirse con una `String` que especifique la etiqueta del botón:

```
Button b = new Button("No");
p.add(b); // Añadirlo al Panel p
```

La construcción puede especificar un estado inicial del objeto.

### 2.2.3 Recolección de basura

Puesto que todos los objetos deben ser construidos, cabría esperar que sea necesario destruirlos explícitamente una vez que ya no son necesarios. En Java, cuando un objeto construido ya no está siendo referenciado por ninguna variable de objeto, la memoria que consume es reclamada automáticamente, con lo que pasa a estar disponible para ser utilizada de nuevo. Esta técnica se conoce con el nombre de *recolección de basura*.

Java utiliza un mecanismo de recolección de basura. Con la recolección de basura, la memoria no referenciada se redrama automáticamente.

El sistema de tiempo de ejecución (es decir, la Máquina Virtual Java) garantiza que un objeto no sea nunca reclamado mientras sea posible acceder a él mediante una referencia o una cadena de referencias. Una vez que el objeto deje de ser alcanzable mediante una cadena de referencias, puede ser reclamado a discreción del sistema de tiempo de ejecución, en caso de que haya poca memoria. Si la memoria no escasea, es perfectamente posible que la máquina virtual no intente reclamar esos objetos.

### 2.2.4 El significado de =

Suponga que tenemos dos variables primitivas `lhs` y `rhs`, donde `lhs` quiere decir *lado izquierdo* y `rhs` quiere decir *lado derecho*. Entonces la instrucción de asignación

`lhs = rhs;`

```
lhs = rhs;
```

tiene un significado muy sencillo. El valor almacenado en `rhs` se almacena en la variable primitiva `lhs`. Los siguientes cambios que se realicen en `lhs` o `rhs` no afectarán a la otra variable.

Para los objetos, el significado de `=` es el mismo: se copian los valores almacenados. Si `lhs` y `rhs` son referencias (de tipos compatibles), entonces después de la instrucción de asignación, `lhs` hará referencia al mismo objeto que `rhs`. Aquí, lo que se está copiando es una dirección. El objeto al que antes hacía referencia `lhs` ya no estará siendo referenciado por `lhs`. Si `lhs` era la única referencia a dicho objeto, entonces ese objeto no estará siendo ya referenciado por nadie y podrá ser sometido al mecanismo de recolección de basura. Observe que los objetos no se copian.

Para los objetos, `=` es una asignación de referencia, en lugar de una copia de objetos.

He aquí algunos ejemplos. En primer lugar, suponga que queremos dos objetos `Button`. Suponga también que intentamos obtenerlos creando primero el objeto `noButton`. Después intentamos crear `yesButton` modificando `noButton` de la forma siguiente:

```
Button noButton = new Button( "No" );
Button yesButton = noButton;
yesButton.setLabel( "Yes" );
p.add( noButton );
p.add( yesButton );
```

Esto no funciona porque solo se ha construido un objeto `Button`. Por tanto, la segunda instrucción simplemente indica que `yesButton` es ahora otro nombre para el objeto `Button` que hemos construido en la línea 1. Ese `Button` construido es conocido ahora mediante dos nombres. En la línea 3, se cambia la etiqueta del objeto `Button` construido a `Yes`, pero esto quiere decir que ese único objeto `Button` que es conocido por dos nombres distintos, estará ahora etiquetado con `Yes`. Las dos últimas líneas añaden ese objeto `Button` al `Panel` p dos veces.

El hecho de que `yesButton` nunca hiciera referencia a su propio objeto no tiene ninguna importancia en este ejemplo. El problema es la asignación. Considere el siguiente fragmento de programa.

```
Button noButton = new Button( "No" );
Button yesButton = new Button( );
yesButton = noButton;
yesButton.setLabel( "Yes" );
p.add( noButton );
p.add( yesButton );
```

Las consecuencias son las mismas. Aquí, se han construido dos objetos `Button`. Al final de la secuencia, el primer objeto está siendo referenciado tanto por `noButton` como por `yesButton`, mientras que al segundo objeto no le referencia nadie.

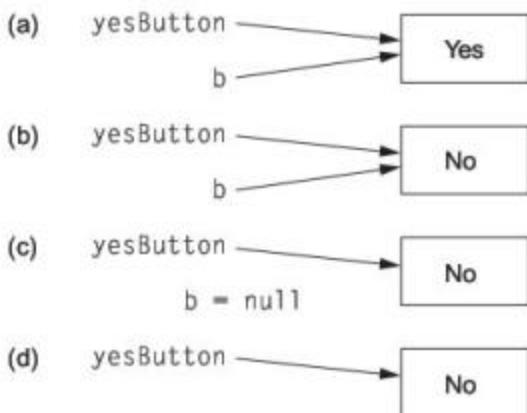
A primera vista, el hecho de que los objetos no puedan copiarse parece una grave limitación. Pero en la práctica no lo es, aunque sí es cierto que hace falta acostumbrarse a esta característica del lenguaje. Algunos objetos necesitan ser copiados. Para ellos, debe utilizarse el método `clone`, si es que hay uno disponible. Sin embargo, no vamos a utilizar `clone` en este texto.

## 2.2.5 Paso de parámetros

Gracias al paso por valor, los argumentos reales se envían dentro de los parámetros formales, utilizando la asignación normal. Si el parámetro es un tipo de referencia, entonces sabemos que

la asignación normal quiere decir que el parámetro formal hace ahora referencia al mismo objeto que el argumento real. Cualquier método aplicado al parámetro formal se estará también aplicando al argumento real. En otros lenguajes, esto se conoce con el nombre de *paso de parámetros por referencia*. Sin embargo, utilizar esta terminología para Java sería algo confuso, porque parece implicar que el paso de parámetros es distinto, cuando en realidad el

El *paso por valor* indica que para los tipos de referencia, el parámetro formal hace referencia al mismo objeto que el argumento real.



**Figura 2.3** El resultado del paso por valor. (a) `b` es una copia de `yesButton`; (b) después de `b.setLabel("No")`: los cambios en el estado del objeto referenciado por `b` se verán reflejados en el objeto referenciado por `yesButton`, porque se trata del mismo objeto; (c) después de `b=null`: el cambio en el valor de `b` no afecta al valor de `yesButton`; (d) después de que el método vuelva, `b` cae fuera de ámbito.

paso de parámetros no ha cambiado; más bien son los parámetros los que han cambiado, de tipos de no referencia a tipos de referencia.

Como ejemplo, suponga que pasamos `yesButton` como parámetro a la rutina `clearButton` que se define de la forma siguiente:

```
public static void clearButton( Button b )
{
    b.setLabel( "No" );
    b = null;
}
```

Entonces, como muestra la Figura 2.3, `b` hace referencia al mismo objeto que `yesButton`, y los cambios realizados en el estado de este objeto por los métodos invocados a través de `b` serán visibles cuando se vuelva de ejecutar `clearButton`. Los cambios en el valor de `b` (es decir, los cambios que provoquen que `b` haga referencia a otros objetos) no tendrán ningún efecto sobre `yesButton`.

## 2.2.6 El significado de ==

Para los tipos primitivos, `==` es `true` si los valores almacenados son idénticos. Para los tipos de referencia su significado es diferente, pero completamente coherente con las explicaciones anteriores.

Dos tipos de referencia son iguales según `==` si hacen referencia al mismo objeto almacenado (o si ambos son `null`). Considere, por ejemplo, lo siguiente:

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

Para los tipos de referencia,  
— es `true` solo si las  
dos referencias hacen  
referencia al mismo objeto.

Aquí, tenemos dos objetos. El primero se conoce con el nombre de `a` y el segundo se conoce con dos nombres: `b` y `c`. `b==c` es `true`. Sin embargo, aun cuando `a` y `b` están haciendo referencia a objetos que parecen tener el mismo valor, `a==b` es `false`, ya que hacen referencia a objetos diferentes. Se aplican reglas similares a `!=`.

El método `equals` puede utilizarse para ver si dos referencias hacen referencia a objetos que tengan estados idénticos.

En ocasiones, es importante conocer si son idénticos los estados de los objetos a los que se está haciendo referencia. Todos los objetos pueden compararse utilizando `equals`, pero para muchos objetos (incluyendo `Button`) `equals` devuelve `false` a menos que las dos referencias estén haciendo referencia al mismo objeto (en otras palabras, para algunos objetos, `equals` es exactamente lo mismo que la comprobación `=`). Veremos un ejemplo de dónde resulta `equals` útil cuando hablamos del tipo `String` en la Sección 2.3.

### 2.2.7 No hay sobrecarga de operadores para los objetos

Salvo por la única excepción descrita en la siguiente sección, no pueden definirse nuevos operadores como `+`, `-`, `*` y `/` para manejar objetos. Por tanto, no hay disponible ningún operador `<` para ningún objeto. En su lugar, deberá definirse para ese tipo de comparación un método nominado, como por ejemplo `lessThan`.

## 2.3 Cadenas

El tipo `String` se comporta como un tipo de referencia.

Las cadenas en Java se manejan con el tipo de referencia `String`. El lenguaje hace parecer que el tipo `String` es un tipo primitivo, porque proporciona los operadores `+` y `+=` para la concatenación. Sin embargo, este es el único tipo de referencia para el que se permite una sobrecarga de operadores. Por lo demás, `String` se comporta como cualquier otro tipo de referencia.

### 2.3.1 Conceptos básicos sobre manipulación de cadenas

Las cadenas de caracteres son inmutables; es decir, un objeto `String` no será modificado.

Hay dos reglas fundamentales acerca de los objetos `String`. En primer lugar, con la excepción de los operadores de concatenación, se comportan como un objeto. En segundo lugar, un objeto `String` es *inmutable*. Esto quiere decir que, una vez que se ha construido un objeto `String`, su contenido no puede modificarse.

Puesto que un objeto `String` es inmutable, se puede usar sin problemas el operador `=`. Por tanto, podemos declarar un objeto `String` de la forma siguiente:

```
String empty = "";
String message = "Hello";
String repeat = message;
```

Después de estas declaraciones habrá dos objetos `String`. El primero será la cadena vacía, que está referenciada por `empty`. El segundo será el objeto `String` `"Hello"`, que estará referenciado

tanto por `message` como por `repeat`. Para la mayoría de los objetos, estar referenciado tanto por `message` como por `repeat` podría dar problemas. Sin embargo, como las cadenas de caracteres son inmutables, la compartición de objetos `String` es segura, además de eficiente. La única forma de cambiar el valor de la cadena a la que `repeat` hace referencia es construir un nuevo objeto `String` y obligar a `repeat` a hacer referencia a él. Esto no tiene ningún efecto sobre el objeto `String` referenciado por `message`.

### 2.3.2 Concatenación de cadenas

Java no permite la sobrecarga de operadores para los tipos de referencia. Sin embargo, en el caso de la concatenación de cadenas existe una excepción en el lenguaje.

El operador `+`, cuando al menos uno de sus operandos es un objeto `String`, realiza la concatenación. El resultado es una referencia a un objeto `String` de nueva construcción. Por ejemplo,

```
"this" + " that2" // Genera "this that"  
"abc" + 5         // Genera "abc5"  
5 + "abc"        // Genera "5abc"  
"a" + "b" + "c"  // Genera "abc"
```

La concatenación de cadenas se realiza con `+` (`y +=`).

Las cadenas de caracteres de un solo carácter no deberían sustituirse por constantes de carácter; en el Ejercicio 2.8 le pedimos que demuestre por qué. Observe que el operador `+` es asociativo a la izquierda y por tanto

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

Asimismo, también se proporciona el operador `+=` para `String`. El efecto de `str+=exp` es igual que el de `str= str+exp`. Específicamente, esto quiere decir que `str` hará referencia al objeto `String` de nueva construcción generado por `str+exp`.

### 2.3.3 Comparación de cadenas

Puesto que el operador básico de asignación funciona para objetos `String`, resulta tentador suponer que también funcionan los operadores relacionales y de igualdad. Sin embargo, esto no es así.

Utilice `equals` y `compareTo` para comparar cadenas.

De acuerdo con la prohibición de sobrecarga de operadores, los operadores relacionales (`<`, `>`, `<-` y `>-`) no están definidos para el tipo `String`. Además, `-` y `!=` tienen el significado típico para las variables de referencia. Para dos objetos `String` `lhs` y `rhs`, por ejemplo, `lhs - rhs` será `true` solo si `lhs` y `rhs` hacen referencia al mismo objeto `String`. Por tanto, si hacen referencia a objetos distintos que tienen un contenido idéntico, `lhs == rhs` será `false`. Para el caso de `!=` se aplica una lógica similar.

Para comparar la igualdad de dos objetos `String`, utilizamos el método `equals`. `lhs.equals(rhs)` será `true` si `lhs` y `rhs` hacen referencia a objetos `String` que almacenan valores idénticos.

Se puede llevar a cabo una comprobación más general mediante el método `compareTo`. `lhs.compareTo(rhs)` compara dos objetos `String`, `lhs` y `rhs`. Devuelve un número negativo, cero, o positivo, dependiendo de si `lhs` es lexicográficamente menor, igual o mayor que `rhs`, respectivamente.

### 2.3.4 Otros métodos `String`

La longitud de un objeto `String` (una cadena de caracteres vacía tiene longitud cero) se puede obtener mediante el método `length`. Puesto que `length` es un método, es necesario utilizar paréntesis.

Utilice `length`, `charAt` y `substring` para calcular la longitud de una cadena, extraer un solo carácter y extraer una subcadena, respectivamente.

Hay definidos dos métodos para acceder a los caracteres individuales de un objeto `String`. El método `charAt` obtiene un único carácter especificando una posición (la primera posición es la posición 0). El método `substring` devuelve una referencia a un objeto `String` de nueva construcción. La llamada a este método se realiza especificando el punto de inicio y la primera posición no incluida.

He aquí un ejemplo de estos tres métodos:

```
String greeting = "hello";
int len = greeting.length(); // len es 5
char ch = greeting.charAt( 1 ); // ch es 'e'
String sub = greeting.substring( 2, 4 ); // sub es "ll"
```

### 2.3.5 Conversión de otros tipos a cadenas de caracteres

`toString` convierte tipos primitivos (y objetos) a objetos `String`.

La concatenación de cadenas proporciona una forma simple de convertir cualquier valor primitivo a un objeto `String`. Por ejemplo, `""+45.3` devuelve el objeto `String` de nueva construcción `"45.3"`. También existen métodos para hacer esto directamente.

El método `toString` se puede utilizar para convertir cualquier tipo primitivo en un objeto `String`. Por ejemplo, `Integer.toString(45)` devuelve una referencia al objeto `String` de nueva construcción `"45"`. Todos los tipos de referencia proporcionan también una implementación de `toString` de calidad variable. De hecho, cuando el operador `+` solo tiene un argumento de tipo `String`, el argumento que no es una cadena de caracteres se convierte en un objeto `String` aplicándole automáticamente un método `toString` apropiado. Para los tipos enteros, una forma alternativa de `Integer.toString` permite la especificación de una base de numeración. Así,

```
System.out.println( "55 in base 2: " + Integer.toString( 55, 2 ) );
```

imprime la representación binaria de 55.

El valor `int` representado por el objeto `String` puede obtenerse invocando el método `Integer.parseInt`. Este método genera una excepción si el objeto `String` no representa un valor `int`. Las excepciones se explican en la Sección 2.5. Se pueden aplicar conceptos similares a los valores `double`. He aquí algunos ejemplos:

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

## 2.4 Matrices

Un *agregado* es una colección de entidades almacenadas en una unidad. Una *matriz* es el mecanismo básico para almacenar una colección de entidades de tipo idéntico. En Java, la matriz no es un tipo primitivo. En lugar de ello, se comporta de forma bastante similar a un objeto. Por tanto, muchas de las reglas aplicables a los objetos también se aplican a las matrices.

Se puede acceder a cada entidad de la matriz mediante el *operador de indexación de matriz* `[]`. Decimos que el operador `[]` indexa la matriz, lo que quiere decir que especifica a qué objeto hay que acceder. A diferencia de lo que sucede en C y C++, la comprobación de los límites se realiza automáticamente.

En Java, las matrices siempre se indexan comenzando por cero. Por tanto, una matriz `a` de tres elementos almacenará `a[0]`, `a[1]` y `a[2]`. El número de elementos que se puede almacenar en una matriz `a` puede obtenerse siempre mediante `a.length`. Observe que no se usan paréntesis. Un bucle típico para una matriz utilizaría

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

Una matriz almacena una colección de entidades de tipo idéntico.

El operador de indexación de matriz `[]` proporciona acceso a cualquier objeto de la matriz.

Las matrices se indexan comenzando por cero. El número de elementos almacenados en la matriz se obtiene mediante el campo `length`. No se utilizan paréntesis.

### 2.4.1 Declaración, asignación y métodos

Una matriz es un objeto, por lo que cuando se formula la declaración de la matriz

```
int [ ] array1;
```

todavía no habrá ninguna memoria asignada para almacenar la matriz. `array1` es simplemente un nombre (referencia) para una matriz y en este punto su valor es `null`. Por ejemplo, para disponer de 100 valores `int`, utilizaríamos `new`:

```
array1 = new int [ 100 ];
```

Para asignar memoria a una matriz, utilice `new`.

Ahora, `array1` hará referencia a una matriz de 100 valores `int`.

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

```
int [ ] array2 = new int [ 100 ];
```

resultaría aceptable. Asimismo, pueden utilizarse listas de inicializadores, como en C o C++, para especificar los valores iniciales. En el siguiente ejemplo, se construye una matriz de cuatro enteros y luego se hace referencia a ella mediante `array3`.

```
int [ ] array3 = { 3, 4, 10, 6 };
```

Los corchetes pueden ir antes o después del nombre de la matriz. Colocarlos antes hace que sea más fácil ver que ese nombre es un tipo de matriz, de modo que ese es el estilo que utilizaremos aquí. Para declarar una matriz de tipos de referencia (en vez de tipos primitivos) se utiliza la misma sintaxis. Observe, sin embargo, que cuando creamos una matriz de tipos de referencia, cada referencia inicialmente almacena un valor `null`. Asimismo, cada una de esas referencias deberá ser configurada para hacer referencia a un objeto construido. Por ejemplo, una matriz de cinco botones se construiría de la forma siguiente:

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

La Figura 2.4 ilustra el uso de las matrices en Java. El programa de la Figura 2.4 elige repetidamente números comprendidos entre 1 y 100, ambos inclusive. La salida es el número de veces que se ha seleccionado cada número. La directiva `import` de la línea 1 se explicará en la Sección 3.8.1.

Asegúrese siempre de declarar el tamaño correcto de las matrices. Los errores de una unidad en el dimensionamiento son bastante comunes.

Los elementos de las matrices se inicializan con cero para los tipos primitivos y con `null` para las referencias.

El resto del programa es relativamente simple. Utiliza el objeto `Random` definido en la librería `java.util` (de aquí la directiva `import` de la línea 1). El método `nextInt` proporciona repetidamente un número (hasta cierto punto) aleatorio en el rango que va desde cero hasta uno menos que el parámetro pasado a `nextInt`; por tanto, sumando 1, obtendremos un número dentro del rango deseado. Los resultados se imprimen en las líneas 25 y 26.

Puesto que una matriz es un tipo de referencia, —no copia matrices. En lugar de ello, si `lhs` y `rhs` son matrices, el efecto de

```
int [ ] lhs = new int [ 100 ];
int [ ] rhs = new int [ 100 ];
...
lhs = rhs;
```

es que el objeto matriz que estaba siendo referenciado por `rhs` estará ahora referenciado también por `lhs`. Por tanto, modificar `rhs[0]` también hace que se modifique `lhs[0]`. (Para hacer que `lhs` sea una copia independiente de `rhs`, podría utilizarse el método `clone`, pero a menudo no hace falta realmente realizar copias completas.)

Por último, puede utilizarse una matriz como un parámetro de un método. Las reglas se deducen, desde el punto de vista lógico, del hecho de que un nombre de matriz es una referencia. Suponga que

La línea 14 declara un matriz de enteros que cuenta las veces que aparece cada número. Puesto que las matrices se indexan comenzando con cero, el `+1` es crucial si queremos acceder al elemento situado en la posición `DIFF_NUMBERS`. Sin él, tendríamos una matriz cuyo rango indexable iría de 0 a 99, por lo que cualquier acceso al índice 100 estaría fuera de límites. El bucle de las líneas 15 y 16 inicializa con el valor cero las entradas de la matriz; esto es, en realidad, innecesario, ya que de manera predeterminada

```
1 import java.util.Random;
2
3 public class RandomNumbers
4 {
5     // Generar números aleatorios (entre 1-100)
6     // Imprimir el número de apariciones de cada número
7
8     public static final int DIFF_NUMBERS = 100;
9     public static final int TOTAL_NUMBERS = 1000000;
10
11    public static void main( String [ ] args )
12    {
13        // Crear la matriz, inicializarla con valores 0
14        int [ ] numbers = new int [ DIFF_NUMBERS + 1 ];
15        for( int i = 0; i < numbers.length; i++ )
16            numbers[ i ] = 0;
17
18        Random r = new Random( );
19
20        // Generar los números
21        for( int i = 0; i < TOTAL_NUMBERS; i++ )
22            numbers[ r.nextInt( DIFF_NUMBERS ) + 1 ]++;
23
24        // Imprimir el resumen
25        for( int i = 1; i <= DIFF_NUMBERS; i++ )
26            System.out.println( i + ":" + numbers[ i ] );
27    }
28 }
```

Figura 2.4 Ilustración simple del uso de matrices.

tenemos un método `methodCall` que acepta una matriz de `int` como parámetro. Las vistas llamante/llamado serían

```
methodCall( actualArray );                                // llamada a método
void methodCall( int [ ] formalArray )                  // declaración de método
```

De acuerdo con los convenios de paso de parámetros para los tipos de referencia en Java, `formalArray` hace referencia al mismo objeto matriz que `actualArray`. Por tanto, `formalArray[i]` accede a `actualArray[i]`. Esto quiere decir que si el método modifica cualquier elemento de la matriz, esas modificaciones serán observables después de haberse completado la ejecución del método. Observe también que una instrucción como

El contenido de una matriz se pasa por referencia.

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

no tiene ningún efecto sobre `actualArray`. Finalmente, puesto que los nombres de matriz son simplemente referencias pueden ser devueltos por un método.

## 2.4.2 Expansión dinámica de matrices

La expansión dinámica de matrices nos permite construir matrices de tamaño arbitrario y hacerlas más grandes en caso necesario.

Suponga que queremos leer una secuencia de números y almacenarlos en una matriz para su procesamiento. La propiedad fundamental de una matriz requiere que declaremos un tamaño, para que el compilador pueda asignar la cantidad correcta de memoria. Asimismo, debemos hacer esta declaración antes de acceder por primera vez a la matriz. Si no tenemos ni idea de cuántos elementos podemos esperar, entonces es difícil elegir un tamaño razonable para la matriz. Esta sección muestra cómo expandir matrices si el tamaño inicial es demasiado pequeño. Esta técnica se denomina *expansión dinámica de matrices* y nos permite construir matrices de tamaño arbitrario y hacerlas más grandes o más pequeñas durante la ejecución del programa.

El método de construcción para las matrices que hemos visto hasta ahora es:

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

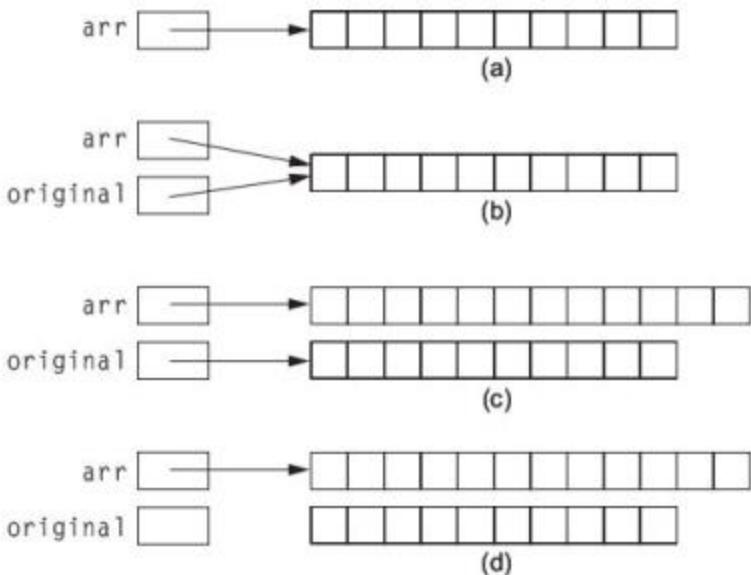
Suponga que decidimos después de las declaraciones, que en realidad necesitamos 12 valores `int` en lugar de 10. En este caso, podemos usar la siguiente técnica, que se ilustra en la Figura 2.5.

```
int [ ] original = arr;           // 1. Guardar referencia a arr
arr = new int [ 12 ];             // 2. Hacer que arr refiera más memoria
for( int i = 0; i < 10; i++ )    // 3. Copiar los datos antiguos
    arr[ i ] = original[ i ];
original = null;                 // 4. Desreferenciar la matriz original
```

Expanda siempre la matriz hasta un tamaño que sea múltiplo del anterior. Duplicar el tamaño de la matriz suele ser una buena elección.

Unos momentos de reflexión le permitirán convencerse de que se trata de una operación bastante cara en términos de procesamiento. Esto se debe a que tenemos que copiar todos los elementos de `original` a `arr`. Si esta expansión de la matriz fuera en respuesta, por ejemplo, a operaciones de lectura de entrada, sería bastante ineficiente volver a realizar la expansión cada vez que leemos unos cuantos elementos. Por tanto, cuando se implementa la expansión de matrices, lo que hacemos siempre es *multiplicar* su tamaño por una cierta constante. Por ejemplo, podríamos expandirla para que fuera el doble de grande. De esta forma, cuando expandamos la matriz de  $N$  elementos a  $2N$  elementos, el coste de las  $N$  copias se distribuye entre los siguientes  $N$  elementos que podrán insertarse en la matriz sin la necesidad de una nueva expansión.

Para concretar aun más las cosas, las Figuras 2.6 y 2.7 muestran un programa que lee un número ilimitado de cadenas de la entrada estándar y almacena el resultado en una matriz dinámicamente expansible. Se utiliza una línea vacía para indicar el final de la entrada. (Los mínimos detalles de E/S utilizados aquí no son importantes para este ejemplo y se explican en la Sección 2.6.) La rutina `resize` realiza la expansión (o la contracción) de la matriz, devolviendo una referencia a la



**Figura 2.5** Expansión de una matriz internamente: (a) en el momento inicial, `arr` representa 10 enteros; (b) después del paso 1, `original1` representa los mismos 10 enteros; (c) después de los pasos 2 y 3, `arr` representa 12 enteros, de los cuales los 10 primeros han sido copiados de `original1`; y (d) después del paso 4, los 10 enteros están disponibles para ser reclamados.

nueva matriz. De forma similar, el método `getStrings` devuelve (una referencia a) la matriz donde residirá.

Al principio de `getStrings`, se asigna a `itemsRead` el valor 0 y comenzamos con una matriz inicial de cinco elementos. Leemos repetidamente nuevos elementos en la línea 23. Si la matriz está llena, lo cual es indicado por la prueba de la línea 26, entonces la matriz se expande invocando a `resize`. Las líneas 42 a 48 realizan la expansión de la matriz utilizando la estrategia exacta que hemos esbozado anteriormente. En la línea 28, se asigna a la matriz el elemento de entrada real y se incrementa el número de elementos leídos. Si se produce un error en la entrada, simplemente detenemos el procesamiento. Finalmente, en la línea 36 contraemos la matriz para ajustarla al número de elementos leídos antes de volver del método.

### 2.4.3 ArrayList

La técnica utilizada en la Sección 2.4.2 es tan común que la librería Java contiene un tipo `ArrayList` con una funcionalidad integrada para implementarla. La idea básica es que un `ArrayList` mantiene no solo un tamaño, sino también una capacidad; la capacidad es la cantidad de memoria que ha reservado. La capacidad de `ArrayList` es realmente un detalle interno, no algo por lo que debamos preocuparnos.

El método `add` incrementa el tamaño en una unidad y añade a la matriz un nuevo elemento en la posición apropiada. Esta es una operación trivial si la capacidad máxima no se ha alcanzado. Si se ha hecho, la capacidad se expande automáticamente utilizando la estrategia descrita en la Sección 2.4.2. Los objetos `ArrayList` se inicializan con un tamaño igual a 0.

El tipo `ArrayList` se utiliza para expandir matrices.

El método `add` incrementa el tamaño en uno y añade un nuevo elemento a la matriz en la posición apropiada, expandiendo la capacidad en caso necesario.

```
1 import java.util.Scanner;
2
3 public class ReadStrings
4 {
5     // Leer un número ilimitado de objetos String; devolver un String [ ]
6     // Los mínimos detalles de E/S utilizados aquí no son importantes
7     // para este ejemplo y se explican en la Sección 2.6.
8     public static String [ ] getStrings( )
9     {
10         Scanner in = new Scanner( System.in );
11         String [ ] array = new String[ 5 ];
12         int itemsRead = 0;
13
14         System.out.println( "Enter strings, one per line: " );
15         System.out.println( "Terminate with empty line: " );
16
17         while( in.hasNextLine( ) )
18         {
19             String oneLine = in.nextLine( );
20             if( oneLine.equals( "" ) )
21                 break;
22             if( itemsRead == array.length )
23                 array = resize( array, array.length * 2 );
24             array[ itemsRead++ ] = oneLine;
25         }
26
27         return resize( array, itemsRead );
28     }
```

---

Figura 2.6 Código para leer un número ilimitado de objetos `String` e imprimirlas (parte 1).

Puesto que la indexación mediante `[]` está reservada solo para matrices primitivas, como era el caso en buena medida para objetos `String`, necesitamos utilizar un método para acceder a los elementos de un objeto `ArrayList`. El método `get` devuelve el objeto en el índice concreto y el método `set` puede utilizarse para modificar el valor de una referencia en un índice determinado; `get` se comporta por tanto como el método `charAt`. Describiremos los detalles de implementación de `ArrayList` en diversos puntos a lo largo del texto, y llegaremos incluso a escribir nuestra propia versión.

El código de la Figura 2.8 muestra cómo se utiliza `add` en `getStrings`; claramente, es mucho más simple que la función `getStrings` de la Sección 2.4.2. Como se muestra en la línea 19, el `ArrayList` especifica el tipo de objetos que almacena. Solo puede añadirse el tipo especificado al `ArrayList`; otros tipos provocarán un error de tiempo de compilación. Es importante mencionar, sin embargo, que a un `ArrayList` solo pueden añadirse objetos (a los que se accede mediante variables

```
29 // Redimensionar una matriz String[ ]; devolver una nueva matriz
30 public static String [ ] resize( String [ ] array,
31                               int newSize )
32 {
33     String [ ] original = array;
34     int numToCopy = Math.min( original.length, newSize );
35
36     array = new String[ newSize ];
37     for( int i = 0; i < numToCopy; i++ )
38         array[ i ] = original[ i ];
39     return array;
40 }
41
42 public static void main( String [ ] args )
43 {
44     String [ ] array = getStrings( );
45     for( int i = 0; i < array.length; i++ )
46         System.out.println( array[ i ] );
47 }
48 }
```

Figura 2.7 Código para leer un número ilimitado de objetos `String` e imprimirlas (parte 2).

de referencia). Los ocho tipos primitivos no pueden añadirse. Sin embargo, existe una solución fácil para este problema, de la que hablaremos en la Sección 4.6.2.

La especificación del tipo es una característica añadida en Java 5 que se conoce con el nombre de *genéricos*. Antes de Java 5, `ArrayList` no especificaba el tipo de los objetos y podía añadirse cualquier tipo al `ArrayList`. De cara a mantener la compatibilidad descendente, sigue permitiéndose no especificar el tipo de los objetos en la declaración de `ArrayList`, pero si se usa `ArrayList` de esta manera se obtendrá una advertencia de compilación, porque estamos impidiendo al compilador detectar las desadaptaciones de tipo y forzando a que esos errores sean detectados mucho más tarde por la Máquina Virtual, en el momento de ejecutarse realmente el programa. En las Secciones 4.6 y 4.8 se describen tanto el estilo antiguo como el nuevo.

#### 2.4.4 Matrices multidimensionales

En ocasiones, necesitamos acceder a las matrices utilizando más de un índice. Un ejemplo común serían las matrices bidimensionales. Una *matriz multidimensional* es una matriz a la que se accede mediante más de un índice. Se construye especificando el tamaño de sus índices y luego se accede a cada elemento colocando cada uno de los índices en su propio par de corchetes. Por ejemplo, la declaración

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

Una matriz multidimensional es una matriz a la que se accede empleando más de un índice.

```
1 import java.util.Scanner;
2 import java.util.ArrayList;
3
4 public class ReadStringsWithArrayList
5 {
6     public static void main( String [ ] args )
7     {
8         ArrayList<String> array = getStrings( );
9         for( int i = 0; i < array.size( ); i++ )
10             System.out.println( array.get( i ) );
11     }
12
13     // Leer un número ilimitado de String; devolver un ArrayList
14     // Los mínimos detalles de E/S utilizados aquí no son importantes
15     // para este ejemplo y se explican en la Sección 2.6.
16     public static ArrayList<String> getStrings( )
17     {
18         Scanner in = new Scanner( System.in );
19         ArrayList<String> array = new ArrayList<String>( );
20
21         System.out.println( "Enter any number of strings, one per line: " );
22         System.out.println( "Terminate with empty line: " );
23
24         while( in.hasNextLine( ) )
25         {
26             String oneLine = in.nextLine( );
27             if( oneLine.equals( "" ) )
28                 break;
29
30             array.add( oneLine );
31         }
32
33         System.out.println( "Done reading" );
34         return array;
35     }
36 }
```

---

Figura 2.8 Código para leer un número ilimitado de objetos String utilizando un ArrayList.

define la matriz bidimensional *x*, en la que el primer índice (que representa el número de filas) va de 0 a 1 y el segundo índice (el número de columnas) va de 0 de 2 (lo que nos da un total de seis valores enteros). Se reservan seis posiciones de memoria para esos valores enteros.

En el ejemplo anterior, la matriz bidimensional es en realidad una matriz de matrices. Como tal, el número de filas es `x.length`, lo que da un valor igual a 2. El número de columnas es `x[0].length` o `x[1].length`, siendo ambos valores iguales a 3.

La Figura 2.9 ilustra cómo imprimir el contenido de una matriz bidimensional. El código funciona no solo para matrices bidimensionales rectangulares, sino también para *matrices bidimensionales irregulares*, en las que el número de columnas varía de una fila a otra. Esto se maneja fácilmente utilizando `m[i].length` en la línea 11 para representar el número de columnas de la fila `i`. También se tiene en cuenta la posibilidad de que una fila pueda ser `null` (lo que es distinto de tener longitud 0), para lo cual se emplea la prueba de la línea 7. La rutina `main` ilustra la declaración de matrices bidimensionales para el caso en el que los valores iniciales son conocidos. Se trata simplemente de una extensión del caso unidimensional expuesto en la Sección 2.4.1. La matriz `a` es una matriz rectangular simple, la matriz `b` tiene una fila `null` y la matriz `c` es irregular.

```
1 public class MatrixDemo
2 {
3     public static void printMatrix( int [ ][ ] m )
4     {
5         for( int i = 0; i < m.length; i++ )
6         {
7             if( m[ i ] == null )
8                 System.out.println( "(null)" );
9             else
10            {
11                for( int j = 0; j < m[i].length; j++ )
12                    System.out.print( m[ i ][ j ] + " " );
13                System.out.println( );
14            }
15        }
16    }
17
18    public static void main( String [ ] args )
19    {
20        int [ ][ ] a = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
21        int [ ][ ] b = { { 1, 2 }, null, { 5, 6 } };
22        int [ ][ ] c = { { 1, 2 }, { 3, 4, 5 }, { 6 } };
23
24        System.out.println( "a: " ); printMatrix( a );
25        System.out.println( "b: " ); printMatrix( b );
26        System.out.println( "c: " ); printMatrix( c );
27    }
28 }
```

Figura 2.9 Impresión de una matriz bidimensional.

### 2.4.5 Argumentos de la Línea de comandos

Los argumentos de la línea de comandos están disponibles examinando el parámetro de `main`.

Hay disponibles argumentos de la línea de comandos para examinar el parámetro empleado con `main`. La matriz de cadenas de caracteres representa los argumentos adicionales de línea de comandos. Por ejemplo, cuando se invoca el programa con

```
java Echo this that
```

`args[0]` hace referencia al objeto `String "this"` y `args[1]` hace referencia al objeto `String "that"`. Por tanto, el programa de la Figura 2.10 simula el comando `echo` estándar.

### 2.4.6 Bucle for avanzado

Java 5 añade una nueva sintaxis que permite acceder a cada elemento de una matriz o `ArrayList` sin necesidad de utilizar índices de matriz. Su sintaxis es

```
for( tipo var : colección )
    instrucción
```

Dentro de `instrucción`, `var` representa el elemento actual de la iteración. Por ejemplo, para imprimir los elementos de `arr`, que tiene tipo `String[]`, podemos escribir:

```
for( String val : arr )
    System.out.println( val );
```

El mismo código funcionaría sin cambios si `arr` tuviera tipo `ArrayList<String>`, lo cual es una ventaja, porque sin el bucle `for` avanzado, sería necesario reescribir el código del bucle cuando cambiáramos el tipo y pasáramos de utilizar una matriz a emplear un `ArrayList`.

```
1 public class Echo
2 {
3     // Enumerar 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 arguments to echo" );
12    }
13 }
```

---

Figura 2.10 El comando echo.

El bucle `for` avanzado tiene algunas limitaciones. En primer lugar, en muchas aplicaciones necesitamos disponer del índice, especialmente si estamos haciendo cambios a los valores de la matriz (o `ArrayList`). En segundo lugar, el bucle `for` avanzado solo es útil si estamos accediendo a todos los elementos en orden secuencial. Si hay que excluir un elemento, es necesario emplear el bucle `for` estándar. Como ejemplos de bucles que no pueden reescribirse fácilmente utilizando el bucle avanzado tendríamos

```
for( int i = 0; i < arr1.length; i++ )
    arr1[ i ] = 0;

for( int i = 0; i < args.length - 1; i++ )
    System.out.println( args[ i ] + " " );
```

Además de permitir la interacción a través de matrices y de objetos `ArrayList`, el bucle `for` avanzado puede utilizarse con otros tipos de colecciones. Hablaremos de esa aplicación de esta nueva sintaxis en el Capítulo 6.

## 2.5 Tratamiento de excepciones

Las *excepciones* son objetos que almacenan información y que se transmiten fuera de la secuencia normal de retorno. Se propagan hacia atrás a través de la secuencia de invocaciones hasta que alguna rutina *captura* la excepción. En ese momento, puede extraerse la información almacenada en el objeto con el fin de realizar el tratamiento de errores.

Dicha información incluirá siempre detalles acerca de dónde se creó la excepción. La otra parte de información importante es el tipo del objeto excepción. Por ejemplo, cuando se propaga una excepción `ArrayIndexOutOfBoundsException` está claro que el problema básico es un índice incorrecto. Las excepciones se utilizan para señalizar *sucesos excepcionales*, como por ejemplo los errores.

Las excepciones se utilizan para manejar situaciones excepcionales como por ejemplo los errores.

### 2.5.1 Procesamiento de excepciones

El código de la Figura 2.11 ilustra el uso de las excepciones. El código que pudiera provocar la propagación de una excepción se encierra en un bloque `try`. El bloque `try` se extiende desde la línea 13 a la 17. Inmediatamente después del bloque `try` se encuentran las rutinas de tratamiento de excepciones. A esta parte del código solo se salta si se genera una excepción; en el punto donde se genera la excepción, el bloque `try` del que proviene la misma se considera terminado. Cada bloque `catch` (este código solo tiene uno) se va comprobando por orden, hasta que se encuentra una rutina de tratamiento adecuada. `parseInt` genera una excepción `NumberFormatException` si `oneLine` no es convertible a un valor `int`.

El código del bloque `catch` –en este caso la línea 18– se ejecuta en caso de que se produzca una excepción apropiada. Después, el bloque `catch` y

Un bloque `try` encierra un código que podría generar una excepción.

Un bloque `catch` procesa una excepción.

```

1 import java.util.Scanner;
2
3 public class DivideByTwo
4 {
5     public static void main( String [ ] args )
6     {
7         Scanner in = new Scanner( System.in );
8         int x;
9
10        System.out.println( "Enter an integer: " );
11        try
12        {
13            String oneLine = in.nextLine( );
14            x = Integer.parseInt( oneLine );
15            System.out.println( "Half of x is " + ( x / 2 ) );
16        }
17        catch( NumberFormatException e )
18        {
19            System.out.println( e );
20        }
21    }
22 }
```

**Figura 2.11** Programa simple para ilustrar las excepciones.

la secuencia `try/catch` se consideran terminados.<sup>1</sup> Se imprime un mensaje significativo a partir del objeto de excepción `e`. Alternativamente, podríamos incluir instrucciones adicionales de procesamiento y mensajes de error más detallados.

## 2.5.2 La cláusula `finally`

La cláusula `finally` siempre se ejecuta antes de completar un bloque, independientemente de las excepciones.

Con algunos objetos creados dentro de un bloque `try` es necesario realizar ciertas tareas de limpieza. Por ejemplo, puede que sea necesario cerrar los archivos abiertos dentro del bloque `try` antes de salir de dicho bloque. Un problema que tiene esto es que, si se genera un objeto excepción durante la ejecución del bloque `try`, las tareas de limpieza podrían llegar a omitirse, porque la excepción provocaría una salida inmediata del bloque `try`. Aunque podemos colocar esas instrucciones de limpieza inmediatamente después de la última cláusula `catch`, esto solo funcionará si la excepción es atrapada por una de las cláusulas `catch`. Y es posible que resulte difícil garantizar que esto sea así.

<sup>1</sup> Observe que tanto `try` como `catch` requieren un bloque y no solo una única instrucción. Por tanto, las llaves no son opcionales. Para ahorrar espacio, a menudo escribimos las cláusulas `catch` simples en una sola línea, junto con sus llaves, sangradas dos espacios adicionales, en lugar de emplear tres líneas. Posteriormente en el texto utilizaremos este estilo para los métodos de una sola línea.

En esta situación, lo que se hace es utilizar una cláusula `finally`, que puede incluirse después del último bloque `catch` (o del bloque `try` si no hay bloques `catch`). La cláusula `finally` está compuesta por la palabra clave `finally` seguida del bloque `finally`. Existen tres casos básicos.

1. Si el bloque `try` se ejecuta sin que se genere ninguna excepción, el control pasa al bloque `finally`. Esto es cierto incluso aunque el bloque `try` salga antes de la última instrucción mediante un `return`, `break` o `continue`.
2. Si se encuentra una excepción no capturada dentro del bloque `try`, el control pasa al bloque `finally`. Después, tras ejecutar el bloque `finally`, la excepción se propaga.
3. Si se encuentra una excepción capturada en el bloque `try`, el control pasa al bloque `catch` apropiado. Después, tras ejecutar el bloque `catch`, se ejecuta el bloque `finally`.

### 2.5.3 Excepciones comunes

En Java existen varios tipos de excepciones estándar. Las *excepciones estándar de tiempo de ejecución* incluyen sucesos tales como la división entera por cero y el acceso ilegal a una matriz. Puesto que estos sucesos pueden ocurrir prácticamente en cualquier parte, sería demasiado engorroso exigir que se escribieran rutinas de tratamiento de excepciones para ellos. Si se proporciona un bloque `catch`, estas excepciones se comportan como cualquier otra excepción. Si no se proporciona un bloque `catch` para excepción estándar y se genera una de estas excepciones, entonces la excepción se propagará de la forma usual, posiblemente más allá de `main`. En este caso, provocará una terminación anormal del programa con un mensaje de error. En la Figura 2.12 se muestran algunas de las excepciones estándar de tiempo de ejecución más comunes. En términos generales, se trata de errores de programación y no se debería intentar capturarlos. Una violación notable de este principio es `NumberFormatException`, aunque `NullPointerException` resulta más típica.

La excepciones de tiempo de ejecución no tienen que ser tratadas.

Excepción estándar de tiempo de ejecución	Significado
<code>ArithmaticException</code>	Desbordamiento o división entera por cero.
<code>NumberFormatException</code>	Conversión ilegal de <code>String</code> a tipo numérico.
<code>IndexOutOfBoundsException</code>	Índice ilegal a una matriz o <code>String</code> .
<code>NegativeArraySizeException</code>	Intento de crear una matriz de longitud negativa.
<code>NullPointerException</code>	Intento ilegal de utilizar una referencia <code>null</code> .
<code>SecurityException</code>	Violación de seguridad de tiempo de ejecución.
<code>NoSuchElementException</code>	Intento fallido de obtener el "siguiente" elemento.

Figura 2.12 Excepciones estándar de tiempo de ejecución comunes.

Las excepciones comprobadas deben ser tratadas o incluidas dentro de una cláusula `throws`.

La mayor parte de las excepciones pertenecen a la categoría de *excepciones comprobadas estándar*. Si se invoca un método que pueda generar directa o indirectamente una excepción comprobada estándar, entonces el programador debe proporcionar un bloque `catch` para ella, o indicar explícitamente que hay que propagar la excepción, incluyendo una cláusula `throws` en la declaración del método. Observe que esa excepción debe terminar procesándose en último término, porque constituye un estilo de programación terrible que `main` tenga una cláusula `throws`. En la Figura 2.13 se muestran algunas de las excepciones comprobadas estándar.

Los errores son excepciones no recuperables.

Los errores son problemas de la máquina virtual. El error más común es `OutOfMemoryError`. Otros errores comunes son `InternalError` y el infame `UnknownError`, en el que la máquina virtual ha decidido que tiene problemas y que no sabe por qué, pero no quiere continuar. En términos generales, un `Error` es irrecuperable y no debe ser capturado.

#### 2.5.4 Las cláusulas `throw` y `throws`

La cláusula `throw` se utiliza para generar una excepción.

El programador puede generar una excepción mediante el uso de la cláusula `throw`. Por ejemplo, podemos crear y luego generar un objeto `ArithmetricException` mediante

```
throw new ArithmetricException( "Divide by zero" );
```

Puesto que la intención es señalizar al llamante que existe un problema, nunca debe generarse una excepción solo para tratarla unas pocas líneas después dentro del mismo ámbito. En otras palabras, no incluye una cláusula `throw` dentro de un bloque `try` para luego tratarla inmediatamente en el bloque `catch` correspondiente. En lugar de ello, déjela sin tratar y pase la excepción al llamante. De otro modo, estaría utilizando las excepciones como una instrucción barata de salto, lo cual no constituye un buen estilo de programación y no es, ciertamente, para lo que están pensadas las excepciones –que lo que hacen es señalizar un suceso excepcional.

Java permite a los programadores crear sus propios tipos de excepción. En el Capítulo 4 se proporcionan los detalles sobre cómo crear y generar excepciones definidas por el usuario.

Como hemos mencionado anteriormente, las excepciones comprobadas estándar deben ser capturadas o propagadas explícitamente hacia la rutina llamante; como último recurso, deberían ser

Excepción comprobada estándar	Significado
<code>java.io.EOFException</code>	Encontrado fin de archivo antes de completar la entrada.
<code>java.io.FileNotFoundException</code>	No se ha encontrado el archivo para abrirlo.
<code>java.io.IOException</code>	Incluye la mayoría de las excepciones de E/S.
<code>InterruptedException</code>	Generada por el método <code>Thread.sleep</code> .

Figura 2.13 Excepciones comprobadas estándar comunes.

```
1 import java.io.IOException;
2
3 public class ThrowDemo
4 {
5     public static void processFile( String toFile )
6                         throws IOException
7     {
8         // La implementación omitida propaga hacia el llamante
9         // todas las excepciones IOException generadas
10    }
11
12    public static void main( String [ ] args )
13    {
14        for( String fileName : args )
15        {
16            try
17            {
18                processFile( fileName );
19            catch( IOException e )
20                {
21                    System.err.println( e );
22                }
23            }
24        }
25    }
26 }
```

Figura 2.14 Ilustración de la cláusula throws.

tratadas en `main`. Para propagar la excepción hacia el llamante, el método que no quiera capturar la excepción, deberá indicar mediante una cláusula `throws`, qué excepciones puede propagar.

La cláusula `throws` se incluye al final de la cabecera del método. La Figura 2.14 ilustra un método que propaga cualquier excepción de tipo `IOException` que se encuentre; estas excepciones deberán terminar siendo capturadas en `main` (puesto que no vamos a incluir una cláusula `throws` en `main`).

La cláusula `throws` indica las excepciones propagadas.

## 2.6 Entrada y salida

La *entrada y salida* (E/S) en Java se lleva a cabo utilizando el paquete `java.io`. Los tipos del paquete de E/S utilizan todos como prefijo `java.io`, incluyendo, como ya hemos visto, `java.io.IOException`. La directiva `import` permite no tener que utilizar los nombres completos. Por ejemplo, con

```
import java.io.IOException;
```

se puede utilizar `IOException` como abreviatura de `java.io.IOException` al principio del código. (Muchos tipos comunes, como `String` y `Math`, no requieren directivas `import`, ya que son automáticamente visibles a través de sus abreviaturas, gracias a que están incluidos en `java.lang`.)

La librería Java es muy sofisticada y tiene una gran variedad de opciones. Aquí vamos a examinar únicamente los usos más básicos, concentrándonos exclusivamente en la E/S formateada. En la Sección 4.5.3, hablaremos del diseño de la librería.

### 2.6.1 Operaciones básicas de flujos

Al igual que muchos lenguajes, Java utiliza para la E/S la noción de flujos. Para realizar la E/S hacia o desde el terminal, un archivo o a través de Internet, el programador crea un *flujo de datos* asociado. Una vez hecho eso, todos los comandos de E/S se dirigen hacia ese flujo de datos. El programador define un flujo para cada destino de E/S (por ejemplo, cada archivo que requiere entrada y salida).

Existen tres flujos predefinidos para la E/S de terminal: `System.in`, la entrada estándar; `System.out`, la salida estándar y `System.err`, la salida estándar de error.

Los flujos predefinidos son  
`System.in`, `System.out` y `System.err`.

Como ya hemos mencionado, los métodos `print` y `println` se utilizan para la salida formateada. Cualquier tipo puede convertirse a un objeto `String` adecuado para su impresión invocando a su método `toString`; en muchos casos, esto se hace automáticamente. A diferencia de C y C++, que tienen una cantidad enorme de opciones de formato, la salida en Java se realiza casi exclusivamente mediante concatenación de objetos `String`, sin ningún formato predefinido.

### 2.6.2 El tipo Scanner

El método más simple de leer una entrada con formato consiste en utilizar un `Scanner`. Un `Scanner` permite al usuario leer líneas de una en una mediante `nextLine`; leer objetos `String` de uno en uno utilizando `next` o leer tipos primitivos de uno en uno utilizando métodos como `nextInt` y `nextDouble`. Antes de intentar realizar una lectura, es habitual verificar que la lectura puede llevarse a cabo correctamente, utilizando métodos como `hasNextLine`, `hasNext`, `hasNextInt` y `hasNextDouble`, que proporcionan resultados de tipo `boolean`; debido a ello, normalmente hay menos necesidad de preocuparse por el tratamiento de excepciones. Cuando se emplea un `Scanner` es costumbre proporcionar la directiva de importación

```
import java.util.Scanner;
```

Para utilizar un `Scanner` que lea de la entrada estándar, primero tenemos que construir un objeto `Scanner` a partir de `System.in`. Esto se ilustró ya anteriormente en la Figura 2.11, en la línea 7. En la Figura 2.11, vemos que se emplea `nextLine` para leer un objeto `String` y que luego el objeto `String` se convierte en `int`. Teniendo en cuenta lo que hemos hablado sobre `Scanner` en el párrafo anterior, sabemos que existen otras opciones.

Una opción alternativa, que quizás sea la más limpia, sería la siguiente sustitución, que evita las excepciones por completo y utiliza `nextInt` y `hasNextInt`:

```
System.out.println("Enter an integer: ");
if( in.hasNextInt() )
{
```

```
    x = in.nextInt();
    System.out.println( "Half of x is " + ( x / 2 ) );
}
else
    { System.out.println("Integer was not entered." ) }
```

Utilizar las diversas combinaciones de `next` y `hasNext` de `Scanner` suele funcionar correctamente, aunque pueden presentarse algunas limitaciones. Por ejemplo, suponga que queremos leer dos enteros e imprimir el máximo.

La Figura 2.15 muestra una idea que resulta algo engorrosa si queremos llevar a cabo una apropiada comprobación de errores sin utilizar excepciones. Cada llamada a `nextInt` va precedida por una llamada a `hasNextInt`, alcanzándose al final una línea donde se imprime un mensaje de error, a menos que haya dos valores `int` disponibles en el flujo de entrada de datos estándar.

La Figura 2.16 muestra una alternativa que no utiliza llamadas a `hasNextInt`. En su lugar, las llamadas a `nextInt` generarán una excepción `NoSuchElementException` si el valor `int` no está disponible, y esto hace que el código parezca más limpio. El uso de la excepción es quizás una

```
1 import java.util.Scanner;
2
3 class MaxTestA
4 {
5     public static void main( String [ ] args )
6     {
7         Scanner in = new Scanner( System.in );
8         int x, y;
9
10        System.out.println( "Enter 2 ints: " );
11
12        if( in.hasNextInt( ) )
13        {
14            x = in.nextInt();
15            if( in.hasNextInt( ) )
16            {
17                y = in.nextInt();
18                System.out.println( "Max: " + Math.max( x, y ) );
19                return;
20            }
21        }
22
23        System.err.println( "Error: need two ints" );
24    }
25 }
```

Figura 2.15 Leer dos enteros e imprimir el máximo utilizando `Scanner` y sin recurrir a las excepciones.

```
1 class MaxTestB
2 {
3     public static void main( String [ ] args )
4     {
5         Scanner in = new Scanner( System.in );
6
7         System.out.println( "Enter 2 ints: " );
8
9         try
10        {
11             int x = in.nextInt( );
12             int y = in.nextInt( );
13
14             System.out.println( "Max: " + Math.max( x, y ) );
15         }
16         catch( NoSuchElementException e )
17         { System.err.println( "Error: need two ints" ); }
18     }
19 }
```

Figura 2.16 Leer dos enteros e imprimir el máximo utilizando Scanner y excepciones.

```
1 import java.util.Scanner;
2
3 public class MaxTestC
4 {
5     public static void main( String [ ] args )
6     {
7         Scanner in = new Scanner( System.in );
8
9         System.out.println( "Enter 2 ints on one line: " );
10        try
11        {
12            String oneLine = in.nextLine( );
13            Scanner str = new Scanner( oneLine );
14
15            int x = str.nextInt( );
16            int y = str.nextInt( );
17
18            System.out.println( "Max: " + Math.max( x, y ) );
19        }
20        catch( NoSuchElementException e )
21        { System.err.println( "Error: need two ints" ); }
22    }
23 }
```

Figura 2.17 Leer dos enteros de la misma línea e imprimir el máximo, utilizando dos objetos Scanner.

decisión razonable, porque en realidad se consideraría inusual que el usuario no introdujera dos enteros para este programa.

Sin embargo, ambas opciones están limitadas porque en muchos casos podríamos insistir en que los dos enteros se introdujeran en la misma línea de texto. Podríamos incluso insistir en que no hubiera ningún otro dato en dicha línea. En la Figura 2.17 se muestra una opción distinta. Podemos construir un Scanner proporcionando un objeto String. Así que podemos crear primero un Scanner a partir de System.in (línea 7) para leer una única línea (línea 12) y luego crear un segundo Scanner a partir de la única línea (línea 13) con el fin de extraer los dos enteros (líneas 15 y 16). Si algo va mal, el programa captura y trata una excepción NoSuchElementException.

El uso del segundo Scanner en la Figura 2.17 puede funcionar bien y resulta cómodo; sin embargo, si es importante asegurarse de que no haya más de dos enteros por línea, necesitaríamos código adicional. En particular, tendríamos que añadir una llamada a str.hasNext() y, si esa llamada devolviera un valor verdadero, sabríamos que hay un problema. Esto se ilustra en la Figura 2.18. Existen otras opciones, como por ejemplo el método split de String, tal y como se describe en los ejercicios.

```
1 class MaxTestD
2 {
3     public static void main( String [ ] args )
4     {
5         Scanner in = new Scanner( System.in );
6
7         System.out.println( "Enter 2 ints on one line: " );
8         try
9         {
10            String oneLine = in.nextLine( );
11            Scanner str = new Scanner( oneLine );
12
13            int x = str.nextInt( );
14            int y = str.nextInt( );
15
16            if( !str.hasNext( ) )
17                System.out.println( "Max: " + Math.max( x, y ) );
18            else
19                System.err.println( "Error: extraneous data on the line." );
20        }
21        catch( NoSuchElementException e )
22        { System.err.println( "Error: need two ints" ); }
23    }
24 }
```

Figura 2.18 Leer exactamente dos enteros de la misma línea e imprimir el máximo utilizando dos objetos Scanner.

### 2.6.3 Archivos secuenciales

FileReader se utiliza para la entrada desde archivo.

Una de las reglas básicas de Java es que lo que funciona para la E/S a través de terminal también funciona para los archivos. Para tratar con un archivo, no construimos ningún objeto `BufferedReader` a partir de un `InputStreamReader`. En lugar de ello, lo construimos a partir de un objeto `FileReader`, que a su vez puede ser construido proporcionando un nombre de archivo.

En la Figura 2.19 se muestra un ejemplo que ilustra estas ideas básicas. En ella, tenemos un programa que muestra el contenido de los archivos de texto especificados como argumentos de la línea de comandos. La rutina `main` simplemente recorre los argumentos de la línea de comandos, pasando cada uno de ellos a `listFile`. En `listFile`, construimos el objeto `FileReader` en la línea 22 y luego lo usamos para construir un objeto `Scanner`, `fileIn`. Llegados a ese punto, la lectura es idéntica al procedimiento que ya hemos visto.

```
1 import java.util.Scanner;
2 import java.io.FileReader;
3 import java.io.IOException;
4
5 public class ListFiles
6 {
7     public static void main( String [ ] args )
8     {
9         if( args.length == 0 )
10             System.out.println( "No files specified" );
11         for( String fileName : args )
12             listFile( fileName );
13     }
14
15     public static void listFile( String fileName )
16     {
17         Scanner fileIn = null;
18
19         System.out.println( "FILE: " + fileName );
20         try
21         {
22             fileIn = new Scanner( new FileReader( fileName ) );
23             while( fileIn.hasNextLine( ) )
24             {
25                 String oneLine = fileIn.nextLine( );
```

Continúa

---

Figura 2.19 Programa para mostrar el contenido de un archivo.

```
26         System.out.println( oneLine );
27     }
28 }
29 catch( IOException e )
30 {
31     System.out.println( e );
32 }
33 // Cerrar el flujo de datos
34 if( fileIn != null )
35     fileIn.close();
36 }
37 }
38 }
```

Figura 2.19 (Continuación).

Después de terminar con el archivo, debemos cerrarlo; en caso contrario, podríamos terminar quedándonos sin flujos de datos. Observe que esto no puede hacerse al final del bloque `try`, ya que una excepción podría provocar una salida prematura del bloque. Por tanto, cerramos el archivo en un bloque `finally`, que se garantiza que se ejecutará independientemente de si hay excepciones, y de si estas se tratan o no. El código para gestionar la instrucción `close` es complejo porque

1. `fileIn` debe declararse fuera del bloque `try` para que sea visible en el bloque `finally`.
2. `fileIn` debe inicializarse con `null` para evitar que el compilador se queje de la existencia de una variable posiblemente no inicializada.
3. Antes de llamar a `close`, tenemos que comprobar que `fileIn` no sea `null` para evitar generar una excepción `NullPointerException` (`fileIn` sería `null` si el archivo no fuera encontrado, lo que daría como resultado una excepción `IOException` antes de esta asignación).
4. En algunos casos, pero no en nuestro ejemplo, `close` puede generar por sí mismo una excepción comprobada, por lo que requeriría un bloque `try/catch` adicional.

La salida formateada a archivo es similar a la entrada desde archivo. `FileWriter`, `PrintWriter` y `println` sustituyen a `FileReader`, `Scanner` y `nextLine`, respectivamente. La Figura 2.20 ilustra un programa que escribe a doble espacio los archivos especificados en la línea de comandos (los archivos resultantes se colocan en un archivo con una extensión `.ds`).

`FileWriter` se utiliza para la salida a archivo.

Esta descripción de la E/S en Java, aunque es suficiente para realizar una E/S básica formateada, oculta un interesante diseño orientado a objetos del que hablaremos con más detalle en la Sección 4.5.3.

```
1 // Escribir a doble espacio los archivos especificados en la linea de comandos.
2
3 import java.io.FileReader;
4 import java.io.FileWriter;
5 import java.io.PrintWriter;
6 import java.io.IOException;
7 import java.util.Scanner;
8
9 public class DoubleSpace
10 {
11     public static void main( String [ ] args )
12     {
13         for( String fileName : args )
14             doubleSpace( fileName );
15     }
16
17     public static void doubleSpace( String fileName )
18     {
19         PrintWriter fileOut = null;
20         Scanner fileIn = null;
21
22         try
23         {
24             fileIn = new Scanner( new FileReader( fileName ) );
25             fileOut = new PrintWriter( new FileWriter( fileName + ".ds" ) );
26
27             while( fileIn.hasNextLine( ) )
28             {
29                 String oneLine = fileIn.nextLine( );
30                 fileOut.println( oneLine + "\n" );
31             }
32         }
33         catch( IOException e )
34         {
35             e.printStackTrace( );
36         }
37         finally
38         {
39             if( fileOut != null )
40                 fileOut.close( );
41             if( fileIn != null )
42                 fileIn.close( );
43         }
44     }
45 }
```

---

Figura 2.20 Programa para escribir archivos a doble espacio.

## Resumen

En este capítulo hemos examinado los tipos de referencia. Una *referencia* es una variable que almacena la dirección de memoria en la que un objeto reside o la referencia especial `null`. Solo puede hacerse referencia a objetos y cada objeto puede ser referenciado por varias variables de referencia. Cuando se comparan dos referencias mediante `=`, el resultado es `true` si ambas referencias se refieren al mismo objeto. De forma similar, `=` hace que una variable de referencia refiera a otro objeto. Solo hay disponibles unas cuantas operaciones adicionales. La más significativa es el operador punto, que permite seleccionar un método de un objeto o acceder a sus datos internos.

Puesto que solo hay ocho tipos primitivos, casi todas las cosas significativas en Java son objetos y se accede a ellas mediante referencias. Esto incluye los objetos de tipo `String`, las matrices, los objetos de excepción, los flujos de datos de archivo y los analizadores sintácticos de cadenas.

`String` es un tipo de referencia especial, porque pueden utilizarse los operadores `+` y `+=` para la concatenación. Por lo demás, un objeto `String` es como cualquier otra referencia; hace falta `equals` para comprobar si el contenido de dos objetos `String` son idénticos. Una *matriz* es una colección de valores con tipo idéntico. La matriz se indexa empezando por 0 y se garantiza la realización de comprobaciones del rango del índice. Las matrices se pueden expandir dinámicamente utilizando `new` para asignar una cantidad mayor de memoria y luego copiando los elementos individuales. Este proceso lo realizan automáticamente los objetos `ArrayList`.

Las *excepciones* se emplean para señalar sucesos excepcionales. Una excepción se señaliza mediante la cláusula `throw`; la excepción se propaga hasta ser tratada por un bloque `catch` asociado con un bloque `try`. Salvo por las excepciones de tiempo de ejecución y los errores, cada método debe señalizar las excepciones que pueda propagar utilizando una lista `throws`.

La entrada se gestiona mediante objetos `Scanner` y `FileReader`. En el siguiente capítulo veremos cómo diseñar nuevos tipos definiendo una `clase`.



## Conceptos clave

**agregado** Una colección de objetos almacenados en una unidad. (37)

**argumento de línea de comandos** Se accede a él mediante un parámetro de `main`. (46)

**ArrayList** Almacena una colección de objetos en formato de matriz, permitiendo fácilmente expandir la colección mediante el método `add`. (41)

**bloque catch** Utilizado para procesar una excepción. (47)

**bucle for avanzado** Añadido en Java 5, permite la iteración a través de una colección de elementos. ()

**concatenación de cadenas** Se lleva a cabo con los operadores `+` y `+=`. (35)

**construcción** Para los objetos, se realiza mediante la palabra clave `new`. (31)

**equals** Se utiliza para comprobar si los valores almacenados en dos objetos son iguales. (34)

**entrada y salida (E/S)** Se lleva a cabo utilizando el paquete `java.io`. (51)

**Error** Una excepción no recuperable. (50)

**excepción** Utilizada para tratar sucesos excepcionales, como por ejemplo los errores. (47)

**excepción comprobada** Debe ser atrapada o debe permitirse explícitamente que se propague utilizando una cláusula `throws`. (50)

**excepción de tiempo de ejecución** No es necesario tratarla. Como ejemplos podríamos citar `ArithmException` y `NullPointerException`. (49)

**expansión dinámica de matrices** Nos permite hacer más grandes las matrices en caso necesario. (40)

**FileReader** Usado para entrada desde archivo. (56)

**FileWriter** Usado para salida a archivo. (57)

**finally, cláusula** Siempre se ejecuta antes de salir de una secuencia `try/catch`. (48)

**immutable** Objeto cuyo estado no puede cambiar. Específicamente, los objetos `String` son inmutables. (34)

**java.io** Paquete utilizado para la E/S no trivial. (51)

**length, campo** Utilizado para determinar el tamaño de una matriz. (37)

**length, método** Utilizado para determinar la longitud de una cadena. (36)

**lhs y rhs** Hacen referencia al lado izquierdo y al lado derecho, respectivamente. (31)

**matriz** Almacena una colección de objetos de tipo idéntico. (37)

**matriz multidimensional** Una matriz a la que se accede mediante más de un índice. (43)

**new** Utilizado para construir un objeto. (30)

**null, referencia** El valor de una referencia a objeto que no hace referencia a ningún objeto. (30)

**NullPointerException** Se genera cada vez que se intenta aplicar un método a una referencia `null`. (30)

**objeto** Una entidad no primitiva. (29)

**operador de indexación de matriz** `[]` Proporciona acceso a cualquier elemento de la matriz. (37)

**operador punto (.)** Permite acceder a cada miembro del objeto. (29)

**paso por referencia** En muchos lenguajes de programación significa que el parámetro formal es una referencia al argumento real. Este es el efecto natural que se consigue en Java al utilizar el paso por valor con tipos de referencia. (32)

**recolección de basura** Reclamación automática de la memoria no referenciada. (31)

**Scanner** Se utiliza para llevar a cabo la entrada línea a línea. También se emplea para extraer líneas, cadenas y tipos primitivos a partir de una única fuente de caracteres, como por ejemplo un flujo de datos de entrada o un objeto `String`. Se encuentra en el paquete `java.util`. (52)

**String** Un objeto especial utilizado para almacenar una colección de caracteres. (34)

**System.in, System.out y System.err** Los flujos de datos de E/S predefinidos. (52)

**tipo de referencia** Cualquier tipo que no sea un tipo primitivo. (30)

**throw, cláusula** Utilizada para generar una excepción. (50)

**throws, cláusula** Indica que un método puede propagar una excepción. (51)

**toString, método** Convierte un objeto o un tipo primitivo a un objeto `String`. (36)

**try, bloque** Encierra código que podría generar una excepción. (47)



## Errores comunes

1. Para los tipos de referencia y las matrices, = no hace una copia de los valores de los objetos. En lugar de ello, hace una copia de las direcciones.
2. Para los tipos de referencia y las cadenas de caracteres es preciso utilizar `equals` en lugar de == para comprobar si dos objetos tienen un estado idéntico.
3. Los errores de una unidad durante la indexación son comunes en todos los lenguajes.
4. Los tipos de referencia se inicializan con `null` de forma predeterminada. No se puede construir ningún objeto sin invocar `new`. Un error de "variable de referencia no inicializada" o una excepción `NullPointerException` indica que nos hemos olvidado de asignar memoria al objeto.
5. En Java, las matrices se indexan de 0 a `N-1`, donde `N` es el tamaño de la matriz. Sin embargo, se efectúa una comprobación de rango, por lo que los accesos fuera de límites a una matriz se detectan en tiempo de ejecución.
6. Las matrices bidimensionales se indexan como `A[i][j]`, no `A[i,j]`.
7. Las excepciones comprobadas deben ser capturadas o se las debe permitir explícitamente programarse mediante una cláusula `throws`.
8. Utilice " " y no ' ' para imprimir un espacio en blanco.



## Internet

A continuación se indican los archivos disponibles para este capítulo. Todo está autocontenido y nada de ello se utiliza posteriormente en el texto.

**RandomNumbers.java**

Contiene el código del ejemplo de la Figura 2.4.

**ReadStrings.java**

Contiene el código de los ejemplos de las Figuras 2.6 y 2.7.

**ReadStringsWithArrayList.java**

Contiene el código del ejemplo de la Figura 2.8.

**MatrixDemo.java**

Contiene el código del ejemplo de la Figura 2.9.

**Echo.java**

Contiene el código del ejemplo de la Figura 2.10.

**ForEachDemo.java**

Ilustra el bucle `for` avanzado.

**DivideByTwo.java**

Contiene el código del ejemplo de la Figura 2.11.

**MaxTest.java**

Contiene el código de los ejemplos de las Figuras 2.15–2.18.

**ListFiles.java****DoubleSpace.java**

Contiene el código del ejemplo de la Figura 2.19.

Contiene el código del ejemplo de la Figura 2.20.



## Ejercicios

### EN RESUMEN

- 21** Indique cinco operaciones que puedan aplicarse a un tipo de referencia.
- 22** Describa cómo funcionan las excepciones en Java.
- 23** Enumere las principales diferencias entre los tipos de referencia y los tipos primitivos.
- 24** Explique el papel de `next` y `hasnext` de `Scanner`.
- 25** ¿Cuáles son las diferencias entre una matriz y un `ArrayList`?
- 26** Enumere las operaciones básicas que pueden llevarse a cabo con objetos de tipo `String`.

### EN TEORÍA

- 27** El bloque `finally` introduce complicaciones en la especificación del lenguaje Java. Escriba un programa para determinar qué valor es devuelto por `foo` y qué excepción es generada por `bar` en la Figura 2.21.
- 28** Si `x` e `y` tienen los valores 3 y 1, respectivamente, ¿cuál será la salida del siguiente fragmento de código?

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

### EN LA PRÁCTICA

- 29** Escriba un método que devuelva `true` si el objeto `String str1` es un sufijo del objeto `String str2`. No utilice ninguna de las rutinas generales de búsqueda en cadenas de caracteres, salvo `charAt`.
- 210** ¿Cuál es el error en el siguiente fragmento de código?

```
public static void resize( int [ ] arr )
{
    int [ ] old = arr;
    arr = new int[ old.length * 2 + 1 ];

    for( int i = 0; i < old.length; i++ )
        arr[ i ] = old[ i ];
}
```

```
public static void foo( )
{
    try
    {
        return 0;
    }
    finally
    {
        return 1;
    }
}
public static void bar( )
{
    try
    {
        throw new NullPointerException( );
    }
    finally
    {
        throw new ArithmeticException( );
    }
}
```

Figura 2.21 Complicaciones causadas por el bloque finally.

- 2.11** Una *suma de comprobación* es el entero de 32 bits resultante de sumar todos los caracteres Unicode de un archivo (permitimos el desbordamiento silencioso de esas sumas, aunque es poco probable que se produzca si todos los caracteres son ASCII). Dos archivos idénticos tienen la misma suma de comprobación. Escriba un programa que calcule la suma de comprobación de un archivo suministrado como argumento de la línea de comandos.
- 2.12** Implemente los siguientes métodos, que afectan a una matriz de tipo *double* y devuelven la suma, la media y la moda (el elemento más común) de la matriz.

```
public static double sum( double [ ] arr )
public static double average( double [ ] arr )
public static double mode( double [ ] arr )
```

- 2.13** Modifique el programa de la Figura 2.19 para que si no se proporciona ningún argumento de la línea de comandos, entonces se utilice la entrada estándar.
- 2.14** Implemente los siguientes métodos que invierten una matriz o un *ArrayList* de objetos *String*.

```
public static void reverse( String [ ] arr )
public static void reverse( ArrayList<String> arr )
```

- 2.15** Escriba una rutina que imprima la longitud total de los objetos `String` de una matriz `String[]` pasada como parámetro. La rutina debe poder funcionar sin modificaciones si se cambia el parámetro a una colección `ArrayList<String>`.

- 2.16** Implemente los siguientes métodos que aceptan una matriz bidimensional de tipo `double` y devuelven la suma, la media y la moda (el elemento más común) de la matriz bidimensional.

```
public static double sum( double [ ] arr )
public static double average( double [ ] arr )
public static double mode( double [ ] arr )
```

- 2.17** Implemente los siguientes métodos que devuelven el máximo del grupo de elementos pasado como parámetro. En el caso de objetos `String`, el máximo es el elemento alfabéticamente más grande, tal como se determina mediante `compareTo`.

```
public static int min( int [ ] arr )
public static int min( int [ ][ ] arr )
public static String max( String [ ] arr )
public static String max( ArrayList<String> arr )
```

- 2.18** Implemente los diversos métodos `hasDuplicates`, todos los cuales devuelven `true` si hay entradas duplicadas en el grupo de elementos especificado.

```
public static boolean hasDuplicates( int [ ] arr )
public static boolean hasDuplicates( int [ ][ ] arr )
public static boolean hasDuplicates( String [ ] arr )
public static boolean hasDuplicates( ArrayList<String> arr )
```

- 2.19** Implemente los siguientes métodos que devuelven el índice de la fila que contiene mayor número de unos.

```
public static int rowWithMostOnes( int [ ] [ ] arr )
```

- 2.20** Implemente ambos métodos `countChars`, que devuelven el número de apariciones de `ch` en `str`.

```
public static int countChars( String str, char ch )
public static int countChars( String [ ] str, char ch )
```

- 2.21** El método `isDecreasing` devuelve `true` si en cada fila de la matriz bidimensional todas las entradas decrecen monotónicamente y si en cada columna todas las entradas también decrecen del mismo modo. Implemente `isDecreasing`.

```
public static boolean isDecreasing( int [ ] [ ] arr )
```

- 2.22** Implemente ambos métodos `howMany`, que devuelven el número de apariciones de `val` dentro de `arr`.

```
public static int howMany( int [ ] arr, int val )
public static int howMany( int [ ] [ ] arr, int val )
```

- 2.23** Tanto Scanner como split se pueden configurar para utilizar delimitadores que sean distintos de los espacios en blanco normales. Por ejemplo, en un archivo delimitado por comas, el único delimitador es la coma. Para split, utilice "[,]" como parámetro y para Scanner utilice la instrucción

```
scan.useDelimiter( "[,]" )
```

Con esta información modifique el código de la Sección 2.6.2 para que funcione con una línea de entrada delimitada por comas.

- 2.24** Una alternativa a utilizar Scanner consiste en utilizar el método split para un objeto String. Específicamente, en la instrucción siguiente

```
String [ ] arr = str.split( "\\s" );
```

si str es "this is a test", entonces arr será una matriz de longitud cuatro que almacenará las cadenas de caracteres "this", "is", "a" y "test". Modifique el código de la Sección 2.6.2 para utilizar split en lugar de un objeto Scanner.

- 2.25** Implemente el método startsWith que devuelve un ArrayList que contiene todas las cadenas de caracteres de arr que comienzan con el carácter ch.

```
public ArrayList<String> startsWith( String [ ] arr, char ch )
```

- 2.26** Implemente un método split que devuelva una matriz de objetos String que contenga las unidades sintácticas del objeto String. Utilice un Scanner. La firma del método split es

```
public static String [ ] split( String str )
```

- 2.27** Utilizando el método toLowerCase de String que crea un nuevo objeto String, que es el equivalente en minúsculas de un objeto String existente (es decir, str.toLowerCase() devuelve el equivalente en minúsculas de str, dejando str sin modificar), implemente los métodos getLowerCase y makeLowerCase siguientes. getLowerCase devuelve una nueva colección de objetos String, mientras que makeLowerCase modifica la colección existente.

```
public static String [ ] getLowerCase( String [ ] arr )
public static void makeLowerCase( String [ ] arr )
public static ArrayList<String> getLowerCase( ArrayList<String> arr )
public static void makeLowerCase( ArrayList<String> arr )
```

## PROYECTOS DE PROGRAMACIÓN

- 2.28** Escriba un programa que muestre el número de caracteres, palabras y líneas de los archivos que se le suministren como argumentos de la línea de comandos.

- 2.29** En Java, la división por cero de coma flotante es legal y no provoca una excepción (en lugar de ello, proporciona una representación del infinito, del infinito negativo o un símbolo especial que indica que el valor no es un número).

a. Verifique la descripción anterior realizando algunas divisiones en coma flotante.

- b. Escriba un método `divide` estático que tome dos parámetros y devuelva su cociente. Si el divisor es 0.0, genere una excepción `ArithmeticException`. ¿Es necesario utilizar una cláusula `throws`?
- c. Escriba un programa `main` que invoque `divide` y capture la excepción `ArithmeticException`. ¿En qué método habría que colocar la cláusula `catch`?
- 2.30** Cree un archivo de datos `double.txt` que contenga números en coma flotante y que sea adecuado para utilizarlo en el Ejercicio 2.12. Escriba un método que invoque las funciones de ese ejercicio para los datos contenidos en el archivo. Asegúrese de que haya solo un elemento por línea y encárguese de tratar todos los problemas que se produzcan.
- 2.31** Cree un archivo de datos `double2.txt` que contenga números en coma flotante en una matriz bidimensional adecuada para utilizarla en el Ejercicio 2.16. Escriba un método que invoque las funciones de dicho ejercicio para los datos contenidos en el archivo. Si su código del Ejercicio 2.16 requiere que la matriz bidimensional sea rectangular, entonces, antes de invocar su método, genere una excepción si el archivo de datos no representa una matriz rectangular.
- 2.32** Cree un archivo de datos `double3.txt` que contenga números en coma flotante en una matriz bidimensional adecuada para utilizarla en el Ejercicio 2.16. Los números de cada fila deberían estar separados por comas. Escriba un método que invoque las funciones de dicho ejercicio para los datos contenidos en el archivo. Si su código del Ejercicio 2.16 requiere que la matriz bidimensional sea rectangular, entonces, antes de invocar su método, genere una excepción si el archivo de datos no representa una matriz rectangular.
- 2.33** Escriba un programa que calcule las notas para un determinado curso. El programa debe pedir al usuario el nombre de un archivo en el que estén almacenadas las puntuaciones de los exámenes. Cada línea del archivo tiene el siguiente formato:

Apellido : Nombre : Examen1 : Examen2 : Examen3

Los exámenes se ponderan de la manera siguiente: 20% para el primer examen, 25% para el segundo examen y 55% para el tercer examen. Teniendo esto en cuenta hay que asignar una nota final al alumno: A si el total es al menos 90, B si es al menos 80, C si es al menos 70, D si es al menos 60 y F en cualquier otro caso. Siempre se asigna la nota más alta basada en el total de puntos obtenidos, por lo que un 75 equivaldría a una C.

El programa debería presentar en el terminal una lista de estudiantes con la letra correspondiente a su nota de la forma siguiente:

Apellido Nombre LetraNota

También debe guardar la salida en un archivo, cuyo nombre será proporcionado por el usuario. Las líneas del archivo deben tener el formato

Apellido Nombre Examen1 Examen2 Examen3 TotalPuntos LetraNota

Después de guardar en el archivo los datos, deberá proporcionar como salida la distribución de notas. Si la entrada es

```
Doe:John:100:100:100  
Pantz:Smartee:80:90:80
```

Entonces la salida a través de terminal será

```
Doe John A  
Pantz Smartee B
```

Y el archivo de salida contendrá

```
Doe John 100 100 100 100 A  
Pantz Smartee 80 90 80 83 B  
A 1  
B 1  
C 0  
D 0  
F 0
```

- 2.34** Modifique el Ejercicio 2.33 para utilizar una escala de puntuación muy generosa, en la que el examen con mayor puntuación tiene una ponderación del 45%, el examen con la siguiente mayor puntuación tiene una ponderación del 30% y el examen con la menor puntuación tiene una ponderación del 25%. Por lo demás, la especificación del programa es idéntica.
- 2.35** Cada línea de un archivo contiene un nombre (en forma de cadena de caracteres) y una edad (en forma de número entero).
- Escriba un programa que muestre la persona más joven. En caso de que haya un empate, el programa puede mostrar a cualquiera de las personas con la edad más baja.
  - Escriba un programa que muestre la persona más joven. En caso de empate, hay que mostrar a todas las personas con la edad más baja. (*Pista:* mantenga el grupo actual de personas con la edad más baja en un `ArrayList`).
- 2.36** Implemente un programa de copia de archivos de texto. Incluya una comprobación para garantizar que los archivos de origen y de destino sean diferentes.



## Referencias

Puede encontrar más información en las referencias especificadas al final del Capítulo 1.



# Capítulo 3

## Objetos y clases

En este capítulo comienza la exposición acerca de la *programación orientada a objetos*. Un componente fundamental de la programación orientada a objetos es la especificación, la implementación y el uso de objetos. En el Capítulo 2, hemos visto varios ejemplos de objetos, incluyendo cadenas de caracteres y archivos, que forman parte de la librería Java obligatoria. También hemos visto que estos objetos tienen un estado interno que puede manipularse aplicando el operador punto para seleccionar un método. En Java, el estado y la funcionalidad de un objeto queda determinado mediante la definición de una *clase*. Por tanto, un objeto es una instancia de una clase.

En este capítulo, vamos a ver

- Cómo utiliza Java el concepto de clase para conseguir los objetivos de *encapsulación* y *ocultación de la información*.
- Cómo se implementan y documentan automáticamente las clases.
- Cómo se agrupan las clases en *paquetes*.

### 3.1 ¿Qué es la programación orientada a objetos?

La *programación orientada a objetos* emergió como el paradigma dominante a mediados de la década de 1990. En esta sección vamos a exponer algunas de las cosas que Java proporciona de cara al soporte de la orientación a objetos y vamos a mencionar algunos de los principios de la programación orientada a objetos.

En el núcleo mismo de la programación orientada a objetos se encuentra el *objeto*. Un objeto es un tipo de datos que tiene una estructura y un estado. Cada objeto define operaciones que permiten acceder a ese estado o manipularlo. Como ya hemos visto, en Java los objetos son distintos de los tipos primitivos, pero esto es una característica concreta de Java, más que formar parte del paradigma de orientación a objetos. Además de realizar operaciones generales podemos hacer lo siguiente:

- Crear nuevos objetos, posiblemente con una inicialización.
- Copiar objetos o comprobar su igualdad.
- Realizar operaciones de E/S con esos objetos.

Los objetos son entidades que tienen una estructura y un estado. Cada objeto define operaciones que permiten acceder a dicho estado o manipularlo.

Un objeto es una *unidad atómica*: sus partes no pueden ser disecadas por los usuarios generales del objeto.

La *ocultación de información* hace que sean inaccesibles los detalles de implementación, incluyendo los componentes de un objeto.

La *encapsulación* es la agrupación de datos y de las operaciones que se aplican a ellos, para formar un agregado, al mismo tiempo que se oculta la implementación de ese agregado.

Asimismo, consideramos el objeto como una *unidad atómica* que el usuario no debería disecar. A la mayoría de nosotros ni siquiera se nos ocurriría enredar con los bits que forman un número en coma flotante, y consideraríamos completamente ridículo tratar de incrementar algún objeto de coma flotante modificando nosotros mismos su representación interna.

El principio de atomicidad se conoce con el nombre de *ocultación de la información*. El usuario no dispone de acceso directo a las distintas partes del objeto ni a sus implementaciones; solo se puede acceder a esas partes del objeto indirectamente mediante métodos suministrados junto con el propio objeto. Podemos considerar cada objeto como si viniera acompañado de una advertencia: "No abrir. No contiene ninguna parte que el usuario pueda modificar". En la vida real, la mayoría de las personas que tratan de arreglar cosas que vienen con esa advertencia, terminan haciendo más daño, en lugar de arreglar lo que no funciona. A este respecto, la programación se asemeja al mundo real. La agrupación de una serie de datos y de las operaciones que se aplican a ellos con el fin de formar un agregado, al mismo tiempo que se ocultan los detalles de implementación de ese agregado, se conoce con el nombre de *encapsulación*.

Un objetivo importante de la programación orientada a objetos consiste en fomentar la reutilización de código. Al igual que los ingenieros utilizan componentes una y otra vez en sus diseños, los programadores deberían ser capaces de reutilizar objetos en lugar de reimplementarlos varias veces.

Cuando disponemos de una implementación del objeto exacto que necesitamos emplear, la reutilización es bastante sencilla. El desafío está en utilizar un objeto existente cuando el objeto que necesitamos no se corresponde exactamente con él, sino que simplemente es bastante similar.

Los lenguajes orientados a objetos proporcionan varios mecanismos para apoyar este objetivo. Uno de esos mecanismos es el uso de *código genérico*. Si la implementación es idéntica, salvo por el tipo básico del objeto, no hay necesidad de reescribir completamente el código: en lugar de ello, lo que hacemos es escribir el código genéricamente para que pueda funcionar con cualquier tipo de objeto. Por ejemplo, la lógica utilizada para ordenar una matriz de objetos es independiente del tipo de los objetos que se estén ordenando, por lo que en este caso podríamos utilizar un algoritmo genérico.

El mecanismo de *herencia* permite ampliar la funcionalidad de un objeto. En otras palabras, podemos crear nuevos tipos con propiedades restringidas (o ampliadas) con respecto al tipo original. La herencia nos permite avanzar enormemente en nuestro camino hacia la reutilización de código.

Otro principio importante de la orientación a objetos es el *polimorfismo*. Un tipo de referencia polimórfico puede hacer referencia a varios tipos de objetos distintos. Cuando se aplican métodos al tipo polimórfico, se selecciona automáticamente la operación que resulte apropiada para el objeto actualmente referenciado. En Java, esto se implementa como parte del mecanismo de herencia. El polimorfismo nos permite implementar clases que comparten una lógica común. Como se explica en el Capítulo 4, este principio está ilustrado en las propias librerías Java. El uso de la herencia para crear estas jerarquías distingue a la programación orientada a objetos de la *programación basada en objetos*, que es más simple.

En Java, los algoritmos genéricos se implementan como parte del mecanismo de herencia. En el Capítulo 4 se explica la herencia y el polimorfismo. En este capítulo, vamos a describir cómo el lenguaje Java utiliza las clases para conseguir la encapsulación y la ocultación de información.

Un *objeto* en Java es una instancia de una clase. Una *clase* es similar a una estructura en C o a un registro en Pascal/Ada, salvo por dos mejoras importantes. En primer lugar, los miembros pueden ser tanto funciones como datos, a los que se conoce con los nombres de *métodos* y *campos*, respectivamente. En segundo lugar, la visibilidad de estos miembros puede estar restringida. Puesto que los métodos que manipulan el estado del objeto son miembros de la clase, se accede a ellos mediante el operador punto, al igual que pasa con los campos. En terminología de orientación a objetos, cuando hacemos una llamada a un método lo que hacemos es pasar un mensaje al objeto. Los tipos que hemos explicado en el Capítulo 2, como por ejemplo `String`, `ArrayList`, `Scanner` y `FileReader`, son todos ellos clases implementadas en la librería Java.

Una clase en Java está compuesta por campos que almacenan datos y de métodos que se aplican a las instancias de esa clase.

## 3.2 Un ejemplo simple

Recuerde que a la hora de diseñar una clase es importante ser capaz de ocultar los detalles internos a ojos del usuario de esa clase. Esto se lleva a cabo de dos maneras. En primer lugar, la clase puede definir funcionalidad mediante miembros de la clase, denominados *métodos*. Algunos de esos métodos describen cómo se crea e inicializa una instancia de la estructura, cómo se realizan las comprobaciones de igualdad y cómo se lleva a cabo la salida.

La funcionalidad se suministra mediante miembros adicionales; estos métodos manipulan el estado del objeto.

Otros métodos serían específicos de cada estructura concreta. La idea es que los campos de datos internos que representan el estado de un objeto no deberían poder ser manipulados directamente por el usuario de la clase, sino que deberían manipularse solamente a través de los métodos. Esta idea puede ser reforzada ocultando los miembros a ojos del usuario. Para hacer esto, podemos especificar que esos miembros se almacenen en una sección *privada*. El compilador impondrá la regla de que los miembros contenidos en la sección privada sean inaccesibles para los métodos que no pertenezcan a esa clase de objetos. Hablando en términos generales, todos los datos miembro deberían ser privados.

La Figura 3.1 ilustra la declaración de clase para un objeto `IntCell`.<sup>1</sup> La declaración está compuesta de dos partes: *pública* y *privada*. Los miembros *públicos* representan la parte que es visible para el usuario del objeto. Puesto que esperamos poder ocultar los datos, en la sección *pública* solo se deberían incluir, generalmente, los métodos y constantes. En nuestro ejemplo, disponemos de métodos que leen y escriben en el objeto `IntCell`. La sección *privada* contiene los datos; estos son invisibles para el usuario del objeto. Para acceder al miembro `storedValue` deben utilizarse las rutinas públicamente visibles `read` y `write`; `main` no puede acceder directamente a ese miembro. En la Figura 3.2 se muestra otra forma de ver esto.

Los miembros públicos son visibles para las rutinas que no pertenecen a la clase, los miembros privados no lo son.

<sup>1</sup> Las clases públicas deben almacenarse en archivos que tengan el mismo nombre. Por tanto, `IntCell` debe estar en un archivo llamado `IntCell.java`. Hablaremos del significado de `public` en la línea 5 cuando expliquemos lo que son los paquetes.

```

1 // clase IntCell
2 // int read( )          --> Devuelve el valor almacenado
3 // void write( int x ) --> se almacena x
4
5 public class IntCell
6 {
7     // Métodos públicos
8     public int read( ) { return storedValue; }
9     public void write( int x ) { storedValue = x; }
10
11    // Representación privada interna de los datos
12    private int storedValue;
13 }

```

Figura 3.1 Una declaración completa de una clase IntCell.

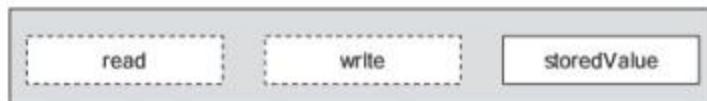


Figura 3.2 Miembros de IntCell: read y write son accesibles, pero storedValue está oculto.

Los miembros que se declaran como `private` no son visibles para las rutinas que no pertenecen a la clase.

La Figura 3.3 muestra cómo se utilizan los objetos IntCell. Puesto que `read` y `write` son miembros de la clase `IntCell`, se accede a ellos utilizando el operador punto. Al miembro `storedValue` también se podría acceder utilizando el operador punto, pero como es de tipo `private`, el acceso de la línea 14 sería ilegal si no estuviera desactivado como comentario.

```

1 // Ejemplo de uso de la clase IntCell
2
3 public class TestIntCell
4 {
5     public static void main( String [ ] args )
6     {
7         IntCell m = new IntCell( );
8
9         m.write( 5 );
10        System.out.println( "Cell contents: " + m.read( ) );
11
12        // La siguiente linea seria ilegal si no estuviera desactiva por un
13        // comentario, porque storedValue es un miembro privado
14        // m.storedValue = 0;
15    }
16 }

```

Figura 3.3 Una sencilla de rutina de comprobación para mostrar cómo se accede a los objetos IntCell.

He aquí un resumen de la terminología. La clase define *miembros*, que pueden ser *campos* (datos) o *métodos* (funciones). Los métodos pueden actuar sobre los campos y pueden invocar a otros métodos. El modificador de visibilidad `public` significa que el miembro es accesible para todo el mundo, mediante el operador punto. El modificador de visibilidad `private` significa que a ese miembro solo pueden acceder otros miembros de esta clase. Si no se especifica ningún modificador de visibilidad, lo que tendremos es un acceso con visibilidad de paquete, del que hablaremos en la Sección 3.8.4. También existe un cuarto modificador, conocido como `protected`, que veremos en el Capítulo 4.

Un campo es un miembro que almacena datos; un método es un miembro que realiza una acción.

### 3.3 javadoc

Al diseñar una clase, la *especificación de la clase* representa el diseño de la clase y nos dice lo que podemos hacer con un objeto. La *implementación* representa los detalles internos de cómo se lleva eso a cabo. En lo que concierne al usuario de la clase, estos detalles internos no son importantes. En muchos casos, la implementación representa información confidencial que el diseñador de la clase podría no querer compartir. Sin embargo, la especificación sí que debe ser compartida, de lo contrario, la clase no podría utilizarse.

La especificación de la clase describe lo que puede hacerse con un objeto. La implementación representa los detalles internos acerca de cómo se satisfacen las especificaciones.

En muchos lenguajes, el compartir la especificación al mismo tiempo que se oculta la implementación se consigue colocando la especificación y la implementación en archivos fuente separados. Por ejemplo, C++ dispone de la interfaz de clase que se almacena en un archivo `.h` y de una implementación de la clase, que se almacena en un archivo `.cpp`. En el archivo `.h`, la interfaz de la clase vuelve a enumerar los métodos (proporcionando cabeceras de método) implementados por la clase.

Java adopta un enfoque diferente. Es fácil darse cuenta de que una lista de métodos de una clase, con sus signaturas y tipos de retorno, puede documentarse automáticamente a partir de la implementación. Java utiliza la siguiente idea: el programa `javadoc`, que se suministra con todos los sistemas Java, puede ejecutarse para generar automáticamente documentación para las clases. La salida de `javadoc` es un conjunto de archivos HTML que pueden visualizarse o imprimirse utilizando un explorador web.

El programa `javadoc` genera automáticamente documentación para las clases.

El archivo de implementación Java también puede añadir comentarios `javadoc` que comienzan con el símbolo de inicio de comentario `/**`. Esos comentarios se añaden automáticamente, de manera uniforme y coherente a la documentación producida por `javadoc`.

También hay varios marcadores especiales que pueden incluirse en los comentarios `javadoc`. Entre ellos están `@author`, `@param`, `@return` y `@throws`. La Figura 3.4 ilustra el uso de la funcionalidad de comentarios `javadoc` para la clase `IntCell`. En la línea 3, se utiliza el marcador `@author`. Este marcador debe preceder a la definición de la clase. La línea 10 ilustra el uso del marcador `@return` y la línea 19 el del marcador `@param`. Estos marcadores deben aparecer antes de la declaración de un método. El primer símbolo situado a continuación del marcador `@param` es el nombre del parámetro. El marcador `@throws` no se muestra, pero utiliza la misma sintaxis que `@param`.

Entre los marcadores `javadoc` se encuentran `@author`, `@param`, `@return` y `@throws`. Se utilizan en los comentarios `javadoc`.

```
1 /**
2 * Una clase para simular una celda de memoria entera
3 * @author Mark A. Weiss
4 */
5
6 public class IntCell
7 {
8     /**
9      * Obtener el valor almacenado.
10     * @return el valor almacenado.
11    */
12    public int read( )
13    {
14        return storedValue;
15    }
16
17    /**
18     * Almacenar un valor.
19     * @param x es el número que hay que almacenar.
20    */
21    public void write( int x )
22    {
23        storedValue = x;
24    }
25
26    private int storedValue;
27 }
```

---

Figura 3.4 La declaración de `IntCell` con comentarios `javadoc`.

En la Figura 3.5 se muestra parte de la salida resultante de la ejecución de `javadoc`. Ejecute `javadoc` suministrando un nombre (incluyendo la extensión `.java`) de archivo fuente.

La salida de `javadoc` son puros comentarios, salvo por las cabeceras de métodos. El compilador no comprueba que esos comentarios estén implementados. Sin embargo, es imposible sobrevalorar la importancia de una adecuada documentación de las clases. `javadoc` facilita la tarea de generar una documentación bien formateada.

## 3.4 Métodos básicos

Algunos métodos son comunes a todas las clases. En esta sección vamos a hablar de los mutadores, accesores y tres métodos especiales: los constructores, `toString` y `equals`. También nos ocuparemos de `main`.

The screenshot shows a Mozilla Firefox browser window displaying JavaDoc for the `IntCell` class. The title bar reads "IntCell - Mozilla Firefox". The menu bar includes File, Edit, View, History, Bookmarks, Yahoo!, Tools, and Help. The toolbar below the menu bar has buttons for PREV CLASS, NEXT CLASS, SUMMARY, NESTED, FIELD, CONSTR, METHOD, FRAMES, NO FRAMES, ALL Classes, DETAIL, FIELD, CONSTR, and METHOD.

## Class IntCell

java.lang.Object  
└ IntCell

`public class IntCell  
extends java.lang.Object`

A class for simulating an integer memory cell

### Constructor Summary

`IntCell()`

### Method Summary

int	<code>read()</code> Get the stored value.
void	<code>write(int x)</code> Store a value

### Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

### Constructor Detail

Figura 3.5 La salida de Javadoc correspondiente a la Figura 3.4 (salida parcial).

### 3.4.1 Constructores

Un constructor dice cómo se declara e inicializa un objeto.

El constructor predeterminado consiste en una aplicación miembro a miembro de una inicialización predeterminada.

Como hemos mencionado anteriormente, una propiedad básica de los objetos es que pueden definirse, añadiendo posiblemente una inicialización. En Java, el método que controla cómo se crea e inicializa un objeto es el *constructor*. Gracias al mecanismo de sobrecarga, una clase puede definir múltiples constructores.

Si no se proporciona ningún constructor, como en el caso de la clase `IntCe11` de la Figura 3.1, se genera un constructor predeterminado que inicializa cada dato miembro utilizando los valores predeterminados normales. Esto quiere decir que los campos primitivos se inicializan con cero y que los campos de referencia se inicializan con la referencia `null`. (Estos valores predeterminados pueden sustituirse mediante la inicialización de campos en línea, que se ejecuta antes de ejecutar los cuerpos de los constructores.) Por tanto, en el caso de `IntCe11`, el componente `storedValue` es 0.

Para escribir un constructor, tenemos que proporcionar un método que tenga el mismo nombre que la clase y ningún tipo de retorno (es fundamental que se omita el tipo de retorno; un error frecuente es incluir `void` como tipo de retorno, lo que da como resultado que se declare un método que no es un constructor). En la Figura 3.6 hay dos constructores: uno comienza en la línea 7 y el otro en la línea 15. Utilizando estos constructores podemos construir objetos `Date` de cualquiera de las formas siguientes:

```
Date d1 = new Date();
Date d2 = new Date( 4, 15, 2010 );
```

Observe que una vez que se ha escrito un constructor, ya no se genera un constructor predeterminado con cero parámetros. Si deseamos uno, tenemos que escribirlo explícitamente. Por tanto, el constructor de la línea 7 es obligatorio de cara a permitir la construcción del objeto que `d1` hace referencia.

### 3.4.2 Mutadores y accesores

Un método que examina pero que no modifica el estado de un objeto se denomina *accesor*. Un método que modifica el estado es un *mutador*.

Generalmente, los campos de una clase se declaran como de tipo `private`. Por tanto, las rutinas que no pertenezcan a esa clase no pueden acceder directamente a ellos. En ocasiones, sin embargo, nos gustaría poder examinar el valor de un campo. Es posible incluso que deseemos modificarlo.

Una alternativa para hacer esto consiste en declarar los campos como `public`. Sin embargo, esta elección no suele ser conveniente, porque viola el principio de ocultación de la información. En lugar de ello, podemos proporcionar métodos para examinar y modificar cada campo. Un método que examina pero no modifica el estado de un objeto se denomina *accesor*. Un método que modifica el estado se denomina *mutador* (porque hace que mute el estado del objeto).

Algunos casos especiales de accesores y mutadores examinan únicamente un solo campo. Estos accesores suelen tener nombres que comienzan con `get`, como `getMonth`, mientras que los correspondientes mutadores suelen tener nombres que comienzan con `set`, como `setMonth`.

```
1 // Clase Date mínima que ilustra algunas características Java
2 // No hay comprobaciones de errores ni comentarios javadoc
3
4 public class Date
5 {
6     // Constructor con cero parámetros
7     public Date( )
8     {
9         month = 1;
10        day = 1;
11        year = 2010;
12    }
13
14     // Constructor con tres parámetros
15     public Date( int theMonth, int theDay, int theYear )
16     {
17         month = theMonth;
18         day = theDay;
19         year = theYear;
20     }
21
22     // Devuelve true si los valores son iguales
23     public boolean equals( Object rhs )
24     {
25         if( !( rhs instanceof Date ) )
26             return false;
27         Date rhDate = ( Date ) rhs;
28         return rhDate.month == month && rhDate.day == day &&
29             rhDate.year == year;
30     }
31
32     // Conversión a String
33     public String toString( )
34     {
35         return month + "/" + day + "/" + year;
36     }
37
38     // Campos
39     private int month;
40     private int day;
41     private int year;
42 }
```

Figura 3.6 Una clase Date mínima que ilustra los constructores y los métodos equals y toString.

La ventaja de utilizar un mutador es que este puede garantizar que los cambios efectuados en el estado del objeto sean coherentes. Por tanto, un mutador que modifique el campo `day` de un objeto `Date` puede asegurarse de que solo se usen fechas correctas.

### 3.4.3 Salida de información y el método `toString`

Puede proporcionarse un método `toString`. Este método devuelve un objeto `String` basado en el estado del objeto.

Normalmente, querremos poder mostrar el estado de un objeto utilizando `print`. Esto se lleva a cabo escribiendo el método de clase `toString`. Este método devuelve un objeto `String` adecuado para la salida. Por ejemplo, la Figura 3.6 muestra una implementación básica del método `toString` para la clase `Date`.

### 3.4.4 `equals`

Puede proporcionarse un método `equals` para comprobar si dos referencias se refieren al mismo valor.

El parámetro de `equals` es de tipo `Object`.

El método `equals` se utiliza para comprobar si dos objetos representan el mismo valor. La firma es siempre

```
public boolean equals( Object rhs )
```

Observe que el parámetro es del tipo de referencia `Object` en lugar de ser del tipo de la clase (en el Capítulo 4 se explica la razón de esto). Normalmente, el método `equals` de la clase `NombreClase` se implementa para devolver `true` solo si `rhs` es una instancia de `NombreClase`, y si después de la conversión a `NombreClase`, todos los campos primitivos son iguales (vía `==`) y todos los campos de referencia son iguales (aplicando miembro a miembro `equals`).

En la Figura 3.6 se muestra un ejemplo de cómo se implementa `equals` para la clase `Date`. El operador `instanceof` se explica en la Sección 3.6.3.

### 3.4.5 `main`

Cuando se ejecuta el comando `java` para iniciar el intérprete se invoca el método `main` del archivo de clase referenciado por el comando `java`. Por tanto, cada clase puede disponer de su propio método `main`, sin ningún problema. Esto hace que resulte fácil probar la funcionalidad básica de las clases individuales. Sin embargo, aunque puede probarse la funcionalidad, el incluir `main` en la clase proporciona a `main` más visibilidad de la que en general queríamos permitir. De ese modo, las llamadas a `main` desde métodos no públicos de la misma clase se compilarán correctamente, aun cuando serían ilegales en un entorno más general.

## 3.5 Ejemplo: utilización de `java.math.BigInteger`

En la Sección 3.3 hemos descrito cómo generar documentación a partir de una clase y en la Sección 3.4 se han descrito algunos componentes típicos de una clase, como los constructores, accesores, mutadores y, en particular, los métodos `equals` y `toString`. En esta sección vamos a mostrar las partes de la documentación que más habitualmente usan los programadores.

La Figura 3.7 muestra una sección de la documentación en línea para la clase de librería `java.math.BigInteger`. Falta la sección que proporciona una panorámica de la clase en algo que se asemeja al inglés (compare con la Figura 3.5 para ver el preámbulo que falta). Ese preámbulo nos dice, entre otras cosas, que los objetos `BigInteger` son inmutables, como los objetos `String`: una vez que creamos un objeto `BigInteger`, su valor no puede variar.

**Field Summary**

<code>static BigInteger ONE</code>	The BigInteger constant one.
<code>static BigInteger ZERO</code>	The BigInteger constant zero.

**Constructor Summary**

<code>BigInteger(String val)</code>	Translates the decimal String representation of a BigInteger into a BigInteger.
-------------------------------------	---

**Method Summary**

<code>BigInteger abs()</code>	Returns a BigInteger whose value is the absolute value of this BigInteger.
<code>BigInteger add(BigInteger val)</code>	Returns a BigInteger whose value is (this + val).
<code>int compareTo(BigInteger val)</code>	Compares this BigInteger with the specified BigInteger.
<code>BigInteger divide(BigInteger val)</code>	Returns a BigInteger whose value is (this / val).
<code>boolean equals(Object x)</code>	Compares this BigInteger with the specified Object for equality.
<code>BigInteger gcd(BigInteger val)</code>	Returns a BigInteger whose value is the greatest common divisor of abs(this) and abs(val).
<code>BigInteger multiply(BigInteger val)</code>	Returns a BigInteger whose value is (this * val).
<code>BigInteger negate()</code>	Returns a BigInteger whose value is (-this).
<code>BigInteger subtract(BigInteger val)</code>	Returns a BigInteger whose value is (this - val).
<code>String toString()</code>	Returns the decimal String representation of this BigInteger.

Figura 3.7 Javadoc simplificado para `java.math.BigInteger`.

A continuación hay una lista de campos, que en nuestro caso son las constantes `ZERO` y `ONE`. Si a continuación hacemos clic en los hipervínculos `ZERO` o `ONE`, obtendremos una descripción más completa que nos dice que se trata de entidades de tipo `public static final`.

La siguiente sección enumera los constructores disponibles. En este caso hay seis, pero solo se muestra uno en nuestro listado resumido, y ese constructor exige un objeto `String`. De nuevo, si hacemos clic sobre el nombre del constructor, el hipervínculo nos lleva a una descripción más completa que se muestra en la Figura 3.8. Entre otras cosas, vemos que si el objeto `String` contiene espacios en blanco extra, el constructor fallará y generará una excepción. Estos son los tipos de detalles que siempre merece la pena conocer.

A continuación vemos una serie de métodos (de nuevo, este es un listado abreviado). Dos métodos importantes son `equals` y `toString`; puesto que se enumeran específicamente aquí, podemos estar seguros de que los objetos `BigInteger` pueden compararse con total seguridad utilizando `equals` e imprimirse obteniendo una representación razonable. También hay un método `compareTo` y si hacemos clic en el hipervínculo, vemos que el comportamiento general de `compareTo` es idéntico al método `compareTo` de la clase `String`. Esto no es por casualidad, como veremos en el Capítulo 4. Observe también que examinando las signatures y las breves descripciones podemos ver que métodos como `add` y `multiply` devuelven objetos `BigInteger` de nueva creación, dejando los originales sin modificar. Esto es por supuesto obligatorio, ya que `BigInteger` es una clase inmutable.

Más adelante en el capítulo utilizaremos la clase `BigInteger` como componente para implementar nuestra propia clase `BigRational`, una clase con la que representaremos los números racionales.

## 3.6 Constructores adicionales

Tres palabras clave adicionales son `this`, `instanceof` y `static`. La palabra clave `this` tiene varios usos en Java; veremos dos de ellos en esta sección. La palabra clave `instanceof` también tiene varios usos generales; aquí se emplea para garantizar que la conversión de tipos se realizará correctamente. Asimismo, `static` también tiene varios usos. Ya hemos visto los métodos estáticos. En esta sección veremos qué es un *campo estático* y un *inicializador estático*.

---

### `BigInteger`

`public BigInteger (String val)`

Translates the decimal `String` representation of a `BigInteger` into a `BigInteger`. The `String` representation consists of an optional minus sign followed by a sequence of one or more decimal digits. The character-to-digit mapping is provided by `Character.digit`. The `String` may not contain any extraneous characters (whitespace, for example).

**Parameters:**

`val` - decimal `String` representation of `BigInteger`.

**Throws:**

`NumberFormatException` - `val` is not a valid representation of a `BigInteger`.

**See also:**

`Character.digit(char, int)`

---

Figura 3.8 Detalles del constructor `BigInteger`.

### 3.6.1 La referencia this

El primer uso de `this` es como referencia al objeto actual. Piense en la referencia `this` como si fuera un dispositivo de orientación que nos dijera, en cualquier momento, dónde nos encontramos. Un uso importante de la referencia `this` es a la hora de gestionar el caso especial de las autoasignaciones. Un ejemplo de este tipo de uso sería un programa que copiara un archivo en otro. Un algoritmo normal comenzaría truncando el archivo de destino, para dejarlo con longitud cero. Si no efectuamos ninguna comprobación para cerciorarnos de que los archivos de origen y de destino sean diferentes, entonces el archivo de origen podría llegar a ser truncado, lo que no suele ser lo que deseamos. A la hora de manejar dos objetos, en uno de ellos escribimos y en el otro leemos, primero debemos comprobar este caso especial, que se conoce con el nombre de *utilización de alias*.

`this` es una referencia al objeto actual. Puede utilizarse para enviar el objeto actual, como una unidad, a algún otro método.

Para el segundo ejemplo, suponga que tenemos una clase `Account` con un método `finalTransfer`. Este método transfiere todo el dinero de una cuenta a otra. En principio, se trata de una rutina bastante fácil de escribir:

```
// Transferir todo el dinero de rhs a la cuenta actual
public void finalTransfer( Account rhs )
{
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
```

Sin embargo, considere el resultado:

```
Account account1;
Account account2;
...
account2 = account1;
account1.finalTransfer( account2 );
```

Puesto que estamos transfiriendo dinero de una cuenta a sí misma, no debería haber ningún cambio en el saldo. Sin embargo, la última instrucción de `finalTransfer` garantiza que la cuenta quedará vacía. Una forma de evitar esto, consiste en utilizar una comprobación de alias:

```
// Transferir todo el dinero de rhs a la cuenta actual
public void finalTransfer( Account rhs )
{
    if( this == rhs ) // Comprobación de alias
        return;
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
```

La utilización de `alias` es un caso especial que se presenta cuando el mismo objeto aparece cumpliendo más de una función.

### 3.6.2 La abreviatura `this` para constructores

`this` puede utilizarse para hacer una llamada a otro constructor de la misma clase.

Muchas clases tienen varios constructores que se comportan de forma similar. Podemos utilizar `this` dentro de un constructor para invocar a uno de los otros constructores de la clase. Una alternativa al constructor de cero parámetros de `Date` de la Figura 3.6 sería

```
public Date()
{
    this( 1, 1, 2010 ); // Invocar el constructor de 3 parámetros
}
```

También son posibles otros usos más complicados, pero la llamada a `this` puede ser la primera instrucción del constructor; después de ella se pueden incluir otras instrucciones.

### 3.6.3 El operador `instanceof`

El operador `instanceof` se utiliza para comprobar si una expresión es una instancia de alguna clase determinada.

El operador `instanceof` realiza una comprobación en tiempo de ejecución. El resultado de

```
exp instanceof NombreClase
```

será `true` si `exp` es una instancia de `NombreClase` y `false` en caso contrario.

Si `exp` es `null`, el resultado será siempre `false`. El operador `instanceof` suele utilizarse, típicamente, antes de realizar una conversión de tipos y será `true` si la conversión de tipos puede realizarse correctamente.

### 3.6.4 Miembros de instancia y miembros estáticos

Los miembros de instancia son campos o métodos declarados sin el modificador `static`.

Los campos y métodos declarados con la palabra clave `static` son *miembros estáticos*. Si se los declara sin la palabra clave `static`, decimos que son *miembros de instancia*. En la siguiente subsección se explica la diferencia entre miembros de instancia y miembros estáticos.

### 3.6.5 Campos y métodos estáticos

Un método estático es un método que no necesita un objeto que lo controle.

Un *método estático* es un método que no necesita un objeto que lo controle, y que por tanto se suele llamar proporcionando un nombre de clase en lugar del nombre del objeto controlador. El método estático más común es `main`. Puede encontrar otros métodos estáticos en las clases `Integer` y `Math`. Como ejemplos tendríamos los métodos `Integer.parseInt`, `Math.sin` y `Math.max`.

El acceso a un método estático utiliza las mismas reglas de visibilidad que los campos estáticos. Estos métodos se asemejan a las funciones globales que podemos encontrar en los lenguajes no orientados a objetos.

Los *campos estáticos* se utilizan cuando tenemos una variable que tiene que ser compartida por todos los miembros de una cierta clase. Normalmente, se tratará de una constante simbólica, pero

tampoco es obligatorio que sea así. Cuando una variable de clase se declara como `static`, solo se creará una instancia de esa variable. No forma parte de ninguna instancia de la clase. En lugar de ello, se comporta como una única variable global, pero cuyo ámbito es el de la clase. En otras palabras, en la declaración

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

cada objeto `Sample` almacena su propia `x`, pero solo existe una `y` compartida.

Un uso común de un campo estático es como constante. Por ejemplo, la clase `Integer` define el campo `MAX_VALUE` como

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

Si esta constante no fuera un campo estático, entonces cada instancia de `Integer` tendría un campo de datos denominado `MAX_VALUE`, desperdiциando así espacio y tiempo de inicialización. En lugar de ello, solo hay una única variable denominada `MAX_VALUE`. Cualquiera de los métodos de `Integer` puede acceder a esa variable utilizando el identificador `MAX_VALUE`. También se puede acceder a ella a través de un objeto `obj` de tipo `Integer` utilizando `obj.MAX_VALUE`, como con cualquier otro campo. Observe que esto se permite únicamente porque `MAX_VALUE` es pública. Por último, también se puede acceder a `MAX_VALUE` utilizando el nombre de la clase como en `Integer.MAX_VALUE` (lo que de nuevo está permitido debido a que es un campo público). Esto no se permitiría para un campo no estático. Esta última forma es la preferible, porque comunica al lector que el campo es, de hecho, de tipo estático. Otro ejemplo de campo estático sería la constante `Math.PI`.

Incluso sin el cualificador `final`, los campos estáticos siguen siendo útiles. La Figura 3.9 ilustra un ejemplo típico. Aquí lo que queremos es construir objetos `Ticket`, dando a cada `Ticket` un número de serie distintivo. Para poder hacer esto, tenemos que disponer de alguna forma de llevar la cuenta de todos los números de serie utilizados previamente; esto es evidentemente un dato compartido y que no forma parte de ningún objeto `Ticket` concreto.

Cada objeto `Ticket` tendrá su miembro de instancia `serialNumber`; esto es un dato de instancia, porque cada instancia de `Ticket` tiene su propio campo `serialNumber`. Todos los objetos `Ticket` compartirán la variable `ticketCount`, que indica el número de objetos `Ticket` que se han creado. Esta variable forma parte de la clase, en lugar de ser específica de un objeto, por lo que se la declara como de tipo `static`. Solo hay un `ticketCount`, independientemente de que haya un objeto `Ticket`, 10 objetos `Ticket` o incluso ningún objeto `Ticket`. Este último punto –el de que los datos estáticos existen incluso antes de crear ninguna instancia de la clase– es importante, porque significa que los datos estáticos no pueden inicializarse en los constructores. Una forma de realizar la inicialización es en línea, en el momento de declarar el campo. En la Sección 3.6.6 se describe un método de inicialización más complejo.

En la Figura 3.9, ahora podemos ver que la construcción de los objetos `Ticket` se realiza utilizando `ticketCount` como número de serie e incrementando `ticketCount`. También proporcionamos un método estático, `getTicketCount`, que devuelve el número de tickets. Puesto que

*Los campos estáticos son esencialmente variables globales cuyo ámbito es el de una clase.*

*Un campo estático es compartido por todas las instancias (posiblemente zero) de la clase.*

```
1 class Ticket
2 {
3     public Ticket( )
4     {
5         System.out.println( "Calling constructor" );
6         serialNumber = ++ticketCount;
7     }
8
9     public int getSerial( )
10    {
11        return serialNumber;
12    }
13
14    public String toString( )
15    {
16        return "Ticket #" + getSerial( );
17    }
18
19    public static int getTicketCount( )
20    {
21        return ticketCount;
22    }
23
24    private int serialNumber;
25    private static int ticketCount = 0;
26 }
27
28 class TestTicket
29 {
30     public static void main( String [ ] args )
31     {
32         Ticket t1;
33         Ticket t2;
34
35         System.out.println( "Ticket count is " +
36                             Ticket.getTicketCount( ) );
37         t1 = new Ticket( );
38         t2 = new Ticket( );
39
40         System.out.println( "Ticket count is " +
41                             Ticket.getTicketCount( ) );
42
43         System.out.println( t1.getSerial( ) );
44         System.out.println( t2.getSerial( ) );
45 }
46 }
```

---

Figura 3.9 La clase Ticket: un ejemplo de campos y métodos estáticos.

es estático, se puede invocar sin proporcionar una referencia a objeto, como se muestra en las líneas 36 y 41. La llamada de la línea 41 podría haberse realizado utilizando `t1` o `t2`, aunque muchos afirman que invocar un método estático utilizando una referencia a objeto es un estilo de programación inadecuado, por lo que nosotros no utilizaremos esa técnica en este texto. Sin embargo, es significativo que la llamada de la línea 36 no podría, claramente, realizarse a través de una referencia a objeto, porque en ese punto no existen todavía objetos `Ticket` válidos. Esta es la razón por la que es importante que `getTicketCount` se declare como método estático: si se declarara como un método de instancia, solo podría invocarse a través de una referencia a objeto.

Cuando se declara un método como estático, no existe referencia `this` implícita. Por ello, no puede acceder a datos de instancia o invocar métodos de instancia, sin proporcionar una referencia a objeto. En otras palabras, desde dentro de `getTicketCount`, el acceso no cualificado a `serialNumber` implicaría `this.serialNumber`, pero como no existe `this`, el compilador generará un mensaje de error. Por tanto, un método de clase estático solo podrá acceder a un campo no estático, que forma parte de cada instancia de la clase, si se proporciona un objeto controlador.

Un método static no tiene referencia this implícita, y puede invocarse sin una referencia a objeto.

### 3.6.6 Inicializadores estáticos

Los campos estáticos se inicializan en el momento de cargar la clase. Ocasionalmente, necesitaremos un mecanismo de inicialización complejo. Por ejemplo, suponga que necesitamos una matriz estática que almacene la raíz cuadrada de los 100 primeros números enteros. Lo mejor sería hacer que esos valores se calcularan automáticamente. Una posibilidad consiste en proporcionar un método estático y obligar al programador a invocarlo antes de utilizar la matriz.

Otra alternativa es el *inicializador estático*. En la Figura 3.10 se muestra un ejemplo. Allí, el inicializador estático abarca de las líneas 5 a 9. El uso más simple del inicializador estático coloca el código de inicialización para los campos estáticos en un bloque precedido por la palabra clave `static`. El inicializador estático debe seguir a la declaración del miembro estático.

Un inicializador estático es un bloque de código que se utiliza para inicializar campos estáticos.

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

Figura 3.10 Un ejemplo de inicializador estático.

### 3.7 Ejemplo: implementación de una clase BigRational

En esta sección, vamos a escribir una clase que ilustra muchos de los conceptos que hemos descrito en el capítulo, incluyendo:

- Constantes `public static final`.
- Uso de una clase existente, en concreto `BigInteger`.
- Múltiples constructores.
- Generación de excepciones.
- Implementación de un conjunto de accesores.
- Implementación de `equals` y `toString`.

La clase que vamos a escribir representará números racionales. Un número racional almacena un numerador y un denominador y utilizaremos objetos `BigInteger` para representar el numerador y el denominador. Por ello, nuestra clase se llamará de forma bastante lógica `BigRational`.

La Figura 3.11 muestra la clase `BigRational`. El código en línea está completamente comentado. Aquí, hemos omitido los comentarios, para que el código pueda caber en las páginas de texto. Las líneas 5 y 6 son constantes `BigRational.ZERO` y `BigRational.ONE`. También vemos que la representación de los datos se hace mediante dos objetos `BigInteger`, `num` y `den`, y nuestro código se implementará de una forma que garantice que el denominador nunca sea negativo. Proporcionamos cuatro constructores, y dos de ellos se implementan utilizando la palabra clave `this`. Los otros dos constructores tienen implementaciones más complicadas, mostradas en la Figura 3.12. Aquí podemos ver que el constructor de dos parámetros `BigRational` inicializa el numerador y el denominador de la forma especificada, pero luego debe garantizar que el denominador no sea negativo (lo que lo hace invocando el método privado `fixSigns`) y luego eliminando los factores comunes (invocando el método privado `reduce`). También proporcionamos una comprobación para garantizar que se no acepte `0/0` como un `BigRational`, y esto se hace en `check00` que generará una excepción si se intenta construir ese tipo de objeto como `BigRational`. Los detalles de `check00`, `fixSigns` y `reduce` son menos importantes que el hecho de que su uso en un constructor y en otros métodos permita al diseñador de la clase garantizar que los objetos siempre se configuren en estados válidos.

La clase `BigRational` también incluye métodos para devolver valores absolutos y un negativo. Se trata de métodos simples, que se muestran en las líneas 24 a 27 de la Figura 3.11. Observe que estos métodos devuelven nuevos objetos `BigRational`, dejando el original intacto.

`add`, `subtract`, `multiply` y `divide` se muestran en las líneas 29 a 36 de la Figura 3.11 y están implementados en la Figura 3.13. Los aspectos matemáticos son menos interesantes que el concepto fundamental de que, debido a que cada una de las cuatro rutinas termina creando un nuevo `BigRational`, y el constructor de `BigRational` tiene llamadas a `check00`, `fixSigns` y `reduce`, las respuestas resultantes están siempre en la forma simplificada correcta, y cualquier intento de dividir cero entre cero será capturado por `check00`.

```
1 import java.math.BigInteger;
2
3 public class BigRational
4 {
5     public static final BigRational ZERO = new BigRational( );
6     public static final BigRational ONE = new BigRational( "1" );
7
8     public BigRational( )
9         { this( BigInteger.ZERO ); }
10    public BigRational( BigInteger n )
11        { this( n, BigInteger.ONE ); }
12    public BigRational( BigInteger n, BigInteger d )
13        { /* Implementación en la Figura 3.12 */ }
14    public BigRational( String str )
15        { /* Implementación en la Figura 3.12 */ }
16
17    private void check00( )
18        { /* Implementación en la Figura 3.12 */ }
19    private void fixSigns( )
20        { /* Implementación en la Figura 3.12 */ }
21    private void reduce( )
22        { /* Implementación en la Figura 3.12 */ }
23
24    public BigRational abs( )
25        { return new BigRational( num.abs( ), den ); }
26    public BigRational negate( )
27        { return new BigRational( num.negate( ), den ); }
28
29    public BigRational add( BigRational other )
30        { /* Implementación en la Figura 3.13 */ }
31    public BigRational subtract( BigRational other )
32        { /* Implementación en la Figura 3.13 */ }
33    public BigRational multiply( BigRational other )
34        { /* Implementación en la Figura 3.13 */ }
35    public BigRational divide( BigRational other )
36        { /* Implementación en la Figura 3.13 */ }
37
38    public boolean equals( Object other )
39        { /* Implementación en la Figura 3.14 */ }
40    public String toString( )
41        { /* Implementación en la Figura 3.14 */ }
42
43    private BigInteger num; // solo esto puede ser negativo
44    private BigInteger den; // nunca negativo
45 }
```

Figura 3.11 La clase BigRational, con una implementación parcial.

```
1  public BigRational( BigInteger n, BigInteger d )
2  {
3      num = n; den = d;
4      check00( ); fixSigns( ); reduce( );
5  }
6
7  public BigRational( String str )
8  {
9      if( str.length( ) == 0 )
10         throw new IllegalArgumentException( "Zero-length string" );
11
12     // Buscar '/'
13     int slashIndex = str.indexOf( '/' );
14     if( slashIndex == -1 )
15     {
16         num = new BigInteger( str.trim( ) );
17         den = BigInteger.ONE; // No hay denominador... usar 1
18     }
19     else
20     {
21         num = new BigInteger( str.substring( 0, slashIndex ).trim( ) );
22         den = new BigInteger( str.substring( slashIndex + 1 ).trim( ) );
23         check00( ); fixSigns( ); reduce( );
24     }
25 }
26
27 private void check00( )
28 {
29     if( num.equals( BigInteger.ZERO ) && den.equals( BigInteger.ZERO ) )
30         throw new ArithmeticException( "ZERO DIVIDE BY ZERO" );
31 }
32
33 private void fixSigns( )
34 {
35     if( den.compareTo( BigInteger.ZERO ) < 0 )
36     {
37         num = num.negate( );
38         den = den.negate( );
39     }
40 }
41
42 private void reduce( )
43 {
44     BigInteger gcd = num.gcd( den );
45     num = num.divide( gcd );
46     den = den.divide( gcd );
47 }
```

---

Figura 3.12 Los constructores de BigRational y métodos check00, fixSigns y reduce.

Finalmente, `equals` y `toString` están implementados en la Figura 3.14. La signatura de `equals`, como hemos explicado anteriormente, requiere un parámetro de tipo `Object`. Después de una comprobación `instanceof` estándar y una conversión de tipo podemos comparar los numeradores y denominadores. Observe que utilizamos `equals` (`==`) para comparar los numeradores y denominadores, y observe también que, como los objetos `BigRational` están siempre en forma simplificada, la prueba es relativamente simple. Para `toString`, que devuelve una representación `String` del objeto `BigRational`, la implementación podría ser una sola línea, pero hemos añadido código para manejar los valores infinito y  $-\infty$ , así como para no proporcionar como salida el denominador en caso de que tenga el valor 1.

```
1  public BigRational add( BigRational other )
2  {
3      BigInteger newNumerator =
4          num.multiply( other.den ).add(
5              other.num.multiply( den ) );
6      BigInteger newDenominator = den.multiply( other.den );
7
8      return new BigRational( newNumerator, newDenominator );
9  }
10
11 public BigRational subtract( BigRational other )
12 {
13     return add( other.negate() );
14 }
15
16 public BigRational multiply( BigRational other )
17 {
18     BigInteger newNumer = num.multiply( other.num );
19     BigInteger newDenom = den.multiply( other.den );
20
21     return new BigRational( newNumer, newDenom );
22 }
23
24 public BigRational divide( BigRational other )
25 {
26     BigInteger newNumer = num.multiply( other.den );
27     BigInteger newDenom = den.multiply( other.num );
28
29     return new BigRational( newNumer, newDenom );
30 }
```

Figura 3.13 Métodos `add`, `subtract`, `multiply` y `divide` de `BigRational`.

```
1  public boolean equals( Object other )
2  {
3      if( ! ( other instanceof BigRational ) )
4          return false;
5
6      BigRational rhs = (BigRational) other;
7
8      return num.equals( rhs.num ) && den.equals( rhs.den );
9  }
10
11 public String toString( )
12 {
13     if( den.equals( BigInteger.ZERO ) )
14         if( num.compareTo( BigInteger.ZERO ) < 0 )
15             return "-infinity";
16         else
17             return "infinity";
18
19     if( den.equals( BigInteger.ONE ) )
20         return num.toString( );
21     else
22         return num + "/" + den;
23 }
```

---

Figura 3.14 Métodos `equals` y `toString` de `BigRational`.

Observe que la clase `BigRational` no tiene métodos mutadores: rutinas como `add` simplemente devuelven un nuevo `BigRational` que representa una suma. Por tanto, `BigRational` es un tipo inmutable.

## 3.8 Paquetes

Un paquete se utiliza para organizar una colección de clases.

Los *paquetes* se utilizan para organizar clases similares. Cada paquete está compuesto por un conjunto de clases. Dos clases contenidas en el mismo paquete tienen restricciones de visibilidad ligeramente menores entre ellas mismas que las que tendrían si se encontraran en paquetes distintos.

Java proporciona varios paquetes predefinidos, incluyendo `java.io`, `java.lang` y `java.util`. El paquete `java.lang` incluye, entre otras, las clases `Integer`, `Math`, `String` y `System`. Algunas de las clases del paquete `java.util` son `Date`, `Random` y `Scanner`. El paquete `java.io` se utiliza para la E/S e incluye las diversas clases de flujos de datos vistos en la Sección 2.6.

La clase `C` del paquete `p` se especifica como `p.C`. Por ejemplo, podemos construir un objeto `Date` con la fecha y hora actuales como estado inicial utilizando

```
java.util.Date today = new java.util.Date( );
```

Observe que al incluir un nombre de paquete, evitamos conflictos con otras clases de nombre idéntico contenidas en otros paquetes (como por ejemplo nuestra propia clase `Date`). Observe también el convenio de denominación típico: los nombres de las clases empiezan por mayúscula y los nombres de paquete no.

Por convenio, los nombres de las clases empieza por mayúscula y los nombres de paquetes no.

### 3.8.1 La directiva `import`

Utilizar un nombre de paquete y de clase completo puede ser engorroso. Para evitarlo, utilice la directiva `import`. Hay dos formas de la directiva `import` que permiten al programador especificar una clase sin utilizar como prefijo el nombre del paquete.

```
import nombrePaquete.NombreClase;  
import nombrePaquete.*;
```

La directiva `import` se utiliza para proporcionar una abreviatura para un nombre de clase completamente cualificado.

En la primera forma, `NombreClase` puede utilizarse como abreviatura en lugar del nombre de clase completamente cualificado. En la segunda forma, todas las clases de un paquete pueden abreviarse mediante el nombre de clase correspondiente. Por ejemplo, con las directivas `import`

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

podemos usar

```
Date today = new Date( );  
FileReader theFile = new FileReader( name );
```

La utilización de la directiva `import` nos ahorra esfuerzo de teclado. Puesto que el mayor ahorro se consigue aplicando la segunda forma, podrá comprobar que esa forma se emplea a menudo en muchos programas. Las directivas `import` tienen dos desventajas. En primer lugar, la abreviatura hace que resulte difícil, leyendo el código, determinar qué clase es la que está siendo utilizada cuando existen múltiples directivas `import`. Además, la segunda forma puede permitir que se usen abreviaturas para clases que no deseábamos e introducir conflictos de denominación que tendremos que resolver empleando nombres de clase completamente cualificados.

El uso descuidado de la directiva `import` puede introducir conflictos de denominación.

Suponga que utilizamos

```
import java.util.*; // Paquete de librería  
import weiss.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 tenemos nuestra propia clase `Random` en `weiss.util`, la directiva `import` generará un conflicto con `weiss.util.Random` y tendremos que cualificar completamente el nombre de la clase. Además, si estamos utilizando una clase perteneciente a uno de estos paquetes,

al leer el código no se podrá determinar si esa clase proviene del paquete de librería o de nuestro propio paquete. Podríamos haber evitado estos problemas si hubiéramos utilizado la forma

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

y por esta razón solo vamos a utilizar esa primera forma en este texto, evitando las directivas `import` con caracteres "comodín".

Java.lang.\* se importa de manera automática.

Las directivas `import` deben aparecer antes del comienzo de la declaración de clase. Hemos visto un ejemplo de esto en la Figura 2.19. Asimismo, todo el paquete `java.lang` se importa de manera automática. Esta es la razón por la que podemos utilizar abreviaturas como `Math.max`, `Integer.parseInt`, `System.out`, etc.

En las versiones de Java anteriores a Java 5, los miembros estáticos como `Math.max` e `Integer.MAX_VALUE` no podían abreviarse para utilizar simplemente `max` y `MAX_VALUE`. Los programadores que hacían un uso intensivo de la librería matemática habían estado esperando mucho tiempo a que se generalizara la directiva de importación y se permitiría utilizar métodos como `sin`, `cos`, `tan` en lugar de los nombres más largos `Math.sin`, `Math.cos`, `Math.tan`. En Java 5, se ha añadido esta característica al lenguaje a través de la directiva de importación estática. La *directiva de importación estática* permite acceder a los miembros estáticos (métodos y campos) y proporcionar explícitamente el nombre de la clase. La directiva de importación estática tiene dos formas: la de importación de un único miembro y la de importación comodín. Así,

```
import static java.lang.Math.*;
import static java.lang.Integer.MAX_VALUE;
```

permite al programador escribir `max` en lugar de `Math.max`, `PI` en lugar de `Math.PI` y `MAX_VALUE` en lugar de `Integer.MAX_VALUE`.

### 3.8.2 La instrucción package

La instrucción `package` indica que una clase es parte de un paquete. Debe preceder a la definición de la clase.

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

En este texto vamos a utilizar los dos paquetes utilizados en la Figura 3.15. Otros programas, incluyendo los programas de prueba y los programas de aplicación de la Parte Tres del libro, son clases autónomas que no forman parte de un paquete.

Paquete	Uso
<code>weiss.util</code>	Una reimplementación de un subconjunto del paquete <code>java.util</code> que contiene varias estructuras de datos.
<code>weiss.nonstandard</code>	Varias estructuras de datos en una forma simplificada, utilizando convenios no estándar que difieren de los de <code>java.util</code> .

Figura 3.15 Paquetes definidos en este texto.

```
1 package weiss.math;  
2  
3 import java.math.BigInteger;  
4  
5 public class BigRational  
6 {  
7 /* La clase completa se muestra en el código en línea*/  
8 }
```

Figura 3.16 Inclusión de la clase `BigRational` en el paquete `weiss.math`.

En la Figura 3.16 se muestra un ejemplo de cómo se utiliza la instrucción `package`, en este caso para incluir la clase `BigRational` en un nuevo paquete `weiss.math`.

### 3.8.3 La variable de entorno CLASSPATH

Los paquetes se buscan en las ubicaciones designadas en la variable `CLASSPATH`. ¿Qué significa esto? He aquí algunas posibles configuraciones de `CLASSPATH`, primero para un sistema Windows y en segundo lugar para un sistema Unix:

```
SET CLASSPATH=.;C:\bookcode\  
setenv CLASSPATH .:$HOME/bookcode/
```

La variable `CLASSPATH` especifica los archivos y directorios que hay que explorar para encontrar las clases.

En ambos casos, la variable `CLASSPATH` enumera directorios (o archivos jar<sup>2</sup>) que contienen los archivos de clase del paquete. Por ejemplo, si la variable `CLASSPATH` está corrompida, no se podrá ejecutar ni siquiera el programa más trivial, porque no podrá encontrarse el directorio actual.

Una clase en un paquete `p` debe estar en un directorio `p` que pueda ser encontrado buscando en la lista de `CLASSPATH`; cada `.` en el nombre del paquete representa un subdirectorio. A partir de Java 1.2, el directorio actual (directorío `.`) se explora siempre si `CLASSPATH` no está configurada, así que si estamos trabajando desde un único directorio principal, basta simplemente con crear subdirectorios en él y no configurar `CLASSPATH`. Lo más probable, sin embargo, es que queramos crear un subdirectorío Java separado y luego crear subdirectorios de paquete dentro de él. Entonces lo que haríamos sería ampliar la variable `CLASSPATH` para incluir `.` y el subdirectorío Java. Esto es lo que hacímos en la anterior declaración Unix al añadir `$HOME/bookcode/` a la variable `CLASSPATH`. Dentro del directorio `bookcode`, crearemos un subdirectorío llamado `weiss`, y dentro de él otros subdirectorios denominados `math`, `util` y `nonstandard`. En el subdirectorío `math`, incluiremos el código para la clase `BigRational`.

Una clase en un paquete `p` debe estar en un directorio `p` que pueda ser encontrado buscando en la lista de `CLASSPATH`.

Una aplicación, escrita en cualquier directorio, podrá entonces utilizar la clase `BigRational` bien con su nombre completo.

<sup>2</sup> Un archivo *jar* es básicamente un archivo comprimido (como un archivo zip), con archivos adicionales que contienen información específica de Java. La herramienta *jar*, suministrada con el JDK, puede utilizarse para crear y expandir archivos *jar*.

```
weiss.math.BigRational;
```

o simplemente utilizando `BigRational`, si se proporciona una directiva `import` apropiada.

Mover una clase de un paquete a otro puede ser tedioso, porque puede requerir revisar una secuencia de directivas `import`. Muchas herramientas de desarrollo realizan esta tarea automáticamente como una de las opciones del proceso de *refactorización*.

### 3.8.4 Reglas de visibilidad de paquete

Los campos sin modificadores de visibilidad tienen *visibilidad de paquete*, lo que quiere decir que solo son visibles para otras clases que están en el mismo paquete.

Las clases no públicas solo son visibles para otras clases que se encuentren en el mismo paquete.

Los paquetes tienen varias reglas importantes de visibilidad. En primer lugar, si no se especifica ningún modificador de visibilidad para un campo, entonces el campo tendrá *visibilidad de paquete*. Esto quiere decir que será visible únicamente para las restantes clases del mismo paquete. Esto significa una mayor visibilidad que con `private` (que es invisible incluso para otras clases del mismo paquete), pero una menor visibilidad que con `public` (que es visible también para las clases que no son de ese paquete).

En segundo lugar, fuera del paquete solo se pueden utilizar las clases públicas de ese paquete. Esa es la razón por la que a menudo hemos utilizado el calificador `public` antes de `class`. Las clases no pueden declararse como `private`.<sup>3</sup> El acceso con visibilidad de paquete también se extiende a las clases. Si una clase no se declara `public`, entonces solo podrá ser accedida por otras clases del mismo paquete; será entonces una *clase con visibilidad de paquete*. En la Parte Cuatro, veremos que las clases con visibilidad de paquete pueden utilizarse sin violar el principio de ocultación de la información. Por tanto, existen algunos casos en los que las clases con visibilidad de paquete pueden resultar muy útiles.

Todas las clases que no forman parte de un paquete pero que son alcanzables a través de la variable `CLASSPATH` se consideran parte del mismo paquete predeterminado. Como resultado, la visibilidad de paquete se aplica entre todas ellas. Esta es la razón por la que la visibilidad no se ve afectada si se omite el modificador `public` en las clases que no pertenezcan a un paquete. Sin embargo, este es un uso poco conveniente del acceso a miembros con visibilidad de paquete. Solamente lo empleamos para colocar varias clases en un mismo archivo, porque eso hace que el examen y la impresión de los ejemplos sean más sencillos. Puesto que una clase `public` debe estar en un archivo con el mismo nombre, solo puede haber una clase pública por cada archivo.

## 3.9 Un patrón de diseño: compuesto (par)

Aunque el diseño y la programación de software representan a menudo un enorme desafío, muchos ingenieros de software experimentados argumentan que la ingeniería del software solo trata, en realidad, con un conjunto relativamente pequeño de problemas básicos. Quizá esto sea una exageración, pero es cierto que muchos problemas básicos se presentan una y otra vez en los proyectos software. Los ingenieros software que están familiarizados con estos problemas

<sup>3</sup> Esto se aplica a las clases de alto nivel que hemos visto hasta ahora. Posteriormente, veremos las clases anidadas e internas, que sí que pueden declararse como `private`.

y, en particular, con los esfuerzos que otros programadores han hecho a la hora de resolver esos problemas, tienen la ventaja de no tener que "reinventar la rueda".

La idea de los patrones de diseño consiste en documentar un problema y su solución de modo que otros puedan aprovecharse de la experiencia colectiva de toda la comunidad del ingeniería del software. Escribir un patrón se parece bastante a escribir una receta de un libro de cocina; se han descrito muchos patrones comunes y, en lugar de invertir energía en reinventar la rueda, pueden utilizarse esos patrones para escribir mejores programas. Por tanto, un *patrón de diseño* describe un problema que se presenta una y otra vez en la ingeniería del software y luego describe la solución de una manera lo suficientemente genérica como para poder aplicarla en una amplia variedad de contextos.

A lo largo del libro analizaremos diversos problemas que surgen a menudo dentro de un diseño, junto con una solución típica utilizada para resolver dichos problemas. Vamos a comenzar con el siguiente problema simple.

En la mayoría de los lenguajes, una función solo puede devolver un único objeto. ¿Qué podemos hacer si necesitamos devolver dos o más cosas? La forma más fácil de conseguir esto consiste en combinar los objetos en un único objeto, utilizando una matriz o una clase. La situación más común en la que hace falta devolver múltiples objetos es el caso de dos objetos. Por tanto, un patrón de diseño común consiste en devolver los dos objetos como un *par*. Este es el *patrón compuesto*.

Además de la situación que acabamos de describir, los pares son útiles para implementar mapas y diccionarios. En estas dos abstracciones, lo que hacemos es mantener parejas clave-valor: los pares se añaden al mapa o diccionario y luego buscamos una clave, obteniendo como resultado su valor. Una forma común de implementar un mapa consiste en implementar un conjunto. En un conjunto, tenemos una colección de elementos y lo que hacemos es buscar correspondencias. Si esos elementos son parejas y el criterio de búsqueda de correspondencias se basa exclusivamente en la componente de clave de la pareja, entonces es sencillo escribir una clase que construya un mapa basándose en un conjunto. Veremos esta idea con más detalle en el Capítulo 19.

Un patrón de diseño describe un problema que se presenta una y otra vez en la ingeniería del software, y luego describe la solución de una forma lo suficientemente genérica como para poder aplicarla en una amplia variedad de contextos.

Un patrón de diseño común consiste en devolver dos objetos como un par.

Los pares son útiles para implementar parejas clave-valor en mapas y diccionarios.

## Resumen

En este capítulo hemos descrito las estructuras sintácticas de Java para clases y paquetes. La clase es el mecanismo Java utilizado para crear nuevos tipos de referencia; el paquete se utiliza para agrupar clases relacionadas. Para cada clase, podemos

- Definir la construcción de objetos.
- Encargarnos de garantizar la ocultación de información y la atomicidad.
- Definir métodos para manipular los objetos.

La clase consta de dos partes: la especificación y la implementación. La especificación le dice al usuario de la clase lo que esa clase hace; la implementación se encarga de hacerlo. La implementación suele contener código confidencial y en algunos casos solo se distribuye como un archivo .class. Sin embargo, la especificación es de conocimiento público. En Java, puede generarse a partir de la implementación que enumere los métodos de la clase utilizando *javadoc*.

La ocultación de la información puede conseguirse utilizando la palabra clave `private`. La inicialización de los objetos está controlada por los constructores, y los componentes de cada objeto pueden examinarse o modificarse mediante los métodos accesores y mutadores, respectivamente. La Figura 3.17 ilustra muchos de estos conceptos, tal como se aplican a una versión simplificada de `ArrayList`. Esta clase, `StringArrayList`, soporta `add`, `get` y `size`. En el código en línea podrá encontrar una versión más completa que incluye `set`, `remove` y `clear`.

Las características expuestas en este capítulo implementan los aspectos fundamentales de la programación basada en objetos. En el siguiente capítulo hablaremos de la herencia, que es fundamental para la programación orientada a objetos.

```
1 /**
2 * StringArrayList implementa una matriz ampliable de objetos String.
3 * Las inserciones siempre se hacen al final.
4 */
5 public class StringArrayList
6 {
7     /**
8      * Devuelve el número de elementos de esta colección.
9      * @return el número de elementos de esta colección.
10     */
11    public int size( )
12    {
13        return theSize;
14    }
15
16    /**
17     * Devuelve el elemento en la posición idx.
18     * @param idx el índice que se quiere buscar.
19     * @throws ArrayIndexOutOfBoundsException si el índice es incorrecto.
20     */
21    public String get( int idx )
22    {
23        if( idx < 0 || idx >= size( ) )
24            throw new ArrayIndexOutOfBoundsException( );
25        return theItems[ idx ];
26    }
```

Continúa

---

Figura 3.17 StringArrayList simplificado con add, get y size.

```
27
28     /**
29      * Añade un elemento al final de esta colección.
30      * @param x cualquier objeto.
31      * @return true (como con java.util.ArrayList).
32     */
33     public boolean add( String x )
34     {
35         if( theItems.length == size( ) )
36         {
37             String [ ] old = theItems;
38             theItems = new String[ theItems.length * 2 + 1 ];
39             for( int i = 0; i < size( ); i++ )
40                 theItems[ i ] = old[ i ];
41         }
42
43         theItems[ theSize++ ] = x;
44         return true;
45     }
46
47     private static final int INIT_CAPACITY = 10;
48
49     private int theSize = 0;
50     private String [ ] theItems = new String[ INIT_CAPACITY ];
51 }
```

Figura 3.17 (Continuación).



## Conceptos clave

**acceso con visibilidad de paquete** Los miembros que no tienen modificadores de visibilidad solo son accesibles para los métodos de las clases pertenecientes al mismo paquete. (94)

**accesor** Un método que examina un objeto pero no cambia su estado. (76)

**alias** Un caso especial que se produce cuando el mismo objeto aparece desempeñando más de un papel. (81)

**campo** Un miembro de una clase que almacena datos. (73)

**campo estático** Un campo compartido por todas las instancias de una clase. (83)

**clase** Está compuesta por campos y métodos que se aplican a instancias de la clase. (71)

**clase con visibilidad de paquete** Una clase que no es pública y solo es accesible desde otras clases contenidas en el mismo paquete. (94)

**CLASSPATH, variable** Especifica los directorios y archivos en los que hay que buscar para encontrar las clases. (93)

**constructor** Establece cómo se declara e inicializa un objeto. El constructor predeterminado consiste en una inicialización predeterminada miembro a miembro, en la que los campos primitivos se inicializan a cero y los campos de referencia se inicializan con null. (76)

**encapsulación** La agrupación de datos y las operaciones que se aplican a los mismos con el fin de formar un agregado, al mismo tiempo que se oculta la implementación del agregado. (70)

**equals, método** Puede implementarse para ver si dos objetos representan el mismo valor. El parámetro formal es siempre de tipo Object. (78)

**especificación de clase** Describe la funcionalidad, pero no la implementación. (73)

**implementación** Representa las interioridades relativas a cómo se cumplen las especificaciones. En lo que respecta al usuario de la clase, la implementación no es importante. (73)

**import, directiva** Utilizada para proporcionar una abreviatura para un nombre de clase completamente cualificado. Java 5 añade el mecanismo de importación estática, que permite utilizar una abreviatura para un miembro estático. (91)

**inicializador estático** Un bloque de código utilizado para inicializar campos estáticos. (85)

**instanceof, operador** Comprueba si una expresión es una instancia de una clase. (82)

**javadoc** Genera automáticamente documentación para las clases. (73)

**Llamada a this en un constructor** Utilizada para hacer una llamada a otro constructor de la misma clase. (82)

**marcador javadoc** Como ejemplos tendríamos @author, @param, @return y @throws. Se utilizan dentro de los comentarios javadoc. (73)

**método** Una función suministrada como miembro que, si no es estática, opera sobre una instancia de la clase. (71)

**método estático** Un método que no tiene referencia this implícita y que por tanto puede invocarse sin una referencia a un objeto controlador. (82)

**miembros de instancia** Miembros declarados sin el modificador estático. (82)

**mutador** Un método que modifica el estado del objeto. (76)

**objeto** Una entidad que tiene una estructura y un estado y define operaciones que pueden acceder a dicho estado o manipularlo. Una instancia de una clase. (70)

**ocultación de información** Hace que sean inaccesibles los detalles de implementación, incluyendo los componentes de un objeto. (70)

**package, instrucción** Indica que una clase es un miembro de un paquete. Debe preceder a la definición de la clase. (92)

**paquete** Se utiliza para organizar una colección de clases. (90)

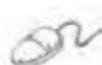
**par** El patrón compuesto por dos objetos. (95)

- patrón compuesto** El patrón en el que almacenamos dos o más objetos en una entidad. (95)
- patrón de diseño** Describe un problema que se presenta una y otra vez en la ingeniería del software, y luego describe la solución de una forma lo suficientemente genérica como para poder aplicarla en una amplia variedad de contextos. (95)
- privado** Un miembro que no es visible para los métodos que no son de esa clase. (72)
- programación basada en objetos** Utiliza los mecanismos de encapsulación y de ocultación de la información de los objetos, pero no emplea la herencia. (70)
- programación orientada a objetos** Se distingue de la programación basada en objetos porque emplea la herencia para formar jerarquías de clases. (69)
- público** Un miembro que es visible para los métodos que no son de esa clase. (71)
- this, referencia** Una referencia al objeto actual. Puede utilizarse para enviar el objeto actual como una unidad a algún otro método. (81)
- toString, método** Devuelve un objeto `String` basado en el estado del objeto. (78)
- unidad atómica** En referencia a un objeto, sus partes no pueden ser diseccionadas por la generalidad de los usuarios del objeto. (70)



## Errores comunes

1. No se puede acceder a los miembros desde fuera de la clase. Recuerde que, de manera predeterminada, los miembros de la clase tienen visibilidad de paquete: solo son visibles dentro del paquete.
2. Utilice `public class` en lugar de `class`, a menos que esté escribiendo una clase auxiliar de la que luego vaya a prescindir.
3. El parámetro formal de `equals` debe ser de tipo `Object`. En caso contrario, aunque el programa se podrá compilar, habrá situaciones en las que se utilizará un `equals` predeterminado (que simplemente se comporta como `==`) en lugar del método definido por nosotros.
4. Los métodos estáticos no pueden acceder a miembros no estáticos sin un objeto controlador.
5. Las clases que forman parte de un paquete deben colocarse en un directorio con nombre idéntico que sea alcanzable desde `CLASSPATH`.
6. `this` es una referencia final y no puede ser modificada.
7. Los constructores no tienen tipos de retorno. Si se escribe un "constructor" con el tipo de retorno `void`, lo que se habrá escrito es en realidad un método con el mismo nombre que la clase, pero eso *NO*es un constructor.



## Internet

A continuación se indican los archivos disponibles para este capítulo.

### **TestIntCell.java**

Contiene un método `main` que comprueba `IntCell`, mostrado en la Figura 3.3.

### **IntCell.java**

Contiene la clase `IntCell`, mostrada en la Figura 3.4. La salida de `javadoc` también se puede encontrar en `IntCell.html`.

### **Date.java**

Contiene la clase `Date`, mostrada en la Figura 3.6.

### **BigRational.java**

Contiene la clase `BigRational`, mostrada en la Figura 3.7 y que está incluida en el paquete `weiss.math`.

### **Ticket.java**

Contiene el ejemplo del miembro estático de `Ticket` de la Figura 3.9.

### **Squares.java**

Contiene el código de ejemplo de un inicializador estático de la Figura 3.10.

### **StringArrayList.java**

Contiene una versión más completa del código de `StringArrayList` de la Figura 3.17.

### **ReadStringsWithStringArrayList.java**

Contiene un programa de prueba para `StringArrayList`.



## Ejercicios

### EN RESUMEN

- 31** Explique los usos de `this` en Java.
- 32** Describa la función del constructor.
- 33** Si una clase no proporciona ningún constructor, ¿cuál es el resultado?
- 34** Explique las secciones pública y privada de una clase.
- 35** ¿Qué es la *ocultación de información*? ¿Qué es la *encapsulación*? ¿Cómo soporta Java estos conceptos?
- 36** Indique las dos formas de directiva `import` que permiten utilizar `BigRational` sin proporcionar el nombre de paquete `weiss.math`.
- 37** ¿En qué circunstancias puede un método estático hacer referencia a un campo de instancia perteneciendo a la misma clase?
- 38** ¿Qué es el *acceso con visibilidad de paquete*?
- 39** ¿Qué sucede si, al tratar de escribir un constructor, se incluye un tipo de retorno `void`?

```
1 class Person
2 {
3     public static final int NO_SSN = -1;
4
5     private int SSN = 0;
6     String name = null;
7 }
8
9 class TestPerson
10 {
11     private Person p = new Person();
12
13     public static void main( String [ ] args )
14     {
15         Person q = new Person();
16
17         System.out.println( p );           // illegal
18         System.out.println( q );           // legal
19
20         System.out.println( q.NO_SSN );    // ?
21         System.out.println( q.SSN );      // ?
22         System.out.println( q.name );     // ?
23         System.out.println( Person.NO_SSN ); // ?
24         System.out.println( Person.SSN );  // ?
25     }
26 }
```

Figura 3.18 Código para el Ejercicio 3.11.

**310** ¿Qué es un *patrón de diseño*?

**311** Para el código de la Figura 3.18, que reside completamente en un solo archivo:

- La línea 17 es ilegal, aun cuando la línea 18 es legal. Explique por qué.
- ¿Cuáles de las líneas 20 a 24 son legales y cuáles no? Explique por qué.

**312** ¿Cuál es la diferencia entre un campo de instancia y un campo estático?

**313** Para una clase NombreClase, ¿cómo se realiza la salida?

## EN TEORÍA

**314** Suponga que compilamos los códigos de la Figura 3.3 (`TestIntCe11`) y de la Figura 3.4 (`IntCe11`). A continuación, modificamos la clase `IntCe11` de la Figura 3.4 añadiendo un constructor de un parámetro (eliminando por tanto el constructor predeterminado de cero parámetros). Por supuesto, si volvemos a compilar

`TestIntCell`, se producirá un error de compilación. Pero si `TestIntCell` no se compila de nuevo y recompilamos solo `IntCell` no habrá ningún error. ¿Qué sucederá cuando se ejecute entonces `TestIntCell`.

- 315** ¿Es legal la siguiente directiva de importación, que trata de importar directamente toda la librería Java?

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

- 316** Suponga que el método `main` de la Figura 3.3 fuera parte de la clase `IntCell`.

- Continuará funcionando el programa?
- Se podría activar la línea de `main` que ha sido desactivada mediante un comentario, sin generar ningún error?

- 317** Una clase proporciona un único constructor privado. ¿En qué sentido podría ser esto útil?

## EN LA PRÁCTICA

- 318** Un `BinaryArray` representa secuencias arbitrariamente largas de variables binarias. La representación privada de los datos es una matriz de variables booleanas. Por ejemplo, la representación del `BinaryArray` "TFTTF" sería una matriz de longitud cinco que almacenaría `true, false, true, true, false` en los índices 0, 1, 2, 3 y 4 de la matriz, respectivamente. La clase `BinaryArray` tiene la siguiente funcionalidad:

- Un constructor de un único parámetro que contiene un objeto `String`. Genera una excepción `IllegalArgumentException` si hay caracteres ilegales.
- Sendos métodos `get` y `set` para acceder a una variable en un índice concreto o modificarla.

Implemente la clase `BinaryArray`, incluyéndola en un paquete de su elección.

- 319** El paquete `java.math` contiene una clase `BigDecimal`, utilizada para representar un número decimal de precisión arbitraria. Lea la documentación de `BigDecimal` y responda a las siguientes preguntas:

- ¿Es `BigDecimal` una clase inmutable?
- Si `bd1.equals(bd2)` es `true`, ¿cómo será `bd1.compareTo(bd2)`?
- Si `bd1.compareTo(bd2)` es 0, ¿cuándo dará `bd1.equals(bd2)` como resultado `false`?
- Si `bd1` representa 1.0 y `bd2` representa 7.0, ¿qué valor tendrá de manera predeterminada `bd1.divide(bd2)`?
- Si `bd1` representa 1.0 y `bd2` representa 8.0, ¿qué valor tendrá de manera predeterminada `bd1.divide(bd2)`?

- 320** Escriba un programa que lea un archivo de datos que contenga números racionales, uno por línea, que almacene los números en un `ArrayList`, que elimine los duplicados y que luego muestre la suma, la media aritmética y la media armónica de los restantes números racionales distintos.

- 3.21** Modifique la clase `BigRational` de modo que `0/0` sea legal y sea interpretado como "indeterminado" por `toString`.
- 3.22** Mueva la clase `IntCell` (Figura 3.3) al paquete `weiss.nonstandard` y revise `TestIntCell` (Figura 3.4) de la forma correspondiente.
- 3.23** Una clase `Account` almacena el saldo actual y proporciona los métodos `getBalance` (obtener saldo), `deposit` (depositar fondos), `withdraw` (retirar fondos) y `toString` además de al menos un constructor. Escriba y pruebe una clase `Account`. Asegúrese que su método de retirada de fondos genera una excepción en el caso apropiado.
- 3.24** Una caja fuerte tiene las siguientes propiedades básicas: la combinación (una secuencia de tres números) está oculta. La caja puede abrirse proporcionando la combinación; la combinación puede ser modificada, pero solo por alguien que conozca la combinación actual. Diseñe una clase con los métodos públicos `open` (abrir) y `changeCombo` (cambiar combinación) y con campos privados de datos que almacenen la combinación. La combinación debe configurarse en el constructor.
- 3.25** Añada los siguientes métodos a la clase `BigRational`, asegurándose de generar las excepciones apropiadas:

```
BigRational pow( int exp ) // excepción si exp<0  
BigInteger toBigInteger( ) // excepción si el denominador no es 1  
int toInteger( ) // excepción si el denominador no es 1
```

- 3.26** Las directivas de importación comodín son peligrosas debido a que pueden introducir ambigüedades y otras sorpresas. Recuerde que tanto `java.awt.List` como `java.util.List` son clases. Partiendo del código de la Figura 3.19:
- Compile el código; debería obtener una ambigüedad.
  - Añada una directiva `import` para utilizar explícitamente `java.awt.List`. El código debería ahora poder compilarse y ejecutarse.

```
1 import java.util.*;  
2 import java.awt.*;  
3  
4 class List // Desactive esta clase con un comentario  
5 {  
6     public String toString( ) { return "My List!!"; }  
7 }  
8  
9 class WildCardIsBad  
10 {  
11     public static void main( String [ ] args )  
12     {  
13         System.out.println( new List( ) );  
14     }  
15 }
```

Figura 3.19 Este código para el Ejercicio 3.26 ilustra por qué las importaciones comodín no son convenientes.

- c. Quite el comentario que desactiva la clase local `List` y elimine la directiva `import` que acaba de añadir. El código se debería poder compilar y ejecutar.
  - d. Vuelva a desactivar la clase local `List` con un comentario, volviendo a la situación del principio. Compile de nuevo para ver el sorprendente resultado. ¿Qué sucede si añade la directiva `import` explícita del paso (b)?
- 3.27** Para la clase `BigRational`, añada un constructor adicional que acepte dos objetos `BigInteger` como parámetros y asegúrese de generar las excepciones apropiadas.

## PROYECTOS DE PROGRAMACIÓN

- 3.28** Suponga que deseamos imprimir una matriz bidimensional en la que todos los números están comprendidos entre 0 y 999. La forma normal de imprimir cada número podría hacer que la matriz no estuviera alineada. Por ejemplo,

```
54 4 12 366 512
756 192 18 27 4
14 18 99 300 18
```

Examine la documentación del método `format` de la clase `String` y escriba una rutina que imprima la matriz bidimensional con un formato más elegante, como por ejemplo

```
054 004 012 366 512
756 192 018 027 004
014 018 099 300 018
```

- 3.29** Implemente una clase `IntType` completa que soporte un conjunto razonable de constructores y los métodos `add`, `subtract`, `multiply`, `divide`, `equals`, `compareTo` y `toString`. Mantenga un `IntType` en forma de una matriz suficientemente grande. Para esta clase, la operación difícil es la división, seguida de cerca por la multiplicación.

- 3.30** En ocasiones, un número complejo se representa mediante su módulo y un ángulo (en el rango semiabierto que va de 0 a 360 grados). Proporcione una implementación de una clase `BigComplex`, en la que la representación de los datos sea un `BigDecimal` para representar el módulo y un número de coma flotante de doble precisión para representar el ángulo.

- 3.31** Implemente una clase, `Polynomial`, para representar polinomios de una sola variable y escriba un programa de pruebas. La funcionalidad de la clase `Polynomial` es la siguiente:

- Proporciona al menos tres constructores: un constructor de cero parámetros que hace que el polinomio sea cero, un constructor que crea una copia independiente separada de un polinomio existente y un constructor que crea un polinomio basado en una especificación `String`. El último constructor puede generar una excepción si la especificación `String` no es válida y dejamos a criterio del lector decidir qué es una especificación válida.
- `negate` devuelve el negado del polinomio que se proporciona como parámetro.

- `add`, `subtract` y `multiply` devuelven un nuevo polinomio que es igual a la suma, diferencia o producto, respectivamente, de este polinomio y de otro polinomio, `rhs`. Ninguno de estos métodos modifica ninguno de los polinomios originales.
  - `equals` y `toString` siguen la especificación estándar para estas funciones. Para `toString` haga que la representación en forma de cadena de caracteres tenga el mejor formato posible.
  - El polinomio está representado por dos campos. Uno, `degree`, representa el grado del polinomio. Por tanto,  $x^2 + 2x + 1$  es de grado 2,  $3x + 5$  es de grado 1 y 4 es de grado 0. Cero es automáticamente de grado 0. El segundo campo, `coeff`, representa los coeficientes (`coeff[1]` representa el coeficiente de  $x^1$ ).
- 3.32** Modifique la clase del ejercicio anterior para almacenar los coeficientes como objetos `BigRational`.
- 3.33** Un objeto `PlayingCard` representa una carta utilizada en juegos como el póker y el black jack, y almacena el palo (corazones, diamantes, tréboles o picas) y el valor (2 a 10, sota, reina, rey o as). Un objeto `Deck` representa un mazo completo de 52 cartas `PlayingCard`. Un objeto `MultipleDeck` representa uno o más mazos (objetos `Deck`) de cartas (el número exacto se especifica en el constructor). Implemente las tres clases `PlayingCard`, `Deck` y `MultipleDeck`, proporcionando una funcionalidad razonable para `PlayingCard`, y para el caso de `Deck` y `MultipleDeck`, proporcionando mínimamente la capacidad de barajar, de repartir una carta y de comprobar si todavía quedan cartas.
- 3.34** Implemente una clase `Date` simple. Debería poder representar cualquier fecha desde el 1 de enero de 1800 hasta el 31 de diciembre de 2500; restar dos fechas; incrementar una fecha en un cierto número de días y comparar dos fechas utilizando tanto `equals` como `compareTo`. Un objeto `Date` se representa internamente como el número de días transcurridos desde un cierto momento inicial, que aquí es el inicio del año 1800. Esto hace que todos los métodos sean triviales, salvo los constructores y `toString`.

La regla para los años bisiestos es que un año es bisiesto si es divisible por 4 y no lo es por 100, a menos que también sea divisible por 400. Por tanto, 1800, 1900 y 2100 no son años bisiestos, pero 2000 sí lo es. El constructor debe comprobar la validez de la fecha, como también debe hacerlo `toString`. El objeto `Date` podría llegar a tener un valor incorrecto si un operador de incremento o de sustracción le hiciera salirse fuera del rango permitido.

Una vez que haya decidido las especificaciones, puede realizar una implementación. La parte difícil es convertir entre las representaciones interna y externa de una fecha. A continuación se presenta un posible algoritmo.

Configure dos matrices que sean campos estáticos. La primera matriz, `daysTillFirstOfMonth`, contendrá el número de días hasta el primero de cada mes en un año no bisiesto. Por tanto, contendrá 0, 31, 59, 90, etc. La segunda matriz, `daysTillJan1`, contendrá el número de días hasta el primer día de cada año, comenzando con `firstYear`. Por tanto, contendrá 0, 365, 730, 1095, 1460,

1826, y así sucesivamente, porque 1800 no es un año bisiesto, pero 1804 sí lo es. Debería hacer que su programa inicializara esta matriz una sola vez utilizando un inicializador estático. Después, puede emplear la matriz para convertir de la representación interna a la representación externa.

- 3.35** Un número complejo almacena una parte real y una parte imaginaria. Proporcione una implementación de una clase `BigComplex`, en la que la representación de los datos se analice mediante dos objetos `BigDecimal` que representen las partes real e imaginaria.



## referencias

Puede encontrar más información sobre las clases en las referencias especificadas al final del Capítulo 1. La referencia clásica sobre patrones de diseño es [1]. Este libro describe 23 patrones estándar, algunos de los cuales comentaremos posteriormente.

1. E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

## Herencia

Como hemos mencionado en el Capítulo 3, un objetivo importante de la programación orientada a objetos es la reutilización de código. Al igual que los ingenieros utilizan una y otra vez determinados componentes en sus diseños, los programadores deberían ser capaces de reutilizar los objetos, en lugar de tener que implementarlos repetidamente. En un lenguaje de programación orientado a objetos, el mecanismo fundamental de reutilización del código es la *herencia*. La herencia nos permite ampliar la funcionalidad de un objeto. En otras palabras, podemos crear nuevos tipos con propiedades restringidas (o ampliadas) con respecto al tipo original, formando en la práctica una jerarquía de clases.

Sin embargo, la herencia es algo más que una simple reutilización de código. Utilizando la herencia correctamente, el programador puede mantener y actualizar el código más fácilmente, tareas ambas que resultan esenciales en las aplicaciones comerciales de gran envergadura. Comprender el uso de la herencia es esencial para poder escribir programas Java de una cierta entidad, y los mecanismos de herencia también son empleados por Java para implementar clases y métodos genéricos.

En este capítulo, veremos

- Los principios generales de la herencia, incluyendo el *polimorfismo*.
- Cómo se implementa la herencia en Java.
- Cómo puede construirse 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 utilizando la herencia.
- Cómo implementa Java 5 la programación genérica utilizando clases genéricas.

### 4.1 ¿Qué es la herencia?

La *herencia* es el principio fundamental de orientación a objetos empleado para reutilizar código entre clases relacionadas. Los mecanismos de herencia modelan la *relación ES-UN*. En una relación *ES-UN* decimos que la clase derivada *es una* (variación de la) clase base. Por ejemplo, un Círculo *ES-UNA FormaGeometrica* y un Automóvil *ES-UN Vehículo*. Sin embargo, una Elipse *NO-ES-UN Círculo*. Las relaciones de herencia forman *jerarquías*. Por ejemplo, podemos ampliar Automóvil a

En una relación *ES-UN*, decimos que una clase derivada es una (variación de la) clase base.

En una relación *TIENE-UN*, decimos que una clase derivada tiene una (instancia de la) clase base. Para modelar las relaciones de tipo *TIENE-UN* se utiliza la técnica de composición.

otras clases, ya que un AutomovilDeImportacion *ES-UN* Automóvil (y paga impuestos especiales) y un AutomovilNacional *ES-UN* Automóvil (y no paga impuestos especiales), etc.

Otro tipo de relación es la *relación TIENE-UN* (o *ESTÁ-COMPUESTO-POR*). Este tipo de relación no posee las propiedades que son naturales dentro de una jerarquía de herencia. Un ejemplo de relación *TIENE-UN* es que un automóvil *TIENE-UN* volante. Las relaciones de tipo *TIENE-UN* no deben ser modeladas mediante la herencia. En lugar de ello, deberían emplear la técnica de *composición*, en la que los componentes se tratan simplemente como campos privados de datos.

Como veremos en los próximos capítulos, el propio lenguaje Java hace un amplio uso de la herencia a la hora de implementar sus librerías de clases.

#### 4.1.1 Creación de nuevas clases

Nuestras explicaciones acerca de la herencia se centrarán en un determinado ejemplo. En la Figura 4.1 se muestra una clase típica. La clase `Person` se emplea para almacenar información acerca de una persona; en nuestro caso, tenemos datos privados que incluyen el nombre, la edad, la dirección y el número de teléfono, junto con algunos métodos públicos que pueden acceder a esta información y posiblemente modificarla. Cabe imaginar que, en la práctica, esta clase sería significativamente más compleja, almacenando quizás unos 30 campos de datos, junto con unos 100 métodos.

Ahora suponga que queremos tener una clase `Student` para representar estudiantes o una clase `Employee` para representar empleados, o ambas. Imagine que un objeto `Student` es similar a un objeto `Person`, con la adición de solo unos cuantos métodos y miembros de datos. En nuestro sencillo ejemplo, imagine que la diferencia es que un objeto `Student` añade un campo `gpa` para almacenar la nota media y un método accesror `getGPA`. De forma similar, imagine que el objeto `Employee` tiene los mismos componentes que `Person`, pero que además dispone de un campo `salary` para representar el salario y de métodos para manipular ese salario.

Una opción a la hora de diseñar estas clases sería la técnica clásica de *copiar y pegar*: copiamos la clase `Person`, cambiamos el nombre de la clase y de los constructores y luego añadimos los nuevos miembros. Esta estrategia se ilustra en la Figura 4.2.

El recurrir a la técnica de copiar y pegar es una opción de diseño bastante poco conveniente, que presenta numerosas desventajas. En primer lugar, tenemos el problema de que si copiamos código erróneo, terminaremos teniendo más errores en el código. Esto hace que sea muy difícil corregir los errores de programación detectados, especialmente cuando se detectan de forma tardía.

En segundo lugar, tenemos el problema, relacionado con el anterior, del mantenimiento y el versionado. Supongamos que decidimos en una versión del programa que sería mejor almacenar los nombres con el formato apellido, nombre de pila, en lugar de emplear un único campo. O quizás, que sería mejor almacenar las direcciones utilizando una clase especial `Address`. Para poder mantener la coherencia, deberíamos realizar estos cambios en todas las clases. Utilizando la técnica de copiar y pegar, estos cambios de diseño tienen que efectuarse en numerosos sitios distintos.

Un tercer problema, quizás más sutil que los anteriores, es el que hecho de que empleando la técnica de copiar y pegar, `Person`, `Student` y `Employee` son tres entidades separadas sin ninguna relación entre sí, a pesar de sus similitudes. Así que, por ejemplo, si tenemos una rutina que acepta

```
1  class Person
2 {
3     public Person( String n, int ag, String ad, String p )
4         { name = n; age = ag; address = ad; phone = p; }
5
6     public String toString( )
7         { return getName( ) + " " + getAge( ) + " "
8             + getPhoneNumber( ); }
9
10    public String getName( )
11        { return name; }
12
13    public int getAge( )
14        { return age; }
15
16    public String getAddress( )
17        { return address; }
18
19    public String getPhoneNumber( )
20        { return phone; }
21
22    public void setAddress( String newAddress )
23        { address = newAddress; }
24
25    public void setPhoneNumber( String newPhone )
26        { phone = newPhone; }
27
28    private String name;
29    private int age;
30    private String address;
31    private String phone;
32 }
```

Figura 4.1 La clase Person almacena el nombre, la edad, la dirección y el número de teléfono.

un objeto Person como parámetro, no podríamos enviar a esa rutina un objeto Student. Por tanto, tendríamos que copiar y pegar todas esas rutinas para que funcionaran para esos nuevos tipos.

El mecanismo de herencia resuelve estos tres problemas. Utilizando la herencia, dirímos que un objeto Student *ES UN* objeto Person. A continuación, especificaremos los cambios que un objeto Student tiene en relación con un objeto Person. Solo se permiten tres tipos de cambios:

Student puede añadir nuevos campos, por ejemplo gpa.

Student puede añadir nuevos métodos, por ejemplo, getGPA.

Student puede sustituir métodos existentes, por ejemplo, *toString*.

```
1 class Student
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5         { name = n; age = ag; address = ad; phone = p; gpa = g; }
6
7     public String toString( )
8         { return getName( ) + " " + getAge( ) + " "
9             + getPhoneNumber( ) + " " + getGPA( ); }
10
11    public String getName( )
12        { return name; }
13
14    public int getAge( )
15        { return age; }
16
17    public String getAddress( )
18        { return address; }
19
20    public String getPhoneNumber( )
21        { return phone; }
22
23    public void setAddress( String newAddress )
24        { address = newAddress; }
25
26    public void setPhoneNumber( String newPhone )
27        { phone = newPhone; }
28
29    public double getGPA( )
30        { return gpa; }
31
32    private String name;
33    private int age;
34    private String address;
35    private String phone;
36    private double gpa
37 }
```

---

**Figura 4.2** La clase Student almacena el nombre, la edad, la dirección, el número de teléfono y la nota media mediante la técnica de copiar y pegar.

Hay dos cambios que están específicamente prohibidos porque violarían el concepto de una relación *ES-UN*:

Student no puede eliminar campos.

Student no puede eliminar métodos.

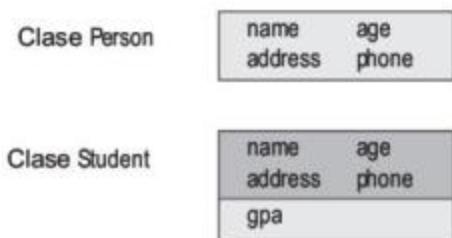
Por último, la nueva clase debe especificar sus propios constructores; es bastante probable que esto implique cierta sintaxis de la que hablaremos en la Sección 4.1.6.

La Figura 4.3 muestra la clase `Student`, mientras que en la Figura 4.4 se muestra la estructura de datos para las clases `Person` y `Student`. En esta última figura se ilustra que la huella de memoria de cualquier objeto `Student` incluye todos los campos que estarían contenidos en un objeto `Person`. Sin embargo, puesto que esos campos están declarados como privados por `Person`, no son accesibles desde los métodos de la clase `Student`. Esa es la razón por la que el constructor resulta problemático en este punto. No podemos tocar los campos de datos en ningún método de `Student`, y la única manera que tenemos de manipular los campos privados heredados es utilizando los métodos públicos de `Person`. Por supuesto, podríamos hacer que los campos heredados fueran públicos, pero eso resultaría, en general, una decisión de diseño errónea. Permitiría a los implementadores de las clases `Student` y `Employee` acceder directamente a los campos heredados. Pero, si hiciéramos eso

```

1 class Student extends Person
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5     {
6         /* ¡OJO! Hace falta cierta sintaxis; consulte la Sección 4.1.6 */
7         gpa = g; }
8
9     public String toString( )
10    { return getName( ) + " " + getAge( ) + " "
11      + getPhoneNumber( ) + " " + getGPA( ); }
12
13    public double getGPA( )
14    { return gpa; }
15
16    private double gpa;
17 }
```

**Figura 4.3** Herencia utilizada para crear la clase `Student`.



**Figura 4.4** Estructura de memoria utilizando herencia. El sombreado más claro indica campos que son privados y a los que solo se puede acceder mediante métodos de la clase. El sombreado más oscuro en la clase `Student` indica campos que no son accesibles desde la propia clase `Student`, pero sin embargo están presentes.

y posteriormente se realizaran modificaciones en la clase `Person`, como por ejemplo un cambio en la representación de los datos de nombre y dirección en `Person`, nos veríamos obligados a repasar todas las dependencias, lo que volvería a plantearnos los problemas relacionados con la técnica de copiar y pegar.

La herencia nos permite derivar clases a partir de una clase base sin perturbar la implementación de la clase base.

o pequeña que fuera la clase `Person` y tenemos la ventaja de la *reutilización directa del código* y de un fácil mantenimiento. Observe también que hemos hecho nuestra implementación sin perturbar la implementación de la clase existente.

Resumamos la sintaxis de lo que hemos visto hasta ahora. Una *clase derivada* hereda todas las propiedades de una clase base. Después podemos añadir miembros de datos, sustituir métodos y añadir métodos nuevos. Cada clase derivada es una clase completamente nueva.

En la Figura 4.5 se muestra la disposición típica en el caso de la herencia, utilizando la clase `extends`. Una cláusula `extends` declara que una clase se deriva de otra clase. Decimos que una clase derivada *extiende* (amplia) una clase base. He aquí una breve descripción de una clase derivada:

Una clase derivada hereda todos los miembros de datos de la clase base y puede añadir más miembros de datos.

■ Generalmente, todos los datos son privados, por lo que añadimos campos de datos adicionales en la clase derivada especificándolos dentro de la sección privada.

■ Cualquier método de la clase base que no se especifique en la clase derivada se heredará sin cambios, con la excepción del constructor. El caso especial del constructor se analiza en la Sección 4.1.6.

```

1 public class Derivada extends Base
2 {
3     // Cualquier miembro no enumerado se hereda sin modificaciones,
4     // salvo por el constructor.
5
6     // miembros públicos
7     // Constructor(es) si el predeterminado no es aceptable
8     // Métodos de Base cuyas definiciones deban cambiar en Derivada
9     // Métodos públicos adicionales
10
11    // miembros privados
12    // Campos de datos adicionales (generalmente privados)
13    // Métodos privados adicionales
14 }
```

Figura 4.5 Estructura general en el caso de herencia.

- Cualquier método de la clase base que se declara en la sección pública de la clase derivada será sustituido. La nueva definición del método se aplicará a los objetos de la clase derivada.
- Los métodos públicos de la clase base no pueden sustituirse en la sección privada de la clase derivada, porque eso equivaldría a eliminar métodos y violaría los principios de la relación *ES-UN*.
- Pueden añadirse métodos adicionales en la clase derivada.

La clase derivada hereda todos los métodos de la clase base. Puede aceptarlos o redefinirlos. También puede definir nuevos métodos.

#### 4.1.2 Compatibilidad de tipos

La reutilización directa del código descrita en el párrafo anterior representa una ventaja significativa. Sin embargo, la ventaja más significativa es la *reutilización indirecta del código*. Esta ventaja procede del hecho de que un objeto *Student* *ES-UN* objeto *Person* y un objeto *Employee* *ES-UN* objeto *Person*.

Cada clase derivada es una clase completamente nueva que, de todos modos, presenta una cierta compatibilidad con la clase de la que se deriva.

Puesto que un objeto *Student* *ES-UN* objeto *Person*, puede accederse a un objeto *Student* utilizando una referencia a *Person*. El siguiente código sería, por tanto, legal:

```
Student s = new Student( "Joe", 26, "1 Main St",
                        "202-555-1212", 4.0 );
Person p = s;
System.out.println( "Age is " + p.getAge( ) );
```

Es legal porque el tipo estático (es decir, el tipo en tiempo de compilación) de *p* es *Person*. Por tanto, *p* puede hacer referencia a cualquier objeto del que podamos decir que *ES-UN* objeto *Person*, y cualquier método que invoquemos a través de la referencia *p* tendrá siempre sentido, ya que una vez que se ha definido un método para *Person*, este no puede ser eliminado por una clase derivada.

Puede que el lector se esté preguntando por qué esto representa una ventaja. La razón es que esto se aplica no solo a las asignaciones, sino también al paso de parámetros. Un método cuyo parámetro formal sea de tipo *Person* puede recibir cualquier cosa de la que podamos decir que *ES-UN* objeto *Person*, incluyendo *Student* y *Employee*.

Considere por tanto el siguiente código escrito en *cualquier clase*:

```
public static boolean isOlder( Person p1, Person p2 )
{
    return p1.getAge( ) > p2.getAge( );
}
```

Considere las siguientes declaraciones, en las que omitimos los argumentos de los constructores para ahorrar espacio:

```
Person p = new Person( ... );
Student s = new Student( ... );
Employee e = new Employee( ... );
```

Podemos utilizar esa única rutina `isOlder` para todas las llamadas siguientes: `isOlder(p,p)`, `isOlder(s,s)`, `isOlder(e,e)`, `isOlder(p,e)`, `isOlder(p,s)`, `isOlder(s,p)`, `isOlder(s,e)`, `isOlder(e,p)`, `isOlder(e,s)`.

Con ello hemos conseguido que una rutina que no pertenece a esa clase funcione en nueve casos distintos. De hecho, no hay ningún límite al nivel de reutilización que puede conseguirse de esta forma. En cuanto utilicemos la herencia para añadir una cuarta clase a la jerarquía, tendremos 4 por 4, es decir 16 métodos diferentes, sin cambiar `isOlder` en absoluto. El nivel de reutilización sería todavía más significativo si un método admitiera como parámetros tres referencias a `Person`. E imagine el increíble nivel de reutilización de código que se puede alcanzar si un método utiliza como parámetro una matriz de referencias a `Person`.

Es por eso que, para muchas personas, la compatibilidad de tipos entre las clases derivadas y sus clases base es el aspecto más importante de la herencia, porque conduce a una masiva *reutilización indirecta del código*. Y, como ilustra el caso de `isOlder`, también hace que sea muy fácil añadir nuevos tipos que funcionen automáticamente con los métodos existentes.

#### 4.1.3 Despacho dinámico y polimorfismo

Obviamente, existe el problema de la sustitución de métodos: si no concuerdan el tipo de referencia y la clase del objeto al que se está haciendo referencia (en el ejemplo anterior, serían `Person` y `Student`, respectivamente), y ambos tienen diferentes implementaciones, ¿qué implementación hay que utilizar?

Como ejemplo, considere el siguiente fragmento de código:

```
Student s = new Student( "Joe", 26, "1 Main St",
                        "202-555-1212", 4.0 );
Employee e = new Employee( "Boss", 42, "4 Main St.",
                           "203-555-1212", 100000.0 );
Person p = null;
if( getTodaysDay( ).equals( "Tuesday" ) )
    p = s;
else
    p = e;
System.out.println( "Person is " + p.toString( ) );
```

Aquí, el tipo estático de `p` es `Person`. Al ejecutar el programa, el tipo dinámico (es decir, el tipo del objeto al que se está haciendo referencia en realidad) será `Student` o `Employee`. Es imposible deducir el tipo dinámico hasta que se ejecuta el programa. Naturalmente, sin embargo, lo que queríamos es que se empleara el tipo dinámico, y eso es lo que sucede en Java. Cuando se ejecute este fragmento de código, el método `toString` utilizado será el que resulte apropiado para el tipo dinámico de la referencia al objeto controlador.

Existe un importante principio de orientación a objetos que se conoce con el nombre de *polimorfismo*. Una variable de referencia polimórfica puede hacer referencia a objetos de varios tipos diferentes. Cuando se aplican operaciones a la referencia, se selecciona automáticamente la operación que

Una variable polimórfica puede hacer referencia a objetos de varios tipos distintos. Cuando se aplican operaciones a la variable polimórfica, se selecciona automáticamente la operación apropiada para el objeto referenciado

es apropiada para el objeto realmente referenciado. Todos los tipos de referencia son polimórficos en Java. Este mecanismo se conoce también con el nombre de *despacho dinámico* o *acoplamiento tardío* (o en ocasiones *acoplamiento dinámico*).

Una clase derivada es *compatible en cuanto a tipo* con su clase base, lo que quiere decir que una variable de referencia del tipo de la clase base puede hacer referencia a un objeto de la clase derivada, pero no a la inversa. Las clases hermanas (es decir, clases derivadas de una clase común) no son compatibles en cuanto a tipo.

#### 4.1.4 Jerarquías de herencia

Como hemos mencionado anteriormente, el uso de la herencia suele producir una jerarquía de clases. La Figura 4.6 ilustra una posible jerarquía para Person. Observe que Faculty deriva de Person indirectamente, en lugar de directamente –¡así que los profesores representados por la clase Faculty, también son personas! Este hecho es transparente para el usuario de las clases, porque las relaciones de tipo *ES-UN* son transitivas. En otras palabras, si  $X \text{ ES-UN } Y$  e  $Y \text{ ES-UN } Z$ , entonces  $X \text{ ES-UN } Z$ . La jerarquía Person ilustra los problemas típicos de diseño a la hora de consolidar los aspectos comunes en una serie de clases base y luego especializar estas en las clases derivadas. En esta jerarquía, decimos 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, y además el operador `instanceof` funciona con las subclases. Así, si `obj` es de tipo `Undergrad` (y distinto de `null`), entonces `obj instanceof Person` es `true`.

Si  $X \text{ ES-UN } Y$ , entonces  
 $X$  es una subclase de  $Y$   
 $Y$  es una superclase de  $X$ . Estas relaciones son  
transitivas.

#### 4.1.5 Reglas de visibilidad

Sabemos que cualquier miembro que se declare con visibilidad privada solo es accesible para los métodos de esa clase. Así, como ya hemos visto, los miembros privados de la clase base no son accesibles por parte de la clase derivada.

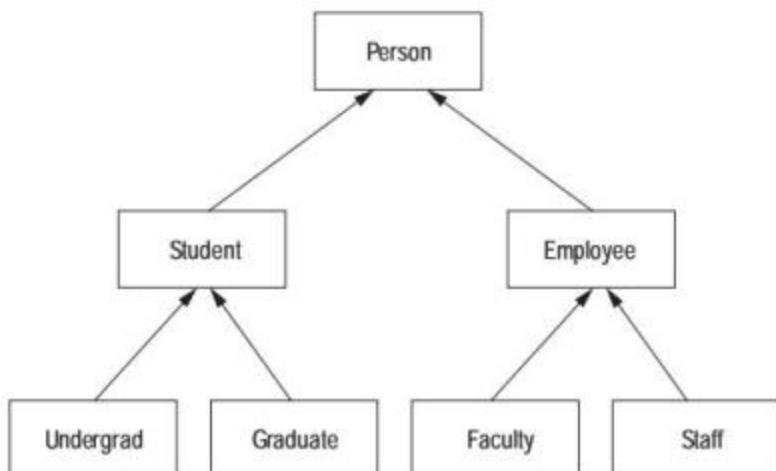


Figura 4.6 La jerarquía de Person.

Ocasionalmente, queríamos que la clase derivada tuviera acceso a los miembros de la clase base. Para esto, existen dos opciones fundamentales. La primera consiste en utilizar acceso con visibilidad pública o de paquete (si las clases base y derivada se encuentran en el mismo paquete), según resulte apropiado. Sin embargo, esto permitiría que también accedieran otras clases además de las clases derivadas.

*Un miembro protegido de la clase será visible para la clase derivada y también para las clases contenidas en el mismo paquete.*

Si queremos restringir el acceso de forma que solo puedan acceder las clases derivadas, podemos hacer que los miembros sean protegidos. Un *miembro protegido de una clase* es visible por parte de los métodos de una clase derivada y también por parte de los métodos de las clases contenidas en el mismo paquete, pero no es visible para nadie más.<sup>1</sup> Declarar los miembros de datos como `protected` o `public` viola el espíritu de encapsulación y ocultamiento de la información, y solo se suele hacer, generalmente, con el fin

de simplificar la programación. Normalmente, una mejora alternativa consiste en escribir métodos accesores y mutadores. Sin embargo, si una declaración de visibilidad protegida nos permite evitar el tener que emplear un código enrevesado, entonces no deja de ser razonable hacerlo. En este texto, se emplean miembros de datos protegidos precisamente por esa razón. En el texto utilizamos también métodos protegidos. Esto permite a una clase derivada heredar un método interno, sin hacer que este sea accesible desde fuera de la jerarquía de clases. Observe que en aquellos *códigos de práctica* en los que todas las clases se encuentran en el paquete sin nombre predeterminado, los miembros protegidos son visibles.

#### 4.1.6 El constructor y super

Cada clase derivada debe definir sus propios constructores. Si no se escribe ningún constructor, se genera entonces un único constructor predeterminado con cero parámetros. Este constructor invocará el constructor de cero parámetros de la clase base para la parte heredada y luego aplicará la inicialización predeterminada para todos los campos de datos adicionales (lo que quiere decir asignar un valor 0 a los tipos primitivos y `null` a los tipos de referencia).

Es bastante común construir un objeto de la clase derivada construyendo primero la parte heredada. De hecho, eso es lo que se hace de manera predeterminada, e incluso aunque se proporcione un constructor explícito para la clase derivada. Esto es bastante natural, porque el punto

<sup>1</sup> La regla de la visibilidad protegida es bastante compleja. Un miembro protegido de la clase B será visible para todos los métodos de todas las clases que se encuentren en el mismo paquete que B. También será visible para los métodos de cualquier clase D que se encuentre en un paquete distinto de B, siempre y cuando D amplíe B, pero solo si se accede a través de una referencia que sea compatible en cuanto al tipo con D (incluyendo un `this` implícito o explícito). Específicamente, ese miembro NO SERÁ VISIBLE en la clase D si se accede a él a través de una referencia de tipo B. El siguiente ejemplo ilustra esta regla.

```

1 class Demo extends java.io.FilterInputStream
2 { // FilterInputStream tiene un campo de datos protegido llamado in
3     public void foo()
4     {
5         java.io.FilterInputStream b = this; // legal
6         System.out.println( in );           // legal
7         System.out.println( this.in );     // legal
8         System.out.println( b.in );       // illegal
9     }
10 }
```

```
1 class Student extends Person
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5         { super( n, ag, ad, p ); gpa = g; }
6
7     // Omitidos toString y getAge
8
9     private double gpa;
10 }
```

Figura 4.7 Un constructor para la nueva clase Student; utiliza super.

de vista de la encapsulación nos dice que la parte heredada es una entidad en sí misma y el constructor de la clase base nos dice cómo inicializar esa entidad diferenciada.

Los constructores de la clase base pueden ser invocados explícitamente utilizando el método `super`. Así, el constructor predeterminado para una clase derivada es, en realidad,

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

El método `super` puede invocarse con parámetros que se correspondan con los de un constructor de la clase base. Como ejemplo, la Figura 4.7 ilustra la implementación del constructor `Student`.

El método `super` solo puede emplearse como primera línea de un constructor. Si no se incluye, se genera una llamada automática a `super` sin ningún parámetro.

Si no se escribe ningún constructor, entonces se genera un único constructor predeterminado de cero parámetros que invoca el constructor de cero parámetros de la clase base para la parte heredada, y luego aplica la inicialización predeterminada para los campos de datos adicionales.

`super` se usa para llamar al constructor de la clase base.

#### 4.1.7 Clases y métodos final

Como hemos descrito anteriormente, la clase derivada acepta o sustituye los métodos de la clase base. En muchos casos, está claro que un método concreto de la clase base debe ser invariante en toda la jerarquía, lo que quiere decir que una clase derivada no debería sustituirlo. En este caso, podemos declarar que el método es `final` y que no puede ser sustituido.

Un método `final` es invariante en toda la jerarquía de herencia y no puede ser sustituido.

El declarar los métodos invariantes como `final` no solo es una buena práctica de programación, sino que también permite obtener código más eficiente. Es una buena práctica de programación porque, además de declarar nuestras intenciones al lector del programa y de la documentación, evitamos la sustitución accidental de un método que no debería ser sustituido.

Para ver por qué la utilización de `final` permite obtener un código más eficiente, suponga que la clase base `Base` declara un método final `f` y suponga que `Derived` amplía `Base`. Considere la rutina:

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

El acoplamiento estático podría utilizarse cuando el método sea invariante en toda la jerarquía de herencia.

durante la compilación, en lugar de en tiempo de ejecución, el programa debería ejecutarse más rápidamente. El que esto sea perceptible dependerá de cuántas veces evitemos tomar la decisión en tiempo de ejecución mientras se ejecuta el programa.

Los métodos estáticos no tienen ningún objeto controlador y por tanto se resuelven en tiempo de compilación, mediante acoplamiento estático.

tienen ningún elemento controlador y por tanto se resuelven en tiempo de compilación, utilizando el mecanismo de acoplamiento estático.

Una clase final no puede ampliarse mediante herencia. Una clase hoja es una clase final.

Puesto que `f` es un método final, no importa si `obj` hace referencia a un objeto de tipo `Base` o `Derived`; la definición de `f` es invariante, así que sabemos positivamente qué es lo que hace `f`. Como resultado, puede tomarse una decisión en tiempo de compilación para resolver la llamada al método, en lugar de tomar la decisión en tiempo de ejecución. Esto se conoce con el nombre de *acoplamiento estático*.

Puesto que el acoplamiento se realiza durante la compilación, en lugar de en tiempo de ejecución, el programa debería ejecutarse más rápidamente. El que esto sea perceptible dependerá de cuántas veces evitemos tomar la decisión en tiempo de ejecución mientras se ejecuta el programa.

Un corolario de esta observación es que si `f` es un método trivial, como por ejemplo un accesor a un único campo y se declara como `final`, el compilador podría sustituir la llamada a `f` por su definición en línea. Por tanto, la llamada al método sería sustituida por una única línea que accediera a un campo de datos, ahorrando así tiempo. Si `f` no se declara `final`, entonces es imposible hacer esto, ya que `obj` podría estar haciendo referencia a un objeto de una clase derivada, para el que la definición de `f` podría ser diferente.<sup>2</sup> Los métodos estáticos no son métodos finales, pero no tienen ningún elemento controlador y por tanto se resuelven en tiempo de compilación, utilizando el mecanismo de acoplamiento estático.

Similar al concepto de método final es el concepto de *clase final*. Una clase final no puede ser ampliada mediante herencia. Como resultado, todos sus métodos son, automáticamente métodos finales. Por ejemplo, la clase `String` es una clase final. Observe que el hecho de que una clase tenga solo métodos finales no implica necesariamente que ella misma sea una clase final. Las clases finales también se conocen con el nombre de *clases hoja*, porque en la jerarquía de herencia, que se asemeja a un árbol, las clases finales se encuentran en los extremos finales, como si fueran las hojas del árbol.

En la clase `Person`, los métodos triviales accesores y mutadores (los que comienzan con `get` y `set`) son buenos candidatos para ser definidos como métodos finales, y así hemos elegido declararlos en el código que los lectores tienen a su disposición a través de la web.

<sup>2</sup> En los dos párrafos anteriores, decimos que el acoplamiento estático y las optimizaciones en línea "podrían" realizarse porque, aunque el tomar esas decisiones en tiempo de compilación parece tener sentido, la Sección 8.4.3.3 de la especificación del lenguaje deja claro que las optimizaciones en línea para los métodos finales de carácter trivial pueden realizarse, pero que esta optimización debe ser hecha por la máquina virtual en tiempo de ejecución, en lugar de por el compilador en tiempo de compilación. Esto garantiza que las clases dependientes no queden desincronizadas como resultado de la optimización.

### 4.1.8 Sustitución de un método

Los métodos de la clase base se sustituyen en la clase derivada simplemente proporcionando en la clase derivada un método que tenga la misma *signatura*.<sup>3</sup> El método de la clase derivada debe tener el mismo tipo de retorno y no puede añadir excepciones a la lista `throws`.<sup>4</sup> La clase derivada no puede reducir la visibilidad, ya que eso violaría el espíritu de una relación *ES-UN*. Por tanto, no se puede sustituir un método público por un método con visibilidad de paquete.

En ocasiones, el método de la clase derivada desea invocar el método de la clase base. Normalmente, esto se conoce con el nombre de *sustitución parcial*. Es decir, queremos hacer lo que hace la clase base y un poco más, en lugar de hacer algo completamente distinto. Las llamadas al método de la clase base pueden realizarse utilizando `super`. He aquí un ejemplo:

```
public class Workaholic extends Worker
{
    public void doWork( )
    {
        super.doWork( ); // Trabaja como un Worker
        drinkCoffee( ); // Se toma un descanso
        super.doWork( ); // Trabaja como un Worker un poco más
    }
}
```

Un ejemplo más típico es la sustitución de los métodos estándar, como por ejemplo `toString`. La Figura 4.8 ilustra este uso en las clases `Student` y `Employee`.

El método de la clase derivada debe tener el mismo tipo de retorno y signatura, y no puede añadir excepciones a la lista `throws`.

La sustitución parcial implica invocar un método de la clase base utilizando `super`.

### 4.1.9 Un nuevo análisis de la compatibilidad de tipos

La Figura 4.9 ilustra el uso típico del polimorfismo con matrices. En la línea 17, creamos una matriz de cuatro referencias a `Person`, cada una de las cuales se inicializará con `null`. Los valores de estas referencias pueden establecerse en las líneas 19 a 24, y sabemos que todas las asignaciones son legales, debido a la capacidad que tiene una referencia a un tipo base para hacer referencia a objetos de un tipo derivado.

La rutina `printAll` simplemente recorre la matriz e invoca el método `toString`, utilizando el mecanismo de despacho dinámico. La comprobación de la línea 7 es importante porque, como hemos visto, algunas de las referencias de la matriz podrían ser `null`.

<sup>3</sup> Si se utiliza una *signatura* distinta, lo que hacemos es simplemente sobrecargar el método, y ahora existirán dos métodos con distintas *signaturas* entre los que podrá elegir el compilador.

<sup>4</sup> Java 5 relaja este requisito y permite que el tipo de retorno del método de la clase derivada sea ligeramente distinto siempre y cuando siga siendo “compatible”. La nueva regla se explica en la Sección 4.1.11.

```
1 class Student extends Person
2 {
3     public Student( String n, int ag, String ad, String p,
4                     double g )
5     { super( n, ag, ad, p ); gpa = g; }
6
7     public String toString( )
8     { return super.toString( ) + getGPA( ); }
9
10    public double getGPA( )
11    { return gpa; }
12
13    private double gpa;
14 }
15
16 class Employee extends Person
17 {
18     public Employee( String n, int ag, String ad,
19                      String p, double s )
20     { super( n, ag, ad, p ); salary = s; }
21
22     public String toString( )
23     { return super.toString( ) + " $" + getSalary( ); }
24
25     public double getSalary( )
26     { return salary; }
27
28     public void raise( double percentRaise )
29     { salary *= ( 1 + percentRaise ); }
30
31     private double salary;
32 }
```

---

Figura 4.8 Las clases `Student` y `Employee` completas utilizando ambas formas de `super`.

En el ejemplo, suponga que antes de completar la impresión queremos conceder un aumento a `p[3]`, que sabemos que es un empleado. Puesto que `p[3]` es de tipo `Employee`, podría parecer que  
`p[3].raise( 0.04 );`

sería legal. Pero no lo es. El problema es que el tipo estático de `p[3]` es `Person` y `raise` no está definida para `Person`. En tiempo de compilación, solo pueden aparecer a la derecha del operador punto los miembros (visibles) del tipo estático de la referencia.

Podemos cambiar el tipo estático utilizando una cláusula de modificación del tipo:

`((Employee) p[3]).raise( 0.04 );`

```

1 class PersonDemo
2 {
3     public static void printAll( Person [ ] arr )
4     {
5         for( int i = 0; i < arr.length; i++ )
6         {
7             if( arr[ i ] != null )
8             {
9                 System.out.print( "[" + i + " ] " );
10                System.out.println( arr[ i ].toString( ) );
11            }
12        }
13    }
14
15    public static void main( String [ ] args )
16    {
17        Person [ ] p = new Person[ 4 ];
18
19        p[0] = new Person( "joe", 25, "New York",
20                           "212-555-1212" );
21        p[1] = new Student( "jill", 27, "Chicago",
22                           "312-555-1212", 4.0 );
23        p[3] = new Employee( "bob", 29, "Boston",
24                           "617-555-1212", 100000.0 );
25
26        printAll( p );
27    }
28 }

```

Figura 4.9 Una ilustración del polimorfismo con matrices.

La línea anterior hace que el tipo estático de la referencia situada a la izquierda del operador punto sea `Employee`. Si esto es imposible (por ejemplo, porque `p[3]` se encuentre en una jerarquía de herencia completamente distinta), el compilador se quejará. Si es posible que el cambio de tipo tenga sentido, el programa se compilará, por lo que el código anterior proporcionará correctamente a `p[3]` un tanto por ciento de aumento. Esta estructura sintáctica, en la que modificamos el tipo específico de una expresión, cambiando una clase base por otra situada por debajo de ella dentro de la jerarquía de herencia, se conoce con el nombre de *downcast o especialización*.

¿Qué pasaría si `p[3]` no fuera de tipo `Employee`? Por ejemplo, ¿qué pasaría si empleáramos la línea siguiente?

Una especialización o *downcast* es un cambio de tipo por el que se desciende dentro de la jerarquía de herencia. Los cambios de tipo son siempre verificados en tiempo de ejecución por la Máquina Virtual.

```
((Employee) p[1]).raise( 0.04 ); // p[1] es un objeto Student
```

En ese caso, el programa se compilaría, pero la Máquina Virtual generaría una excepción `ClassCastException`, que es una excepción de tiempo de ejecución que indica un error de programación. Los cambios de tipo siempre se comprueban exhaustivamente en tiempo de ejecución, para cerciorarse que el programador (o un pirata informático malicioso) no esté tratando de pervertir el sólido sistema de tipos de Java. La forma segura de hacer este tipo de llamadas consiste en utilizar primero `instanceof`:

```
if( p[3] instanceof Employee )
    ((Employee) p[3]).raise( 0.04 );
```

#### 4.1.10 Compatibilidad de tipos matriciales

Una de las dificultades en el diseño de lenguajes es cómo gestionar la herencia en el caso de tipos agregados. En nuestro caso, sabemos que `Employee ES-UN Person`. ¿Pero quiere esto decir que `Employee[] ES-UN Person[]`? En otras palabras, si escribimos una rutina para aceptar `Person[]` como parámetro, ¿podemos pasarle un objeto de tipo `Employee[]` como argumento?

Las matrices de subclases son compatibles en cuanto a tipo con las matrices de superclase. Esto se conoce con el nombre de *matrices covariantes*.

A primera vista, parece que se trata de una cuestión muy sencilla y que `Employee[]` debería ser compatible en cuanto a tipo con `Person[]`. Sin embargo, el problema es más sutil de lo que parece. Suponga que además de `Employee`, `Student ES-UN Person`. Suponga que `Employee[]` es compatible en cuanto tipo con `Person[]`. Entonces, considere esta secuencia de asignaciones:

```
Person[] arr = new Employee[ 5 ]; // se compila: las matrices son compatibles
arr[ 0 ] = new Student( ... ); // se compila: Student ES-UN Person
```

Si se inserta un tipo incompatible en la matriz, la Máquina Virtual generará una excepción `ArrayStoreException`.

Ambas asignaciones se compilan, a pesar de lo cual `arr[0]` estará haciendo realmente referencia a un objeto de tipo `Employee` y `Student NO-ES-UN Employee`. Por tanto, tenemos una confusión de tipos. El sistema de tiempo de ejecución no puede generar una excepción `ClassCastException`, ya que no hay ninguna conversión de tipo explícita.

La forma más fácil de evitar este problema consiste en especificar que las matrices no son compatibles en cuanto a tipo. Sin embargo, en Java, las matrices *son* compatibles en cuanto a tipo. Esto se conoce con el nombre de *tipo matricial covariante*. Cada matriz sabe el tipo de objeto que tiene permitido almacenar. Si se inserta un tipo incompatible dentro de la matriz, la Máquina Virtual generará una excepción `ArrayStoreException`.

#### 4.1.11 Tipos de retorno covariantes

Antes de Java 5, al sustituir un método, el método de la subclase tenía obligatoriamente que tener el mismo tipo de retorno que el método de la superclase. En Java 5 se ha relajado esta regla, y el tipo de retorno del método de la subclase solo necesita ser compatible en cuanto a tipo (es decir, puede

ser una subclase de) con el tipo de retorno del método de la superclase. Esto se conoce con el nombre de *tipo de retorno covariante*. Por ejemplo, suponga que la clase Person tiene un método makeCopy

```
public Person makeCopy();
```

que devuelve una copia del objeto Person. Antes de Java 5, si la clase Employee sustituía este método, el tipo de retorno tenía que ser Person. En Java 5, el método tiene que sustituirse de la forma siguiente:

```
public Employee makeCopy();
```

En Java 5, el tipo de retorno de un método de la subclase solo necesita ser compatible en cuanto a tipo (es decir, puede ser una subclase de) con el tipo de retorno del método de la superclase. Esto se conoce con el nombre de *tipo de retorno covariante*.

## 4.2 Diseño de jerarquías

Suponga que tenemos una clase Circle y que para cualquier Circle c no nulo, c.area() devuelve el área del objeto Circle c. Suponga también que tenemos una clase Rectangle y que para cualquier Rectangle r no nulo, r.area() devuelve el área del objeto Rectangle r. Posiblemente tendríamos otras clases como Ellipse, Triangle y Square, todas con métodos que permitan calcular el área de la forma geométrica correspondiente. Suponga que tenemos una matriz que contiene referencias a estos objetos y que queremos calcular el área total de todos los objetos. Puesto que todas las clases disponen de un método area, el polimorfismo constituye una opción atractiva, permitiéndonos escribir un código como el siguiente:

```
public static double totalArea( WhatType [ ] arr )
{
    double total = 0.0;
    for( int i = 0; i < arr.length; i++ )
        if( arr[ i ] != null )
            total += arr[ i ].area();
    return total;
}
```

Para que este código funcione, tenemos que decidir la declaración de tipo para `WhatType`. Ni `Circle`, ni `Rectangle`, etc., funcionarían, ya que no existe una relación de tipo *ES-UN*. Por tanto, necesitamos definir un tipo, como por ejemplo `Shape`, que represente una forma geométrica cualquiera, de modo `Circle ES-UN Shape`, `Rectangle ES-UN Shape`, etc. En la Figura 4.10 se ilustra una posible jerarquía. Además, para que `arr[i].area()` tenga sentido, `area` debe ser un método disponible para `Shape`.

Esto sugiere una clase para `Shape`, como se muestra en la Figura 4.11. Una vez que disponemos de la clase `Shape`, podemos proporcionar otras, como se muestra en la Figura 4.12. Estas clases también incluyen un método `perimeter` para calcular el perímetro.

El código de la Figura 4.12, con clases que amplian la clase `Shape` simple de la Figura 4.11, que devuelve -1 para `area`, puede utilizarse ahora polimórficamente, como se muestra en la Figura 4.13.

Una gran ventaja de este diseño es que podemos añadir una nueva clase a la jerarquía sin perturbar las implementaciones. Por ejemplo, suponga que queremos añadir triángulos a nuestro diseño. Lo único que necesitamos hacer es que `Triangle` amplíe `Shape`, sustituya el método `area`

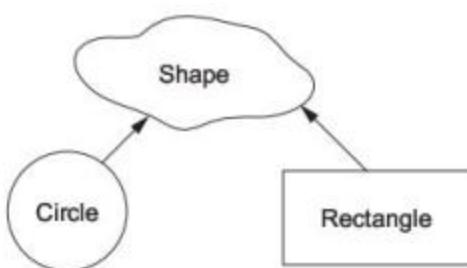


Figura 4.10 La jerarquía de formas geométricas usada en un ejemplo de herencia.

```

1 public class Shape
2 {
3     public double area( )
4     {
5         return -1;
6     }
7 }
  
```

Figura 4.11 Una posible clase Shape.

apropiadamente, y con ello se podrán incluir objetos `Triangle` en cualquier objeto `Shape[]`. Observe que esto implica lo siguiente:

La existencia de muchos operadores `instanceof` es un síntoma de un mal diseño orientado a objetos.

- NO HAY NINGÚN CAMBIO en la clase `Shape`,
- NO HAY NINGÚN CAMBIO en las clases `Circle`, `Rectangle` u otras clases existentes,
- NO HAY NINGÚN CAMBIO en el método `totalArea`,

lo que hace difícil que el código existente deje de funcionar durante el proceso de adición de nuevo código. Observe también que no hay ninguna prueba de tipo `instanceof`, lo que es típico de un buen código polimórfico.

#### 4.2.1 Clases y métodos abstractos

Aunque el código del ejemplo anterior funciona, podemos realizar mejoras en la clase `Shape` que hemos escrito en la Figura 4.11. Observe que la propia clase `Shape`, y el método `area` en particular, son simples *parámetros de sustitución*: no se pretende que el método `area` de `Shape` sea nunca invocado de forma directa. Está allí simplemente para que el compilador y el sistema de tiempo de ejecución puedan colaborar para utilizar el despacho dinámico e invocar un método `area` apropiado. De hecho, examinando `main`, vemos que tampoco se pretende que se creen nunca objetos `Shape`. Esa clase existe simplemente como superclase común para las otras.<sup>5</sup>

<sup>5</sup> Declarar un constructor `Shape` privado NO RESUELVE el segundo problema: el constructor es necesario para las subclases.

```
1 public class Circle extends Shape
2 {
3     public Circle( double rad )
4         { radius = rad; }
5
6     public double area( )
7         { return Math.PI * radius * radius; }
8
9     public double perimeter( )
10    { return 2 * Math.PI * radius; }
11
12    public String toString( )
13    { return "Circle: " + radius; }
14
15    private double radius;
16 }
17
18 public class Rectangle extends Shape
19 {
20     public Rectangle( double len, double wid )
21         { length = len; width = wid; }
22
23     public double area( )
24         { return length * width; }
25
26     public double perimeter( )
27         { return 2 * ( length + width ); }
28
29     public String toString( )
30         { return "Rectangle: " + length + " " + width; }
31
32     public double getLength( )
33         { return length; }
34
35     public double getWidth( )
36         { return width; }
37
38     private double length;
39     private double width;
40 }
```

Figura 4.12 Clases Circle y Rectangle.

```

1 class ShapeDemo
2 {
3     public static double totalArea( Shape [ ] arr )
4     {
5         double total = 0;
6
7         for( Shape s : arr )
8             if( s != null )
9                 total += s.area( );
10
11    return total;
12 }
13
14 public static void printAll( Shape [ ] arr )
15 {
16     for( Shape s : arr )
17         System.out.println( s );
18 }
19
20 public static void main( String [ ] args )
21 {
22     Shape [ ] a = { new Circle( 2.0 ), new Rectangle( 1.0, 3.0 ), null };
23
24     System.out.println( "Total area = " + totalArea( a ) );
25     printAll( a );
26 }
27 }
```

**Figura 4.13** Un programa de ejemplo que utiliza la jerarquía de formas geométricas.

El programador ha intentado dejar claro que invocar el método de cálculo del área de `Shape` es un error devolviendo el valor `-1`, que es obviamente un área imposible. Pero este es un valor que podría ser ignorado. Además, se trata de un valor que será devuelto si no se sustituye el método del cálculo del área al ampliar `Shape` con una clase heredada. Esta no sustitución del método podría producirse debido a un error tipográfico: imagine que se escribe una función `Area` en lugar de `area`, haciendo difícil localizar el error en tiempo de ejecución.

Una solución mejor para `area` consiste en generar una excepción de tiempo de ejecución (una adecuada sería `UnsupportedOperationException`) en la clase `Shape`. Esto es preferible a devolver `-1` porque la excepción no será ignorada.

Sin embargo, incluso utilizando esa solución, el problema solo se resuelve en tiempo de ejecución. Sería mejor disponer de una sintaxis que indicara explícitamente que `area` es un método pensado para ser sustituido

Los métodos y clases abstractos representan elementos que habrá que sustituir.

y que no necesita ninguna implementación en absoluto, y que además `Shape` es una clase pensada para ser ampliada y no pueden construirse objetos de la misma, aun cuando se declaren en ella constructores y disponga de un constructor predeterminado si no se declara ninguno. Si esta sintaxis estuviera disponible, entonces el compilador podría, en tiempo de compilación, declarar como ilegal cualquier intento de construir una instancia de `Shape`. También podría declarar como ilegal cualquier clase, como `Triangle`, en la que se intentara construir una instancia sin haber sustituido el método `area`. Esto describe exactamente lo que son los métodos abstractos y las clases abstractas.

Un *método abstracto* es un método que declara funcionalidad que todos los objetos de las clases derivadas deben terminar implementando. En otras palabras, dice lo que esos objetos pueden hacer. Sin embargo, no proporciona ninguna implementación predeterminada. En lugar de ello, cada objeto debe proporcionar su propia implementación.

Una clase que tenga al menos un método abstracto se denomina *clase abstracta*. Java exige que todas las clases abstractas se declaren explícitamente como tales. Cuando una clase derivada se olvida de sustituir un método abstracto con una implementación, el método continúa siendo abstracto en la clase derivada. Como resultado, si una clase que no se pretendía que fuera abstracta se olvida de sustituir un método abstracto, el compilador detectará la incoherencia e informará del error.

En la Figura 4.14 se muestra un ejemplo de cómo podemos hacer que `Shape` sea abstracta. No hace falta efectuar ningún cambio en el código de las Figuras 4.12 y 4.13. Observe que una clase abstracta puede tener métodos que no sean abstractos, como es el caso de `semiperimeter`.

Una clase abstracta también puede declarar tanto campos estáticos como de instancia. Al igual que en las clases no abstractas, estos campos serán típicamente privados, y los campos de instancia se inicializarían mediante constructores. Aunque no pueden crearse instancias de clases abstractas, esos constructores serán invocados cuando las clases derivadas usen `super`. En un ejemplo más amplio, la clase `Shape` podría incluir las coordenadas de los vértices de los objetos, que serían configuradas mediante constructores, y podría proporcionar la implementación de métodos como `positionOf`, que fueran independientes del tipo concreto del objeto; `positionOf` sería un método final.

Como hemos mencionado anteriormente, la existencia de al menos un método abstracto hace que la clase base sea abstracta e impide crear instancias de la misma. Por tanto, no se puede crear

```

1 public abstract class Shape
2 {
3     public abstract double area();
4     public abstract double perimeter();
5
6     public double semiperimeter()
7         { return perimeter() / 2; }
8 }
```

Un método abstracto no tiene ninguna definición significativa y se define siempre, por tanto, en la clase derivada.

Una clase con al menos un método abstracto debe ser definida como clase abstracta.

Figura 4.14 Una clase `Shape` abstracta. El código de las Figuras 4.12 y 4.13 no sufre modificaciones.

un objeto de tipo `Shape`; solo pueden crearse los objetos derivados. Sin embargo, como es habitual, una variable de tipo `Shape` puede hacer referencia a cualquier objeto derivado concreto, como un `Circle` o un `Rectangle`. Así

```
Shape a, b;
a = new Circle( 3.0 ); // Legal
b = new Shape(); // Illegal
```

Antes de continuar, resumamos los cuatro tipos de métodos de una clase:

1. *Métodos finales.* La Máquina Virtual puede decidir en tiempo de ejecución realizar una optimización en línea, evitando así el mecanismo de despacho dinámico. Utilizamos un método final solo cuando el método es invariante en toda la jerarquía de herencia (es decir, cuando el método no se redefine nunca).
2. *Métodos abstractos.* La sustitución del método se resuelve en tiempo de ejecución. La clase base no proporciona ninguna implementación y es abstracta. La ausencia de una implementación predeterminada exige que las clases derivadas proporcionen una implementación o que esas clases derivadas se definan ellas mismas como abstractas.
3. *Métodos estáticos.* La sustitución se resuelve en tiempo de compilación, porque no hay ningún objeto controlador.
4. *Otros métodos.* La sustitución de métodos se resuelve en tiempo de ejecución. La clase base proporciona una implementación predeterminada, que puede ser sustituida por las clases derivadas o aceptada sin modificaciones por estas.

#### 4.2.2 Diseño pensando en el futuro

Considere la siguiente implementación para la clase `Square`:

```
public class Square extends Rectangle
{
    public Square( double side )
    { super( side, side ); }
}
```

Puesto que obviamente un cuadrado es un rectángulo cuya longitud y anchura son iguales, parece razonable hacer que `Square` amplíe `Rectangle`, para así evitar tener que reescribir métodos como `area` y `perimeter`. Aunque es cierto que, debido a la no sustitución de `toString`, los objetos de tipo `Square` siempre se mostrarán como objetos de tipo `Rectangle` de longitud y anchura idénticas, podemos corregir esa situación proporcionando un método `toString` para `Square`. De ese modo, la clase `Square` puede hacerse realmente escueta y podemos reutilizar el código de `Rectangle`. ¿Pero es un diseño razonable? Para responder a esta pregunta, debemos considerar de nuevo la regla fundamental de la herencia.

La cláusula `extends` es apropiada solo si es cierto que `Square` ES-UN `Rectangle`. Desde una perspectiva de programación, esto no quiere decir simplemente que un cuadrado deba ser geométricamente un tipo de rectángulo; más bien, lo que significa es que cualquier operación que

podamos realizar con un objeto de tipo `Rectangle` pueda ser realizada también con un objeto de tipo `Square`. Y lo más importante es que esta no es una decisión estática, lo que quiere decir que no debemos simplemente mirar al conjunto actual de operaciones soportadas por `Rectangle`. En lugar de ello, debemos preguntarnos si es razonable asumir que en el futuro puedan añadirse operaciones a la clase `Rectangle` que no tendrían sentido para un objeto de tipo `Square`. En ese caso, entonces el argumento de que un `Square ES-UN Rectangle` se debilita considerablemente. Por ejemplo, suponga que la clase `Rectangle` tuviera un método `stretch` para modificar la forma del rectángulo y cuya especificación indicara que `stretch` aumenta la dimensión más larga del objeto `Rectangle`, dejando intacta la dimensión más pequeña. Claramente, la operación no puede estar disponible para un objeto `Square`, porque si lo hiciéramos el objeto dejaría de ser un cuadrado.

Si sabemos que la clase `Rectangle` tiene un método `stretch`, entonces probablemente no sea una buena decisión de diseño hacer que `Square` amplíe `Rectangle`. Si `Square` ya amplía `Rectangle` y posteriormente deseamos añadir un método `stretch` a `Rectangle`, hay dos formas básicas de hacerlo.

La opción 1 sería que `Square` sustituya `stretch` con una implementación que genere una excepción:

```
public void stretch( double factor )
{ throw new UnsupportedOperationException( ); }
```

Con este tipo de diseño, al menos los cuadrados nunca dejarán de ser cuadrados.

La opción 2 sería rediseñar toda la jerarquía para que `Square` deje de ser una ampliación de `Rectangle`. Esta forma de actuar se conoce con el nombre de *refactorización*. Dependiendo de lo complicada que sea la jerarquía completa, podría tratarse de una tarea increíblemente compleja. Sin embargo, algunas herramientas de desarrollo permiten automatizar buena parte del proceso. El mejor plan, especialmente para una jerarquía de gran tamaño, consiste en pensar en ese tipo de problemas durante el diseño y preguntarse cuál será el aspecto más razonable de la jerarquía en el futuro. Aunque, por supuesto, esto es fácil de decir, pero bastante difícil de llevar a cabo.

Una filosofía similar se aplica a la hora de definir qué excepciones deben enumerarse en la lista de excepciones generadas por un método. Debido a la relación *ES-UN*, cuando se sustituye un método no pueden añadirse nuevas excepciones comprobadas a la lista de excepciones generadas. La implementación sustituta puede reducir la lista original de excepciones comprobadas, pero nunca ampliarla. Por ello, a la hora de determinar la lista de excepciones generadas por un método, el diseñador no solo debe pensar en las excepciones que puedan ser generadas en la implementación actual del método, sino también en las excepciones que podrían llegar a ser generadas por ese método en el futuro (en caso de que cambie la implementación) y en las excepciones que podrían ser generadas por las implementaciones sustitutas proporcionadas en las subclases futuras.

## 4.3 Herencia múltiple

Todos los ejemplos de herencia que hemos visto hasta ahora hacían derivar una clase de otra única clase base. En la *herencia múltiple*, una clase puede derivar de más de una clase base. Por ejemplo, podemos tener una clase `Student` y una clase `Employee`. Podríamos pensar en otra nueva clase `StudentEmployee` que derivara de ambas clases.

La *herencia múltiple* se usa para derivar una clase de varias clases base. Java no permite la herencia múltiple.

Aunque la herencia múltiple parece atractiva, y algunos lenguajes (incluyendo C++) la soportan, su gestión está llena de sutilezas que hacen que el diseño resulte difícil. Por ejemplo, las dos clases base pueden contener dos métodos que tengan la misma firma pero diferentes implementaciones. Alternativamente, podrían tener dos campos con un nombre idéntico; ¿cuál de los dos habría que utilizar?

Por ejemplo, suponga que en el caso de la clase `StudentEmployee` anterior, `Person` es una clase con el campo de datos `name` y el método `toString`. Suponga también que `Student` amplía `Person` y sustituye `toString` para añadir el año de graduación del estudiante. Además, suponga que `Employee` amplía `Person` pero no sustituye `toString`; en lugar de ello, declara que es `final`.

1. Puesto que `StudentEmployee` hereda los miembros de datos tanto de `Student` como de `Employee`, ¿obtenemos dos copias de `name`?
2. Si `StudentEmployee` no sustituye `toString`, ¿qué método `toString` habría que utilizar?

Cuando hay implicadas muchas clases, los problemas son aun mayores. Sin embargo, parece que los problemas típicos con la herencia múltiple pueden atribuirse a la existencia de implementaciones conflictivas o de campos de datos conflictivos. Como resultado, Java no permite la herencia múltiple de implementaciones.

Sin embargo, permitir la herencia múltiple con propósitos de compatibilidad de tipos puede resultar muy útil, siempre y cuando podamos garantizar que no haya conflictos de implementación.

Volviendo a nuestro ejemplo de `Shape`, suponga que nuestra jerarquía contiene muchas formas geométricas como `Circle`, `Square`, `Ellipse`, `Rectangle`, `Triangle`. Suponga que para algunas de estas formas geométricas, pero no para todas, tenemos un método `stretch`, tal como se describe en la Sección 4.2.2, que alarga la dimensión más grande, dejando las restantes sin modificar. Podemos pensar razonablemente que el método `stretch` está implementado para `Ellipse`, `Rectangle` y `Triangle`, pero no para `Circle` o `Square`. Imagine ahora que quisiéramos un método para aplicar `stretch` a todas las formas geométricas contenidas en una matriz:

```
public static void stretchAll( WhatType [ ] arr, factor )
{
    for( WhatType s : arr )
        s.stretch( factor );
}
```

La idea es que `stretchAll` funcionaría para las matrices de `Ellipse`, las matrices de `Rectangle`, las matrices de `Triangle`, o incluso para una matriz que contuviera objetos `Ellipse`, `Rectangle` y `Triangle`.

Para que este código funcione, tenemos que decidir cuál será la declaración de tipo para `WhatType`. Una posibilidad es que `WhatType` sea de tipo `Shape`, siempre y cuando `Shape` tenga un método abstracto `stretch`. De nuevo podríamos sustituir `stretch` en cada tipo de `Shape`, haciendo que `Circle` y `Square` generaran excepciones `UnsupportedOperationException`. Pero, como hemos explicado en la Sección 4.2.2, esta solución parece violar la noción de una relación *ES-UN*, y además no se puede generalizar bien a casos más complicados.

Otra idea sería tratar de definir una clase abstracta `Stretchable` de la forma siguiente:

```
abstract class Stretchable
{
    public abstract void stretch( double factor );
}
```

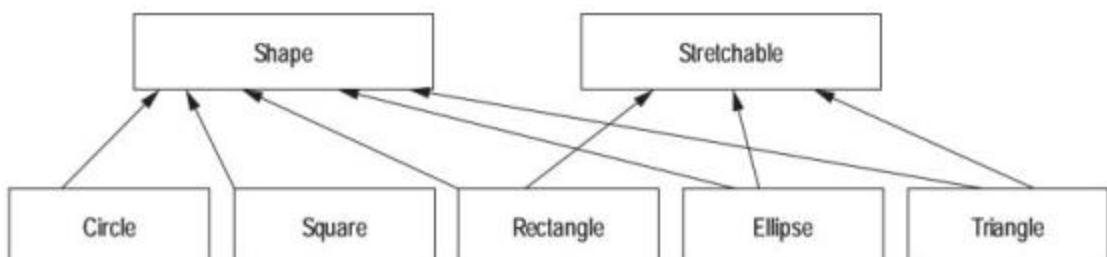
Podríamos utilizar `Stretchable` como el tipo que nos falta en el método `stretchAll`. Si lo hicierámos así, trataríamos de que `Rectangle`, `Ellipse` y `Triangle` ampliaran `Stretchable` y proporcionaran el método `stretch`:

```
// No funciona
public class Rectangle extends Shape, Stretchable
{
    public void stretch( double factor )
    {
        ...
    }
    public void area()
    {
        ...
    }
    ...
}
```

La jerarquía que tendríamos llegados a este punto se muestra en la Figura 4.15.

En principio, esto sería correcto, pero entonces tendríamos herencia múltiple, que ya hemos dicho anteriormente que es ilegal, debido al problema de la posible herencia de implementaciones conflictivas. Con lo que hemos dicho hasta ahora, solo la clase `Shape` dispone de una implementación; `Stretchable` es puramente abstracta, así que podría argumentarse que el compilador debería ser benévolos en este caso. Pero es posible que después de compilarlo todo, `Stretchable` fuera modificada para proporcionar una implementación, en cuyo caso tendríamos un problema. Lo que nos gustaría es disponer de algo más que de una promesa vacía de contenido; necesitamos algún tipo de sintaxis que obligue a `Stretchable` a carecer ahora y en un futuro de implementación. Si esto fuera posible, entonces el compilador podría permitir heredar de dos clases, mediante una jerarquía parecida a la mostrada en la Figura 4.15.

Esa sintaxis que buscamos es precisamente la de la interfaz.



**Figura 4.15** Herencia de múltiples clases. Esto no funciona a menos que `Shape` o `Stretchable` se diseñen específicamente como carentes de implementación.

## 4.4 La interfaz

La interfaz es una clase abstracta que no contiene ningún detalle de implementación.

La *interfaz* en Java es la clase más abstracta posible. Está compuesta únicamente de métodos públicos abstractos y campos finales estáticos públicos.

Decimos que una clase *implementa* la interfaz si proporciona definiciones para todos los métodos abstractos de la interfaz. Una clase que implemente la interfaz se comporta como si hubiera ampliado una clase abstracta especificada por la interfaz.

En principio, la diferencia principal entre una interfaz y una clase abstracta es que, aunque ambas proporcionan una especificación de lo que las subclases deben hacer, a la interfaz no se le permite proporcionar ningún detalle de implementación, ni en forma de campos de datos ni de métodos implementados. El efecto práctico de esto es que la utilización de interfaces múltiples no sufre los mismos problemas potenciales que la herencia múltiple, porque no pueden aparecer implementaciones conflictivas. Así, aunque una clase solo puede ampliar a otra clase, sin embargo sí que puede implementar más de una interfaz.

### 4.4.1 Especificación de una interfaz

Sintácticamente, no hay nada más sencillo que especificar una interfaz. La interfaz parece una declaración de clase, salvo porque utiliza la palabra clave `interface`. Está compuesta de un listado de los métodos que hay que implementar. Un ejemplo sería la interfaz `Stretchable` mostrada en la Figura 4.16.

La interfaz `Stretchable` especifica el método que toda subclase debe implementar: `stretch`. Observe que no tenemos que especificar que estos métodos sean de tipo `public` y `abstract`. Puesto que estos modificadores son obligatorios para los métodos de una interfaz, se pueden omitir y usualmente se omiten.

### 4.4.2 Implementación de una interfaz

Una clase implementa una interfaz de la manera siguiente:

1. Declarando que implementa la interfaz.
2. Definiendo implementaciones para todos los métodos de la interfaz.

```
1 /**
2 * Interfaz que define el método stretch para alargar la dimensión
3 * más larga de un objeto Shape
4 */
5 public interface Stretchable
6 {
7     void stretch( double factor );
8 }
```

---

Figura 4.16 La interfaz `Stretchable`.

En la Figura 4.17 se muestra un ejemplo. Aquí, completamos la clase `Rectangle`, que hemos utilizado en la Sección 4.2.

La línea 1 muestra que a la hora de implementar una interfaz utilizamos la palabra clave `implements` en lugar de `extends`. Podemos proporcionar cualquier método que deseemos, pero estamos obligados a proporcionar al menos los que se enumeran en la interfaz. La interfaz se implementa en las líneas 5 a 14. Observe que debemos implementar el *método exacto* especificado en la interfaz.

Una clase que implemente una interfaz puede ser ampliada por herencia, siempre y cuando no sea final. La clase derivada implementa automáticamente la interfaz.

Como podemos ver a partir de nuestro ejemplo, una clase que implemente una interfaz puede continuar ampliando mediante herencia alguna otra clase. La cláusula `extends` debe preceder a la cláusula `implements`.

La cláusula `implements` se utiliza para declarar que una clase implementa una interfaz. La clase debe implementar todos los métodos de la interfaz o continuará siendo abstracta.

#### 4.4.3 Interfaces múltiples

Como hemos mencionado anteriormente, una clase puede implementar múltiples interfaces. La sintaxis para hacerlo es sencilla. Una clase implementa múltiples interfaces de la manera siguiente:

1. Enumerando las interfaces (separadas por comas) que implementa.
2. Definiendo implementaciones para todos los métodos de las interfaces.

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

```
1 public class Rectangle extends Shape implements Stretchable
2 {
3     /* El resto de la clase no cambia con respecto a la Figura 4.12 */
4
5     public void stretch( double factor )
6     {
7         if( factor <= 0 )
8             throw new IllegalArgumentException( );
9
10        if( length > width )
11            length *= factor;
12        else
13            width *= factor;
14    }
15 }
```

Figura 4.17 La clase `Rectangle` (abreviada), que implementa la interfaz `Stretchable`.

#### 4.4.4 Las interfaces son clases abstractas

Puesto que una interfaz es una clase abstracta, se aplican todas las reglas de la herencia. Específicamente,

1. Se cumple la relación de tipo *ES-UN*. Si una clase *C* implementa la interfaz *I*, entonces *C ES-UN I* y es compatible en cuanto a tipo con *I*. Si una clase *C* implementa las interfaces *I<sub>1</sub>*, *I<sub>2</sub>* e *I<sub>3</sub>*, entonces *C ES-UN I<sub>1</sub>*, *C ES-UN I<sub>2</sub>* y *C ES-UN I<sub>3</sub>*, y es compatible en cuanto tipo con *I<sub>1</sub>*, *I<sub>2</sub>* e *I<sub>3</sub>*.
2. Puede utilizarse el operador `instanceof` para determinar si una referencia es compatible en cuanto a tipo con una interfaz.
3. Cuando una clase implementa un método de interfaz, no puede reducir la visibilidad. Puesto que todos los métodos de las interfaces son públicos, todas las implementaciones tienen que ser públicas.
4. Cuando una clase implementa un método de interfaz, no puede añadir excepciones comprobadas a la lista `throws`. Si una clase implementa múltiples interfaces en las que el mismo método incluye una lista `throws` diferente, la lista `throws` de la implementación solo puede enumerar excepciones comprobadas que pertenezcan al conjunto intersección de las listas `throws` de las distintas interfaces.
5. Cuando una clase implementa un método de interfaz, debe implementar la signatura exacta (no incluyendo la lista `throws`); en caso contrario, heredará una versión abstracta del método de la interfaz y habrá proporcionado un método no abstracto sobrecargado, pero diferente.
6. Una clase no puede implementar dos interfaces que contengan un método con la misma signatura y tipos de retorno incompatibles, ya que sería imposible proporcionar ambos métodos en una clase.
7. Si una clase deja de implementar cualquiera de los métodos de una interfaz, debe ser declarada como abstracta.
8. Las interfaces pueden ampliar mediante herencia otras interfaces (incluyendo múltiples interfaces).

### 4.5 Herencia fundamental en Java

Dos lugares importantes en los que se emplea la herencia en Java son la clase `Object` y la jerarquía de excepciones.

#### 4.5.1 La clase `Object`

Java especifica que si una clase no amplía a otra clase, entonces amplía implícitamente a la clase `Object` (definida en `java.lang`). Como resultado, toda clase es una subclase directa o indirecta de `Object`.

La clase `Object` contiene varios métodos, y puesto que no es abstracta, todos ellos contienen implementaciones. El método más comúnmente utilizado es `toString`, del que ya hemos hablado.

Si no se escribe `toString` para una clase, se proporciona automáticamente una implementación que concatena el nombre de la clase, un @ y el "código hash" de la clase.

Otros métodos importantes son `equals` y `hashCode`, del que hablaremos más detalladamente en el Capítulo 6, así como un conjunto de métodos algo complicados con los que los programadores avanzados de Java necesitan familiarizarse.

#### 4.5.2 La jerarquía de excepciones

Como se describe en la Sección 2.5, existen varios tipos de excepciones. La raíz de la jerarquía, una parte de la cual se muestra en la Figura 4.18, es `Throwable`, que define un conjunto de métodos `printStackTrace`, proporciona una implementación de `toString`, una pareja de constructores y poco más. La jerarquía se divide en `Error`, `RuntimeException` y excepciones comprobadas. Una excepción comprobada es cualquier objeto `Exception` que no sea una `RuntimeException`. Por regla general, cada nueva clase amplía otra clase de excepción, proporcionando solo un par de constructores. Se pueden proporcionar más, pero ninguna de las excepciones estándar se preocupa

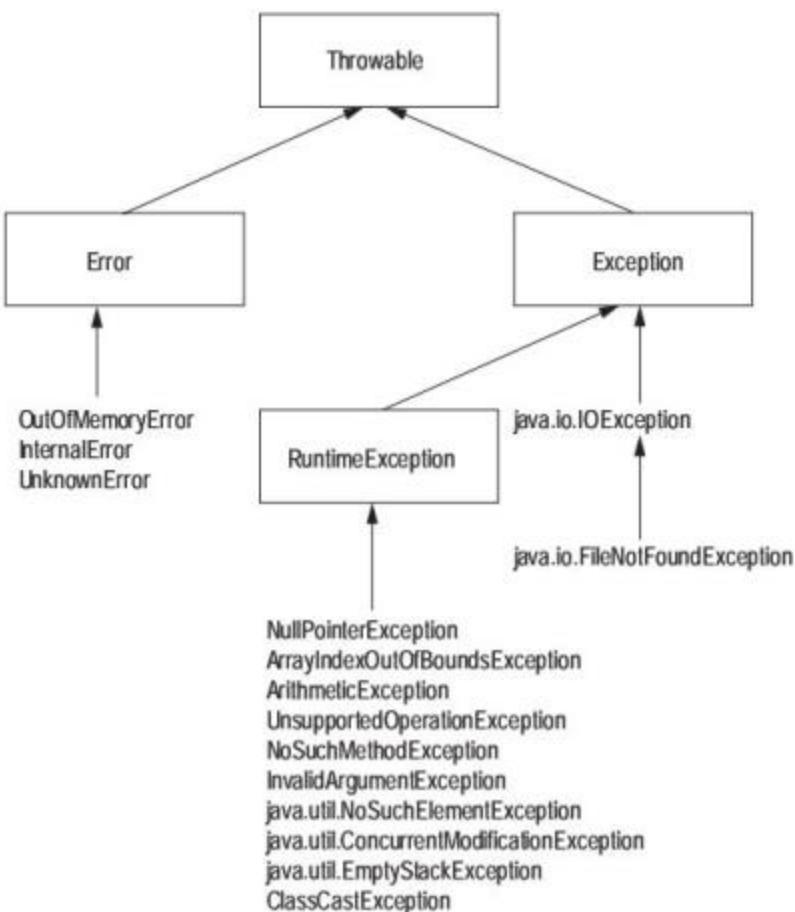


Figura 4.18 La jerarquía de excepciones (lista parcial).

```
1 package weiss.util;
2
3 public class NoSuchElementException extends RuntimeException
4 {
5     /**
6      * Construye una excepción NoSuchElementException
7      * sin ningún mensaje detallado.
8      */
9     public NoSuchElementException( )
10    {
11    }
12
13    /**
14     * Construye una excepción NoSuchElementException
15     * con un mensaje detallado.
16     * @param msg el mensaje detallado.
17     */
18    public NoSuchElementException( String msg )
19    {
20        super( msg );
21    }
22 }
```

Figura 4.19 NoSuchElementException, implementada en weiss.util.

de hacerlo. En `weiss.util`, implementamos tres de las excepciones estándar de `java.util`. Una de esas implementaciones, que ilustra que las nuevas clases de excepción suelen proporcionar poco más que constructores se muestra en la Figura 4.19.

#### 4.5.3 E/S: el patrón decorador

La E/S en Java parece bastante compleja de utilizar, pero funciona muy bien para llevar a cabo la E/S con diferentes orígenes, como el terminal, archivos o sockets Internet. Puesto que está diseñada para ser ampliable, existen un montón de clases (más de 50 en total). Resulta complicado de utilizar para tareas triviales; por ejemplo, leer un número desde el terminal requiere un trabajo sustancial.

La entrada se hace utilizando clases de flujos de datos. Puesto que Java fue diseñado para programación Internet, la mayor de la E/S se centra alrededor de la lectura y escritura orientada a bytes.

La E/S orientada a bytes se lleva a cabo con clases de flujos de datos que amplían `InputStream` o `OutputStream`. `InputStream` y `OutputStream` son clases abstractas y no interfaces, por lo que no existe el concepto de flujo de datos abierto tanto para entrada como para salida. Estas clases declaran sendos métodos `read` y `write` abstractos para E/S de un único byte, respectivamente, así como un pequeño conjunto de métodos concretos como el método `close` y métodos de E/S

en bloque (que pueden implementarse en términos de llamadas a operaciones de E/S de un único byte). Como ejemplos de estas clases se incluyen `FileInputStream` y `FileOutputStream`, así como las clases ocultas `SocketInputStream` y `SocketOutputStream`. (Los flujos de datos de tipo `socket` son generados por métodos que devuelven un objeto cuyo tipo estático es `InputStream` o `OutputStream`.)

La E/S orientada a caracteres se lleva a cabo mediante clases que amplían las clases abstractas `Reader` y `Writer`. Estas contienen también métodos `read` y `write`. No existen tantas clases `Reader` y `Writer` como clases `InputStream` y `OutputStream`.

Sin embargo, esto no es problema gracias a las clases `InputStreamReader` y `OutputStreamWriter`. Estas clases se denominan *clase puente* porque enlazan la jerarquía `Stream` con las jerarquías `Reader` y `Writer`. Un `InputStreamReader` se construye con cualquier `InputStream` y crea un objeto que *ES-UN Reader*. Por ejemplo, podemos crear un `Reader` para archivos utilizando

```
InputStream fis = new FileInputStream( "foo.txt" );
Reader fin = new InputStreamReader( fis );
```

Resulta que existe una clase `FileReader` de utilidad que ya se encarga de hacer esto. La Figura 4.20 proporciona una posible implementación.

A partir de un `Reader`, podemos realizar una E/S limitada; el método `read` devuelve un carácter. Si queremos en su lugar una línea, necesitamos una clase denominada `BufferedReader`. Al igual que otros objetos `Reader`, un `BufferedReader` se construye a partir de cualquier otro `Reader`, pero proporciona tanto un buffer de entrada como un método `readLine`. Por tanto, continuando con el ejemplo anterior,

```
BufferedReader bin = new BufferedReader( fin );
```

Envolver un `InputStream` dentro de un `InputStreamReader` y este dentro de un `BufferedReader` funciona para cualquier `InputStream`, incluyendo `System.in` o sockets. La Figura 4.21, que se asemeja a la Figura 2.17, ilustra el uso de este patrón para leer dos números desde la entrada estándar.

La idea de envolver una clase con otra es un ejemplo de un patrón de diseño Java comúnmente utilizado y con el que nos volveremos a encontrar en la Sección 4.6.2.

Similar a `BufferedReader` es `PrintWriter`, que nos permite realizar operaciones `println`.

La jerarquía `OutputStream` incluye varios envoltorios, como por ejemplo `DataOutputStream`, `ObjectOutputStream` y `GZIPOutputStream`.

Las clases  
`InputStreamReader` y  
`OutputStreamWriter`  
son clases puente que  
permiten al programador  
cruzar de la jerarquía  
`Stream` a las jerarquías  
`Reader` y `Writer`.

```
1 class FileReader extends InputStreamReader
2 {
3     public FileReader( String name ) throws FileNotFoundException
4         { super( new FileInputStream( name ) ); }
5 }
```

Figura 4.20 La clase de utilidad `FileReader`.

```
1 import java.io.InputStreamReader;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4 import java.util.Scanner;
5 import java.util.NoSuchElementException;
6
7 class MaxTest
8 {
9     public static void main( String [ ] args )
10    {
11        BufferedReader in = new BufferedReader( new
12                            InputStreamReader( System.in ) );
13
14        System.out.println( "Enter 2 ints on one line: " );
15        try
16        {
17            String oneLine = in.readLine( );
18            if( oneLine == null )
19                return;
20            Scanner str = new Scanner( oneLine );
21
22            int x = str.nextInt( );
23            int y = str.nextInt( );
24
25            System.out.println( "Max: " + Math.max( x, y ) );
26        }
27        catch( IOException e )
28        {
29            System.err.println( "Unexpected I/O error" );
30        }
31        catch( NoSuchElementException e )
32        {
33            System.err.println( "Error: need two ints" );
34        }
35    }
36 }
```

---

Figura 4.21 Un programa que ilustra cómo envolver flujos dentro de clases lectoras.

`DataOutputStream` nos permite escribir primitivas en forma binaria (en lugar de en un formato de texto legible); por ejemplo, una llamada a `writeInt` escribe los 4 bytes que representan un entero de 32 bits. El escribir los datos de esa forma evita las conversiones a formato de texto, lo que permite ahorrar tiempo y (en ocasiones) espacio. `ObjectOutputStream` nos permite escribir en un flujo de datos un objeto completo, incluyendo todos sus componentes, los componentes de sus componentes, etc. El objeto y todos sus componentes deben implementar la interfaz `Serializable`.

No hay ningún método en la interfaz; lo único que se debe declarar es que una clase es serializable.<sup>6</sup> `GZIPOutputStream` envuelve `OutputStream` y comprime los datos de salida antes de enviarlos a `OutputStream`. Además, hay una clase `BufferedOutputStream`. En el lado de `InputStream` existen unas clases envoltorios similares. Por ejemplo, suponga que tenemos una matriz de objetos `Person` serializables. Podemos escribir los objetos como una unidad, comprimidos de la forma siguiente:

```
Person [ ] p = getPersons( ); // rellenar la matriz
FileOutputStream fout = new FileOutputStream( "people.gzip" );
BufferedOutputStream bout = new BufferedOutputStream( fout );
GZIPOutputStream gout = new GZIPOutputStream( bout );
ObjectOutputStream oout = new ObjectOutputStream( gout );
oout.writeObject( p );
oout.close( );
```

Más adelante, podemos volver a leerlo todo:

```
FileInputStream fin = new FileInputStream( "people.gzip" );
BufferedInputStream bin = new BufferedInputStream( fin );
GZIPInputStream gin = new GZIPInputStream( bin );
ObjectInputStream oin = new ObjectInputStream( gin );
Person [ ] p = (Person[ ]) oin.readObject( );
oin.close( );
```

El código proporcionado en línea amplía este ejemplo, haciendo que cada objeto `Person` almacene un nombre, una fecha de nacimiento y los dos objetos `Person` que representan a los padres.

La idea de anidar envoltorios con el fin de añadir funcionalidad se conoce con el nombre de *patrón decorador*. Al hacer esto, disponemos de numerosas clases de pequeño tamaño que se combinan para proporcionar una potente interfaz. Sin este patrón, cada origen de E/S tendría que disponer de su propia funcionalidad para compresión, serialización, E/S orientada a carácter, E/S orientada a byte, etc. Utilizando el patrón, cada origen es solo responsable de una E/S básica mínima, y las características adicionales son añadidas por las clases decoradoras.

La idea de emplear envoltorios anidados con el fin de añadir funcionalidad se conoce con el nombre de *patrón decorador*.

<sup>6</sup> La razón de esto es que la serialización es, de manera predeterminada, insegura. Cuando se escribe un objeto en un `ObjectOutputStream`, el formato es bien conocido, por lo que sus miembros privados pueden ser leídos por un usuario malicioso. De forma similar, cuando se vuelve a leer un objeto así escrito, los datos del flujo de datos de entrada no se comprueban para verificar su corrección, por lo que es posible leer un objeto corrupto. Existen técnicas avanzadas que pueden emplearse para garantizar la seguridad de la integridad en caso de emplear la serialización, pero esa técnica queda fuera- del alcance de este texto. Los diseñadores de la librería de serialización pensaron que la serialización no debería ser el mecanismo predeterminado, porque un uso correcto de la misma requiere conocer estos problemas, y esa es la razón de que pusieran un pequeño obstáculo en el camino.

## 4.6 Implementación de componentes genéricos mediante la herencia

La programación genérica nos permite implementar lógica dependiente del tipo de objeto.

podemos escribir un método para ordenar una matriz de elementos; la *lógica* es independiente del tipo de los objetos que se estén ordenando, por lo que podría utilizarse un método genérico.

En Java, el carácter genérico se obtiene utilizando la herencia.

A diferencia de muchos de los lenguajes más recientes (como C++, que utilizan plantillas para implementar la programación genérica), antes de la versión 1.5, Java no soportaba directamente las implementaciones genéricas. En lugar de ello, la programación genérica se implementaba utilizando los conceptos básicos de la herencia. Esta sección describe cómo pueden implementarse métodos y clases genéricos en Java utilizando los principios básicos de herencia.

El soporte directo para métodos y clases genéricos fue anunciado por Sun en junio de 2001 como futura adición al lenguaje. Finalmente, a finales de 2004, se lanzó la versión Java 5 y se proporcionó un soporte para métodos y clases genéricos. Sin embargo, la utilización de clases genéricas requiere comprender las estructuras sintácticas de programación genérica anteriores a Java 5. Como resultado, comprender cómo se emplea la herencia para implementar programas genéricos resulta esencial, e incluso en la versión Java 5.

### 4.6.1 Utilización de Object para la programación genérica

La idea básica en Java es que podemos implementar una clase genérica utilizando una superclase apropiada, como `Object`.

Considere la clase `IntCell` mostrada en la Figura 3.2. Recuerde que `IntCell` soporta los métodos `read` y `write`. En principio, podemos convertir esto en una clase `MemoryCell` genérica que almacene cualquier tipo de `Object`, sustituyendo las instancias de `int` por `Object`. En la Figura 4.22 se muestra la clase `MemoryCell` resultante.

Hay dos detalles que debemos considerar al utilizar esta estrategia. El primero se ilustra en la Figura 4.23, que muestra un método `main` que escribe un "37" en un objeto `MemoryCell` y luego lee del objeto `MemoryCell`. Para acceder a un método específico del objeto, debemos hacer una especialización al tipo correcto. (Por supuesto, en este ejemplo no necesitamos la especialización, puesto que simplemente estamos invocando el método `toString` en la línea 9, y esto puede hacerse para cualquier objeto.)

Un segundo detalle de importancia es que no se pueden utilizar los tipos primitivos. Solo los tipos de referencia son compatibles con `Object`. En breve expondremos una solución bastante común a este problema.

`MemoryCell` es un ejemplo relativamente sencillo. La Figura 4.24 constituye un ejemplo de más entidad y que es típico de la reutilización de código genérico; en ella se muestra una clase genérica `ArrayList` simplificada tal y como se escribiría antes de Java 5, pudiendo el lector interesado encontrar algunos métodos adicionales en el código que se suministra en línea.

```

1 // Clase MemoryCell
2 // Object read( )           --> Devuelve el valor almacenado
3 // void write( Object x ) --> Se almacena x
4
5 public class MemoryCell
6 {
7     // Métodos públicos
8     public Object read( ) { return storedValue; }
9     public void write( Object x ) { storedValue = x; }
10
11    // Representación privada interna de los datos
12    private Object storedValue;
13 }

```

Figura 4.22 Una clase MemoryCell genérica (pre-Java 5).

```

1 public class TestMemoryCell
2 {
3     public static void main( String [ ] args )
4     {
5         MemoryCell m = new MemoryCell( );
6
7         m.write( "37" );
8         String val = (String) m.read( );
9         System.out.println( "Contents are: " + val );
10    }
11 }

```

Figura 4.23 Utilización de la clase MemoryCell genérica (pre-Java 5).

## 4.6.2 Envoltorios para tipos primitivos

Cuando implementamos algoritmos, a menudo nos encontramos con un problema de tipos en el lenguaje: disponemos de un objeto de un tipo, pero la sintaxis del lenguaje requiere un objeto de un tipo distinto.

Esta técnica ilustra el concepto básico de *clase envoltorio*. Una aplicación típica consiste en almacenar un tipo primitivo y añadir operaciones que el tipo primitivo no soporta o no soporta correctamente. Un segundo ejemplo lo hemos visto en el sistema de E/S, en el que un envoltorio almacena una referencia a un objeto y reenvía solicitudes hacia ese objeto, embelleciendo el resultado de alguna manera (por ejemplo, añadiendo un buffer o un mecanismo de compresión). Un concepto similar es el de *clase adaptadora* (de hecho, los términos envoltorio y adaptador se utilizan a menudo de manera intercambiable). Una clase adaptadora se utiliza típicamente cuando

Una clase envoltorio almacena una entidad (la clase envuelta) y añade operaciones que el tipo original no soportaba correctamente. Una clase adaptadora se utiliza cuando la interfaz de una clase no coincide exactamente con la que necesitamos.

```
1 /**
2  * SimpleArrayList implementa una matriz ampliable de Object.
3  * Las inserciones siempre se hacen al final.
4 */
5 public class SimpleArrayList
6 {
7     /**
8      * Devuelve el número de elementos de esta colección.
9      * @return el número de elementos de esta colección.
10     */
11    public int size( )
12    {
13        return theSize;
14    }
15
16    /**
17     * Devuelve el elemento en la posición idx.
18     * @param idx el índice que hay que buscar.
19     * @throws ArrayIndexOutOfBoundsException si el índice es incorrecto.
20     */
21    public Object get( int idx )
22    {
23        if( idx < 0 || idx >= size( ) )
24            throw new ArrayIndexOutOfBoundsException( );
25        return theItems[ idx ];
26    }
27
28    /**
29     * Añade un elemento al final de esta colección.
30     * @param x cualquier objeto.
31     * @return true (según java.util.ArrayList).
32     */
33    public boolean add( Object x )
34    {
35        if( theItems.length == size( ) )
36        {
37            Object [ ] old = theItems;
38            theItems = new Object[ theItems.length * 2 + 1 ];
39            for( int i = 0; i < size( ); i++ )
40                theItems[ i ] = old[ i ];
41        }
42
43        theItems[ theSize++ ] = x;
44        return true;
45    }
46
47    private static final int INIT_CAPACITY = 10;
48
49    private int theSize = 0;
50    private Object [ ] theItems = new Object[ INIT_CAPACITY ];
51 }
```

---

Figura 4.24 Un ArrayList simplificado con add, get y size (pre-Java 5).

la interfaz de una clase no es exactamente la que necesitamos, y proporciona un efecto de envoltorio al mismo tiempo que modifica la interfaz.

En Java, ya hemos visto que, aunque todo tipo de referencia es compatible con `Object`, los ocho tipos primitivos no lo son. Como resultado, Java proporciona una clase envoltorio para cada uno de los ocho tipos primitivos. Por ejemplo, el envoltorio para un tipo `int` es `Integer`. Cada objeto envoltorio es *immutable* (lo que quiere decir que su estado no puede variar nunca), almacena un valor primitivo que se configura en el momento de construir el objeto y proporciona un método para consultar ese valor. Las clases envoltorio también contienen diversos métodos estáticos de utilidad.

Como ejemplo, la Figura 4.25 muestra cómo podemos utilizar el `ArrayList` de Java 5 para almacenar enteros. Observe especialmente que no podemos utilizar `ArrayList<int>`.

### 4.6.3 Autoboxing/unboxing

El código de la Figura 4.25 resulta incómodo de escribir, porque la utilización de la clase envoltorio requiere crear un objeto `Integer` antes de la llamada a `add`, y luego extraer el valor `int` del `Integer`, utilizando el método `intValue`. Antes de Java 1.4, esto era obligatorio, porque si se pasaba un `int` en algún lugar donde hacía falta un objeto `Integer`, el compilador generaba un mensaje de error, como también lo generaba si se asignaba el resultado de un objeto `Integer` a un `int`. Este código de la Figura 4.25 refleja con precisión la distinción entre tipos primitivos y tipos de referencia, pero no consigue expresar de manera limpia la intención del programador de almacenar valores `int` dentro de la colección.

Java 5 rectifica esta situación. Si se pasa un `int` en un lugar donde hace falta un `Integer`, el compilador inserta entre bastidores una llamada al constructor `Integer`. Esta funcionalidad se conoce con el nombre de *autoboxing* o auto-envolvimiento. Si se pasa un `Integer` en un lugar donde hace falta un `int`, el compilador inserta una llamada entre bastidores al método `intValue`. Esto se conoce con el nombre de *auto-unboxing* o auto-desenvolvimiento. Para los otros siete pares de primitiva/envoltorio se manifiesta un comportamiento similar. La Figura 4.26 ilustra el uso del

```
1 import java.util.ArrayList;
2
3 public class BoxingDemo
4 {
5     public static void main( String [ ] args )
6     {
7         ArrayList<Integer> arr = new ArrayList<Integer>();
8
9         arr.add( new Integer( 46 ) );
10        Integer wrapperVal = arr.get( 0 );
11        int val = wrapperVal.intValue();
12        System.out.println( "Position 0: " + val );
13    }
14 }
```

Figura 4.25 Una ilustración de la clase envoltorio `Integer` utilizando el `ArrayList` genérico de Java 5.

```

1 import java.util.ArrayList;
2
3 public class BoxingDemo
4 {
5     public static void main( String [ ] args )
6     {
7         ArrayList<Integer> arr = new ArrayList<Integer>( );
8
9         arr.add( 46 );
10        int val = arr.get( 0 );
11        System.out.println( "Position 0: " + val );
12    }
13 }

```

Figura 4.26 Autoboxing y unboxing.

auto-envolvimiento y el desenvolvimiento. Observe que las entidades referenciadas en el `ArrayList` siguen siendo objetos `Integer`; no se puede utilizar `int` en lugar de `Integer` en las instantaciones de `ArrayList`.

#### 4.6.4 Adaptadores: modificación de una interfaz

El *patrón adaptador* se utiliza para modificar la interfaz de una clase existente, con el fin de adaptarla a otra especificación. En ocasiones se utiliza para proporcionar una interfaz más simple,

bien con un número menor de métodos o bien con métodos más fáciles de utilizar. Otras veces, se emplea simplemente para cambiar los nombres de algunos métodos. En cualquier caso, la técnica de implementación es similar.

Ya hemos visto un ejemplo de adaptador: las clases puente `InputStreamReader` y `OutputStreamWriter` que convierten flujos de datos orientados a byte en flujos de datos orientados a carácter.

Como ejemplo adicional, nuestra clase `MemoryCell` de la Sección 4.6.1 utiliza `read` y `write`. ¿Pero qué sucedería si quisieramos que la interfaz empleara `get` y `put` en lugar de esos métodos? Hay dos alternativas razonables. Una consiste en cortar y pegar una clase completamente nueva. La otra es usar la técnica de *composición*, en la que diseñamos una nueva clase que envuelve el comportamiento de una clase existente.

En la Figura 4.27 utilizamos esta técnica para implementar la nueva clase, `StorageCell`. Sus métodos están implementados mediante llamadas a la clase `MemoryCell` envuelta. Es tentador utilizar la herencia en lugar de la composición, pero la herencia suplementa la interfaz (es decir, añade métodos adicionales, pero dejando los originales). Si ese es el comportamiento apropiado que buscamos, entonces por supuesto que la herencia puede ser preferible a la composición.

El patrón adaptador se utiliza para modificar la interfaz de una clase existente, con el fin de adaptarla a otra especificación.

```
1 // A class for simulating a memory cell.  
2 public class StorageCell  
3 {  
4     public Object get()  
5     { return m.read(); }  
6  
7     public void put( Object x )  
8     { m.write( x ); }  
9  
10    private MemoryCell m = new MemoryCell();  
11 }
```

Figura 4.27 Una clase adaptadora que modifica la interfaz de `MemoryCell` para utilizar `get` y `put`.

#### 4.6.5 Utilización de tipos de interfaz para la programación genérica

Utilizar `Object` como un tipo genérico funciona solo si las operaciones que se están realizando pueden expresarse empleando solo los métodos disponibles en la clase `Object`.

Considere, por ejemplo, el problema de localizar el elemento máximo de una matriz de elementos. El código básico es independiente del tipo, pero requiere la capacidad de comparar cualesquiera dos objetos y decidir cuál de ellos es más grande y cuál es más pequeño. Por ejemplo, he aquí el código básico para encontrar el `BigInteger` máximo dentro de una matriz:

```
public static BigInteger findMax( BigInteger [ ] arr )  
{  
    int maxIndex = 0;  
  
    for( int i = 1; i < arr.length; i++ )  
        if( arr[i].compareTo( arr[ maxIndex ] ) < 0 )  
            maxIndex = i;  
  
    return arr[ maxIndex ];  
}
```

Encontrar el elemento máximo en una matriz de `String`, donde el máximo se entiende desde el punto de vista lexicográfico (es decir, el último elemento alfabéticamente) utiliza el mismo código básico.

```
public static String findMax( String [ ] arr )  
{  
    int maxIndex = 0;  
  
    for( int i = 1; i < arr.length; i++ )
```

```

if( arr[i].compareTo( arr[ maxIndex ] ) < 0 )
    maxIndex = i;

return arr[ maxIndex ];
}

```

Si deseamos que `findMax` funcione para ambos tipos o incluso para otros tipos que también dispongan de un método `compareTo`, entonces deberíamos poder hacerlo, siempre y cuando seamos capaces de identificar un tipo que sirva como unificador. En este sentido, resulta que el lenguaje Java define la interfaz `Comparable`, que contiene un método `compareTo`. Muchas clases de librería implementan esta interfaz, y también nosotros podemos implementarla en nuestras propias clases. La Figura 4.28 muestra la jerarquía básica. Las versiones anteriores de Java requerían que los parámetros de `compareTo` se enumeraran como de tipo `Object`; las versiones más recientes (desde Java 5) hacen que `Comparable` sea una interfaz genérica, que veremos en la Sección 4.7.

Con esta interfaz, podemos simplemente escribir la rutina `findMax` de modo que acepte una matriz de objetos `Comparable`. El estilo más antiguo, anterior a los genéricos, para `findMax` se muestra en la Figura 4.29, junto con un programa de prueba.

Es importante mencionar unas cuantas desventajas. En primer lugar, solo pueden pasarse como elementos de la matriz `Comparable` los objetos que implementen la interfaz `Comparable`. Los objetos que dispongan de un método `compareTo` pero no declaren que implementan `Comparable` no son `Comparable`, y no satisfacen el requisito de que se trate de una relación del tipo *ES-UN*.

En segundo lugar, si una matriz `Comparable` tuviera dos objetos que son incompatibles (por ejemplo, un objeto `Date` y otro `BigInteger`), el método `compareTo` generaría una `ClassCastException`. Este es el comportamiento esperado (de hecho, es el comportamiento requerido).

En tercer lugar, como antes, las primitivas no se pueden pasar como objetos `Comparable`, pero los envoltorios sí que funcionan porque implementan la interfaz `Comparable`.

En cuarto lugar, no se exige que la interfaz sea una interfaz de librería estándar.

Por último, esta solución no siempre funciona, porque podría ser imposible declarar que una clase implementa una interfaz necesaria. Por ejemplo, la clase podría ser una clase de librería, mientras que la interfaz es una interfaz definida por el usuario. Y si la clase es final, ni siquiera podemos crear una nueva clase. La Sección 4.8 ofrece otra solución para este problema, que es el *objeto función*. El objeto función utiliza también interfaces, y es quizás uno de los conceptos fundamentales que nos podemos encontrar dentro de la librería Java.

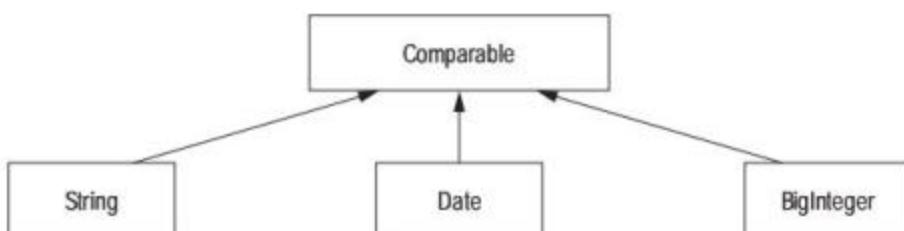


Figura 4.28 Tres clases que implementan la interfaz `Comparable`.

```
1 import java.math.BigInteger;
2
3 class FindMaxDemo
4 {
5     /**
6      * Devuelve el elemento máximo en a.
7      * Precondición: a.length > 0
8      */
9     public static Comparable findMax( Comparable [ ] a )
10    {
11        int maxIndex = 0;
12
13        for( int i = 1; i < a.length; i++ )
14            if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
15                maxIndex = i;
16
17        return a[ maxIndex ];
18    }
19
20    /**
21     * Probar findMax con objetos BigInteger y String.
22     */
23    public static void main( String [ ] args )
24    {
25        BigInteger [ ] bil = { new BigInteger( "8764" ),
26                               new BigInteger( "29345" ),
27                               new BigInteger( "1818" ) };
28
29        String [ ] stl = { "Joe", "Bob", "Bill", "Zeke" };
30
31        System.out.println( findMax( bil ) );
32        System.out.println( findMax( stl ) );
33    }
34 }
```

Figura 4.29 Una rutina `findMax` genérica, con un programa de demostración que utiliza formas geométricas y cadenas de caracteres (pre-Java 5).

## 4.7 Implementación de componentes genéricos con los componentes genéricos de Java 5

Ya hemos visto que Java 5 soporta las clases genéricas y que estas clases son fáciles de utilizar. Sin embargo, la escritura de clases genéricas requiere algo más de trabajo. En esta sección, vamos a ilustrar los fundamentos de la escritura de clases y métodos genéricos. No pretendemos cubrir todas las estructuras relevantes del lenguaje, que son bastante complejas y en ocasiones engañosas. En su lugar, mostraremos la sintaxis y las estructuras más comunes que se utilizan a lo largo de este libro.

### 4.7.1 Interfaces y clases genéricas simples

La Figura 4.30 muestra una versión genérica de la clase `MemoryCell` que antes hemos presentado en la Figura 4.22. Aquí, hemos cambiado el nombre a `GenericMemoryCell` porque ninguna de las clases se encuentra en un paquete y por tanto los nombres no pueden coincidir.

Cuando se especifica una clase genérica, la declaración de la clase incluye uno o más parámetros de tipo, encerrados entre corchetes angulares <>, después del nombre de la clase.

Las interfaces también pueden declararse como genéricas.

Cuando se especifica una clase genérica, la declaración de la clase incluye uno o más parámetros de tipo encerrados entre corchetes angulares <> después del nombre de la clase. La línea 1 muestra que `GenericMemoryCell` admite un parámetro de tipo. En este caso, no hay restricciones explícitas sobre el parámetro de tipo, por lo que el usuario puede crear tipos como `GenericMemoryCell<String>` y `GenericMemoryCell<Integer>` pero no `GenericMemoryCell<int>`. Dentro de la declaración de la clase `GenericMemoryCell`, podemos declarar campos del tipo genérico y métodos que utilicen el tipo genérico como parámetro o tipo de retorno.

Las interfaces también pueden declararse como genéricas. Por ejemplo, antes de Java 5, la interfaz `Comparable` no era genérica y su método `compareTo` tomaba un `Object` como parámetro. Como resultado, cualquier variable de referencia pasada al método `compareTo` se podía compilar, incluso si la variable no era de un tipo adecuado, y solo se informaba del error en tiempo de ejecución mediante una excepción `ClassCastException`. En Java 5, la clase `Comparable` es genérica, como se muestra en la Figura 4.31. Por ejemplo, la clase `String` ahora implementa `Comparable<String>` y tiene un método `compareTo` que admite un `String` como parámetro. Haciendo la clase genérica, muchos de los errores que antes solo se señalizaban en tiempo de ejecución han pasado a convertirse en errores de tiempo de compilación.

### 4.7.2 Comodines con límites

En la Figura 4.13 vimos un método estático que calculaba el área total en una matriz de objetos `Shape`. Suponga que queremos reescribir el método de manera que funcione un parámetro que sea `ArrayList<Shape>`. Debido al bucle `for` avanzado, el código deberá ser idéntico y el resultado se muestra en la Figura 4.32. Si pasamos un `ArrayList<Shape>`, el código funciona. Sin embargo, ¿qué ocurre si pasamos un `ArrayList<Square>`? La respuesta depende de si un `ArrayList<Square>` ES UN `ArrayList<Shape>`. Recuerde de la Sección 4.1.10 que el término técnico para esto es si tenemos covarianza o no.

```

1 public class GenericMemoryCell<AnyType>
2 {
3     public AnyType read( )
4         { return storedValue; }
5     public void write( AnyType x )
6         { storedValue = x; }
7
8     private AnyType storedValue;
9 }
```

Figura 4.30 Implementación genérica de la clase `MemoryCell`.

```

1 package java.lang;
2
3 public interface Comparable<AnyType>
4 {
5     public int compareTo( AnyType other );
6 }

```

Figura 4.31 Interfaz Comparable. Versión Java 5, que es genérica.

En Java, como hemos mencionado en la Sección 4.1.10, las matrices son covariantes. Por tanto, `Square[] ES-UN Shape[]`. Por un lado, la coherencia sugeriría que si las matrices son covariantes, entonces las colecciones también deberían serlo. Por otro lado, como vimos en la Sección 4.1.10, la covarianza de matrices conduce a la obtención de código que se compila pero que luego genera una excepción de tiempo de ejecución (un `ArrayStoreException`). Puesto que la única razón de tener genéricos es generar errores de compilación en lugar de excepciones de tiempo de ejecución cuando haya desadaptaciones de tipos, las colecciones genéricas no son covariantes. Como resultado, no podemos pasar un `ArrayList<Square>` como parámetro al método de la Figura 4.32.

Las colecciones genéricas  
no son covariantes.

La conclusión es que los genéricos (y las colecciones genéricas) no son covariantes (lo que tiene sentido), mientras que las matrices sí lo son. Sin sintaxis adicional, los usuarios tenderían a evitar las colecciones, porque la falta de covarianza hace que el código sea menos flexible.

Java 5 solventa este problema mediante *comodines*. Los comodines se utilizan para expresar subclases (o superclases) de tipos de parámetros. La Figura 4.33 ilustra el uso de comodines con un límite para escribir un método `totalArea` que admite como parámetro un `ArrayList<T>`, donde `T ES-UN Shape`. Así, tanto `ArrayList<Shape>` como `ArrayList<Square>` serían parámetros aceptables. Los comodines también pueden utilizarse sin un límite (en cuyo caso se da por supuesto que lo que se pretende es incluir `extends Object`) o con `super` en lugar de `extends` (para referirse a una superclase en lugar de a una subclase); hay también algunos otros usos sintáticos de los que no vamos a hablar aquí.

Los comodines se utilizan  
para expresar subclases  
(o superclases) de tipos de  
parámetros.

```

1 public static double totalArea( ArrayList<Shape> arr )
2 {
3     double total = 0;
4
5     for( Shape s : arr )
6         if( s != null )
7             total += s.area();
8
9     return total;
10 }

```

Figura 4.32 Método `totalArea` que no funciona si se le pasa un `ArrayList<Square>`.

```

1 public static double totalArea( ArrayList<? extends Shape> arr )
2 {
3     double total = 0;
4
5     for( Shape s : arr )
6         if( s != null )
7             total += s.area();
8
9     return total;
10 }

```

Figura 4.33 Método totalÁrea revisado con comodines y que funciona si se le pasa un `ArrayList<Square>`.

#### 4.7.3 Métodos estáticos genéricos

El método genérico se asemeja bastante a la clase genérica, en el sentido de que la lista de parámetros de tipo utiliza la misma sintaxis. La lista de tipos en un método genérico precede al tipo de retorno.

En un cierto sentido, el método `totalÁrea` de la Figura 4.33 es genérico, ya que funciona para distintos tipos. Pero no hay una lista específica de parámetros de tipo, como se hacía en la declaración de la clase `GenericMemoryCell`. En ocasiones, el tipo específico es importante, quizás porque es aplicable una de las siguientes razones:

1. El tipo se utiliza como tipo de retorno.
2. El tipo se utiliza en más de un tipo parámetro.
3. El tipo se utiliza para declarar una variable local.

En caso de aplicarse alguna de estas razones, entonces hay que declarar un método genérico explícito con parámetros de tipo.

Por ejemplo, la Figura 4.34 ilustra un método estático genérico que realiza una búsqueda secuencial del valor `x` en la matriz `arr`. Utilizando un método genérico en lugar de uno no genérico que emplee `Object` como tipo de parámetro, podemos obtener errores de tiempo de compilación si tratamos de encontrar un objeto `Apple` en una matriz de objetos `Shape`.

```

1 public static <AnyType>
2 boolean contains( AnyType [ ] arr, AnyType x )
3 {
4     for( AnyType val : arr )
5         if( x.equals( val ) )
6             return true;
7
8     return false;
9 }

```

Figura 4.34 Método estático genérico para buscar en una matriz.

El método genérico se parece bastante a la clase genérica, en el sentido de que la lista de parámetros de tipo utiliza la misma sintaxis. Los parámetros de tipo en un método genérico preceden al tipo de retorno.

#### 4.7.4 Límites de tipo

Suponga que queremos escribir una rutina `findMax`. Considere el código de la Figura 4.35. Este código no puede funcionar, porque el compilador no puede comprobar que la llamada a `compareTo` en la línea 6 es válida; solo se garantiza que `compareTo` exista si `AnyType` es `Comparable`. Podemos resolver este problema utilizando un *límite de tipo*. El límite de tipo se especifica dentro de los corchetes angulares `<>`, y especifica las propiedades que los tipos de parámetro deben tener. Un intento simplista sería reescribir la firma como

```
public static <AnyType extends Comparable> ...
```

El límite de tipo se especifica dentro de los corchetes angulares `<>`.

Esto es simplista porque, como ya sabemos, la interfaz `Comparable` ahora es genérica. Aunque este código se compilaría, una mejor solución sería

```
public static <AnyType extends Comparable<AnyType>> ...
```

Sin embargo, este intento no es satisfactorio. Para ver dónde está el problema, suponga que `Shape` implementa `Comparable<Shape>`. Suponga que `Square` amplía `Shape`. Entonces, todo lo que sabemos es que `Square` implementa `Comparable<Shape>`. Por tanto, un `Square` ES-UN `Comparable<Shape>`, pero NO-ES-UN `Comparable<Square>`.

Como resultado, lo que necesitamos decir es que `AnyType` ES-UN `Comparable<T>`, donde `T` es una superclase de `AnyType`. Puesto que no necesitamos conocer el tipo exacto `T`, podemos emplear un comodín. La firma resultante es:

```
public static <AnyType extends Comparable<? super AnyType>> ...
```

La Figura 4.36 muestra la implementación de `findMax`. El compilador aceptará matrices únicamente de tipos `T` que implementen la interfaz `Comparable<S>`, donde `T` ES-UN `S`. Ciertamente,

```
1 public static <AnyType> AnyType findMax( AnyType [ ] a )
2 {
3     int maxIndex = 0;
4
5     for( int i = 1; i < a.length; i++ )
6         if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
7             maxIndex = i;
8
9     return a[ maxIndex ];
10 }
```

Figura 4.35 Método estático genérico para encontrar el elemento más grande de una matriz y que no funciona adecuadamente.

```

1 public static <AnyType extends Comparable<? super AnyType>>
2 AnyType findMax( AnyType [ ] a )
3 {
4     int maxIndex = 0;
5
6     for( int i = 1; i < a.length; i++ )
7         if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
8             maxIndex = i;
9
10    return a[ maxIndex ];
11 }

```

Figura 4.36 Método estático genérico para encontrar el mayor elemento dentro de una matriz. Ilustra un límite impuesto al parámetro de tipo.

la declaración de límites parece un poco liosa. Afortunadamente, no nos vamos a topar con otra cosa más complicada que esta estructura sintáctica.

#### 4.7.5 Borrado de tipos

Las clase genéricas son convertidas por el compilador en clases no genéricas mediante un proceso conocido con el nombre de *borrado de tipos*.

Los genéricos no hacen el código más rápido. Lo que si hacen es que sea más seguro, en cuanto al tratamiento de tipos, en tiempo de compilación.

Los tipos genéricos, en su mayor parte, son estructuras del lenguaje Java, pero no existen como tales en la Máquina Virtual. Las clases genéricas son convertidas por el compilador en clases no genéricas mediante un proceso conocido con el nombre de *borrado de tipos*. La versión simplificada de lo que sucede es que el compilador genera una *clase en bruto* con el mismo nombre que la clase genérica y en la que los parámetros de tipo se han eliminado. Las variables de tipo se sustituyen por sus límites y cuando se realicen llamadas a métodos genéricos que tienen un tipo de retorno borrado, se insertan automáticamente con versiones de tipos. Si se utiliza una clase genérica sin un parámetro de tipo, se emplea directamente la clase en bruto.

Una consecuencia importante del mecanismo de borrado de tipos es que el código generado no es muy diferente del que los programadores habían estado escribiendo durante años antes de la aparición de los genéricos, y de hecho, no es más rápido. La ventaja más significativa es que el programador no tiene que insertar él mismo conversiones de tipos en el código, y que el compilador podrá realizar una significativa verificación de los tipos.

#### 4.7.6 Restricciones a los genéricos

Hay numerosas restricciones que afectan a los tipos genéricos. Cada una de las restricciones que aquí se numeran es de carácter obligatorio, a causa del mecanismo de borrado de tipos.

Los tipos primitivos no se pueden usar como parámetro de tipo.

##### Tipos primitivos

Los tipos primitivos no se pueden usar como parámetro de tipo. Por tanto, `ArrayList<int>` es ilegal. Es necesario utilizar clases envoltorio.

## Pruebas instanceof

Las pruebas `instanceof` y las conversiones de tipos solo funcionan con el tipo en bruto. Así si

```
ArrayList<Integer> list1 = new ArrayList<Integer>();
list1.add( 4 );
Object list = list1;
ArrayList<String> list2 = (ArrayList<String>) list;
String s = list2.get( 0 );
```

Las pruebas `instanceof` y las conversiones de tipos solo funcionan con el tipo en bruto.

fuerá legal, entonces en tiempo de ejecución la conversión de tipo se realizaría correctamente, puesto que todos los tipos son `ArrayList`. Eventualmente, se produciría un error de tiempo de ejecución en la última línea, porque la llamada a `get` trataría de devolver un objeto `String` pero no podría.

## Contextos estáticos

En una clase genérica, los métodos y campos estáticos no pueden hacer referencia a las variables del tipo de la clase, puesto que después de un borrado de tipos, no existe ninguna variable de tipo. Además, puesto que solo hay realmente una clase en bruto, los campos estáticos son compartidos entre todas las instantaciones genéricas de la clase.

Los métodos y campos estáticos no pueden hacer referencia a las variables de tipo de la clase. Los campos estáticos son compartidos por todas las instantaciones genéricas de la clase.

## Instanciación de tipos genéricos

Es ilegal crear una instancia de un tipo genérico. Si `T` es una variable de tipo, la instrucción

```
T obj = new T(); // El lado derecho es ilegal.
```

Es ilegal crear una instancia de un tipo genérico.

es ilegal. `T` es sustituida por sus límites, que podrían ser `Object` (o incluso una clase abstracta), por lo que la llamada a `new` no tiene ningún sentido.

## Objetos matriz genéricos

Es ilegal crear una matriz de tipo genérico. Si `T` es una variable de tipo, la instrucción

```
T [ ] arr = new T[ 10 ]; // El lado derecho es ilegal.
```

Es ilegal crear una matriz de un tipo genérico.

es ilegal. `T` sería sustituida por sus límites, que probablemente serían `Object`, y entonces la conversión de tipo (generada por el mecanismo de borrado de tipos) a `T[]` fallaría, porque `Object[] NO-ES-UN T[]`. La Figura 4.37 muestra una versión genérica de la clase `SimpleArrayList` que hemos visto anteriormente en la Figura 4.24. La única parte complicada es el código de la línea 38. Puesto que no podemos crear matrices de objetos genéricos, debemos crear una matriz de `Object` y luego utilizar una conversión de tipo. Esta conversión de tipo generará una advertencia del compilador acerca de una conversión de tipo no comprobada. Es imposible implementar las clases de colección genéricas con matrices sin obtener esta advertencia de compilación. Si los clientes quieren que su código se compile sin advertencias, deben utilizar únicamente tipos de colección genéricos, no tipos de matriz genéricos.

```
1 /**
2  * GenericSimpleArrayList ampliable de Object.
3  * Las inserciones siempre se hacen al final.
4 */
5 public class GenericSimpleArrayList<AnyType>
6 {
7     /**
8      * Devuelve el número de elementos de esta colección.
9      * @return el número de elementos de esta colección.
10     */
11    public int size( )
12    {
13        return theSize;
14    }
15
16    /**
17     * Devuelve el elemento en la posición idx.
18     * @param idx el índice que hay que buscar.
19     * @throws ArrayIndexOutOfBoundsException si el índice es incorrecto.
20     */
21    public AnyType get( int idx )
22    {
23        if( idx < 0 || idx >= size( ) )
24            throw new ArrayIndexOutOfBoundsException( );
25        return theItems[ idx ];
26    }
27
28    /**
29     * Añade un elemento al final de esta colección.
30     * @param x cualquier objeto
31     * @return true.
32     */
33    public boolean add( AnyType x )
34    {
35        if( theItems.length == size( ) )
36        {
37            AnyType [ ] old = theItems;
38            theItems = (AnyType [ ])new Object[size( )*2 + 1];
39            for( int i = 0; i < size( ); i++ )
40                theItems[ i ] = old[ i ];
41        }
42    }
```

---

*Continúa*

Figura 4.37 Clase SimpleArrayList utilizando genéricos.

```

43     theItems[ theSize++ ] = x;
44     return true;
45   }
46
47   private static final int INIT_CAPACITY = 10;
48
49   private int theSize;
50   private AnyType [ ] theItems;
51 }
```

Figura 4.37 (Continuación).

### Matrices de tipos parametrizados

La instantación de matrices de tipo parametrizado es ilegal. Considere el siguiente código:

```

ArrayList<String> [ ] arr1 = new ArrayList<String>[ 10 ];
Object [ ] arr2 = arr1;
arr2[ 0 ] = new ArrayList<Double>( );
```

La instantación de matrices de tipos parametrizados es ilegal.

Normalmente, esperaríamos que la asignación de la línea 3, que tiene el tipo incorrecto, generara una excepción `ArrayStoreException`. Sin embargo, después del borrado de tipos, el tipo de la matriz es `ArrayList[]`, y el objeto añadido a la matriz es `ArrayList`, por lo que no se produce la excepción `ArrayStoreException`. Por tanto, este código no tiene ninguna conversión de tipos, a pesar de lo cual terminará por generar una excepción `ClassCastException`, que es exactamente la situación que se supone que los genéricos tratan de evitar.

## 4.8 El functor (objetos función)

En las Secciones 4.6 y 4.7, hemos visto cómo pueden utilizarse las interfaces para escribir algoritmos genéricos. Como ejemplo, puede emplearse el método de la Figura 4.36 para encontrar el elemento máximo en una matriz.

Sin embargo, el método `findMax` tiene una importante limitación. Nos referimos a que solo funciona para objetos que implementen la interfaz `Comparable` y sean capaces de proporcionar un método `compareTo` como base para todas las decisiones de comparación. Hay muchas situaciones en la que esto no es factible. Por ejemplo, considere la clase `SimpleRectangle` de la Figura 4.38.

La clase `SimpleRectangle` no tiene una función `compareTo`, y no puede por tanto implementar la interfaz `Comparable`. La razón principal es que, como hay muchas alternativas plausibles, resulta difícil decidir cuál sería un buen significado para `compareTo`. Podríamos basar la comparación en el área de los rectángulos, en su perímetro, en su longitud, en su anchura, etc. Una vez que escribamos `compareTo`, nos vemos limitados a utilizar el criterio elegido. ¿Y qué sucede si queremos que `findMax` funcione con varias alternativas de comparación?

La solución al problema consiste en pasar la función de comparación como segundo parámetro de `findMax`, y hacer que `findMax` utilice la función de comparación, en lugar de asumir la existencia

```

1 // Una clase de rectángulos simples.
2 public class SimpleRectangle
3 {
4     public SimpleRectangle( int len, int wid )
5     { length = len; width = wid; }
6
7     public int getLength( )
8     { return length; }
9
10    public int getWidth( )
11    { return width; }
12
13    public String toString( )
14    { return "Rectangle " + getLength( ) + " by "
15        + getWidth( ); }
16
17    private int length;
18    private int width;
19 }

```

Figura 4.38 La clase SimpleRectangle, que no implementa la interfaz Comparable.

de compareTo. De ese modo, findMax tendrá ahora dos parámetros: una matriz de objetos Object de un tipo arbitrario (que no tiene por qué tener definido compareTo) y una función de comparación.

El principal problema que nos queda por resolver es cómo pasar la función de comparación. Algunos lenguajes permiten que los parámetros sean funciones. Sin embargo, esta solución a menudo presenta problemas de eficiencia y no está disponible en todos los lenguajes orientados a objetos. Java no permite pasar funciones como parámetros; solo podemos pasar valores primitivos y referencias. Por tanto, parece que no tenemos forma de pasar una función.

Sin embargo, recuerde que un objeto está compuesto por datos y funciones. Así que podemos integrar la función en un objeto y pasar una referencia a este. De hecho, esta idea funciona en todos los lenguajes orientados a objetos. El objeto se conoce como *objeto función* y en ocasiones también se denomina *functor*.

El objeto función a menudo no contiene ningún dato. La clase contiene simplemente un único método, con un nombre determinado, que está especificado por el algoritmo genérico (en este caso, findMax). Después, al algoritmo se le pasa una instancia de la clase, que a su vez invoca el único método existente en el objeto función. Podemos diseñar diferentes funciones de comparación simplemente declarando nuevas clases.

Cada nueva clase contendrá una implementación diferente de ese único método previamente acordado.

En Java, para implementar este tipo de estructura utilizaríamos la herencia, y específicamente emplearíamos las interfaces. La interfaz se utiliza para

La palabra *functor* es otro nombre para designar a los objetos función.

La clase del objeto función contiene un método específico para el algoritmo genérico. Al algoritmo se le pasa una instancia de la clase.

```
1 package weiss.util;
2
3 /**
4  * Interfaz del objeto función Comparator.
5  */
6 public interface Comparator<AnyType>
7 {
8     /**
9      * Devuelve el resultado de comparar lhs y rhs.
10     * @param lhs primer objeto.
11     * @param rhs segundo objeto.
12     * @return < 0 si lhs es menor que rhs,
13     *         0 si lhs es igual que rhs,
14     *         > 0 si lhs es mayor que rhs.
15     */
16     int compare( AnyType lhs, AnyType rhs );
17 }
```

Figura 4.39 La interfaz `Comparator`, originalmente definida en `java.util` y reescrita para el paquete `weiss.util`.

declarar la firma de la función previamente acordada. Por ejemplo, la Figura 4.39 muestra la interfaz `Comparator`, que forma parte del paquete estándar `java.util`. Recuerde que para ilustrar cómo está implementada la librería Java, vamos a reimplementar una parte de `java.util` como `weiss.util`. Antes de Java 5, esta clase no era genérica.

La interfaz dice que cualquier clase (no abstracta) que afirme ser un `Comparator` debe proporcionar una implementación del método `compare`; por tanto, cualquier objeto que sea una instancia de una de esas clases dispondrá de un método `compare` al que poder invocar.

Utilizando esta interfaz, podemos ahora pasar un `Comparator` como segundo parámetro a `findMax`. Si este `Comparator` es `cmp`, podemos hacer con total seguridad la llamada `cmp.compare(o1,o2)` con el fin de comparar cualesquiera dos objetos de la forma necesaria. Se emplea un comodín en el parámetro `Comparator` para indicar que el `Comparator` sabe cómo comparar objetos que sean del mismo tipo que los de la matriz o supertipos de ellos. Es responsabilidad del llamante de `findMax` pasar como argumento real una instancia apropiadamente implementada de `Comparator`.

En la Figura 4.40 se muestra un ejemplo. `findMax` acepta ahora dos parámetros. El segundo parámetro es el objeto función. Como se muestra en la línea 11, `findMax` espera que el objeto función implemente un método denominado `compare`, y en efecto debe hacerlo así, ya que implementa la interfaz `Comparator`.

Una vez escrita la interfaz `findMax`, se puede invocar desde `main`. Para ello, necesitamos pasar a `findMax` una matriz de objetos `SimpleRectangle` y un objeto función que implemente la interfaz `Comparator`. Implementamos `OrderRectByWidth`, una nueva clase que contiene el método `compare` requerido. El método `compare` devuelve un entero que indica si el primer rectángulo es menor, igual o mayor que el segundo rectángulo, según sus anchuras. `main` simplemente pasa una instancia de

```

1 public class Utils
2 {
3     // findMax genérico con un objeto función.
4     // Precondición: a.length > 0.
5     public static <AnyType> AnyType
6         findMax( AnyType [ ] a, Comparator<? super AnyType> cmp )
7     {
8         int maxIndex = 0;
9
10        for( int i = 1; i < a.length; i++ )
11            if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
12                maxIndex = i;
13
14        return a[ maxIndex ];
15    }
16 }

```

**Figura 4.40** El algoritmo `findMax` genérico, utilizando un objeto función.

`OrderRectByWidth` a `findMax`.<sup>7</sup> Tanto `main` como `OrderRectByWidth` se muestran en la Figura 4.41. Observe que el objeto `OrderRectByWidth` no tiene miembros de datos. Esto suele pasar con todos los objetos función.

La técnica del objeto función es una ilustración de un patrón que aparece siempre una y otra vez, no solo en Java, sino en cualquier lenguaje que disponga de objetos. En Java, este patrón se emplea casi de manera continua y representa quizás el uso más extendido de las interfaces.

#### 4.8.1 Clases anidadas

Hablando en términos generales, cuando escribimos una clase, esperamos que sea útil en muchos contextos, no solo para la aplicación concreta en la que estamos trabajando.

Una característica un tanto molesta del patrón de objeto función, especialmente en Java, es el hecho de que, debido a que se utiliza tan a menudo, obliga a crear numerosas clases de pequeño tamaño, cada una de las cuales contiene un método y que se utiliza quizás una única vez dentro de un programa, teniendo una aplicabilidad bastante limitada fuera de la aplicación actual.

Esto resulta molesto por al menos dos razones distintas. En primer lugar, puede que tengamos docenas de clases de objetos función. Si son públicos estarán necesariamente dispersos en archivos separados. Si tienen visibilidad de paquete, puede que se encuentren todos en el mismo archivo, pero aun así tendremos que andar recorriendo arriba y abajo el archivo para encontrar sus definiciones, que es probable que se encuentren muy alejadas del pequeño conjunto de lugares de

<sup>7</sup> El truco para implementar `compare` mediante la operación de resta funciona para valores `int`, siempre y cuando ambos tengan el mismo signo. En caso contrario, existe una posibilidad de desbordamiento. Este truco simplificador es también la razón por la que utilizamos `SimpleRectangle`, en lugar de `Rectangle` (que almacenaba las anchuras en forma de valores `double`).

```

1 class OrderRectByWidth implements Comparator<SimpleRectangle>
2 {
3     public int compare( SimpleRectangle r1, SimpleRectangle r2 )
4         { return( r1.getWidth() - r2.getWidth() ); }
5     }
6
7 public class CompareTest
8 {
9     public static void main( String [ ] args )
10    {
11        SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
12        rects[ 0 ] = new SimpleRectangle( 1, 10 );
13        rects[ 1 ] = new SimpleRectangle( 20, 1 );
14        rects[ 2 ] = new SimpleRectangle( 4, 6 );
15        rects[ 3 ] = new SimpleRectangle( 5, 5 );
16
17        System.out.println( "MAX WIDTH: " +
18            Utils.findMax( rects, new OrderRectByWidth( ) ) );
19    }
20 }

```

Figura 4.41 Un ejemplo de objeto función.

todo el programa en los que se las instancia como objetos función. Sería preferible que cada clase de objeto función pudiera declararse lo más próxima a su instantación. En segundo lugar, una vez que se utiliza un nombre, no se puede volver a emplear dentro del paquete sin que se produzca la posibilidad de una colisión de nombres. Aunque los paquetes resuelven algunos problemas relativos al espacio de nombres, no los resuelven todos, especialmente cuando se usa dos veces el mismo nombre de clase dentro del paquete predeterminado.

Con una clase anidada, podemos resolver algunos de estos problemas. Una *clase anidada* es una declaración de clase situada dentro de otra declaración de clase –la clase externa– utilizando la palabra clave `static`. Una clase anidada se considera un miembro de la clase externa. Como resultado, puede ser pública, privada, con visibilidad de paquete o protegida, y dependiendo de la visibilidad puede o no ser accesible por parte de los métodos que no estén incluidos en la clase externa. Normalmente, la clase anidada será privada y por tanto inaccesible desde fuera de la clase externa. Asimismo, puesto que una clase anidada es un miembro de la clase externa, sus métodos pueden acceder a los miembros privados estáticos de la clase externa, y también pueden acceder a los miembros privados de instancia cuando se les proporciona una referencia a un objeto externo.

La Figura 4.42 ilustra el uso de una clase anidada en conjunción con el patrón del objeto función. La palabra clave `static` delante de la declaración de la clase anidada de `OrderRectByWidth` es esencial; sin ella, tendríamos una

Una clase anidada es una declaración de clase situada dentro de otra declaración de clase –la clase externa– utilizando la palabra clave `static`.

Una clase anidada es una parte de la clase externa y puede declararse con un especificador de visibilidad. Todos los miembros de la clase externa son visibles para los métodos de la clase anidada.

```

1 import java.util.Comparator;
2
3 class CompareTestInner1
4 {
5     private static class OrderRectByWidth implements Comparator<SimpleRectangle>
6     {
7         public int compare( SimpleRectangle r1, SimpleRectangle r2 )
8             { return r1.getWidth( ) - r2.getWidth( ); }
9     }
10
11    public static void main( String [ ] args )
12    {
13        SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
14        rects[ 0 ] = new SimpleRectangle( 1, 10 );
15        rects[ 1 ] = new SimpleRectangle( 20, 1 );
16        rects[ 2 ] = new SimpleRectangle( 4, 6 );
17        rects[ 3 ] = new SimpleRectangle( 5, 5 );
18
19        System.out.println( "MAX WIDTH: " +
20            Utils.findMax( rects, new OrderRectByWidth( ) ) );
21    }
22 }

```

Figura 4.42 Utilización de una clase anidada para ocultar la declaración de la clase OrderRectByWidth.

clase interna, que se comporta de forma diferente y que es un tipo de clase del que hablaremos posteriormente en el libro (en el Capítulo 15).

Ocasionalmente, una clase anidada es pública. En la Figura 4.42, si OrderRectByWidth se declarara pública, la clase CompareTestInner1.OrderRectByWidth podría ser utilizada desde fuera de la clase CompareTestInner1.

#### 4.8.2 Clases locales

Además de permitir declarar clases dentro de otras clases, Java también permite declarar clases dentro de métodos. Estas clases se denominan clases locales. Esto se ilustra en la Figura 4.43.

Observe que cuando se declara una clase dentro de un método, no se puede declarar `private` o `static`. Sin embargo, la clase solo es visible dentro del método en el que ha sido declarada. Esto facilita escribir la clase justo antes de su primera (y quizás única) utilización, y evitar así la polución de los espacios de nombres.

Una ventaja de declarar una clase dentro de un método es que los métodos de esa clase (en este caso, `compare`) tienen acceso a las variables locales de la función que hayan sido declaradas antes que la clase. Esto puede ser importante en algunas aplicaciones. Existe una regla técnica: para poder

Java también permite declarar clases dentro de métodos. Dichas clases se conocen con el nombre de *clases locales* y no pueden declararse con un modificador de visibilidad, ni con el modificador `static`.

```

1 class CompareTestInner2
2 {
3     public static void main( String [ ] args )
4     {
5         SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
6         rects[ 0 ] = new SimpleRectangle( 1, 10 );
7         rects[ 1 ] = new SimpleRectangle( 20, 1 );
8         rects[ 2 ] = new SimpleRectangle( 4, 6 );
9         rects[ 3 ] = new SimpleRectangle( 5, 5 );
10
11     class OrderRectByWidth implements Comparator<SimpleRectangle>
12     {
13         public int compare( SimpleRectangle r1, SimpleRectangle r2 )
14             { return r1.getWidth( ) - r2.getWidth( ); }
15     }
16
17     System.out.println( "MAX WIDTH: " +
18         Utils.findMax( rects, new OrderRectByWidth( ) ) );
19 }
20 }
```

**Figura 4.43** Utilización de una clase local para ocultar aun más la declaración de la clase OrderRectByWidth.

acceder a las variables locales, las variables deben ser declaradas como `final`. No vamos a utilizar estos tipos de clases en el texto.

### 4.8.3 Clases anónimas

Uno tendería a sospechar que al colocar una clase en la línea inmediatamente anterior a la línea de código en la que se la usa, hemos declarado la clase en el punto más próximo posible al punto de utilización. Sin embargo, en Java, podemos hacerlo todavía mejor.

Una clase anónima es una clase que no tiene nombre.

La Figura 4.44 ilustra el concepto de clase interna anónima. Una *clase anónima* es una clase que no tiene ningún nombre. La sintaxis es que en lugar de escribir `new Inner()` y de proporcionar la implementación de `Inner` como clase nominada, escribimos `new Interface()`, y luego proporcionamos la implementación de la interfaz (todo, desde el corchete de apertura al de cierre) inmediatamente después de la expresión `new`. En lugar de implementar una interfaz anónimamente, también es posible ampliar anónimamente mediante herencia una clase, proporcionando solo los métodos sustituidos.

La sintaxis parece aterradora, pero al cabo de poco tiempo uno llega a acostumbrarse. Complica el lenguaje significativamente, porque la clase anónima es una clase. Un ejemplo de estas complicaciones es que puesto que el nombre de un constructor es el nombre de una clase, ¿cómo se define un constructor para una clase anónima? La respuesta es que no se puede definir.

Las clases anónimas introducen significativas complicaciones en el lenguaje.

```

1 class CompareTestInner3
2 {
3     public static void main( String [ ] args )
4     {
5         SimpleRectangle [ ] rects = new SimpleRectangle[ 4 ];
6         rects[ 0 ] = new SimpleRectangle( 1, 10 );
7         rects[ 1 ] = new SimpleRectangle( 20, 1 );
8         rects[ 2 ] = new SimpleRectangle( 4, 6 );
9         rects[ 3 ] = new SimpleRectangle( 5, 5 );
10
11        System.out.println( "MAX WIDTH: " +
12                            Utils.findMax( rects, new Comparator<SimpleRectangle>()
13                            {
14                                public int compare( SimpleRectangle r1, SimpleRectangle r2 )
15                                { return r1.getWidth( ) - r2.getWidth( ); }
16                            }
17                        ) );
18    }
19 }

```

**Figura 4.44** Utilización de una clase anónima para implementar el objeto función.

Las clases anónimas suelen utilizarse para implementar objetos función.

La clase anónima es en la práctica muy útil, y su uso suele considerarse parte del patrón de objeto función, en conjunción con el tratamiento de sucesos en las interfaces de usuario. En el tratamiento de sucesos, el programador está obligado a especificar una función, lo que sucede cada vez se produce un cierto suceso.

#### 4.8.4 Clases anidadas y genéricos

Cuando se declara una clase anidada dentro de una clase genérica, la clase anidada no puede hacer referencia a los tipos de parámetro de la clase externa genérica. Sin embargo, la propia clase anidada puede ser genérica y puede reutilizar los nombres de tipos de parámetro de la clase genérica externa. Como ejemplos de sintaxis tendríamos los siguientes:

```

class Outer<AnyType>
{
    public static class Inner<AnyType>
    {

    }

    public static class OtherInner
    {
        // Aquí no puede utilizarse AnyType
    }
}

```

```
Outer.Inner<String> i1 = new Outer.Inner<String>();
Outer.OtherInner i2 = new Outer.OtherInner();
```

Observe que en las declaraciones de `i1` e `i2`, `Outer` no tiene tipos de parámetro.

## 4.9 Detalles sobre el mecanismo de despacho dinámico

Un mito muy común es el de que todos los métodos y todos los parámetros se acoplan en tiempo de ejecución. Esto no es cierto. En primer lugar, hay algunos casos en los que el despacho dinámico nunca se utiliza o ni siquiera tiene sentido:

- Métodos estáticos, independientemente de cómo se invoque el método.
- Métodos finales.
- Métodos privados (puesto que se invocan exclusivamente desde dentro de la clase y por tanto son implícitamente finales).

En otros escenarios, el mecanismo de despacho dinámico se utiliza de una manera significativa. ¿Pero qué significa exactamente eso del despacho dinámico?

El *despacho dinámico* significa que se utilizará en cada momento el método que sea apropiado para el objeto con el que se esté operando. Sin embargo, no quiere decir que siempre se vaya a seleccionar la correspondencia más perfecta para todos los parámetros. Específicamente, en Java, los parámetros de un método siempre se deducen estáticamente, en tiempo de compilación.

Para ver un ejemplo concreto, considere el código de la Figura 4.45. En el método `whichFoo`, se hace una llamada a `foo`. ¿Pero a cuál `foo` se llama? Cabría esperar que la respuesta dependiera de los tipos en tiempo de ejecución de `arg1` y `arg2`.

Puesto que las correspondencias de parámetros siempre se realizan en tiempo de compilación, no importa a qué tipo esté refiriéndose realmente `arg2`. El método `foo` que se seleccionará será

```
public void foo( Base x )
```

La única cuestión es si se empleará la versión de `Base` o de `Derived`. Esa es la decisión que se toma en tiempo de ejecución, una vez que se conoce el objeto al que `arg1` hace referencia.

La metodología precisa utilizada es que el compilador deduce en tiempo de compilación la mejor firma, basándose en los tipos estáticos de los parámetros y de los métodos disponibles para el tipo estático de la referencia controladora. En ese punto, se fija la firma del método. Este paso se denomina *sobrecarga estática*. El único problema que resta es determinar de cuál clase hay que utilizar la versión de ese método. Esto se lleva a cabo haciendo que la Máquina Virtual deduzca el tipo de ese objeto en tiempo de ejecución. Una vez conocido el tipo en tiempo de ejecución, la Máquina Virtual recorre hacia arriba la jerarquía de herencia, buscando la última versión sustituida del método; este será el primer método de la firma

**El mecanismo de despacho dinámico no es importante para los métodos estáticos, finales o privados.**

**En Java, los parámetros de un método siempre se deducen estáticamente, en tiempo de compilación.**

**La sobrecarga estática significa que los parámetros de un método siempre se deducen estáticamente, en tiempo de compilación.**

**El despacho dinámico significa que, una vez que se ha determinado la firma de un método de instancia, la clase del método puede determinarse en tiempo de ejecución basándose en el tipo dinámico del objeto que realiza la invocación.**

```
1 class Base
2 {
3     public void foo( Base x )
4         { System.out.println( "Base.Base" ); }
5
6     public void foo( Derived x )
7         { System.out.println( "Base.Derived" ); }
8 }
9
10 class Derived extends Base
11 {
12     public void foo( Base x )
13         { System.out.println( "Derived.Base" ); }
14
15     public void foo( Derived x )
16         { System.out.println( "Derived.Derived" ); }
17 }
18
19 class StaticParamsDemo
20 {
21     public static void whichFoo( Base arg1, Base arg2 )
22     {
23         // Se garantiza que llamaremos a foo( Base )
24         // El único problema es qué versión de clase utilizar
25         // de foo( Base ): para decidir, se utiliza el tipo
26         // dinámico de arg1.
27         arg1.foo( arg2 );
28     }
29
30     public static void main( String [] args )
31     {
32         Base b = new Base( );
33         Derived d = new Derived( );
34
35         whichFoo( b, b );
36         whichFoo( b, d );
37         whichFoo( d, b );
38         whichFoo( d, d );
39     }
40 }
```

---

Figura 4.45 Ilustración del acoplamiento estático de parámetros.

apropiada que la Máquina Virtual encuentre a medida que vaya subiendo hacia `Object`.<sup>8</sup> El segundo paso se denomina *despacho dinámico*.

La sobrecarga estática puede conducir a errores sutiles cuando un método que se suponía que tenía que ser sustituido ha sido, en su lugar, sobrecargado. La Figura 4.46 ilustra un error de programación común que se produce a la hora de implementar el método `equals`.

```
1 final class SomeClass
2 {
3     public SomeClass( int i )
4     { id = i; }
5
6     public boolean sameVal( Object other )
7     { return other instanceof SomeClass && equals( other ); }
8
9     /**
10      * ¡Esta es una mala implementación!
11      * other tiene el tipo incorrecto, por lo que esto
12      * no sustituye el método equals de Object.
13      */
14     public boolean equals( SomeClass other )
15     { return other != null && id == other.id; }
16
17     private int id;
18 }
19
20 class BadEqualsDemo
21 {
22     public static void main( String [ ] args )
23     {
24         SomeClass obj1 = new SomeClass( 4 );
25         SomeClass obj2 = new SomeClass( 4 );
26
27         System.out.println( obj1.equals( obj2 ) ); // true
28         System.out.println( obj1.sameVal( obj2 ) ); // false
29     }
30 }
```

Figura 4.46 Ilustración de cómo se puede sobrecargar `equals` en lugar de sustituirlo. Aquí, la llamada a `sameVal` devuelve `false`.

<sup>8</sup> Si no encuentra tal método, quizás porque solo se recompiló parte del programa, entonces la Máquina Virtual genera una excepción `NoSuchMethodException`.

El método `equals` se define en la clase `Object` y está pensado para devolver `true` si dos objetos tienen estados idénticos. Toma un `Object` como parámetro y el `Object` proporciona una implementación predeterminada que devuelve `true` si los dos objetos son el mismo. En otras palabras, en la clase `Object`, la implementación de `equals` es aproximadamente

```
public boolean equals( Object other )
{ return this == other; }
```

Al sustituir `equals`, el parámetro debe ser de tipo `Object`; en caso contrario, lo que estaremos haciendo es una sobrecarga del método. En la Figura 4.46, `equals` no es sustituido, en lugar de ello se sobrecarga (de forma no intencionada). Como resultado, la llamada a `sameVal` devolverá `false`, lo que parece sorprendente, ya que la llamada a `equals` devuelve `true` y `sameVal` llama a `equals`.

El problema es que la llamada en `sameVal` es `this.equals(other)`. El tipo estático de `this` es `SomeClass`. En `SomeClass`, hay dos versiones de `equals`: el `equals` allí indicado que toma un objeto `SomeClass` como parámetro y el `equals` heredado que admite un `Object`. El tipo estático del parámetro (`other`) es `Object`, por lo que la correspondencia más perfecta es el método `equals` que admite un objeto de tipo `Object`. En tiempo de ejecución, la máquina virtual busca ese método `equals` y encuentra el de la clase `Object`. Y puesto que `this` y `other` son objetos distintos, el método `equals` de la clase `Object` devuelve `false`.

Por tanto, `equals` debe escribirse para admitir un `Object` como parámetro, y normalmente hará falta una especialización del tipo, después de verificar que el tipo es apropiado. Una forma de hacer esto es utilizar una prueba `instanceof`, pero eso solo resulta seguro para las clases finales. Sustituir `equals` es, en realidad, bastante complicado en presencia de herencia, y hablaremos de ello en la Sección 6.7.

## Resumen

---

La herencia es una potente característica que constituye una parte esencial de la programación orientada a objetos y de Java. Nos permite abstraer funcionalidad en clases base abstractas y hacer que las clases derivadas implementen y amplíen dicha funcionalidad. En la clase base pueden especificarse varios tipos de métodos, como se ilustra en la Figura 4.47.

La clase más abstracta, en la que no se permite ninguna implementación, es la *interfaz*. La interfaz enumera los métodos que tienen que ser implementados con una clase derivada. La clase derivada debe implementar todos esos métodos (o ser ella misma abstracta) y especificar, mediante la cláusula `implements`, que está implementando una interfaz. Una clase puede implementar múltiples interfaces, proporcionando así una alternativa más simple al mecanismo de herencia múltiple.

Por último, la herencia nos permite escribir más fácilmente clases y métodos genéricos, que funcionen para un amplio rango de tipos genéricos. Esto implicará normalmente utilizar una cantidad significativa de conversiones de tipos. Java 5 añade clases y métodos genéricos que ocultan esas conversiones de tipos. Las interfaces también se utilizan ampliamente para componentes genéricos, así como para implementar el patrón de objeto función.

Este capítulo concluye la primera parte del texto, en la que hemos proporcionado una panorámica de Java y de la programación orientada a objetos. Ahora continuaremos examinando los algoritmos y los componentes fundamentales del proceso de resolución de problemas.

Método	Sobrecarga	Comentarios
<code>final</code>	Potencialmente en línea	Invariante en toda la jerarquía de herencia (el método nunca se redefine).
<code>abstract</code>	En tiempo de ejecución	La clase base no proporciona ninguna implementación y es abstracta. La clase derivada debe proporcionar una implementación.
<code>static</code>	En tiempo de compilación	No hay objeto controlador.
Otros	En tiempo de ejecución	La clase base proporciona una implementación predeterminada que puede ser sustituida por las clases derivadas o aceptadas sin modificaciones por esas clases derivadas.

Figura 4.47 Cuatro tipos de métodos de clase.



## Conceptos clave

**acoplamiento estático** La decisión de cuál es la clase cuya versión de un método hay que utilizar se toma en tiempo de compilación. Solo se usa para métodos estáticos, finales o privados. (118)

**borrado de tipos** El proceso mediante el cual se reescriben las clases genéricas en forma de clases no genéricas. (152)

**boxing** Creación de una instancia en una clase envoltorio para almacenar un tipo primitivo. En Java 5, esto se hace automáticamente. (143)

**clase abstracta** Una clase que no puede construirse pero sirve para especificar la funcionalidad de las clases derivadas. (127)

**clase adaptora** Una clase que se suele utilizar cuando la interfaz de otra clase no coincide exactamente con lo que se necesita. La clase adaptadora proporciona un efecto envoltorio, al mismo tiempo que modifica la interfaz. (141)

**clase anidada** Una clase dentro de otra clase, declarada con el modificador `static`. (158)

**clase anónima** Una clase que no tiene nombre y que es útil para implementar cortos objetos función. (161)

**clase base** La clase en la que está basada la herencia. (112)

**clase derivada** Una clase completamente nueva que tiene, de todos modos, una cierta compatibilidad con la clase de la que se deriva. (112)

**clase en bruto** Una clase en la que se han eliminado los parámetros de tipo genérico. (152)

**clase final** Una clase que no puede ser ampliada mediante herencia. (118)

**clase hoja** Una clase final. (118)

**clase local** Una clase dentro de un método, declarada sin modificador de visibilidad. (160)

- clases genéricas** Añadidas en Java 5, permiten a las clases especificar parámetros de tipo y evitan una significativa cantidad de conversiones de tipo. (148)
- cláusula extends** Una cláusula utilizada para declarar que una nueva clase es una subclase de otra clase. (112)
- cláusula implements** Una cláusula utilizada para declarar que una clase implementa los métodos de una interfaz. (133)
- composición** Mecanismo preferido de herencia cuando no se cumple una relación de tipo *ES-UN*. En lugar de ello, decimos que un objeto de la clase B está compuesto de un objeto de la clase A (y de otros objetos). (108)
- despacho dinámico** Una decisión tomada en tiempo de ejecución para aplicar el método correspondiente al objeto realmente referenciado. (163)
- envoltorio** Una clase utilizada para almacenar otro tipo y añadir operaciones que el tipo primitivo no soporta, o que no soporta correctamente. (141)
- funcitor** Un objeto función. (156)
- herencia** El proceso del que podemos derivar una clase a partir de una clase base, sin perturbar la implementación de la clase base. También permite el diseño de una jerarquía de clases como `Throwable` e `InputStream`. (112)
- herencia múltiple** El proceso de衍生 una clase a partir de varias clases base. La herencia múltiple no está permitida en Java. Sin embargo, sí que está permitida una alternativa, las interfaces múltiples. (129)
- interfaz** Un tipo especial de clase abstracta que no contiene detalles de implementación. (132)
- límites de tipo** Especifican propiedades que los parámetros de tipo deben satisfacer. (151)
- llamada al constructor super** Una llamada al constructor de la clase base. (117)
- matrices covariantes** En Java, las matrices son covariantes, lo que quiere decir que `Derived[]` es compatible en cuanto a tipo con `Base[]`. (122)
- método abstracto** Un método que no tiene ninguna definición significativa y se define, por tanto, en la clase derivada. (127)
- método final** Un método que no puede ser sustituido y es invariante en toda la jerarquía de herencia. Para los métodos finales se utiliza el acoplamiento estático. (117)
- miembro de clase protegido** Accesible por parte de las clases derivadas y de las clases contenidas en el mismo paquete. (116)
- objeto función** Un objeto pasado a una función genérica con la intención de que su único método sea utilizado por la función genérica. (156)
- objeto super** Un objeto utilizado en la sustitución parcial para aplicar un método de la clase base. (119)
- parámetros de tipos** Los parámetros encerrados entre corchetes angulares `<>` en una declaración de un método o una clase genéricos. (148)
- patrón decorador** El patrón que implica la combinación de varios envoltorios con el fin de añadir funcionalidad. (139)

**polimorfismo** La capacidad de una variable de referencia para hacer referencia a objetos de varios tipos distintos. Cuando se aplican operaciones a la variable, se selecciona automáticamente la operación que resulte apropiada para el objeto realmente referenciado. (114)

**programación genérica** Utilizada para implementar lógica independiente del tipo. (140)

**relación ES-UN** Una relación en la que la clase derivada es una (variación de la) clase base. (108, 115)

**relación TIENE-UN** Una relación en la que la clase derivada tiene una (instancia de la) clase base. (108)

**relaciones subclase/superclase** Si  $X$  ES-UN  $Y$ , entonces  $X$  es una subclase de  $Y$  e  $Y$  es una superclase de  $X$ . Estas relaciones son transitivas. (115)

**sobrecarga estática** El primer paso a la hora de deducir el método que será utilizado. En este paso se utilizan los tipos estáticos de los parámetros para deducir la signatura del método que será invocado. La sobrecarga estática siempre se utiliza. (163)

**sustitución parcial** El acto de ampliar un método de una clase base para realizar tareas adicionales, pero no enteramente distintas. (119)

**tipo de retorno covariante** Sustitución de tipo de retorno por un subtipo. Esto se permite a partir de Java 5. (123)

**tipos comodín** Un símbolo ? como parámetro de tipo; permite cualquier tipo (posiblemente con límites). (149)

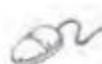
**unboxing** Creación de un tipo primitivo a partir de una instancia de una clase envoltorio. En Java 5, esto se hace automáticamente. (143)



## Errores comunes

1. Los miembros de una clase base no son visibles en la clase derivada.
2. Los objetos de una clase abstracta no pueden construirse.
3. Si la clase derivada no implementa algún método abstracto heredado, entonces la clase derivada pasa a ser abstracta. Si esto no es lo que se pretendía, se obtendrá un error de tiempo de compilación.
4. Los métodos finales no pueden ser sustituidos. Las clases finales no pueden ser ampliadas.
5. Los métodos estáticos utilizan acoplamiento estático, incluso si son sustituidos en una clase derivada.
6. Java utiliza la sobrecarga estática y selecciona siempre en tiempo de compilación la signatura de un método sobrecargado.
7. En una clase derivada, los miembros de la clase base heredados solo deben inicializarse como un agregado utilizando el método `super`. Si estos miembros son públicos o protegidos, luego pueden leerse o modificarse individualmente.

- 8 Cuando se envíe un objeto función como parámetro, hay que enviar un objeto construido, y no simplemente el nombre de la clase.
- 9 El abuso de clases anónimas es un error muy común.
- 10 La lista `throws` en un método de una clase derivada no puede redefinirse para generar una excepción que no sea generada en la clase base. Los tipos de retorno también deben corresponderse.
- 11 Cuando se sustituye un método, es ilegal reducir su visibilidad. Esto también es cierto a la hora de implementar métodos de interfaz, que por definición son siempre `public`.



## Internet

A continuación se indican los archivos disponibles para este capítulo. Parte del código se ha presentado por etapas; para esas clases, solo se proporciona una versión final.

<b>PersonDemo.java</b>	La jerarquía <code>Person</code> y el programa de prueba.
<b>Shape.java</b>	La clase abstracta <code>Shape</code> .
<b>Circle.java</b>	La clase <code>Circle</code> .
<b>Rectangle.java</b>	La clase <code>Rectangle</code> .
<b>ShapeDemo.java</b>	Un programa de prueba para el ejemplo de <code>Shape</code> .
<b>Stretchable.java</b>	La interfaz <code>Stretchable</code> .
<b>StretchDemo.java</b>	El programa de prueba para el ejemplo de <code>Stretchable</code> .
<b>NoSuchElementException.java</b>	La clase de excepción de la Figura 4.19. Forma parte de <code>weiss.util</code> . <b>ConcurrentModificationException.java</b> y <b>EmptyStackException.java</b> también están disponibles en línea.
<b>DecoratorDemo.java</b>	Una ilustración del patrón decorador, incluyendo el mecanismo de buffer, compresión y serialización.
<b>MemoryCell.java</b>	La clase <code>MemoryCell</code> de la Figura 4.22.
<b>TestMemoryCell.java</b>	El programa de prueba para la clase de la celda de memoria mostrada en la Figura 4.23.
<b>SimpleArrayList.java</b>	La clase <code>ArrayList</code> genérica simplificada de la Figura 4.24, con algunos métodos adicionales. Se proporciona un programa de prueba en <b>ReadStringsWithSimpleArrayList.java</b> .
<b>PrimitiveWrapperDemo.java</b>	Ilustra el uso de la clase <code>Integer</code> , como se muestra en la Figura 4.25.

**BoxingDemo.java**

Ilustra los mecanismos de *autoboxing* y *unboxing*, como se muestra en la Figura 4.26.

**StorageCellDemo.java**

El adaptador *StorageCell* como se muestra en la Figura 4.27, junto con un programa de prueba.

**FindMaxDemo.java**

El algoritmo genérico *findMax* de la Figura 4.29.

**GenericMemoryCell.java**

Ilustra la clase *GenericMemoryCell* de la Figura 4.30 actualizada para utilizar los genéricos de Java 5.

**GenericSimpleArrayList.java**

**TestGenericMemoryCell.java** prueba la clase. La clase *ArrayList* genérica simplificada de la Figura 4.37, con algunos métodos adicionales. Se proporciona un programa de prueba en **ReadStringsWithGenericSimpleArrayList.java**.

**GenericFindMaxDemo.java**

Ilustra el método genérico *findMax* de la Figura 4.36.

**SimpleRectangle.java**

Contiene la clase *SimpleRectangle* de la Figura 4.38.

**Comparator.java**

La interfaz *Comparator* de la Figura 4.39. Ilustra el objeto función, sin clases anidadas, como se muestra en la Figura 4.41.

**CompareTestInner1.java**

Ilustra el objeto función con una clase anidada, como se muestra en la Figura 4.42.

**CompareTestInner2.java**

Ilustra el objeto función con una clase local, como se muestra en la Figura 4.43.

**CompareTestInner3.java**

Ilustra el objeto función con una clase anónima, como se muestra en la Figura 4.44.

**StaticParamsDemo.java**

El ejemplo de sobrecarga estática y despacho dinámico mostrado en la Figura 4.45.

**BadEqualsDemo.java**

Ilustra las consecuencias de sobrecargar *equals* en lugar de sustituirlo, como se muestra en la Figura 4.46.

 **Ejercicios****EN RESUMEN**

- 41** Explique el polimorfismo. Explique el despacho dinámico. ¿Cuándo no se utiliza el despacho dinámico?

**42** ¿Qué es un método final?

**43** Considere el programa para probar la visibilidad de la Figura 4.48.

- ¿Qué accesos son ilegales?
- Haga de `main` un método de `Base`. ¿Qué accesos son ilegales?
- Haga de `main` un método de `Derived`. ¿Qué accesos son ilegales?
- ¿Cómo cambian estas respuestas si se elimina `protected` de la línea 4?
- Escriba un constructor de tres parámetros para `Base`. Después escriba un constructor de cinco parámetros para `Derived`.
- La clase `Derived` está compuesta de cinco enteros. ¿Cuáles son accesibles para la clase `Derived`?
- Se pasa un método de la clase `Derived` a un objeto `Base`. ¿A cuáles de los miembros del objeto `Base` puede acceder la clase `Derived`?

```

1 public class Base
2 {
3     public int bPublic;
4     protected int bProtect;
5     private int bPrivate;
6     // Omitidos los métodos públicos
7 }
8
9 public class Derived extends Base
10 {
11     public int dPublic;
12     private int dPrivate;
13     // Omitidos los métodos públicos
14 }
15
16 public class Tester
17 {
18     public static void main( String [ ] args )
19     {
20         Base b = new Base( );
21         Derived d = new Derived( );
22
23         System.out.println( b.bPublic + " " + b.bProtect + " "
24                             + b.bPrivate + " " + d.dPublic + " "
25                             + d.dPrivate );
26     }
27 }
```

---

Figura 4.48 Un programa para probar la visibilidad.

- 44** ¿Qué es la composición?
- 45** ¿Qué miembros de una clase heredada pueden utilizarse en la clase derivada? ¿Qué miembros pasan a ser públicos para los usuarios de la clase derivada?
- 46** ¿Qué son los mecanismos de *autoboxing* y *unboxing*?
- 47** ¿Qué es una interfaz? ¿En qué difiere una interfaz de una clase abstracta? ¿Qué miembros puede haber en una interfaz?
- 48** Explique el diseño de la librería de E/S de Java. Incluya un diagrama de la jerarquía de clases para todas las clases descritas en la Sección 4.5.3.
- 49** ¿Cuál es la diferencia entre una clase final y otras clases? ¿Por qué se utilizan las clases finales?
- 50** ¿Cómo se implementaban los algoritmos genéricos en Java antes de Java 5? ¿Cómo se implementan en Java 5?
- 51** ¿Qué es un método abstracto? ¿Qué es una clase abstracta?
- 52** Explique las reglas de Java para covarianza de matrices y colecciones genéricas. ¿Qué son los comodines y los límites de tipo, y cómo estos elementos intentan hacer que las reglas de covarianza parezcan iguales?
- 53** ¿Qué es una clase local? ¿Qué es una clase anónima?
- 54** Indique dos formas comunes de implementar adaptadores. ¿Cuáles son los compromisos existentes entre estos métodos de implementación? Describa cómo se implementan los objetos función en Java.
- 55** Explique los patrones adaptador y envoltorio. ¿En qué se diferencian?
- 56** ¿Qué es el borrado de tipos? ¿Qué restricciones aplicables a las clases genéricas son aplicables al borrado de tipos? ¿Qué es una clase en bruto?

## EN TEORÍA

- 57** Este ejercicio explora cómo realiza Java el despacho dinámico y también el motivo de que los métodos finales triviales no pueden ser integrados en línea en tiempo de compilación. Coloque cada una de las clases de la Figura 4.49 en su propio archivo.
- Compile `Class2` y ejecute el programa. ¿Cuál es la salida?
  - ¿Cuál es la firma exacta (incluyendo el tipo de retorno) del método `getX` que se deduce en tiempo de compilación en la línea 14?
  - Cambie la rutina `getX` de la línea 5 para devolver un `int`; elimine “” del cuerpo de la línea 6 y recompile `Class2`. ¿Cuál es la salida?
  - ¿Cuál es la firma exacta (incluyendo el de retorno) del método `getX` que ahora se deduce en tiempo de compilación en la línea 14?
  - Vuelva a dejar `Class1` como estaba, pero recompile solo `Class1`. ¿Cuál es el resultado de ejecutar el programa?
  - ¿Cuál habría sido el resultado si se hubiera permitido al compilador realizar una optimización en línea?

```

1 public class Class1
2 {
3     public static int x = 5;
4
5     public final String getX( )
6         { return "" + x + 12; }
7 }
8
9 public class Class2
10 {
11     public static void main( String [ ] args )
12     {
13         Class1 obj = new Class1( );
14         System.out.println( obj.getX( ) );
15     }
16 }
```

Figura 4.49 Las clases para el Ejercicio 4.17.

- 418** En cada uno de los siguientes fragmentos de código, localice los errores existentes y las conversiones de tipo innecesarias.

a. Base [ ] arr = new Base [ 2 ];
arr[ 0 ] = arr[ 2 ] = new Derived( );

Derived x = (Derived) arr[ 0 ];
Derived y = ( (Derived[])arr )[ 0 ];

b. Derived [ ] arr = new Derived [ 2 ];
arr[ 0 ] = arr[ 1 ] = new Derived( );

Base x = arr[ 0 ];
Base y = ( (Base[])arr )[ 0 ];

c. Base [ ] arr = new Derived [ 2 ];
arr[ 0 ] = arr[ 1 ] = new Derived( );

Derived x = (Derived) arr[ 0 ];
Derived y = ( (Derived[])arr )[ 0 ];

d. Base [ ] arr = new Derived [ 1 ];
arr[ 0 ] = arr[ 1 ] = new Base( );

- 419** Responda a cada apartado con Verdadero o Falso.

- Todos los métodos de una clase abstracta deben ser abstractos.
- Una clase abstracta debe proporcionar constructores.
- Una clase abstracta puede declarar datos de instancia.

- d. Una clase abstracta puede ampliar a otra clase abstracta.
- e. Una clase abstracta puede ampliar a una clase no abstracta.
- f. Una interfaz es una clase no abstracta.
- g. Una interfaz puede declarar datos de instancia.
- h. Todo método de una interfaz debe ser público.
- i. Todos los métodos de una interfaz deben ser abstractos.
- j. Una interfaz no puede tener métodos en absoluto.
- k. Una interfaz puede ampliar a otra interfaz.
- l. Una interfaz no puede declarar constructores.
- m. Una clase puede ampliar a otras varias clases.
- n. Una clase puede implementar más de una interfaz.
- o. Una clase puede ampliar una clase e implementar una interfaz.
- p. Una interfaz no puede implementar algunos de sus métodos.
- q. Los métodos de una interfaz pueden proporcionar una lista `throws`.
- r. Todos los métodos de una interfaz deben tener un tipo de retorno `void`.
- s. `Throwable` es una interfaz.
- t. `Object` es una clase abstracta.
- u. `Comparable` es una interfaz.
- v. `Comparator` es un ejemplo de interfaz utilizada para objetos función.

- 4.20** Examine cuidadosamente la documentación en línea para los constructores de `Scanner`. Indique cuáles de los siguientes serían parámetros aceptables para un `Scanner`: `File`, `FileInputStream`, `FileReader`, `String`.
- 4.21** Una clase local puede acceder a variables locales declaradas en ese método (antes que la clase). Demuestre que si se permite esto, es posible para una instancia de la clase local acceder al valor de la variable local, incluso después de que el método haya terminado. (Por esta razón, el compilador insistirá en que estas variables se marquen como `final`.)

## EN LA PRÁCTICA

- 4.22** Añada una clase `Square` a la jerarquía `Shape` y haga que implemente `Comparable` <`Square`>.
- 4.23** Un `SingleBuffer` soporta los métodos `get` y `put`: el `SingleBuffer` almacena un único elemento y una indicación de si el `SingleBuffer` está lógicamente vacío. Un método `put` solo puede aplicarse a un buffer vacío e inserta un elemento en el buffer. Un `get` solo se puede aplicar a un buffer no vacío, en cuyo caso borra y devuelve el contenido del buffer. Escriba una clase genérica para implementar `SingleBuffer`. Defina una excepción para implementar los errores.
- 4.24** Modifique la clase `BigRational` del Capítulo 3 para implementar `Comparable` <`BigRational`>.

- 4.25** Modifique la clase `Person` para que pueda utilizar `findMax` con el fin de averiguar cuál es la última persona desde el punto de vista alfabético.
- 4.26** Escriba un método genérico `min`, que acepte una matriz y devuelva el elemento más pequeño. A continuación, utilice el método con el tipo `String`.
- 4.27** Escriba un método genérico `sort`, que acepte una matriz y la reordene en orden no decreciente. Compruebe su método tanto con `String` como con `BigInteger`.
- 4.28** Escriba una rutina genérica `copy` que mueva elementos de una matriz a otra matriz compatible de tamaño idéntico.
- 4.29** Escriba un método genérico `max2`, que acepte una matriz y devuelva otra matriz de longitud dos que represente los dos elementos mayores de la matriz de entrada. La matriz de entrada no debe ser modificada. Después utilice el método que haya escrito con el tipo `String`.
- 4.30** Escriba métodos genéricos `min` y `max`, cada uno de los cuales acepte dos parámetros y devuelva el valor más pequeño y más grande, respectivamente. Después utilice estos métodos con el tipo `String`.
- 4.31** Añada una clase `Triangle` a la jerarquía `Shape`. Hágala `Stretchable`, pero haga que `stretch` genere una excepción si la llamada a `stretch` fuera a dar como resultado unas dimensiones que violaran las relaciones que necesariamente deben cumplir los lados de un triángulo.
- 4.32** Modifique `MemoryCell` para implementar `Comparable<MemoryCell>`.
- 4.33** En el ejemplo de `Shape`, modifique los constructores de la jerarquía para generar una excepción `IllegalArgumentException` cuando los parámetros sean negativos.
- 4.34** Añada una clase `Ellipse` a la jerarquía `Shape`, pero no la haga `Stretchable`.
- 4.35** Modifique la clase `Circle` para implementar `Comparable<Circle>`.
- 4.36** Modifique la clase `Polynomial` del Ejercicio 3.31 para implementar `Comparable<Polynomial>`. La comparación debe basarse en los grados de los polinomios.
- 4.37** Revise el método `stretchAll` para aceptar un `ArrayList` en lugar de una matriz. Utilice comodines para asegurarse de que tanto `ArrayList<Stretchable>` como `ArrayList<Rectangle>` sean parámetros aceptables.
- 4.38** El método `transform` toma como parámetros dos matrices de tamaño idéntico: `input` y `output`, y un tercer parámetro que representa una función que hay que aplicar a la matriz de entrada.

Por ejemplo, en el siguiente fragmento de código:

```
double [ ] input = { 1.0, -3.0, 5.0 };
double [ ] output1 = new double [ 3 ];
double [ ] output2 = new double [ 3 ];
double [ ] output3 = new double [ 4 ];

transform( input, output1, new ComputeSquare( ) );
transform( input, output2, new ComputeAbsoluteValue( ) );
transform( input, output3, new ComputeSquare( ) );
```

El resultado pretendido es que `output1` contenga 1.0, 9.0, 25.0, `output2` contenga 1.0, 3.0, 5.0, y la tercera llamada a `transform` genere una excepción `IllegalArgumentException` porque las matrices tienen tamaños diferentes. Implemente los siguientes componentes:

- Una interfaz que se utilizará para especificar el tercer parámetro de `transform`.
- El método `transform` (que es un método estático). Recuerde que deberá generar una excepción si las matrices de entrada y de salida no tienen idéntico tamaño.
- Las clases `ComputeSquare` y `ComputeAbsoluteValue`.

- 4.39** El método `contains` admite una matriz de enteros y devuelve `true` si existe algún elemento de la matriz que satisface una condición especificada. Por ejemplo, en el siguiente fragmento de código:

```
int [ ] input = { 100, 37, 49 };

boolean result1 = contains( input, new Prime( ) );
boolean result2 = contains( input, new PerfectSquare( ) );
boolean result3 = contains( input, new Negative( ) );
```

El resultado pretendido es que `result1` sea `true` porque 37 es un número primo, `result2` sea `true` porque tanto 100 como 49 son cuadrados perfectos y `result3` sea `false` porque no hay números negativos en la matriz. Implemente los siguientes componentes:

- Una interfaz que se utilice para especificar un segundo parámetro de `contains`.
- El método `contains` (que es un método estático).
- Las clases `Negative`, `Prime` y `PerfectSquare`.

- 4.40** Un `SortedArrayList` almacena una colección. Es similar a `ArrayList`, salvo que `add` colocará el elemento en su lugar correspondiente, según el orden correcto, en lugar de colocarlo al final (sin embargo, en este punto le resultará difícil utilizar la herencia a la hora de hacer el ejercicio). Implemente una clase separada `SortedArrayList` que soporte `add`, `get`, `remove` y `size`.

- 4.41** Este ejercicio nos pide escribir un método genérico `countMatches`. El método tendrá dos parámetros. El primer parámetro es una matriz de tipo `int`. El segundo parámetro es un objeto función que devuelve un valor booleano.

- Proporcione una declaración para una interfaz que exprese el objeto función necesario.
- `countMatches` devuelve el número de elementos de la matriz para los que el objeto función devuelve `true`. Implemente `countMatches`.
- Pruebe `countMatches` escribiendo un objeto función, `EqualsZero`, que implemente su interfaz para aceptar un parámetro y devolver `true` si el parámetro es igual a cero. Utilice un objeto función `EqualsZero` para probar `countMatches`.

- 4.42** Aunque los objetos función que hemos examinado no almacenan ningún dato, esto no es ninguna exigencia. Reutilice la interfaz del Ejercicio 4.41(a).

- a. Escriba un objeto función `EqualsK`. `EqualsK` contiene un miembro de datos (`k`). `EqualsK` se construye con un único parámetro (el valor predeterminado es cero) que se utiliza para inicializar `k`. Su método devuelve `true` si el parámetro es igual a `k`.
  - b. Utilice `EqualsK` para probar `countMatches` en el Ejercicio 4.41 (c).
- 4.43** Reescriba el Ejercicio 4.39 utilizando genéricos, para permitir que la matriz de entrada sea de un tipo arbitrario.
- 4.44** Proporcione un objeto función que se pueda pasar a `findMax` y que ordene objetos `String` utilizando `compareToIgnoreCase` en lugar de `compareTo`.
- 4.45** Reescriba el Ejercicio 4.38 utilizando genéricos, para permitir que la matriz de entrada y las matrices de salida sean de tipo arbitrario (no necesariamente el mismo).

## PROYECTOS DE PROGRAMACIÓN

- 4.46** Escriba una clase abstracta para `Date` y su clase derivada `GregorianCalendar`.
- 4.47** Reescriba la jerarquía `Shape` para almacenar el área como un miembro de datos y hacer que la calcule el constructor `Shape`. Los constructores de las clases derivadas deben calcular un área y pasar el resultado al método `super`. Haga que `area` sea un método final que devuelva solo el valor de este miembro de datos.
- 4.48** Implemente un programa `gzip` y `gunzip` que lleva a cabo la compresión y descompresión de archivos.
- 4.49** Un libro está compuesto por un autor, un título y un ISBN (ninguno de los cuales puede cambiar una vez que el libro se ha creado).

Un libro de una biblioteca es un libro que también dispone de una fecha de entrega y el nombre del actual depositario del libro, que puede ser un objeto `String` que representa a una persona que ha tomado prestado el libro o `null` si el libro no está actualmente prestado. Tanto la fecha de entrega como el depositario del libro pueden cambiar a lo largo del tiempo.

Una biblioteca contiene libros de la misma y soporta las siguientes operaciones:

1. Añadir un libro a la biblioteca.
  2. Tomar prestado un libro especificando su número de ISBN, el nuevo depositario y la fecha de entrega.
  3. Determinar el depositario actual de un libro dado su código ISBN.
- a. Escriba dos interfaces: `Book` y `LibraryBook`, que abstraigan la funcionalidad descrita anteriormente.
  - b. Escriba una clase `Library` que represente a la biblioteca y que incluya los tres métodos especificados. Al implementar la clase `Library`, debe mantener los libros de la biblioteca en un `ArrayList`. Puede asumir que nunca va a haber solicitudes para añadir libros duplicados.
- 4.50** Añada el concepto de posición a la jerarquía `Shape` incluyendo las coordenadas como miembros de datos. Después añada un método `distance`.

**4.51** Considere las cinco clases siguientes: `Bank`, `Account`, `NonInterestCheckingAccount`, `InterestCheckingAccount` y `PlatinumCheckingAccount`, así como la interfaz denominada `InterestBearingAccount` que interactúan de la manera siguiente:

- Un banco `Bank` almacena un `ArrayList` que puede contener cuentas de todos los tipos, incluyendo cuentas de ahorro y cuentas corrientes, algunas de la cuales dan intereses y otras no. `Bank` contiene un método llamado `totalAssets` que devuelve la suma de los saldos de todas las cuentas. También contiene un método denominado `addInterest` que invoca el método `addInterest` de todas las cuentas del banco que generan intereses.
- `Account` es una clase abstracta que representa a una cuenta. Cada cuenta almacena el nombre del titular de la misma, un número de cuenta (que se asigna secuencialmente de manera automática) y el saldo actual, junto con un constructor apropiado para inicializar estos miembros de datos, así como métodos para acceder al saldo actual, sumar dinero al saldo actual y restar dinero del saldo actual. *Nota:* todos estos métodos se implementan en la clase `Account`, por lo que aunque `Account` sea abstracta, ninguno de los métodos que vamos a implementar en esta clase serán abstractos.
- La interfaz `InterestBearingAccount`, utilizada para cuentas que generan intereses, declara un único método `addInterest` (sin parámetros, tipo de retorno `void`), que incrementa el saldo de acuerdo con la tasa de interés apropiada para esa cuenta.
- Una `InterestCheckingAccount` es una `Account` que también es una `InterestBearingAccount`. Al invocar `addInterest` se incrementa el saldo en un 5,5%.
- Una `PlatinumCheckingAccount` es una `InterestCheckingAccount`. Al invocar `addInterest` se incrementa el saldo con una tasa tres veces superior al de `InterestCheckingAccount` (sea cual sea esa tasa).
- Una `NonInterestCheckingAccount` es una `Account` pero no es una `InterestBearingAccount`. No tiene funcionalidad más allá de la que tiene la clase `Account` básica.

Para esta cuestión, haga lo siguiente. No tiene por qué proporcionar ninguna funcionalidad más allá de las especificaciones mencionadas:

- a. Cinco de las seis clases anteriores forman una jerarquía de herencia. Para esas cinco clases, dibuje la jerarquía.
- b. Implemente `Account`.
- c. Implemente `NonInterestCheckingAccount`.
- d. Escriba la interfaz `InterestBearingAccount`.
- e. Implemente `Bank`.
- f. Implemente `InterestCheckingAccount`.
- g. Implemente `PlatinumCheckingAccount`.

- 4.52** Implemente una jerarquía de contribuyentes que esté compuesta por una interfaz `TaxPayer` que representa un contribuyente y las clases `SinglePayer` para representar a un contribuyente soltero, y `MarriedPayer`, para representar a un contribuyente casado. Estas dos clases implementan la interfaz anterior.
- 4.53** Se utiliza un conjunto de clases para gestionar los distintos tipos de entradas para un cine. Todas las entradas tienen un número de serie único que se asigna cuando se construye la entrada, y tienen también un precio. Hay muchos tipos de entradas.
- Diseñe una jerarquía de clases que abarque las tres clases anteriores.
  - Implemente la clase abstracta `Ticket` para representar las entradas. Esta clase debe almacenar un número de serie como dato privado. Proporcione un método abstracto apropiado que permita obtener el precio de la entrada. Proporcione un método que devuelva el número de serie y una implementación de `toString` que imprima el número de serie y la información de precio. La clase `Ticket` debe suministrar un constructor para inicializar el número de serie. Para ello, utilice la siguiente estrategia: mantenga un `ArrayList<Integer>` estático que represente los números de serie previamente asignados. Genere repetidamente un nuevo número de serie utilizando un generador de números aleatorios, hasta obtener un número de serie que no haya sido previamente asignado.
  - Implemente la clase `FixedPriceTicket` para representar entradas de precio fijo. El constructor acepta un precio como parámetro. La clase es abstracta, pero debe implementar el método que devuelva la información de precio.
  - Implemente la clase `WalkupTicket` para representar las entradas compradas en taquilla y la clase `ComplementaryTicket` para representar entradas gratuitas.
  - Implemente la clase `AdvanceTicket` para representar entradas de venta anticipada. Proporcione un constructor que admita un parámetro que indique el número de días de adelanto con el que se está haciendo dicha venta anticipada. Recuerde que el número de días de adelanto afecta al precio de la entrada.
  - Implemente la clase `StudentAdvanceTicket` para representar entradas para estudiantes de venta anticipada. Proporcione un constructor que admita un parámetro que indique el número de días de adelanto con el que se está haciendo dicha venta. El método `toString` debe incluir una indicación de que se trata de una entrada para estudiante. Esta entrada cuesta la mitad de una entrada `AdvanceTicket`. Si el esquema de precios de `AdvanceTicket` cambia, el precio de una entrada `StudentAdvanceTicket` debe ser calculado correctamente sin tener que modificar la clase `StudentAdvanceTicket`.
  - Escriba una clase `TicketOrder` que represente un pedido y almacene una colección de `Ticket`. `TicketOrder` debe proporcionar los métodos `add`, `toString` y `totalPrice`. Proporcione un programa de prueba que cree un objeto `TicketOrder` y luego invoque al método `add` con todos los tipos de entradas. Imprima el pedido, incluyendo el precio total.

Tipo de entrada	Descripción	Ejemplo de salida de <code>toString</code>
<code>Ticket</code>	Es una clase abstracta que representa todas las entradas.	
<code>FixedPriceTicket</code>	Es una clase abstracta que representa las entradas que siempre tienen el mismo precio. El constructor acepta el precio como parámetro.	
<code>ComplimentaryTicket</code>	Estas entradas son gratuitas (y por tanto de tipo <code>FixedPrice</code> ).	SN: 273, \$0
<code>WalkupTicket</code>	Estas entradas se compran el mismo día de la proyección por 50\$ (por tanto son de tipo <code>FixedPrice</code> ).	SN: 314, \$50
<code>AdvanceTicket</code>	Las entradas compradas con diez o más días de antelación cuestan 30\$. Las entradas compradas con menos de diez días de adelanto cuestan 40\$.	SN: 612, \$40
<code>StudentAdvanceTicket</code>	Estas son entradas de tipo <code>AdvanceTicket</code> que cuestan la mitad de lo que normalmente costaría una entrada <code>AdvanceTicket</code> .	SN: 59, \$15 (estudiante)



## Referencias

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

1. G. Booch, *Object-Oriented Design and Analysis with Applications* (Segunda edición), Benjamin Cummings, Redwood City, CA, 1994.
2. T. Budd, *Understanding Object-Oriented Programming With Java*, Addison-Wesley, Boston, MA, 2001.
3. D. de Champeaux, D. Lea y P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, MA, 1993.
4. I. Jacobson, M. Christerson, P. Jonsson y G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach* (cuarta impresión revisada), Addison-Wesley, Reading, MA, 1992.
5. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, 1988.



parte  
dos

# Algoritmos y bloques fundamentales



Capítulo 5 Análisis de algoritmos

Capítulo 6 La API de Colecciones

Capítulo 7 Recursión

Capítulo 8 Algoritmos de ordenación

Capítulo 9 Aleatorización



# Capítulo 5

## Análisis de algoritmos

En la primera parte hemos examinado cómo nos puede ayudar la programación orientada a objetos durante el diseño y la implementación de sistemas de gran tamaño. No nos hemos fijado en las cuestiones de rendimiento. Generalmente, si utilizamos una computadora es porque necesitamos procesar una gran cantidad de datos. Y cuando ejecutamos un programa con una gran cantidad de datos, tenemos que estar seguros de que el programa termine su tarea en un tiempo razonable. Aunque el tiempo total de ejecución depende en alguna medida del lenguaje de programación que empleemos y en menor medida de la metodología utilizada (como por ejemplo programación procedural en lugar de orientada a objetos), a menudo esos factores son constantes del diseño que no se pueden modificar. Aún así, de lo que más depende el tiempo de ejecución es de la elección de los algoritmos.

Un *algoritmo* es un conjunto claramente especificado de instrucciones que la computadora seguirá para resolver un problema. Una vez que se proporciona un algoritmo para un problema y se verifica que es correcto, el siguiente paso consiste en determinar la cantidad de recursos, como por ejemplo tiempo y espacio de memoria, que el algoritmo requerirá. Este paso se denomina *análisis de algoritmos*. Un algoritmo que requiera varios cientos de gigabytes de memoria principal no será útil para la mayoría de las máquinas actuales, incluso aunque sea completamente correcto.

En este capítulo, vamos a abordar las siguientes cuestiones:

- Cómo estimar el tiempo requerido por un algoritmo.
- Cómo utilizar técnicas que reduzcan drásticamente el tiempo de ejecución de un algoritmo.
- Cómo emplear un marco de análisis matemático que describa con el mayor rigor posible el tiempo de ejecución de un algoritmo.
- Cómo describir una rutina simple de *búsqueda binaria*.

### 5.1 ¿Qué es el análisis de algoritmos?

La cantidad de tiempo que cualquier algoritmo tarda en ejecutarse depende casi siempre de la cantidad de entrada que deba procesar. Cabe esperar, por ejemplo, que ordenar 10.000 elementos requiera más tiempo que ordenar 10 elementos. El tiempo de ejecución de un algoritmo es, por tanto, una función del tamaño de la entrada. El valor exacto de la función dependerá de muchos

Una mayor cantidad de datos implica que el programa necesita más tiempo de ejecución .

factores, como por ejemplo de la velocidad de la máquina utilizada, de la calidad del compilador y, en algunos casos, de la calidad del programa. Para un determinado programa en una determinada computadora, podemos dibujar en una gráfica la función que expresa el tiempo de ejecución. La Figura 5.1 ilustra una de esas gráficas para cuatro programas distintos. Las curvas representan cuatro funciones comúnmente encontradas 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 varían entre 0 y 10 microsegundos. Un rápido vistazo a la Figura 5.1, y a su compañera, la Figura 5.2, sugiere que las curvas lineal,  $O(N \log N)$ , cuadrática y cúbica representan distintos tiempos de ejecución, en orden de preferencia decreciente.

De las funciones que comúnmente nos podemos encontrar en el análisis de algoritmos, las lineales representan los algoritmos más eficientes.

Un ejemplo sería el problema de descargar un archivo de Internet. Suponga que hay un retardo inicial de 2 segundos (para establecer una conexión), después de lo cual la descarga se produce una velocidad de 160 K/seg. Entonces, si el archivo tiene  $N$  kilobytes, el tiempo para la descarga está dado por la fórmula  $T(N) = N/160 + 2$ . Esta es una *función lineal*. Descargar un archivo de 8.000K requiere aproximadamente 52 segundos, mientras que descargar un archivo dos veces mayor (16.000K) requiere unos

102 segundos, lo que aproximadamente es el doble de tiempo. Esta propiedad en la que el tiempo es, en esencia, directamente proporcional al tamaño de la entrada, es la característica fundamental de un *algoritmo lineal*, que es el tipo de algoritmo más eficiente. Por contraste, como muestran estas dos primeras gráficas, algunos de los algoritmos no lineales requieren tiempos de ejecución muy grandes. Por ejemplo, el algoritmo lineal es mucho más eficiente que el algoritmo cúbico.

En este capítulo, vamos a considerar varias cuestiones importantes:

- ¿Es importante disponer siempre de la curva más eficiente?
- ¿Cómo podemos comparar unas curvas con otras?

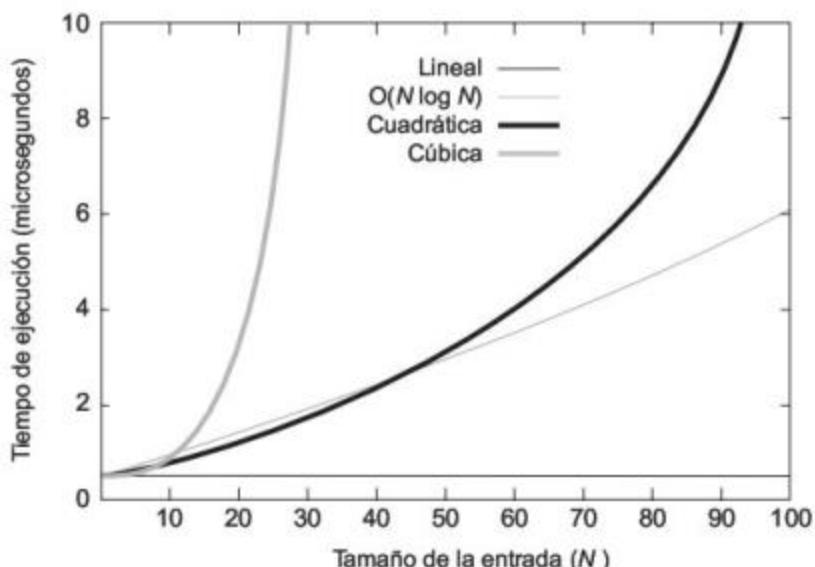


Figura 5.1 Tiempos de ejecución para entradas de pequeño tamaño.

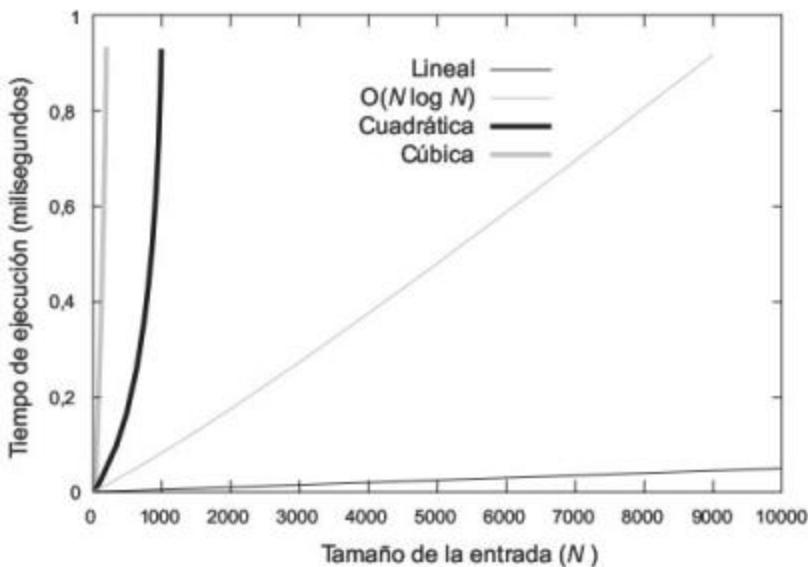


Figura 5.2 Tiempos de ejecución para entradas de tamaño moderado.

- ¿Cómo podemos determinar a qué curva corresponde un determinado algoritmo?
- ¿Cómo podemos diseñar algoritmos que eviten mostrar alguna de las curvas menos eficientes?

Una *función cúbica* es una función cuyo término dominante es una cierta constante multiplicada por  $N^3$ . Por ejemplo,  $10N^3 + N^2 + 40N + 80$  sería una función cúbica. De forma similar, una función cuadrática tiene un término dominante que es igual a una constante multiplicada por  $N^2$ , mientras que una función lineal tiene un término dominante que es igual a una constante multiplicada por  $N$ . La expresión  $O(N \log N)$  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 la típica base 2) es solo 20. El logaritmo crece más lentamente que una raíz cuadrada o cúbica (o cualquier otra raíz). Hablaremos del logaritmo con más profundidad en la Sección 5.5.

Si tenemos dos funciones, cualquiera de ellas puede ser más pequeña que la otra en un punto determinado, así que afirmar que, por ejemplo  $F(N) < G(N)$  no tiene sentido. En lugar de ello, lo que hacemos es medir la tasa de crecimiento de las funciones. Esto está justificado por tres razones distintas. En primer lugar, para las funciones cúbicas, como la que se muestra en la Figura 5.2, cuando  $N$  es 1.000 el valor de la función cúbica está casi completamente determinado por el término cúbico. En la función  $10N^3 + N^2 + 40N + 80$ , para  $N = 1.000$ , el valor de la función es 10.001.040.080, del cual 10.000.000.000 se debe al término  $10N^3$ . Si utilizáramos únicamente el término cúbico para estimar el valor de la función completa, el error que se produciría sería de aproximadamente del 0,01 por ciento. Para un valor de  $N$  suficientemente grande, el valor de una función está principalmente determinado por su término dominante (el significado de *suficientemente grande* varía según la función).

La segunda razón por la que medimos la tasa de crecimiento de las funciones es que el valor exacto de la constante que multiplica al término dominante no tienen ningún significado si tratamos

La tasa de crecimiento de una función tiene su máxima importancia cuando  $N$  es suficientemente grande.

de comparar unas máquinas con otras (aunque los valores relativos de esa constante para funciones que crezcan de manera idéntica sí que pueden ser significativos). Por ejemplo, la calidad del compilador podría tener una gran influencia sobre el valor de esa constante. La tercera razón es que los valores pequeños de  $N$  no suelen ser importantes. Para  $N = 20$ , la Figura 5.1 muestra que todos los algoritmos terminan en los 5  $\mu\text{s}$ . La diferencia entre el algoritmo mejor y el peor es inferior al tiempo que tardamos en parpadear.

La notación  $O$  mayúscula se utiliza para capturar el término más dominante de una función.

Utilizamos la notación  $O$  mayúscula o notación de Landau para capturar el término más dominante de una función y para representar la tasa de crecimiento. Por ejemplo, el tiempo de ejecución de un algoritmo cuadrático se especifica como  $O(N^2)$  (que se lee “de orden ene cuadrado”). La notación  $O$  mayúscula también nos permite establecer un orden relativo entre funciones, comparando los términos dominantes. Veremos la notación  $O$  mayúscula de manera más formal en la Sección 5.4.

Para valores pequeños de  $N$  (por ejemplo, inferiores a 40), la Figura 5.1 muestra que una curva puede ser inicialmente mejor que otra y que sin embargo eso no se cumple para valores de  $N$  más grandes. Por ejemplo, la curva cuadrática es inicialmente mejor que la curva  $O(N \log N)$ , pero a medida que  $N$  se hace suficientemente grande, el algoritmo cuadrático pierde su ventaja. Para pequeñas cantidades de entrada, hacer comparaciones entre funciones resulta difícil, porque las constantes multiplicativas tienen un valor muy significativo. La función  $N + 2.500$  es mayor que  $N^2$  cuando  $N$  es menor que 50. Pero a medida que crece  $N$ , la función lineal siempre termina por tener un valor inferior que la función cuadrática. Además, lo que es aún más importante, para pequeños tamaños de entrada los tiempos de ejecución suelen ser irrelevantes, así que no necesitamos preocuparnos por ellos. Por ejemplo, la Figura 5.1 muestra que cuando  $N$  es inferior a 25, los cuatro algoritmos se ejecutan en menos de 10  $\mu\text{s}$ . En consecuencia, cuando los tamaños de entrada son muy pequeños, una buena regla práctica consiste en utilizar el algoritmo que sea más simple.

La Figura 5.2 demuestra claramente las diferencias entre las distintas curvas para tamaños de entrada grandes. Un algoritmo lineal resuelve el problema de tamaño 10.000 en una pequeña fracción de segundo. El algoritmo  $O(N \log N)$  utiliza un tiempo aproximadamente 10 veces mayor. Observe que las diferencias de tiempo reales dependerán de las constantes implicadas, y pueden por tanto ser mayores o menores de las indicadas. Dependiendo de estas constantes, un algoritmo  $O(N \log N)$  podría ser más rápido que un algoritmo lineal para tamaños de entrada relativamente grandes. Sin embargo, para algoritmos de igual complejidad, los algoritmos lineales tienden a tener un mejor rendimiento que los algoritmos  $O(N \log N)$ .

Los algoritmos cuadráticos no resultan prácticos para tamaños de entrada superiores a unos pocos miles.

Sin embargo, esta relación no es cierta para los algoritmos cuadráticos y cúbicos. Los algoritmos cuadráticos no suelen ser nunca prácticos cuando el tamaño de entrada supera unos pocos miles, mientras que los algoritmos cúbicos dejan de ser prácticos para tamaños de entrada de solo unos pocos centenares. Por ejemplo, no es práctico emplear un algoritmo de ordenación sencillo para un millón elementos, porque los algoritmos de ordenación más simples (como la ordenación por selección o el algoritmo de la burbuja) son algoritmos cuadráticos. Los algoritmos de ordenación presentados en el Capítulo 8 se ejecutan en un tiempo *subcuadrático*; es decir, mejor que  $O(N^2)$ , lo cual hace que sea práctico ordenar matrices de gran tamaño.

La característica más llamativa de estas curvas es que los algoritmos cuadráticos y cúbicos no son competitivos en comparación con los otros para entradas razonablemente grandes. Podemos

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 ordenadas de menor a mayor tasa de crecimiento.

codificar el algoritmo cuadrático utilizando un lenguaje máquina altamente eficiente y no molestarnos apenas en la codificación del algoritmo lineal, y a pesar de todo el algoritmo cuadrático saldrá perdiendo. Ni siquiera los más inteligentes trucos de programación pueden hacer que se ejecute rápidamente un algoritmo no eficiente. Por tanto, antes de perder el tiempo tratando de optimizar el código, lo primero que hay que hacer es optimizar el algoritmo. La Figura 5.3 muestra las funciones que describen comúnmente los tiempos de ejecución de los algoritmos por orden de menor a mayor tasa de crecimiento.

Los algoritmos cúbicos no resultan prácticos para tamaños de entrada de solo unos pocos centenares.

## 5.2 Ejemplos de tiempos de ejecución de diversos algoritmos

En esta sección vamos a examinar tres problemas. También esbozaremos algunas posibles soluciones y determinaremos el tiempo de ejecución que exhibirán los algoritmos, sin proporcionar programas detallados. El objetivo de esta sección es proporcionar al lector una cierta intuición acerca del análisis de algoritmos. En la Sección 5.3 daremos más detalles sobre el proceso y en la Sección 5.4 abordaremos formalmente un problema de análisis de algoritmos.

Vamos a examinar los siguientes problemas en esta sección:

### Elemento mínimo de una matriz

Dada una matriz de  $N$  elementos, determinar el menor de ellos.

### Puntos más próximos en el plano

Dados  $N$  puntos en un plano (es decir, en un sistema de coordenadas  $x$ - $y$ ), encontrar la pareja de puntos más próximos.

### Puntos colineales en el plano

Dados  $N$  puntos en un plano (es decir, en un sistema de coordenadas  $x$ - $y$ ), determinar si cualesquiera tres puntos forman una línea recta.

El problema del elemento mínimo es fundamental en informática. Se puede resolver de la forma siguiente:

1. Mantener una variable  $\min$  que almacene el elemento mínimo.
2. Inicializar  $\min$  con el valor del primer elemento.
3. Hacer un barrido secuencial de la matriz y actualizar  $\min$  de la forma apropiada.

El tiempo de ejecución de este algoritmo será  $O(N)$  o lineal, porque lo que hacemos es repetir una cantidad fija de trabajo para cada elemento de la matriz. Un algoritmo lineal es lo mejor que podemos esperar. Si en este caso se consigue es porque tenemos que examinar cada elemento de la matriz, lo cual es un proceso que requiere un tiempo lineal.

El problema de los puntos más próximos es un problema fundamental en gráficos, que se puede resolver de la forma siguiente:

1. Calcular la distancia entre cada pareja de puntos.
2. Quedarse con la distancia mínima.

Sin embargo, este cálculo es muy costoso, porque hay  $N(N - 1)/2$  pares de puntos.<sup>1</sup> Por tanto, hay aproximadamente  $N^2$  pares de puntos. Examinar todos estos pares y hallar la distancia mínima entre ellos requiere un tiempo cuadrático. Existe un algoritmo mejor que se ejecuta en un tiempo  $O(N \log N)$  y funciona por el procedimiento de evitar tener que calcular todas las distancias. También existe un algoritmo que requiere un tiempo  $O(N)$ . Estos dos algoritmos utilizan una serie de sutiles observaciones para proporcionar resultados más rápidamente, y caen fuera del alcance de este texto.

El problema de los puntos colineales es importante para muchos algoritmos gráficos. La razón es que la existencia de puntos colineales introduce un caso degenerado que requiere un tratamiento especial. El problema se puede resolver directamente enumerando todos los grupos de tres puntos. Esta solución es aún más cara, desde el punto de vista computacional, que la del problema de los puntos más próximos, porque el número de grupos de tres puntos distintos es  $N(N - 1)(N - 2)/6$  (utilizando un razonamiento similar al empleado en el problema de los puntos más próximos). Este resultado nos dice que la solución directa se conseguiría en un algoritmo cúbico. También existe una estrategia más inteligente (que también queda fuera del alcance de este texto) que permite resolver el problema en un tiempo cuadrático (y actualmente se está investigando activamente de manera continua para conseguir mejoras adicionales).

En la Sección 5.3 examinaremos un problema que ilustra las diferencias entre los algoritmos lineales, cuadráticos y cúbicos. También mostraremos cómo se compara el rendimiento de estos algoritmos con una predicción matemática. Finalmente, después de explicar las ideas básicas, examinaremos de manera más formal la notación *O mayúscula*.

---

<sup>1</sup> Cada uno de los  $N$  puntos puede emparejarse con  $N - 1$  puntos, lo que nos da un total de  $N(N - 1)$  parejas. Sin embargo, este emparejamiento hace que se cuenten dos veces las parejas  $A$  y  $B$ , así que es preciso dividir entre 2.

## 5.3 El problema de la suma máxima de una subsecuencia contigua

En esta sección vamos a considerar el siguiente problema:

### Problema de la suma máxima de una subsecuencia contigua

Dada una serie de enteros (posiblemente negativos)  $A_1, A_2, \dots, A_N$ , encontrar el valor máximo de  $\sum_{k=1}^j A_k$  e identificar la secuencia correspondiente. La suma máxima de una secuencia contigua es cero si todos los enteros son negativos.

Por ejemplo, si la entrada es  $\{-2, 11, -4, 13, -5, 2\}$ , entonces la respuesta es 20, que representa la subsecuencia contigua que abarca los elementos 2 a 4 (mostrados en negrita). Como segundo ejemplo, para la entrada  $\{1, -3, 4, -2, -1, 6\}$ , la respuesta es 7 para la subsecuencia que abarca los últimos cuatro elementos.

En Java, las matrices comienzan con cero, por lo que un programa Java representaría la entrada como una secuencia  $A_0$  a  $A_{N-1}$ . Esto no es más que un detalle de programación y no forma parte del diseño de un algoritmo.

Antes de analizar los algoritmos existentes para resolver este problema, necesitamos comentar algo acerca del caso degenerado en el que todos los enteros de entrada sean negativos. El enunciado del problema nos da una suma máxima de subsecuencia contigua igual a 0 para este caso. Podríamos preguntarnos por qué se hace esto, en lugar de limitarse a devolver el mayor de los enteros negativos (es decir, el que tenga un módulo más pequeño) de la entrada. La razón es que la subsecuencia vacía, compuesta de cero enteros, también es una subsecuencia y su suma es claramente 0. Puesto que la subsecuencia vacía es contigua, siempre habrá una subsecuencia contigua cuya suma sea 0. Este resultado es análogo al caso del conjunto vacío, que siempre se considera un subconjunto de cualquier conjunto. Tenga en cuenta que las soluciones vacías siempre son una posibilidad y que en muchas circunstancias no se trata en absoluto de un caso especial.

El problema de la suma máxima de una subsecuencia contigua es interesante, fundamentalmente, porque existen muchos algoritmos distintos disponibles para resolverlo —y el rendimiento de estos algoritmos varía enormemente. En esta sección vamos a analizar tres de estos algoritmos. El primero es un algoritmo obvio de búsqueda exhaustiva, que resulta muy ineficiente. El segundo, es una mejora del primero, que se consigue realizando una simple observación. El tercero es un algoritmo muy eficiente, aunque no tan obvio. Demostraremos que su tiempo de ejecución es lineal.

En el Capítulo 7 veremos un cuarto algoritmo, que tiene un tiempo de ejecución  $O(N \log N)$ . Dicho algoritmo no es tan eficiente como el algoritmo lineal, pero es mucho más eficiente que los otros dos. También es un ejemplo típico de los algoritmos con tiempos de ejecución  $O(N \log N)$ . Las gráficas mostradas en las Figuras 5.1 y 5.2 son representativas de estos cuatro algoritmos.

Los detalles de programación se toman en consideración después del diseño del algoritmo.

Considere siempre las soluciones vacías.

Hay un montón de algoritmos enormemente diferentes (en términos de eficiencia) que pueden utilizarse para resolver el problema de la suma máxima de una subsecuencia contigua.

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

El algoritmo más simple es una búsqueda exhaustiva directa, o un *algoritmo de fuerza bruta*, como se muestra en la Figura 5.4. Las líneas 9 y 10 controlan un par de bucles que iteran a lo

```

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

```

Figura 5.4 Un algoritmo cúbico de suma máxima de subsecuencia contigua.

Un algoritmo de fuerza bruta es, generalmente, el método menos eficiente, pero el más simple de codificar.

largo de todas las posibles subsecuencias. Para cada posible subsecuencia, el valor de su suma se calcula en las líneas 12 a 15. Si esa suma es la mejor de las encontradas hasta el momento, se actualiza el valor de `maxSum`, que se termina devolviendo en la línea 25. También se actualizan dos valores `int`, `seqStart` y `seqEnd` (que son campos de clase estáticos y que indican el principio y el final de la subsecuencia) cada vez que se encuentra una nueva mejor secuencia.

El algoritmo directo de búsqueda exhaustiva tiene como mérito su extremada simplicidad; cuanto menos complejo sea un algoritmo, más probable será que se programe correctamente. Sin embargo, los algoritmos de búsqueda exhaustiva no suelen ser los más eficientes posibles. En el resto de esta sección vamos a ver que el tiempo de ejecución de este algoritmo es cúbico. Contaremos el número de veces (como función del tamaño de la entrada) que se evalúan las expresiones de la Figura 5.4. Lo único que necesitamos es un resultado en notación O mayúscula, por lo que una vez que hayamos encontrado un término dominante, podemos ignorar los términos de menor orden y las constantes multiplicativas.

El tiempo de ejecución del algoritmo está dominado completamente por el bucle `for` más interno de las líneas 14 y 15. Hay cuatro expresiones que se ejecutan repetidamente:

1. La inicialización  $k = i$
2. La comprobación  $test\ k \leq j$
3. El incremento  $thisSum += a[k]$
4. El ajuste  $k++$

El número de veces que se ejecuta la expresión 3 hace que sea el término dominante en las cuatro expresiones. Observe que cada inicialización va acompañada por al menos una comprobación. Estamos ignorando las constantes, por lo que podemos despreciar el coste de las inicializaciones; las inicializaciones no pueden ser el coste dominante de un algoritmo. Puesto que la comprobación representada por la expresión 2 solo da un resultado falso una vez por cada bucle, el número de comprobaciones con resultado falso realizado por la expresión 2 es exactamente igual al número de inicializaciones. En consecuencia, no es dominante. El número de comprobaciones con resultado verdadero en la expresión 2, el número de incrementos realizados por la expresión 3 y el número de ajustes de la expresión 4 son idénticos. Por tanto, el número de incrementos (es decir, el número de veces que se ejecuta la línea 15) es una medida dominante del trabajo realizado en el bucle más interno.

Se utiliza un análisis matemático para contar el número de veces que se ejecutan ciertas instrucciones.

El número de veces que se ejecuta la línea 15 es exactamente igual al número de tripletas ordenadas  $(i, j, k)$  que satisfacen  $1 \leq i \leq k \leq j \leq N$ .<sup>2</sup> La razón es que el índice  $i$  recorre toda la matriz, mientras que  $j$  va de  $i$  al final de la matriz y  $k$  va de  $i$  a  $j$ . Una estimación rápida y aproximada es que el número de tripletas es algo inferior a  $N \times N \times N$ , o  $N^3$ , porque  $i, j$  y  $k$  pueden asumir cada una de ellas uno de  $N$  valores posibles. La restricción adicional  $i \leq k \leq j$  reduce este número. Un cálculo preciso es algo difícil de obtener y lo realizamos en el Teorema 5.1.

La parte más importante del Teorema 5.1 no es la demostración, sino el resultado. Hay dos formas de evaluar el número de tripletas. Una consiste en evaluar la suma  $\sum_{i=1}^N \sum_{j=i}^N \sum_{k=i}^j 1$ . Podríamos evaluar esta suma de dentro hacia fuera (véase el Ejercicio 5.11). Pero, en lugar de ello, vamos a emplear una alternativa.

### Teorema 5.1

El número de tripletas ordenadas  $(i, j, k)$  que satisfacen  $1 \leq i \leq k \leq j \leq N$  es  $N(N + 1)(N + 2)/6$ .

#### Demostración

Coloque las siguientes  $N + 2$  bolas en una caja:  $N$  bolas numeradas de 1 a  $N$ , una bola roja no numerada y una bola azul no numerada. Extraiga tres bolas de la caja. Si se extrae una bola roja, numérela como la más baja de las bolas numeradas extraídas. Si se extrae una bola azul, numérala como la bola más alta de las bolas numeradas extraídas. Observe que si extraemos tanto una bola roja como una azul, entonces el efecto será tener tres bolas con numeración idéntica. Ordene las tres bolas. Cada una de esas ordenaciones corresponde a una tripleta solución de la ecuación del Teorema 5.1. El número de ordenaciones posibles es el número de formas distintas de extraer tres bolas sin sustitución de  $N + 2$  bolas. Esto es similar al problema de seleccionar tres puntos de un grupo de  $N$  que hemos evaluado en la Sección 5.2, así que se obtiene inmediatamente el resultado indicado.

<sup>2</sup> En Java, los índices van de 0 a  $N - 1$ . Hemos utilizado el equivalente algorítmico 1 a  $N$  para simplificar el análisis.

El resultado del Teorema 5.1 es que el bucle `for` más interno representa un tiempo de ejecución cúbico. El trabajo restante del algoritmo no tiene ninguna influencia, porque se realiza, como mucho, una vez por cada iteración del bucle interno. Dicho de otro modo, el coste de las líneas 17 a 22 es irrelevante, porque que esa parte del código se ejecuta únicamente con la misma frecuencia que la inicialización del bucle `for` interno, en lugar de con la misma frecuencia que el cuerpo repetitivo del bucle `for` interno. En consecuencia, el algoritmo es  $O(N^3)$ .

No necesitamos realizar cálculos precisos para estimar la proporcionalidad  $O$  mayúscula. En la mayoría de los casos podemos utilizar la regla simple consistente en multiplicar los tamaños de todos los bucles. Observe que los bucles consecutivos no se multiplican.

El anterior argumento combinatorio nos permite calcular con precisión el número de iteraciones del bucle interno. Para el cálculo de la proporcionalidad  $O$  mayúscula, esto no es realmente necesario; lo único que necesitamos saber es que el término inicial es igual a una cierta constante multiplicada por  $N^3$ . Examinando el algoritmo, vemos un bucle que tiene potencialmente un tamaño  $N$  dentro de un bucle que tiene potencialmente un tamaño  $N$ , dentro de otro bucle que potencialmente también tiene un tamaño  $N$ . Esta configuración nos dice que el triple bucle tiene potencialmente  $N \times N \times N$  iteraciones. Esta cantidad potencial es solo unas seis veces mayor que el cálculo preciso que hemos hecho de lo que realmente ocurre. Pero las constantes se ignoran de todos modos, así que podemos adoptar la regla general de que, cuando tengamos bucles anidados, debemos 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, la cota superior no representará una sobreestimación excesiva.<sup>3</sup> Por tanto, un programa que tenga tres bucles anidados, cada uno de los cuales se ejecuta secuencialmente a lo largo de una gran parte de una matriz, tiene una gran probabilidad de exhibir un comportamiento  $O(N^3)$ . Observe que tres bucles consecutivos (no anidados) exhiben un comportamiento lineal; es el anidamiento lo que conduce a la explosión combinatoria. En consecuencia, para mejorar el algoritmo tenemos que eliminar un bucle.

general de que, cuando tengamos bucles anidados, debemos 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, la cota superior no representará una sobreestimación excesiva.<sup>3</sup> Por tanto, un programa que tenga tres bucles anidados, cada uno de los cuales se ejecuta secuencialmente a lo largo de una gran parte de una matriz, tiene una gran probabilidad de exhibir un comportamiento  $O(N^3)$ . Observe que tres bucles consecutivos (no anidados) exhiben un comportamiento lineal; es el anidamiento lo que conduce a la explosión combinatoria. En consecuencia, para mejorar el algoritmo tenemos que eliminar un bucle.

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

Al eliminar un bucle anidado de un algoritmo, generalmente reducimos el tiempo de ejecución.

Cuando podemos eliminar un bucle anidado del algoritmo, generalmente reducimos el tiempo de ejecución. ¿Cómo hacemos para eliminar un bucle? Obviamente, no siempre podremos hacerlo. Sin embargo, el algoritmo anterior tiene muchos cálculos innecesarios. La inefficiencia que el algoritmo mejorado corrige es el cálculo indebidamente costoso incluido en el bucle `for` interno de la Figura 5.4. El algoritmo mejorado hace uso del hecho

de que  $\sum_{k=i}^j A_k = A_j + \sum_{k=i}^{j-1} A_k$ . En otras palabras, suponga que acabamos de calcular la suma para la subsecuencia  $i, \dots, j - 1$ . Entonces, calcular la suma para la subsecuencia  $i, \dots, j$  no debería exigir mucho esfuerzo, porque lo único que necesitamos es una suma adicional. Sin embargo, en el algoritmo cúbico, esa información se desperdicia. Si usamos esta observación, obtenemos el algoritmo mejorado mostrado en la Figura 5.5. Vemos que hay dos bucles anidados en lugar de tres, y el tiempo de ejecución es  $O(N^2)$ .

<sup>3</sup> El Ejercicio 5.27 ilustra un caso en el que la multiplicación del tamaño de los bucles nos da una sobreestimación de la proporcionalidad  $O$  mayúscula.

### 5.3.3 Un algoritmo lineal

Para pasar de un algoritmo cuadrático a un algoritmo lineal, necesitamos eliminar otro bucle más. Sin embargo, a diferencia de la reducción ilustrada en las Figuras 5.4 y 5.5, en las que la eliminación de un bucle fue simple, el deshacernos de un bucle más no es tan sencillo. El problema reside en que el algoritmo cuadrático sigue constituyendo una búsqueda exhaustiva, es decir, estamos comprobando todas las posibles subsecuencias. La única diferencia entre los algoritmos cuadrático y cúbico es que el coste de probar cada subsecuencia sucesiva es una constante en lugar de ser lineal. Puesto que existe un número cuadrático de subsecuencias, la única forma de poder conseguir una cota subcuadrática es encontrar una manera inteligente de no tomar en consideración un gran número de subsecuencias, sin llegar a calcular su suma y sin comprobar si esa suma es un nuevo máximo. En esta sección vamos a ver cómo se puede hacer esto.

Si eliminamos otro bucle más, tendremos el algoritmo lineal.

El algoritmo tiene su truco. Utiliza una observación inteligente para saltarse rápidamente un gran número de subsecuencias que nunca podrían llegar a ser la mejor.

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

Figura 5.5 Un algoritmo cuadrático de suma máxima de subsecuencia contigua.

En primer lugar, vamos a eliminar un gran número de subsecuencias posibles, evitando tomarlas en consideración. Claramente, la mejor subsecuencia no puede nunca comenzar con un número negativo, por lo que si  $a[i]$  es negativo podemos saltarnos el bucle interno e incrementar  $i$ . Dicho de forma más general, la mejor subsecuencia no puede empezar nunca con una subsubsecuencia negativa.

Por tanto, sea  $A_{i,j}$  la subsecuencia que abarca los elementos comprendidos entre  $i$  y  $j$ , y sea  $S_{i,j}$  su suma.

### Teorema 5.2

Sea  $A_{i,j}$  cualquier secuencia con  $S_{i,j} < 0$ . Si  $q > j$ , entonces  $A_{i,q}$  no es la máxima subsecuencia contigua.

#### Demostración

La suma de los elementos de  $A$  entre  $i$  y  $q$  es la suma de los elementos de  $A$  entre  $i$  y  $j$  más la suma de los elementos de  $A$  comprendidos entre  $j+1$  y  $q$ . Por tanto, tendremos que  $S_{i,q} = S_{i,j} + S_{j+1,q}$ . Puesto que  $S_{i,j} < 0$ , sabemos que  $S_{i,j} < S_{j+1,q}$ . Por tanto,  $A_{i,q}$  no es una subsecuencia contigua máxima.

En las primeras dos líneas de la Figura 5.6 se muestra una ilustración de las sumas generadas por  $i$ ,  $j$  y  $q$ . El Teorema 5.2 demuestra que podemos evitar examinar varias subsecuencias incluyendo una prueba adicional: si `thisSum` es menor que 0, podemos saltar (con `break`) del bucle interno en la Figura 5.5. Intuitivamente, si la suma de una subsecuencia es negativa, entonces no puede tomar parte de la subsecuencia contigua máxima. La razón es que podemos obtener una subsecuencia contigua mayor no incluyendo esa subsecuencia con suma negativa. Esta observación no es suficiente, por sí misma, para reducir el tiempo de ejecución por debajo de la cota cuadrática. También se cumple otra observación similar: todas las subsecuencias contiguas que flanqueen a la subsecuencia contigua máxima deben tener sumas negativas (o 0), ya que en caso contrario las incluiríamos en la subsecuencia máxima. Esta observación tampoco nos permite reducir el tiempo de ejecución por debajo de la cota cuadrática. Sin embargo, una tercera observación, ilustrada en la Figura 5.7, sí que nos lo permite, y vamos a formalizarla con el Teorema 5.3.

### Teorema 5.3

Para cualquier  $i$ , sea  $A_{i,j}$  la primera secuencia con  $S_{i,j} < 0$ . Entonces, para cualquier  $i \leq p \leq j$  y  $p \leq q$  o bien  $A_{i,q}$  no es una subsecuencia contigua máxima o es igual a una subsecuencia contigua máxima que ya hemos visto.

#### Demostración

Si  $p = i$ , entonces se aplica el Teorema 5.2. En caso contrario, como en el Teorema 5.2, tendremos que  $S_{i,q} = S_{i,p-1} + S_{p,q}$ . Puesto que  $j$  es el menor índice para el que  $S_{i,j} < 0$ , se deduce que  $S_{i,p-1} \geq 0$ . Por tanto,  $S_{p,q} \leq S_{i,j}$ . Si  $q > j$  (mostrado en el lado izquierdo de la Figura 5.7), entonces el Teorema 5.2 implica que  $A_{i,q}$  no es una subsecuencia contigua máxima, por lo que tampoco lo será  $A_{p,q}$ . En caso contrario, como se muestra en la parte derecha de la Figura 5.7, la subsecuencia  $A_{p,q}$  tiene una suma igual, como máximo, a la de la subsecuencia  $A_{i,j}$  que ya habremos visto.

$i$	$j$	$j+1$	$q$
$< 0$	$S_{j+1,q}$		
			$< S_{j+1,q}$

Figura 5.6 Las subsecuencias utilizadas en el Teorema 5.2.

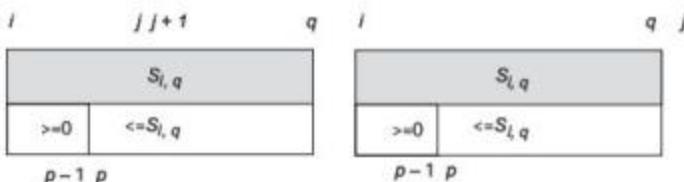


Figura 5.7 Las subsecuencias utilizadas en el Teorema 5.3. La secuencia que va de  $p$  a  $q$  tiene una suma que es, como máximo, igual a la de la subsecuencia que va de  $i$  a  $q$ . En el lado izquierdo, la secuencia que va de  $i$  a  $q$  no es ella misma el máximo (por el Teorema 5.2). En el lado derecho, la secuencia que va de  $i$  a  $q$  ya ha sido vista.

El Teorema 5.3 nos dice que, al detectar una subsecuencia negativa, no solo podemos salir con `break` del bucle interno, sino que también podemos incrementar  $i$  a  $j+1$ . La Figura 5.8 muestra que podemos reescribir el algoritmo utilizando un único bucle. Claramente, el tiempo de ejecución de

```

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

Figura 5.8 Un algoritmo lineal de suma máxima de subsecuencia contigua.

Si detectamos una suma negativa, podemos desplazar  $i$  más allá de  $j$ .

Si un algoritmo es complejo, hace falta una prueba de corrección.

este algoritmo será lineal: en cada paso del bucle, incrementaremos  $j$ , por lo que el bucle iterará como máximo  $N$  veces. La corrección de este algoritmo es mucho menos obvia que para los algoritmos anteriores, lo que es típico de estos procesos de optimización. Es decir, los algoritmos que utilizan la estructura de un problema con el fin de tener mejor rendimiento que una búsqueda exhaustiva requieren, generalmente, algún tipo de prueba de corrección. Hemos demostrado que el algoritmo (aunque no el programa Java resultante) es correcto, utilizando un breve argumento matemático. El propósito es no hacer una exposición completamente matemática, sino más bien dar una ligera pincelada sobre el tipo de técnicas que pueden llegar a ser necesarias a la hora de abordar problemas avanzados.

## 5.4 Reglas generales para el cálculo de cotas $O$ mayúscula

Ahora que tenemos las ideas básicas acerca del análisis de algoritmos, podemos adoptar un enfoque ligeramente más formal. En esta sección vamos a esbozar las reglas generales para la utilización de la notación  $O$  mayúscula. Aunque emplearemos la notación  $O$  mayúscula a todo lo largo de este texto, vamos a definir también otros tipos de notaciones algorítmicas que están relacionadas con la notación  $O$  mayúscula y que ocasionalmente se usarán más adelante en el libro.

**Definición. ( $O$  mayúscula)**  $T(N)$  es  $O(F(N))$  si existen sendas constantes positivas  $c$  y  $N_0$  tales que  $T(N) \leq cF(N)$  cuando  $N \geq N_0$ .

**Definición. (Omega mayúscula)**  $T(N)$  es  $\Omega(F(N))$  si existen sendas constantes positivas  $c$  y  $N_0$  tales que  $T(N) \geq cF(N)$  cuando  $N \geq N_0$ .

**Definición. (Theta mayúscula)**  $T(N)$  es  $\Theta(F(N))$  si y solo si  $T(N)$  es  $O(F(N))$  y  $T(N)$  es  $\Omega(F(N))$ .

**Definición. ( $O$  minúscula)**  $T(N)$  es  $o(F(N))$  si y solo si  $T(N)$  es  $O(F(N))$  y  $T(N)$  no es  $\Theta(F(N))$ .<sup>4</sup>

La primera definición, notación  $O$  mayúscula, indica que existe un punto  $N_0$  tal que para todos los valores de  $N$  más allá de ese punto,  $T(N)$  está acotada por algún múltiplo de  $F(N)$ . Este es el valor de  $N$  suficientemente grande que hemos mencionado anteriormente. Por tanto, si el tiempo de ejecución  $T(N)$  de un algoritmo es  $O(N^2)$ , entonces, ignorando las constantes, estamos garantizando que en algún punto podemos acotar el tiempo de ejecución mediante una función cuadrática. Observe que si el verdadero tiempo de ejecución es lineal, entonces la afirmación de que el tiempo de ejecución es  $O(N^2)$  es técnicamente correcta, porque la desigualdad se cumple. Sin embargo,  $O(N)$  sería la afirmación más precisa.

<sup>4</sup> Nuestra definición de  $O$  minúscula no es precisamente correcta para algunas funciones inusuales, pero es la más simple para expresar los conceptos básicos utilizados a través de todo el texto.

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

La segunda definición,  $T(N) = \Omega(F(N))$ , denominada *Omega mayúscula*, afirma que la tasa de crecimiento de  $T(N)$  es mayor o igual que la de  $F(N)$ . Por ejemplo, podemos decir que cualquier algoritmo que funcione examinando toda posible subsecuencia en el problema de la suma máxima de subsecuencia debe tardar un tiempo  $\Omega(N^2)$ , porque existe un número cuadrático de subsecuencias posibles. Este es un argumento de cota inferior que se utiliza en análisis más avanzado. Posteriormente en el texto, veremos un ejemplo de este argumento y demostraremos que cualquier algoritmo de ordenación de propósito general requiere un tiempo  $\Omega(N \log N)$ .

La tercera definición,  $T(N) = \Theta(F(N))$ , denominada *Theta mayúscula*, dice que la tasa de crecimiento de  $T(N)$  es igual a la tasa de crecimiento de  $F(N)$ . Por ejemplo, el algoritmo de subsecuencia máxima mostrado en la Figura 5.5 se ejecuta en un tiempo  $\Theta(N^2)$ . En otras palabras, el tiempo de ejecución está acotado por una función cuadrática y esta cota no puede mejorarse porque también posee como cota inferior otra función cuadrática. Cuando utilizamos la notación Theta mayúscula, no estamos solo proporcionando una cota superior de un algoritmo, sino también una garantía de que el análisis que conduce a la determinación de esa cota superior es lo mejor (lo más exacto) posible. Sin embargo, a pesar de la precisión adicional ofrecida por Theta mayúscula, se suele utilizar más comúnmente O mayúscula, excepto por parte de los que investigan en el campo de análisis de algoritmos.

La última definición,  $T(N) = o(F(N))$ , denominada *O minúscula*, dice que la tasa de crecimiento de  $T(N)$  es estrictamente inferior a la tasa de crecimiento de  $F(N)$ . Esta función es diferente de O mayúscula porque O mayúscula admite la posibilidad de que las tasas de crecimiento coincidan. Por ejemplo, si el tiempo de ejecución de un algoritmo es  $o(N^2)$ , entonces se garantiza que crece a una tasa más lenta que la cuadrática (es decir, se trata de un *algoritmo subcuadrático*). Por tanto, una cota de  $o(N^2)$  es una cota mejor que  $\Theta(N^2)$ . La Figura 5.9 resume estas cuatro definiciones.

Es conveniente proporcionar un par de notas estilísticas. En primer lugar, no constituye un buen estilo incluir constantes o términos de orden inferior dentro de una expresión O mayúscula. No escriba  $T(N) = O(2N^2)$  ni  $T(N) = O(N^2 + N)$ . En ambos casos, la forma correcta es  $T(N) = O(N^2)$ . En segundo lugar, en cualquier análisis que requiera una respuesta O mayúscula, pueden utilizarse todos los tipos de atajos imaginables. Los términos de menor orden, las constantes multiplicativas y los símbolos relacionales se eliminan siempre.

Ahora que hemos formalizado los aspectos matemáticos, podemos ponerlos en relación con el análisis de algoritmos. La regla más básica es que *el tiempo de ejecución de un bucle es, como máximo, el tiempo de ejecución de las instrucciones contenidas dentro del bucle (incluyendo las comprobaciones) multiplicado por el número de iteraciones*. Como hemos visto anteriormente, la inicialización y prueba de la condición del bucle no suele ser más dominante que las instrucciones que componen el cuerpo del bucle.

La notación O mayúscula es similar a la desigualdad "menor o igual que", cuando hablamos de tasas de crecimiento.

La notación Omega mayúscula es similar a la desigualdad "mayor o igual que", cuando hablamos de tasas de crecimiento.

La notación Theta mayúscula es similar a la igualdad, cuando hablamos de tasas de crecimiento.

La notación O minúscula es similar a la desigualdad "menor que", cuando hablamos de tasas de crecimiento.

Elimine las constantes multiplicativas, los términos de orden inferior y los símbolos relacionales cuando utilice la notación O mayúscula.

Expresión matemática	Tasas relativas de crecimiento
$T(N) = O(F(N))$	Crecimiento de $T(N) \leq$ crecimiento de $F(N)$ .
$T(N) = \Omega(F(N))$	Crecimiento de $T(N) \geq$ crecimiento de $F(N)$ .
$T(N) = \Theta(F(N))$	Crecimiento de $T(N) =$ crecimiento de $F(N)$ .
$T(N) = o(F(N))$	Crecimiento de $T(N) <$ crecimiento de $F(N)$ .

Figura 5.9 Significado de las distintas funciones de crecimiento.

Una cota de caso peor proporciona una garantía para todas las entradas de un cierto tamaño.

El tiempo de ejecución de las instrucciones contenidas dentro de un grupo de bucles anidados es igual al tiempo de ejecución de las instrucciones (incluyendo las comprobaciones del bucle más interno) multiplicado por los tamaños de todos los bucles. El tiempo de ejecución de una secuencia de bucles consecutivos es igual al tiempo de ejecución de bucle dominante. La diferencia de tiempo entre un bucle anidado en el que ambos índices vayan de 1 a  $N$  y dos bucles consecutivos que no estén anidados, pero que se ejecuten a lo largo del mismo rango de índices, es similar a la diferencia de espacio existente entre una matriz bidimensional y dos matrices unidimensionales. El primer caso es cuadrático. El segundo caso es lineal, porque  $N + N$  es  $2N$ , que sigue siendo  $O(N)$ . En ocasiones, esta regla simple puede sobreestimar el tiempo de ejecución, pero en la mayoría de los casos no lo hace. Incluso si lo hace,  $O$  mayúscula no garantiza una respuesta asintótica exacta, simplemente proporciona una cota superior.

Los análisis que hemos realizado hasta ahora implicaban utilizar una *cota de caso peor*, que proporciona una garantía para todas las entradas de un cierto tamaño. Otra forma de análisis es el de *cota de caso promedio*, en el que el tiempo de ejecución se mide como un promedio para todas las posibles entradas de tamaño  $N$ . El promedio puede diferir del caso peor si, por ejemplo, una instrucción condicional que depende de la entrada concreta provoca una salida anticipada de un bucle. Hablaremos con más detalle de las cotas de caso promedio en la Sección 5.8. Por ahora, observe simplemente que el hecho de que un algoritmo tenga una cota de caso peor mejor que otro algoritmo no nos dice nada acerca de cómo se comparan sus cotas de caso promedio respectivas. Sin embargo, en muchos casos, las cotas de caso promedio y de caso peor están estrechamente correlacionadas. Cuando no lo están, esas cotas se analizan por separado.

En una cota de caso promedio, el tiempo de ejecución se mide como promedio para todas las posibles entradas de tamaño  $N$ .

El último aspecto del análisis  $O$  mayúscula que vamos a examinar es el de cómo crece el tiempo de ejecución para cada tipo de curva, como se ilustra en las Figuras 5.1 y 5.2. Lo que queremos es una respuesta más cuantitativa a esta cuestión: si un algoritmo tarda  $T(N)$  en resolver un problema de tamaño  $N$ , ¿cuánto tardará en resolver un problema de mayor tamaño? Por ejemplo, ¿cuánta tarda en resolver un problema cuando la entrada es 10 veces mayor? Las respuestas a estas cuestiones se muestran en la Figura 5.10. Sin embargo, queremos responder a esta cuestión sin ejecutar el programa y esperamos que nuestras respuestas analíticas concuerden con el comportamiento observado.

<b>N</b>	<b>Figura 5.4</b> $O(N^3)$	<b>Figura 5.5</b> $O(N^2)$	<b>Figura 7.20</b> $O(N \log N)$	<b>Figura 5.8</b> $O(N)$
10	0,000001	0,000000	0,000001	0,000000
100	0,000288	0,000019	0,000014	0,000005
1.000	0,223111	0,001630	0,000154	0,000053
10.000	218	0,133064	0,001630	0,000533
100.000	No disponible	13,17	0,017467	0,005571
1.000.000	No disponible	No disponible	0,185363	0,056338

Figura 5.10 Tiempos de ejecución (en segundos) observados para varios algoritmos de suma máxima de subsecuencia contigua.

Comencemos examinando el algoritmo cúbico. Asumimos que el tiempo de ejecución está razonablemente aproximado por  $T(N) = cN^3$ . En consecuencia,  $T(10N) = c(10N)^3$ . Una manipulación matemática nos da

$$T(10N) = 1000cN^3 = 1000T(N)$$

Por tanto, el tiempo de ejecución de un programa cúbico se multiplica por un factor de 1.000 (asumiendo que  $N$  sea suficientemente grande) cuando la cantidad de entrada se multiplica por un factor de 10. Esta relación se ve confirmada aproximadamente por el incremento en tiempo de ejecución entre  $N = 100$  y 1.000 mostrado en la Figura 5.10. Recuerde que no esperamos obtener una respuesta exacta –simplemente una aproximación razonable. También esperaríamos que para  $N = 100.000$ , el tiempo de ejecución se multiplicara de nuevo por 1.000. El resultado sería que un algoritmo cúbico requeriría aproximadamente 60 horas (2 días y medio) de tiempo de computación. En general, si la cantidad de entrada se multiplica por un factor  $f$ , entonces el tiempo de ejecución del algoritmo cúbico se multiplica por un factor  $f^3$ .

Si el tamaño de la entrada se multiplica por un factor  $f$ , el tiempo de ejecución de un programa cúbico se multiplica por un factor aproximadamente igual a  $f^3$ .

Podemos realizar cálculos similares para los algoritmos cuadrático y lineal. Para el algoritmo cuadrático, suponemos que  $T(N) = cN^2$ . De aquí se deduce que  $T(10N) = c(10N)^2$ . Al expandir tenemos

$$T(10N) = 100cN^2 = 100T(N)$$

Si el tamaño de la entrada se multiplica por un factor  $f$ , el tiempo de ejecución de un programa cuadrático se multiplica por un factor aproximadamente igual a  $f^2$ .

Por tanto, cuando el tamaño de la entrada se multiplica por un factor de 10, el tiempo de ejecución de un programa cuadrático se multiplica por un factor de aproximadamente 100. Esta relación se ve también confirmada en la Figura 5.10. En general, para un algoritmo cuadrático, una multiplicación por un factor  $f$  en el tamaño de la entrada nos da una multiplicación por  $f^2$  en el tiempo de ejecución.

Finalmente, para un algoritmo lineal, un cálculo similar nos muestra que si multiplicamos el tamaño de la entrada por 10, obtenemos un tiempo de ejecución que también se multiplica por 10. De nuevo, esta relación se ve confirmada experimentalmente en la Figura 5.10. Observe,

Si el tamaño de la entrada se multiplica por un factor  $f$ , el tiempo de ejecución de un programa lineal también se multiplica por un factor  $f$ . Este es el tiempo de ejecución preferido para un algoritmo.

sin embargo, que para un programa lineal, el término *suficientemente grande* podría significar un tamaño de entrada algo mayor que para los restantes programas, suponiendo que en todos los casos existe un tiempo de computación mínimo fijo que es independiente del tamaño de la entrada. Para un programa lineal, este término podría continuar teniendo un valor significativo para tamaños de entrada moderados.

El análisis utilizado aquí no funciona cuando hay términos logarítmicos.

Cuando a un algoritmo  $O(N \log N)$  se le presenta una entrada 10 veces mayor, el tiempo de ejecución se multiplica por un factor ligeramente mayor de 10. Específicamente, tenemos que  $T(10N) = c(10N) \log (10N)$ . Al expandir, obtenemos

$$T(10N) = 10cN \log(10N) = 10cN \log N + 10cN \log 10 = 10T(N) + c'N$$

Aquí  $c' = 10c \log 10$ . A medida que  $N$  va haciéndose muy grande, la relación  $T(10N) / T(N)$  va haciéndose más y más próxima a 10, porque  $c'N / T(N) \approx (10 \log 10) / \log N$  se hace cada vez más pequeño a medida que aumenta  $N$ . En consecuencia, si el algoritmo es competitivo comparado con un algoritmo lineal para un valor de  $N$  muy grande, es probable que siga siendo competitivo para un valor de  $N$  ligeramente mayor.

¿Quiere esto decir que los algoritmos cuadráticos y cúbicos son inútiles? La respuesta es no. En algunos casos, los algoritmos más eficientes conocidos son cuadráticos o cúbicos. En otros, el algoritmo más eficiente es aún peor (exponencial). Además, cuando la cantidad de entrada es pequeña, cualquier algoritmo nos sirve. Frecuentemente, los algoritmos que no son asintóticamente eficientes, resultan ser, de todos modos, fáciles de programar. Para tamaños de entrada pequeños, esa es la manera correcta de proceder. Finalmente, una buena forma de probar un algoritmo lineal complejo consiste en comparar su salida con la de un algoritmo de búsqueda exhaustiva. En la Sección 5.8 hablaremos, asimismo, de algunas otras limitaciones del modelo O mayúscula.

## 5.5 El logaritmo

La lista de funciones de crecimiento típicas incluye varias entradas que contienen el logaritmo. Un *logaritmo* es el exponente que indica la potencia a la que hay que elevar un número (la base) para obtener otro número dado. En esta sección vamos a ver con algo más de detalle las matemáticas que subyacen al concepto de logaritmo. En la Sección 5.6 veremos cómo se utilizan en un algoritmo sencillo.

Comenzaremos con la definición formal y luego aportaremos algunos puntos de vista más intuitivos.

El logaritmo de  $N$  (en base 2) es el valor  $X$  tal que 2 elevado a la potencia  $X$  es igual a  $N$ . De manera predeterminada, la base del algoritmo es 2.

**Definición.** Para cualquier  $B$ ,  $N > 0$ ,  $\log_B N = K$  si  $B^K = N$

En esta definición,  $B$  es la base del logaritmo. En informática, cuando se omite la base, se asume de manera predeterminada que es 2, lo cual es natural por diversas razones, como veremos posteriormente en el capítulo. Vamos a demostrar un teorema matemático, el Teorema 5.4 para ver que, en lo que respecta a la notación O mayúscula, la base no tiene ninguna importancia, y también para mostrar cómo deducir las relaciones en las que están implicados logaritmos.

**Teorema 5.4**

La base no importa. Para cualquier constante  $B > 1$ ,  $\log_B N = O(\log N)$ .

**Demostración**

Sea  $\log_B N = K$ . Entonces  $B^K = N$ . Sea  $C = \log B$ . Entonces  $2^C = B$ . Por tanto,  $B^K = (2^C)^K = N$ . De aquí, tenemos que  $2^{CK} = N$ , lo que 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 de este texto, utilizaremos exclusivamente logaritmos en base 2. Un hecho importante acerca del logaritmo es que es una función que crece lentamente. Puesto que  $2^{10} = 1.024$ ,  $\log 1.024 = 10$ . Cálculos adicionales nos muestran que el logaritmo de 1.000.000 es aproximadamente 20 y el logaritmo de 1.000.000.000 es solo 30. En consecuencia, el rendimiento de un algoritmo  $O(N \log N)$  está mucho más próximo a un algoritmo lineal  $O(N)$  que a uno cuadrático  $O(N^2)$ , incluso para cantidades de entrada moderadamente grandes. Antes de examinar un algoritmo realista cuyo tiempo de ejecución incluye el logaritmo, veamos unos cuantos ejemplos del papel que desempeñan los logaritmos.

**Bits en un número binario**

¿Cuántos bits hacen falta para representar  $N$  enteros consecutivos?

Un entero *short* de 16 bits representa 65.536 enteros comprendidos en el rango que va de  $-32.768$  a  $32.767$ . En general, bastan  $B$  bits para representar  $2^B$  enteros distintos. Por tanto, el número  $B$  de bits requeridos para representar  $N$  enteros consecutivos satisface la ecuación  $2^B \geq N$ . De aquí, obtenemos que  $B \geq \log N$ , por lo que el número mínimo de bits es  $\lceil \log N \rceil$ . (Aquí  $\lceil X \rceil$  es la función techo y representa el entero más pequeño que tiene al menos un valor igual a  $X$ . La correspondiente función suelo  $\lfloor X \rfloor$  representa el entero que tiene un valor al menos tan pequeño como  $X$ .)

El número de bits requerido para representar números es logarítmico.

**Duplicaciones consecutivas**

Comenzando con  $X = 1$ , ¿cuántas veces hay que multiplicar  $X$  por dos antes de que tenga un valor al menos tan grande como el de  $N$ ?

Suponga que comenzamos 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 tendríamos 4 euros; después de 3 años, tendríamos 8 euros, y así sucesivamente. En general, después de  $K$  años tendríamos  $2^K$  euros, por lo que queremos encontrar el valor de  $K$  más pequeño que satisfaga  $2^K \geq N$ . Esta es la misma ecuación que antes, por lo que  $K = \lceil \log N \rceil$ . Después de 20 años, tendríamos más de un millón de euros. El principio de la duplicación repetida afirma que, partiendo de 1, solo podemos duplicar la cantidad repetidamente  $\lceil \log N \rceil$  veces hasta alcanzar  $N$ .

El principio de la duplicación repetida afirma que, partiendo de 1 solo podemos multiplicar por dos repetidamente un número logarítmico de veces, hasta alcanzar  $N$ .

**Divisiones consecutivas**

Comenzando con  $X = N$ , si dividimos  $N$  entre dos repetidamente, ¿cuántas iteraciones habrá que aplicar para que  $N$  sea menor o igual que 1?

Si se redondea el resultado de la división al entero más próximo (o si efectuamos una división real y no entera), tenemos el mismo problema que con las duplicaciones repetidas, salvo porque ahora

El principio de la división repetida afirma que, partiendo de  $N$ , solo podemos dividir entre dos un número logarítmico de veces. Este proceso se utiliza para obtener rutinas de búsqueda logarítmicas.

El  $N$ -ésimo número armónico es la suma de los reciprocos de los primeros  $N$  enteros positivos. La tasa de crecimiento del número armónico es logarítmica.

vamos en la dirección opuesta. De nuevo, la respuesta es  $\lceil \log N \rceil$  iteraciones. Si se redondea hacia abajo la división, la respuesta es  $\lfloor \log N \rfloor$ . Podemos ver la diferencia comenzando con  $X = 3$ . Hacen falta dos divisiones, a menos que redondeemos las divisiones por defecto, en cuyo caso solo hace falta una.

Muchos de los algoritmos examinados en este texto tendrán logaritmos, introducidos a causa del *principio de la división repetida* que afirma que, si comenzamos con  $N$ , solo podemos dividir entre dos un número logarítmico de veces. En otras palabras, un algoritmo es  $O(N \log N)$  si hace falta un tiempo constante ( $O(1)$ ) para reducir el tamaño del problema según una fracción constante (que normalmente es  $1/2$ ). Esta condición se deduce directamente del hecho de que habrá  $O(\log N)$  iteraciones del bucle. Cualquier fracción constante nos sirve, porque la fracción se ve reflejada en la base del logaritmo, y el Teorema 5.4 nos dice que la base no importa.

Todas las restantes apariciones de los logaritmos se deben (directa o indirectamente) a la aplicación del Teorema 5.5. Este teorema está relacionado con el  $N$ -ésimo número armónico, que es la suma de los reciprocos de los primeros  $N$  enteros positivos, y afirma que el  $N$ -ésimo número armónico,  $H_N$ , satisface  $H_N = \Theta(\log N)$ . La demostración utiliza conceptos de cálculo, pero no hace falta entender la demostración para utilizar el teorema.

### Teorema 5.5

Sea  $H_N = \sum_{i=1}^N 1/i$ . Entonces  $H_N = \Theta(\log N)$ . Una estimación más precisa es  $\ln N + 0,577$ .

### Demostración

La intuición de la demostración es que una suma discreta está bien aproximada por la integral (continua). La demostración utiliza una construcción para demostrar que la suma  $H_N$  puede acotarse por arriba y por abajo mediante  $\int \frac{dx}{x}$ , con límites apropiados. Los detalles se dejan para el Ejercicio 5.28.

En la siguiente sección se muestra cómo el principio de la división repetida conduce a un algoritmo de búsqueda eficiente.

## 5.6 Problema de la búsqueda estática

Uno de los usos más importantes de las computadoras es el de la búsqueda de datos. Si no se permite que los datos varíen (por ejemplo, si están almacenados en CD-ROM), decimos que los datos son estáticos. Una *búsqueda estática* accede a datos que nunca son modificados. El problema de la búsqueda estática se puede formular de manera natural de la forma siguiente:

### Problema de la búsqueda estática

Dado un entero  $X$  y una matriz  $A$ , devolver la posición de  $X$  en  $A$  o una indicación de que no está presente. Si  $X$  aparece más de una vez, devolver cualquiera de las apariciones. La matriz  $A$  nunca es modificada.

Un ejemplo de búsqueda estática sería buscar una persona en una guía telefónica. La eficiencia de un algoritmo de búsqueda estática depende de si la matriz en la que se está buscando está ordenada. En el caso de la guía telefónica, buscar por nombre es muy rápido, pero buscar por

número de teléfono es algo inasumible (para los seres humanos). En esta sección, vamos a examinar algunas soluciones para el problema de la búsqueda estática.

### 5.6.1 Búsqueda secuencial

Cuando la matriz de entrada no está ordenada, no tenemos ninguna opción salvo efectuar una *búsqueda secuencial* lineal, que recorre la matriz secuencialmente hasta encontrar una correspondencia. La complejidad del algoritmo se analiza de tres formas. En primer lugar, proporcionamos el coste de una búsqueda que no tenga éxito. Después, proporcionamos el coste de caso peor de una búsqueda que sí que tenga éxito. Por último, determinamos el coste promedio de una búsqueda que tenga éxito.

Una búsqueda secuencial recorre los datos secuencialmente hasta encontrar una correspondencia.

Analizar por separado las búsquedas que tienen éxito y las que no es algo bastante típico. Las búsquedas que no tienen éxito suelen requerir más tiempo que las que sí lo tienen (piense en la última vez que perdió algo dentro de su domicilio). Para las búsquedas secuenciales el análisis es muy sencillo.

Una búsqueda secuencial es lineal.

Una búsqueda que no tenga éxito requiere examinar cada elemento de la matriz, por lo que el tiempo será  $O(N)$ . En el caso peor, una búsqueda que tenga éxito también requerirá examinar todos los elementos de la matriz, porque puede que no encontremos una correspondencia hasta llegar al último elemento. Por tanto, el tiempo de ejecución para una búsqueda que tenga éxito también es lineal. Sin embargo, como promedio, solo tendremos que explorar la mitad de la matriz; es decir, para toda búsqueda que tenga éxito en la posición  $i$ , existe la búsqueda correspondiente que también tiene éxito en la posición  $N - 1 - i$  (suponiendo que comenzamos la numeración en 0). Sin embargo,  $N/2$  sigue siendo  $O(N)$ . Como hemos mencionado anteriormente en el capítulo, todos estos términos O mayúscula deberían ser también correctamente términos Theta mayúscula. Sin embargo, el uso de la notación O mayúscula es más popular.

### 5.6.2 Búsqueda binaria

Si la matriz de entrada ha sido previamente ordenada, tenemos una alternativa a la búsqueda secuencial. Esa alternativa es la *búsqueda binaria*, que comienza a partir de la mitad de la matriz en lugar de comenzar por uno de los extremos. Mantenemos en todo momento dos indicadores `low` y `high`, que delimitan la parte de la matriz en la que debe residir el elemento buscado, si es que está presente. Inicialmente el rango va de 0 a  $N - 1$ . Si `low` llegara a hacerse mayor que `high`, sabremos inmediatamente que el elemento no está presente, por lo que devolveríamos la indicación `NOT_FOUND` para señalar que no hemos encontrado ninguna correspondencia. En caso contrario, en la línea 15, igualamos `mid` al punto medio del rango (redondeando hacia abajo si el rango tiene un número par de elementos) y comparamos el elemento que estamos buscando con el elemento de la posición `mid`.<sup>5</sup> Si encontramos una correspondencia, habremos terminado y podremos volver. Si el elemento que estamos buscando

Si la matriz de entrada está ordenada, podemos utilizar la búsqueda binaria que se efectúa partiendo del punto medio de la matriz, en lugar de partir de uno de los extremos.

<sup>5</sup> Observe que si `low` y `high` son suficientemente grandes, su suma hará que se desborde un valor `int`, dando un valor negativo incorrecto para `mid`. En el Ejercicio 5.29 se analiza esto con mayor detalle.

es inferior al que estamos buscando, entonces tendrá que estar dentro del rango que va de `low` a `mid - 1`. Si es mayor, entonces deberá estar dentro del rango que va de `mid + 1` a `high`. En la Figura 5.11, las líneas 17 a 20 modifican el rango de valores posibles, reduciéndolo esencialmente a la mitad. Mediante el principio de la división repetida, sabemos que el número de iteraciones será  $O(N \log N)$ .

La búsqueda binaria es logarítmica porque el rango de búsqueda se divide por dos en cada iteración.

Para una búsqueda que no tenga éxito, el número de iteraciones del bucle será  $\lfloor \log N \rfloor + 1$ . La razón es que dividimos el rango a la mitad en cada iteración (redondeando hacia abajo si el rango tiene un número impar de elementos); sumamos 1 porque el rango final tiene cero elementos. Para una búsqueda que tenga éxito, el caso peor será  $\lfloor \log N \rfloor$  iteraciones, porque en el caso peor terminaremos teniendo un rango de un solo elemento. El caso promedio es solo una iteración mejor, porque la mitad de los elementos necesitarán el caso peor para ser encontrados, una cuarta parte de los elementos requerirá una iteración menos y solo uno de cada 2<sup>2</sup> elementos requerirán 1 iteración menos que el caso peor.

```

1  /**
2   * Realiza la búsqueda binaria estándar utilizando
3   * dos comparaciones por nivel.
4   * @return índice donde se encuentra el elemento o NOT_FOUND.
5   */
6  public static <AnyType extends Comparable<? super AnyType>>
7      int binarySearch( AnyType [ ] a, AnyType x )
8  {
9      int low = 0;
10     int high = a.length - 1;
11     int mid;
12
13     while( low <= high )
14     {
15         mid = ( low + high ) / 2;
16
17         if( a[ mid ].compareTo( x ) < 0 )
18             low = mid + 1;
19         else if( a[ mid ].compareTo( x ) > 0 )
20             high = mid - 1;
21         else
22             return mid;
23     }
24
25     return NOT_FOUND; // NOT_FOUND = -1
26 }
```

Figura 5.11 Búsqueda binaria básica que utiliza comparaciones de tres vías.

El aspecto matemático de este cálculo implica determinar el promedio ponderado calculando la suma de una serie finita. En resumen, sin embargo, es que el tiempo de ejecución para cada búsqueda es  $O(N \log N)$ . En el Ejercicio 5.25, le pediremos completar estos cálculos.

Para valores de  $N$  razonablemente grandes, la búsqueda binaria es más rápida que la búsqueda secuencial. Por ejemplo, si  $N$  es 1.000, entonces como promedio una búsqueda secuencial que tenga éxito requerirá unas 500 comparaciones. La búsqueda binaria promedio, utilizando la fórmula anterior, requerirá  $\lceil \log N \rceil - 1$  (es decir, ocho) iteraciones para una búsqueda que tenga éxito. Cada iteración utiliza 1,5 comparaciones como promedio (en ocasiones 1 comparación, en otros casos, 2), por lo que el total es de 12 comparaciones para cada búsqueda que tenga éxito. La búsqueda binaria tiene un rendimiento aun mejor, comparada con la lineal, en el caso peor o cuando las búsquedas no tienen éxito.

Si queremos hacer que la búsqueda binaria sea aun más rápida, tenemos que simplificar el bucle interno. Una posible estrategia consiste en eliminar de ese bucle interno la comprobación (implícita) de si la búsqueda ha tenido éxito, y reducir siempre el rango un elemento en todos los casos. Entonces podemos utilizar una única comprobación fuera del bucle para determinar si el elemento se encuentra en la matriz o no puede ser encontrado, 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 situado en la posición `mid`, entonces se encontrará en el rango que incluye la posición `mid`. Al acabar el bucle, el subrango es 1 y podemos comprobar si hemos encontrado una correspondencia.

La optimización de la búsqueda binaria permite reducir el número de comparaciones a aproximadamente la mitad.

En el algoritmo revisado, el número de iteraciones es siempre  $\lceil \log N \rceil$  porque siempre reducimos el rango a la mitad, posiblemente redondeando hacia abajo. Por tanto, el número de comparaciones utilizadas es siempre  $\lceil \log N \rceil + 1$ .

La búsqueda binaria resulta sorprendentemente complicada de codificar. El Ejercicio 5.2 ilustra algunos errores comunes.

Observe que, para valores de  $N$  pequeños, como por ejemplo los valores menores de 6, puede que no merezca la pena utilizar la búsqueda binaria. Emplea aproximadamente el mismo número de comparaciones que la búsqueda lineal para una búsqueda que tenga éxito típica, pero tiene el gasto adicional de la línea 18 en cada iteración. Ciertamente, las últimas iteraciones de la búsqueda binaria progresan de manera lenta. Eso sugiere que podríamos adoptar una estrategia híbrida, en la que el bucle de búsqueda binaria se terminara cuando el rango fuera ya suficientemente pequeño y se aplicara luego una exploración secuencial para terminar el problema. De forma similar, las personas buscan en una guía telefónica de manera no secuencial, pero una vez que han reducido el rango de búsqueda a una columna, realizan una exploración secuencial. La exploración de una guía telefónica no es secuencial, pero tampoco es una búsqueda binaria. Más bien se parece al algoritmo del que vamos a hablar en la siguiente sección.

### 5.6.3 Búsqueda por interpolación

La búsqueda binaria es muy rápida a la hora de buscar en una matriz estática ordenada. De hecho, es tan rápida que rara vez utilizaremos ninguna otra cosa. Sin embargo, un método de búsqueda estática que en ocasiones es más rápido es la *búsqueda por interpolación*, que tiene como promedio un mejor rendimiento O mayúscula que la búsqueda binaria, pero cuyas aplicaciones son limitadas

```

1  /**
2   * Realiza la búsqueda binaria estándar utilizando
3   * una comparación por nivel.
4   * @return índice donde se encuentra el elemento o NOT_FOUND.
5   */
6  public static <AnyType extends Comparable<? super AnyType>>
7      int binarySearch( AnyType [ ] a, AnyType x )
8  {
9      if( a.length == 0 )
10         return NOT_FOUND;
11
12     int low = 0;
13     int high = a.length - 1;
14     int mid;
15
16     while( low < high )
17     {
18         mid = ( low + high ) / 2;
19
20         if( a[ mid ].compareTo( x ) < 0 )
21             low = mid + 1;
22         else
23             high = mid;
24     }
25
26     if( a[ low ].compareTo( x ) == 0 )
27         return low;
28
29     return NOT_FOUND;
30 }

```

**Figura 5.12** Búsqueda binaria básica que utiliza comparaciones de dos vías.

y que tiene un caso peor bastante pobre. Para que una búsqueda por interpolación resulte práctica, tienen que satisfacerse dos suposiciones:

1. Cada acceso debe ser muy caro comparado con una instrucción típica. Por ejemplo, la matriz puede estar en un disco en lugar de en memoria y cada comparación requiere un acceso a disco.
2. Los datos no solo deben estar ordenados, sino que también deben estar distribuidos de forma bastante uniforme. Por ejemplo, una guía telefónica está distribuida de una manera bastante uniforme. Si los elementos de entrada son  $\{1, 2, 4, 8, 16, \dots\}$ , la distribución no es uniforme.

Estas suposiciones son bastante restrictivas, por lo que es posible que nunca llegue a tener que utilizar una búsqueda por interpolación. Pero es interesante observar que existe más de una forma de resolver cada problema y que no hay ningún algoritmo, ni siquiera la búsqueda binaria clásica, que sea el mejor en todas las situaciones.

La búsqueda por interpolación requiere que invertamos más tiempo en tratar de realizar una estimación precisa de dónde puede estar el elemento que buscamos. La búsqueda binaria siempre utiliza el punto medio. Sin embargo, buscar a *Benítez, Alfonso*, en mitad de la guía telefónica sería estúpido; obviamente, sería más apropiado buscarlo cerca del principio de la guía. Por tanto, en lugar de `mid`, utilizamos `next` para indicar el siguiente elemento al que intentaremos acceder.

He aquí un ejemplo de un algoritmo que podría funcionar bien. Suponga que el rango contiene 1.000 elementos, que el elemento más bajo del rango es 1.000 y que el elemento más alto del rango es 1.000.000, y suponga también que estamos buscando un elemento de valor 12.000. Si los elementos están distribuidos uniformemente, entonces cabría esperar que encontráramos una correspondencia en algún lugar cercano al duodécimo elemento. La fórmula aplicable es:

$$\text{next} = \text{low} + \left\lceil \frac{x - a[\text{low}]}{a[\text{high}] - a[\text{low}]} \times (\text{high} - \text{low} - 1) \right\rceil$$

donde `next` representa el elemento siguiente, `low` representa al elemento inferior del rango en el que estamos buscando y `high` representa al elemento superior del rango.

El restar 1 es un ajuste técnico que se ha demostrado que funciona bien en la práctica. Claramente, este cálculo es más costoso que el utilizado en la búsqueda binaria. Implica una división adicional (la división por 2 en la búsqueda binaria es realmente tan solo un desplazamiento de un bit, al igual que dividir por 10 nos resulta muy sencillo a los seres humanos), una multiplicación y cuatro restas. Estos cálculos se deben realizar llevando a cabo operaciones de punto flotante. Con todo ello, una iteración puede ser más lenta que la búsqueda binaria completa. Sin embargo, si el coste de estos cálculos es insignificante al compararlo con el coste de acceder a un elemento, la velocidad de los cálculos no importa; solo tendremos que preocuparnos por el número de iteraciones.

En el caso peor, en el que los datos no están uniformemente distribuidos, el tiempo de ejecución podría ser lineal, pudiendo ser necesario examinar todos los elementos. En el Ejercicio 5.20 le pediremos que construya ese tipo de caso. Sin embargo, si asumimos que los elementos están razonablemente distribuidos, como sucede en una guía telefónica, se ha demostrado que el número promedio de comparaciones es de  $O(N \log \log N)$ . En otras palabras, aplicamos dos veces sucesivas el logaritmo. Para  $N = 4.000$  millones,  $\log N$  es aproximadamente 32 y  $\log \log N$  es aproximadamente 5. Por supuesto, hay algunas constantes ocultas en la notación O mayúscula, pero el logaritmo adicional puede reducir el número de iteraciones considerablemente, siempre y cuando no nos encontremos con un caso especialmente malo. Sin embargo, demostrar el resultado de forma rigurosa es bastante complicado.

La búsqueda por interpolación tiene una mejor cota O mayúscula, como promedio, que la búsqueda primaria, pero su aplicación práctica es limitada y su caso peor es bastante pobre.

## 5.7 Comprobación del análisis de un algoritmo

Una vez que hemos realizado el análisis de un algoritmo, lo que queremos es determinar si es correcto y si es lo mejor que podemos obtener. Una forma de hacer esto consiste en codificar el

programa y ver si el tiempo de ejecución observado empíricamente se corresponde con el tiempo de ejecución predicho por el análisis.

Cuando  $N$  se multiplica por un factor de 10, el tiempo de ejecución de multiplica por un factor de 10 en el caso de los programas lineales, de 100 en el caso de los programas cuadráticos y de 1000 en el de los programas cúbicos. Los programas que se ejecutan en  $O(N \log N)$  requieren multiplicar por un valor algo superior a 10, para ejecutarse en las mismas circunstancias. Estos incrementos pueden ser difíciles de detectar si los términos de menor orden tienen coeficientes relativamente grandes y  $N$  no tiene un valor suficientemente elevado. Un ejemplo sería el salto de  $N = 10$  a  $N = 100$  en el tiempo de ejecución de las diversas implementaciones del problema de la suma máxima de subsecuencia contigua. Diferenciar los programas lineales de los programas  $O(N \log N)$ , basándose exclusivamente en las evidencias empíricas, puede resultar también bastante difícil.

Otro truco comúnmente utilizado para verificar que un cierto programa es  $O(F(N))$  consiste en calcular los valores  $T(N)/F(N)$  para un cierto rango de  $N$  (usualmente espaciado según factores de 2) donde  $T(N)$  es el tiempo de ejecución empíricamente observado. Si  $F(N)$  constituye una estimación aproximadamente precisa del tiempo de ejecución, entonces los valores calculados convergerán hacia una cierta constante positiva. Si  $F(N)$  es una sobreestimación, entonces los valores convergerán a cero. Si  $F(N)$  es una subestimación, y por tanto es incorrecta, entonces los valores divergirán.

Por ejemplo, suponga que escribimos un programa para realizar  $N$  búsquedas binarias utilizando el algoritmo de búsqueda binaria. Puesto que cada búsqueda es logarítmica, esperamos que el tiempo de ejecución total del programa sea  $O(N \log N)$ . La Figura 5.13 muestra el tiempo de ejecución realmente observado para la rutina en una computadora real (pero extremadamente lenta), para varios tamaños de entrada. La última columna es, con toda probabilidad, la columna convergente y confirma por tanto nuestro análisis, mientras que los números crecientes para  $T/N$  sugieren que  $O(N)$  es una subestimación, y los valores rápidamente decrecientes para  $T/N^2$  sugieren que  $O(N^2)$  es una sobreestimación.

Observe, en particular, que no tenemos una convergencia perfectamente definida. Un problema es que el reloj utilizado para temporizar el programa solo tiene un tic cada 10 milisegundos. Observe

$N$	Tiempo de procesador $T$ (microsegundos)			
	$T/N$	$T/N^2$	$T/(N \log N)$	
10.000	1.000	0,1000000	0,0000100	0,0075257
20.000	2.000	0,1000000	0,0000050	0,0069990
40.000	4.400	0,1100000	0,0000027	0,0071953
80.000	9.300	0,1162500	0,0000015	0,0071373
160.000	19.600	0,1225000	0,0000008	0,0070860
320.000	41.700	0,1303125	0,0000004	0,0071257
640.000	87.700	0,1370313	0,0000002	0,0071046

Figura 5.13 Tiempos de ejecución empíricos para  $N$  búsquedas binarias en una matriz de  $N$  elementos.

también que no hay una gran diferencia entre  $O(N)$  y  $O(N \log N)$ . Ciertamente, un algoritmo  $O(N \log N)$  está mucho más próximo a ser lineal que a ser cuadrático.

## 5.8 Limitaciones del análisis O mayúscula

El análisis O mayúscula es una herramienta muy efectiva, pero también tiene sus limitaciones. Como ya hemos mencionado, su uso no es apropiado para entradas de pequeño tamaño. Para este tipo de entradas, utilice siempre el algoritmo más simple. Asimismo, para un algoritmo concreto, la constante implicada en el análisis O mayúscula puede ser demasiado grande como para resultar práctica. Por ejemplo, si el tiempo de ejecución de un algoritmo está gobernado por la fórmula  $2N \log N$  y otro algoritmo tiene un tiempo de ejecución de  $1000N$ , entonces lo más probable es que el primer algoritmo sea mejor, aun cuando su tasa de crecimiento sea más grande. Las constantes de gran tamaño pueden entrar en acción cuando un algoritmo sea excesivamente complejo. También influyen porque nuestro análisis no tiene en cuenta las constantes y no puede, por tanto, diferenciar entre cosas como los accesos a memoria (que son baratos) y los accesos a disco (que normalmente son varios miles de veces más caros). Nuestro análisis asume que tenemos una memoria infinita, pero las aplicaciones que implican grandes conjuntos de datos, la falta de memoria suficiente puede ser un grave problema.

En ocasiones, aun cuando tengamos en cuenta las constantes y los términos de menor orden, se demuestra empíricamente que el análisis es una sobreestimación. En este caso, tendremos que tratar de hacer más preciso el análisis (usualmente mediante algún tipo de observación inteligente). O bien, puede que la cota del tiempo de ejecución del caso promedio sea significativamente inferior a la del tiempo de ejecución del caso peor, por lo que no es posible mejorar en modo alguno la cota. Para muchos algoritmos complicados, la cota de caso peor se alcanza mediante alguna entrada particularmente mala, pero en la práctica suele tratarse de una sobreestimación. Dos ejemplos serían los algoritmos de ordenación Shell y de ordenación rápida (que describiremos en el Capítulo 8).

Sin embargo, los límites de caso peor suelen ser más fáciles de determinar que los correspondientes casos promedio. Por ejemplo, todavía no ha logrado obtenerse un análisis matemático del tiempo de ejecución promedio de la ordenación Shell. En ocasiones, resulta difícil definir incluso lo que significa *promedio*. Utilizamos el análisis de caso peor porque es sencillo y también porque, en la mayoría de los casos, dicho análisis de caso peor es bastante significativo. En el curso de la realización de ese análisis, frecuentemente podemos determinar si resulta aplicable al caso promedio.

El caso peor es en ocasiones bastante poco común y puede ignorarse sin problemas. En otras ocasiones, es muy común y no puede ser ignorado.

El análisis del caso promedio es casi siempre más difícil que el análisis de caso peor.

## Resumen

En este capítulo hemos presentado el análisis de algoritmos y hemos demostrado que las decisiones algorítmicas suelen influir sobre el tiempo de ejecución de un programa mucho más que los trucos de programación. También hemos visto la enorme diferencia que existe entre los tiempos de ejecución de los programas cuadráticos y lineales, y hemos ilustrado el hecho de que los algoritmos cúbicos son, en su mayor parte, poco satisfactorios. Hemos examinado un algoritmo que podría contemplarse

como base para nuestra primera estructura de datos. La búsqueda binaria soporta de manera eficiente las operaciones estáticas (es decir, la búsqueda pero no la actualización), proporcionando así una búsqueda de caso peor con un tiempo de ejecución logarítmico. Posteriormente en el texto examinaremos estructuras dinámicas de datos que soportan de manera eficiente las actualizaciones (tanto la inserción como el borrado).

En el Capítulo 6 veremos algunas de las estructuras de datos y algoritmos incluidos en la API de Colecciones de Java. También examinaremos algunas aplicaciones de las estructuras de datos y comentaremos su eficiencia.



## Conceptos clave

**algoritmo de tiempo lineal** Un algoritmo que hace que el tiempo de ejecución crezca como  $O(N)$ . Si el tamaño de la entrada se multiplica por un factor  $f$ , entonces el tiempo de ejecución también se multiplica por un factor  $f$ . Es el tiempo de ejecución preferido para un algoritmo. (202)

**búsqueda binaria** El método de búsqueda utilizado si la matriz de entrada ha sido ordenada previamente y que se lleva a cabo comenzando por el medio, en lugar de por uno de los extremos. La búsqueda binaria es logarítmica porque el rango de búsqueda se divide entre dos en cada iteración. (205)

**búsqueda estática** Accede a datos que nunca son modificados. (204)

**búsqueda por interpolación** Un algoritmo de búsqueda estática que tiene un mejor rendimiento O mayúscula como promedio que la búsqueda binaria, pero que tiene pocas aplicaciones prácticas y un caso peor bastante pobre. (209)

**búsqueda secuencial** Un método de búsqueda lineal que recorre una matriz hasta encontrar una correspondencia. (205)

**cota de caso peor** Proporciona una garantía con respecto a todas las entradas de un determinado tamaño. (200)

**cota de caso promedio** Medida del tiempo de ejecución como un promedio para todas las entradas posibles de tamaño  $N$ . (200)

**logaritmo** El exponente que indica la potencia a la que hay que elevar un número para obtener otro número dado. Por ejemplo, el logaritmo de  $N$  (en base 2) es el valor  $X$  tal que 2 elevado a la potencia  $X$  es igual a  $N$ . (202)

**números armónicos** El  $N$ -ésimo número armónico es la suma de los reciprocos de los primeros  $N$  enteros positivos. La tasa de crecimiento de los números armónicos es logarítmica. (204)

**O mayúscula** La notación utilizada para capturar el término más dominante en una función; es similar a menor o igual que, cuando se habla de tasas de crecimiento. (188, 199)

**O minúscula** La notación similar a menor que cuando se habla de tasas de crecimiento. (199)

**Omega mayúscula** La notación similar a mayor o igual que, cuando se habla de tasas de crecimiento. (199)

**principio de la división repetida** Afirma que, partiendo de  $N$ , solo se puede dividir por 2 repetidamente un número logarítmico de veces, antes de alcanzar 1. Este proceso se emplea para obtener rutinas logarítmicas de búsqueda. (204)

**principio de la duplicación repetida** Afirma que, partiendo de 1, solo se puede multiplicar por 2 repetidamente un número logarítmico de veces, antes de alcanzar  $N$ . (203)

**subcuadrático** Un algoritmo cuyo tiempo de ejecución es estrictamente menor que el cuadrático y que puede escribirse como  $O(N^2)$ . (199)

**Theta mayúscula** La notación similar a igual que, cuando se habla de tasas de crecimiento. (199)



## Errores comunes

1. Para bucles anidados, el tiempo total se ve afectado por el producto del tamaño de los bucles. Para bucles consecutivos, no es así.
2. No se limite a contar sin más el número de bucles. Una pareja de bucles anidados que se ejecuten cada uno de ellos desde 1 a  $N^2$  representa un tiempo  $O(N^4)$ .
3. No escriba expresiones como  $O(2N^2)$  u  $O(N^2 + M)$ . Solo es necesario el término dominante con la constante multiplicativa eliminada.
4. Utilice igualdades con O mayúscula, Omega mayúscula, etc. Escribir que el tiempo de ejecución es  $> O(N^2)$  no tiene ningún sentido, porque O mayúscula representa una cota superior. No escriba que el tiempo de ejecución es  $< O(N^2)$ ; si la intención es decir que el tiempo de ejecución es estrictamente inferior al cuadrático, utilice la notación O minúscula.
5. Utilice Omega mayúscula, no O mayúscula para expresar una cota inferior.
6. Emplee el logaritmo para describir el tiempo de ejecución de un problema que se resuelva dividiendo su tamaño a la mitad en un tiempo constante. Si hace falta más de un tiempo constante para dividir el problema a la mitad, el logaritmo no es aplicable.
7. La base (si es una constante) del logaritmo es irrelevante de cara al análisis O mayúscula. Incluirla es un error.



## Internet

Los tres algoritmos de suma máxima de subsecuencia contigua, así como un cuarto tomado de la Sección 7.5 están disponibles en la página web, junto con un procedimiento `main` que realiza las pruebas de temporización.

**MaxSumTest.java**

Contiene cuatro algoritmos para el problema de suma máxima de subsecuencia.

**BinarySearch.java**

Contiene la búsqueda binaria mostrada en la Figura 5.11. El código de la Figura 5.12 no se proporciona, pero una versión similar que forma parte de la API de colecciones y que está implementada en la Figura 6.15 está en **Arrays.java** como parte de `of weiss.util`.



## Ejercicios

### EN RESUMEN

- 51** Resolver un problema requiere ejecutar un algoritmo  $O(N)$  y luego realizar  $N$  búsquedas binarias en una matriz de  $N$  elementos, después de lo cual hay que ejecutar otro algoritmo  $O(N)$ . ¿Cuál es el coste total de resolver el problema?
- 52** Para la rutina de búsqueda binaria de la Figura 5.11, indique cuáles son las consecuencias de efectuar las siguientes sustituciones de fragmentos de código:
  - a. Línea 13: utilizar la comprobación `low < high`
  - b. Línea 15: realizar la asignación `mid = low + high / 2`
  - c. Línea 18: realizar la asignación `low = mid`
  - d. Línea 20: realizar la asignación `high = mid`
- 53** Resolver un problema requiere ejecutar un algoritmo  $O(N^3)$  y luego un algoritmo  $O(N)$ . ¿Cuál es el coste total de resolver el problema?
- 54** Suponga que  $T_1(N) = O(F(N))$  y  $T_2(N) = O(F(N))$ . ¿Cuáles 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.  $T_1(N) = O(T_2(N))$
- 55** Resolver un problema requiere ejecutar un algoritmo  $O(N)$  y después un segundo algoritmo  $O(N)$ . ¿Cuál es el coste total de resolver el problema?
- 56** Agrupe las siguientes funciones según su equivalencia desde el punto de vista del análisis O mayúscula:
 
$$x^2, x, x^3 + x, x^2 - x \text{ y } (x^4 / (x - 1))$$
- 57** Analizamos los programas *A* y *B* y vemos que tienen tiempos de ejecución de caso peor no superiores a  $150N \log N$  y  $N^2$ , respectivamente. Responda a las siguientes cuestiones, en caso de que sea posible.
  - a. ¿Qué programa tiene el mejor tiempo de ejecución garantizado para valores de  $N$  grandes ( $N > 10.000$ )?

- b. ¿Qué programa tiene el mejor tiempo de ejecución garantizado para valores de  $N$  pequeños ( $N < 100$ )?
- c. ¿Qué programa se ejecutará más rápido *como promedio* para  $N = 1.000$ ?
- d. ¿Puede el programa *B* ejecutarse más rápidamente que el programa *A* para *todas* las posibles entradas?
- 58** Extraemos bolas de una caja como se especifica en el Teorema 5.1, en las combinaciones proporcionadas en los apartados (a)-(d). ¿Cuáles son los valores correspondientes de  $i, j$  y  $k$ ?
- Roja, 1, 2
  - Azul, 3, 6
  - Azul, 4, Roja
  - 3, 5, Roja
- 59** ¿Por qué una implementación basada exclusivamente en el Teorema 5.2 no es suficiente para obtener un tiempo de ejecución subcuadrático para el problema de la suma máxima de secuencia contigua?

## EN TEORÍA

- 510** Complete la Figura 5.10 con estimaciones para los tiempos de ejecución que eran demasiado largos como para simularlos. Interpole los tiempos de ejecución para los cuatro algoritmos y estime el tiempo requerido para calcular la suma máxima de subsecuencia contigua de 10.000.000 de números. ¿Qué suposiciones ha hecho?
- 511** Evalúe directamente el sumatorio triple que precede al Teorema 5.1. Verifique que las respuestas son idénticas.
- 512** Un algoritmo requiere 0,4 ms para un tamaño de la entrada de 100. ¿Cuánto tiempo requerirá para un tamaño de entrada igual a 500 (suponiendo que los términos de menor orden sean despreciables), si el tiempo de ejecución es:
- lineal?
  - $O(N \log N)$ ?
  - cuadrático?
  - cúbico?
- 513** Para los algoritmos típicos que emplee para realizar cálculos a mano, determine el tiempo de ejecución necesario para
- Sumar dos enteros de  $N$  dígitos.
  - Multiplicar dos enteros de  $N$  dígitos.
- 514** Para el algoritmo cuadrático correspondiente al problema de la suma máxima de subsecuencia contigua, determine de forma precisa cuántas veces se ejecuta la instrucción más interna.
- 515** Para 1.000 elementos, nuestro algoritmo tarda 10 segundos en ejecutarse en la máquina A, pero ahora sustituimos la máquina A por la máquina B que es 3 veces

más rápida. ¿Aproximadamente cuánto tiempo tardará el algoritmo en ejecutarse en la máquina B para 2.500 elementos si el algoritmo es:

- lineal?
- cuadrático?
- $O(N^3)$ ?
- $O(N \log N)$ ?

- 516** Un algoritmo requiere un tiempo de ejecución de 0,5 ms para un tamaño de entrada igual a 100. ¿Qué tamaño de problema puede resolverse en un minuto (suponiendo que los términos de orden inferior sean despreciables) si el tiempo de ejecución es:
- lineal?
  - $O(N \log N)$ ?
  - cuadrático?
  - cúbico?
- 517** Los datos de la Figura 5.14 muestran el resultado de ejecutar el problema de suma máxima de una subsecuencia en 1991. El programa fue escrito en el lenguaje de programación C y ejecutado en una estación de trabajo Sun 3/60 basada en Unix con 4 Megabytes de memoria principal. Estos son los datos reales de aquella época.
- Verifique que, para cada algoritmo, el cambio en el tiempo observado de ejecución es coherente con el tiempo de ejecución O mayúscula del algoritmo.
  - Estime el tiempo de ejecución para  $N = 100.000$  en el caso del algoritmo con peor rendimiento.
  - ¿Cuánto más rápido es el algoritmo  $O(N^3)$  comparado con 1991?
  - ¿Cuánto más rápido es el algoritmo  $O(N)$  comparado con 1991?
  - Explique por qué las respuestas a los apartados c y d son diferentes. ¿Es esto significativo para dos algoritmos con diferentes tiempos de ejecución O mayúscula? ¿Es esto significativo para dos algoritmos con idénticos tiempos de ejecución O mayúscula?
- 518** Ordene las siguientes funciones según su 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 con la misma tasa.

$N$	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0,00193	0,00045	0,00066	0,00034
100	0,47015	0,0112	0,00486	0,00063
1.000	448,77	1,1233	0,05843	0,00333
10.000	No disponible	111,13	0,6831	0,03042
100.000	No disponible	No disponible	8,0113	0,29832

Figura 5.14 Figura 5.10 utilizando datos de 1991.

- 5.19** En términos de  $N$ , ¿cuál es el tiempo de ejecución del siguiente algoritmo a la hora de calcular  $X^N$ :

```
public static double power( double x, int n )
{
    double result = 1.0;

    for( int i = 0; i < n; i++ )
        result *= x;
    return result;
}
```

- 5.20** Construya un ejemplo en el que una búsqueda por interpolación examine cada elemento de la matriz de entrada.
- 5.21** En un caso judicial, un juez impuso una multa a un ayuntamiento por obstrucción a la justicia de 2\$ el primer día. Cada día posterior, hasta que el ayuntamiento acatara las órdenes del juez, la multa tenía que elevarse al cuadrado (es decir, la multa crecía de la forma siguiente: 2\$, 4\$, 26\$, 256\$, 65536\$,...).
- ¿Cuál será la multa en el día  $N$ ?
  - ¿Cuántos días tardará la multa en alcanzar el valor de  $D$  dólares (basta con dar una respuesta de tipo O mayúscula)?
- 5.22** Juan Desafortunado ha estado trabajando en una serie de empresas que siempre terminan siendo compradas por otras empresas. Cada vez que alguien compra la empresa en la que Juan trabaja, esta termina siendo absorbida por una empresa de mayor tamaño. Actualmente, Juan trabaja en una empresa con  $N$  empleados. ¿Cuál es el número máximo de empresas distintas para las que Juan ha trabajado?
- 5.23** Considere el siguiente método, cuya implementación se muestra:

```
// Precondición: m representa una matriz con N filas, N columnas
// En cada fila, los elementos son crecientes
// En cada columna, los elementos son crecientes
// Postcondición: devolver true si algún elemento de m almacena val;
// devolver false en caso contrario
public static boolean contains( int [ ] [ ] m, int val )
{
    int N = m.length;

    for( int r = 0; r < N; r++ )
        for( int c = 0; c < N; c++ )
            if( m[ r ][ c ] == val )
                return true;
    return false;
}
```

Un ejemplo de una matriz que satisface la precondición indicada sería

```
int [ ] [ ] m1 = { { 4, 6, 8 },
                   { 5, 9, 11 },
                   { 7, 11, 14 } };
```

- ¿Cuál es el tiempo de ejecución de `contains`?
  - Suponga que se tardan 4 segundos en ejecutar `contains` con una matriz de 100 por 100. Asumiendo que los términos de menor orden sean despreciables, ¿cuánto se tardará en ejecutar `contains` para una matriz de 400 por 400?
  - Suponga que escribimos de nuevo `contains` de modo que el algoritmo realice una búsqueda binaria en cada fila, devolviendo un valor verdadero si cualquiera de las búsquedas por fila tiene éxito y falso en caso contrario. ¿Cuál será el tiempo de ejecución de esta versión revisada de `contains`?
- 5.24** El método `hasTwoTrueValues` devuelve un valor verdadero si al menos dos valores de una matriz booleana son verdaderos. Proporcione el tiempo de ejecución O mayúscula para las tres implementaciones propuestas.

```
// Versión 1
public boolean hasTwoTrueValues( boolean [ ] arr )
{
    int count = 0;

    for( int i = 0; i < arr.length; i++ )
        if( arr[ i ] )
            count++;

    return count >= 2;
}

// Versión 2
public boolean hasTwoTrueValues( boolean [ ] arr )
{
    for( int i = 0; i < arr.length; i++ )
        for( int j = i + 1; j < arr.length; j++ )
            if( arr[ i ] && arr[ j ] )
                return true;

    return false;
}

// Versión 3
public boolean hasTwoTrueValues( boolean [ ] arr )
{
    for( int i = 0; i < arr.length; i++ )
        if( arr[ i ] )
            for( int j = i + 1; j < arr.length; j++ )
                if( arr[ j ] )
                    return true;

    return false;
}
```

- 5.25** Analice el coste de una búsqueda con éxito promedio, para el algoritmo de búsqueda binaria de la Figura 5.11.
- 5.26** Para cada uno de los siguientes fragmentos de programa, haga lo siguiente:
- Proporcione un análisis O mayúscula del tiempo de ejecución.
  - Implemente el código y ejecútelo para varios valores de  $N$ .
  - Compare su análisis con los tiempos de ejecución reales.

```

// Fragmento 1
for( int i = 0; i < n; i++ )
    sum++;

// Fragmento 2
for( int i = 0; i < n; i += 2 )
    sum++;

// Fragmento 3
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        sum++;

// Fragmento 4
for( int i = 0; i < n; i++ )
    sum++;
for( int j = 0; j < n; j++ )
    sum++;

// Fragmento 5
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++ )
        sum++;

// Fragmento 6
for( int i = 0; i < n; i++ )
    for( int j = 0; j < i; j++ )
        sum++;

// Fragmento 7
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++ )
        for( int k = 0; k < j; k++ )
            sum++;

// Fragmento 8
for( int i = 1; i < n; i = i * 2 )
    sum++;

```

- 5.27** Demuestre el Teorema 5.5. *Pista:* demuestre que  $\sum_2^N \frac{1}{i} < \int_1^N \frac{dx}{x}$ . Despues, obtenga una cota inferior similar.
- 5.28** Los enteros en Java van de  $-2^{31}$  a  $2^{31} - 1$ . En consecuencia, si tenemos una matriz de gran tamaño con más de  $2^{30}$  elementos, calcular el punto medio de una submatriz utilizando  $mid = (low + high) / 2$  hará que  $low + high$  desborde el rango entero, si el punto medio se encuentra más allá del índice  $2^{30}$  de la matriz.
- ¿Qué tamaño tiene el valor  $2^{30}$ ?
  - Demuestre que  $(low + (high - low) / 2)$  es un cálculo computacionalmente equivalente y que no provoca ningún desbordamiento en esta situación.
  - ¿Cómo de grande puede ser una matriz, utilizando la modificación sugerida en el apartado b?
- 5.29** Ocasionalmente, multiplicar los tamaños de bucles anidados puede dar una sobreestimación del tiempo de ejecución O mayúscula. Esto sucede si uno de los bucles más internos se ejecuta de manera infrecuente. Repita el Ejercicio 5.26 para el siguiente fragmento de programa:

```
for( int i = 1; i <= n; i++ )
    for( int j = 1; j <= i * i; j++ )
        if( j % i == 0 )
            for( int k = 0; k < j; k++ )
                sum++;
```

### EN LA PRÁCTICA

- 5.30** La clase `ArrayList` en `java.util` siempre incrementa la capacidad en un 50%. ¿Cuál es el número máximo de veces que puede incrementarse su capacidad?
- 5.31** La clase `ArrayList` contiene un método `trim` que redimensiona la matriz interna para ajustarla exactamente a la capacidad. El método `trim` se proporciona para utilizarlo después de que se hayan añadido todos los elementos al `ArrayList`, con el fin de no desperdiciar espacio. Suponga, sin embargo, que el programador inexperto invoca `trim` después de cada invocación de `add`. En ese caso, ¿cuál será el tiempo de ejecución de la tarea de construir un `ArrayList` de  $N$  elementos? Escriba una programa que realice 10.000 operaciones de adición a un `ArrayList` e ilustre el error de ese programador inexperto.
- 5.32** Puesto que un objeto `String` es inmutable, una concatenación de objetos `String` de la forma `str1+str2` requiere un tiempo que es proporcional a la longitud de la cadena de caracteres resultante, aun cuando `str2` sea muy corta. ¿Cuál es el tiempo de ejecución del código de la Figura 5.15, suponiendo que la matriz tiene  $N$  elementos?
- 5.33** Un número primo no tiene ningún factor, aparte de 1 y de sí mismo. Haga lo siguiente:
- Escriba un programa para determinar si un entero positivo  $N$  es primo. En términos de  $N$ , ¿cuál es el tiempo de ejecución de caso peor de su programa?

```

public static String toString( Object [ ] arr )
{
    String result = " [";
    for( String s : arr )
        result += s + " ";
    result += "]";

    return result;
}

```

Figura 5.15 Devolución de una representación de tipo `String` de una matriz.

- b. Sea  $B$  igual al número de bits de la representación binaria de  $N$ . ¿Cuál es el valor de  $B$ ?
  - c. En términos de  $B$ , ¿cuál es el tiempo de ejecución de caso peor de su programa?
  - d. Compare los tiempos de ejecución necesarios para determinar si un número de 20 bits y un número de 40 bits son primos.
- 5.34** Un elemento mayoritario en una matriz  $A$  de tamaño  $N$  es un elemento que aparece más de  $N/2$  veces (por tanto, siempre habrá como máximo un elemento mayoritario en cada matriz). Por ejemplo, la matriz
- $$3, 3, 4, 2, 4, 4, 2, 4, 4$$
- tiene un elemento mayoritario (4), mientras que la matriz
- $$3, 3, 4, 2, 4, 4, 2, 4$$
- no lo tiene. Proporcione un algoritmo para determinar el elemento mayoritario, si es que existe uno, o para informar de que no existe tal elemento. ¿Cuál es el tiempo de ejecución de su algoritmo? *Pista:* hay una solución  $O(N)$ .
- 5.35** Suponga que cuando se incrementa la capacidad de un `ArrayList`, esta siempre se duplica. Si un `ArrayList` almacena  $N$  elementos e inicialmente tenía una capacidad igual a 1, ¿cuál es el número máximo de veces que la capacidad del `ArrayList` ha podido incrementarse?
- 5.36** Diseñe algoritmos eficientes que admitan una matriz de números positivos  $a$  y determine
- El valor máximo de  $a[j] + a[i]$ , para  $j \geq i$ .
  - El valor máximo de  $a[j] - a[i]$ , para  $j \geq i$ .
- 5.37** La entrada es una matriz  $N \times N$  de números que ya se encuentra en memoria. Cada fila individual es creciente de izquierda a derecha. Cada columna individual es creciente de arriba hacia abajo. Proporcione un algoritmo  $O(N)$  de caso peor que decida si un número  $X$  se encuentra en la matriz.
- 5.38** Un problema importante en el análisis numérico es el de encontrar una solución a la ecuación  $F(X) = 0$  para alguna  $F$  arbitraria. Si la función es continua y

tiene dos puntos *low* y *high* tales que  $F(\text{low})$  y  $F(\text{high})$  tienen signos opuestos, entonces debe existir una raíz entre *low* y *high* y esa raíz puede encontrarse mediante búsqueda binaria o búsqueda por interpolación. Escriba una función que admita como parámetros  $F$ , *low* y *high* y calcule un cero. ¿Qué hay que hacer para garantizar la terminación del algoritmo?

- 5.39** Proporcione un algoritmo eficiente para determinar si existe un entero  $i$  tal que  $A_i = i$  en una matriz de enteros crecientes. ¿Cuál es el tiempo de ejecución de su algoritmo?

## PROYECTOS DE PROGRAMACIÓN

- 5.40** Como hemos mencionado en el Ejercicio 5.31, la concatenación repetida de objetos *String* puede ser muy cara en tiempo de ejecución. En consecuencia, Java proporciona una clase *StringBuilder*. Una *StringBuilder* es algo parecido a un *ArrayList* que almacena un número ilimitado de caracteres. El objeto *StringBuilder* permite añadir elementos fácilmente al final, expandiendo automáticamente la matriz interna de caracteres (duplicando su capacidad) en caso necesario. Al hacerlo así, el coste de añadir elementos puede asumirse como proporcional al número de caracteres añadidos al objeto *StringBuilder* (en lugar de al número de caracteres del resultado). En cualquier punto, el objeto *StringBuilder* puede utilizarse para construir un objeto *String*. La Figura 5.16 contiene dos métodos que devuelven objetos *String* que contienen  $N$  caracteres *x*. ¿Cuál es el tiempo de ejecución de cada método? Ejecute los métodos para varios valores de  $N$  con el fin de verificar su respuesta.

```
public static String makeLongString1( int N )
{
    String result = "";

    for( int i = 0; i < N; i++ )
        result += "x";

    return result;
}

public static String makeLongString2( int N )
{
    StringBuilder result = new StringBuilder( "" );

    for( int i = 0; i < N; i++ )
        result.append( "x" );

    return new String( result );
}
```

Figura 5.16 Devolución de una cadena de caracteres que contiene un gran número de caracteres *x*.

- 5.41** Suponga que modificamos el código de la Figura 5.5 añadiendo las siguientes líneas inmediatamente después de la línea 15:

```
if( thisSum < 0 )
    break;
```

Esta modificación está sugerida en el texto y evita examinar cualquier secuencia que comience con un número negativo.

- Si todos los números de la matriz son positivos, ¿cuál es el tiempo de ejecución del algoritmo resultante?
- Si todos los números de la matriz son negativos, ¿cuál es el tiempo de ejecución del algoritmo resultante?
- Suponga que todos los números son enteros y que están distribuidos de forma aleatoria y uniforme entre  $-50$  y  $49$ , ambos inclusive. Escriba un programa de prueba para obtener los datos de temporización hasta  $N = 1.000.000$ . ¿Puede deducir el tiempo de ejecución del programa en esta circunstancia especial?
- Ahora suponga que todos los números son enteros distribuidos aleatoriamente y uniformemente entre  $-45$  y  $54$ , ambos inclusive. ¿Afecta esto significativamente al tiempo de ejecución?
- Suponga que todos los números son enteros distribuidos aleatoriamente y uniformemente entre  $-1$  y  $1$ , ambos inclusive. ¿Afecta esto significativamente al tiempo de ejecución?

- 5.42** Suponga que dispone de una matriz ordenada de enteros positivos y negativos, y que desea determinar si existe algún valor  $x$  tal que la matriz contenga tanto  $x$  como  $-x$ . Considere los tres algoritmos siguientes:

**Algoritmo 1.** Para cada elemento de la matriz, llevar a cabo una búsqueda secuencial para ver si su negado también se encuentra en la matriz.

**Algoritmo 2.** Para cada elemento de la matriz, llevar a cabo una búsqueda binaria para ver si su negado también se encuentra en la matriz.

**Algoritmo 3.** Mantener dos índices,  $i$  y  $j$ , inicializados con el primer y el último elemento de la matriz, respectivamente. Si los dos elementos que se están indexando suman 0, entonces habremos encontrado  $x$ . En caso contrario, si la suma es menor que 0, incrementamos  $i$ ; si la suma es mayor que 0, decrementamos  $j$ , y volvemos a comprobar repetidamente la suma, hasta encontrar  $x$  o hasta que  $i$  y  $j$  coincidan.

Determine los tiempos de ejecución de cada algoritmo e implemente los tres, obteniendo datos reales de temporización para varios valores de  $N$ . Confirme sus análisis con los datos de temporización.



## Referencias

El problema de suma máxima de subsecuencia contigua está tomado de [5]. Las referencias [4], [5] y [6] muestran cómo optimizar programas para conseguir una mayor velocidad.

La búsqueda por interpolación fue sugerida por primera vez en [14] y fue analizada en [13]. Las referencias [1], [8] y [17] proporcionan un tratamiento más riguroso del análisis de algoritmos. La serie en tres partes [10], [11] y [12], recientemente actualizada, sigue siendo el principal trabajo de referencia sobre el tema. Los conocimientos matemáticos requeridos para un análisis de algoritmos más avanzado se proporcionan en [2], [3], [7], [15] y [16]. Un libro especialmente bueno sobre 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 MA, 1974.
2. M. O. Albertson y J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, New York, 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, MA, 1986.
6. J. L. Bentley, *More Programming Pearls*, Addison-Wesley, Reading, MA, 1988.
7. R. A. Brualdi, *Introductory Combinatorics*, North-Holland, Nueva York, 1977.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to Algorithms*, 3<sup>a</sup> ed., MIT Press, Cambridge, MA, 2010.
9. R. L. Graham, D. E. Knuth y O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1989.
10. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1997.
11. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1997.
12. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1998.
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<sup>a</sup> ed., John Wiley & Sons, Nueva York, 1984.
17. M. A. Weiss, *Data Structures and Algorithm Analysis in Java*, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 2007.

# Capítulo 6

## La API de Colecciones

Muchos algoritmos requieren el uso de una representación de datos apropiada para ser eficientes. Esta representación y las operaciones permitidas con ella se conocen con el nombre de *estructura de datos*. Cada estructura de datos permite la inserción arbitraria pero difiere en cuanto al modo en que permite acceder a los miembros del grupo. Algunas estructuras permiten accesos y borrados arbitrarios, mientras que otras imponen restricciones, como por ejemplo que se permita acceder solo al elemento del grupo más recientemente insertado o menos recientemente insertado.

Como parte de Java, se proporciona una librería de soporte conocida con el nombre de *API de Colecciones*. La mayor parte de la API de Colecciones reside en `java.util`. Esta API proporciona una serie de estructuras de datos. También proporciona algunos algoritmos genéricos, como por ejemplo de ordenación. La API de Colecciones hace un uso intensivo de la herencia.

Nuestro objetivo principal es describir, en términos generales, algunos ejemplos y aplicaciones de las estructuras de datos. Nuestro objetivo secundario es explicar los fundamentos de la API de Colecciones, para poder utilizarla en la Parte Tres. No trataremos sobre la teoría que subyace a una implementación eficiente de la API de Colecciones hasta la Parte Cuatro, en cuyo momento proporcionaremos implementaciones simplificadas de algunos elementos fundamentales de la API de Colecciones. Pero no representa ningún problema el retrasar las explicaciones sobre la API de Colecciones hasta después de que hayamos aprendido a utilizarla. No necesitamos saber *cómo* está implementado algo, siempre y cuando estemos seguros de que *está* implementado.

En este capítulo, veremos

- Estructuras de datos comunes, sus operaciones permitidas y sus tiempos de ejecución.
- Algunas aplicaciones de las estructura de datos.
- La organización de la API de Colecciones y su integración con el resto del lenguaje.

### 6.1 Introducción

Las estructuras de datos nos permiten conseguir un objetivo importante de la programación orientada a objetos: la reutilización de componentes. Las estructuras de datos descritas en esta sección (e implementadas posteriormente en la Parte Cuatro) tienen usos recurrentes. Una vez que cada estructura de datos ha sido implementada, se la puede emplear todas las veces que se quiera en diversas aplicaciones.

Una estructura de datos es una representación de datos y de las operaciones permitidas con dichos datos.

Las estructuras de datos nos permiten reutilizar los componentes.

eficiente estas operaciones. Esta estimación se basará a menudo en la analogía con aplicaciones no informáticas de la estructura de datos. Nuestro protocolo de alto nivel usualmente soporta tan solo un conjunto fundamental de operaciones básicas. Posteriormente, cuando describamos los fundamentos de cómo pueden implementarse las estructuras de datos (en general, existirán múltiples ideas que compiten entre sí), podremos centrarnos más fácilmente en los detalles algorítmicos independientes del lenguaje si restringimos el conjunto de operaciones a un conjunto mínimo fundamental.

La API de colecciones es una librería de estructuras de datos y algoritmos cuya disponibilidad está garantizada.

Una *estructura de datos* es una representación de los datos y de las operaciones permitidas con esos datos. Muchas de las estructuras de datos comunes, aunque no todas ellas, almacenan una colección de objetos y luego proporcionan métodos para añadir un nuevo objeto de la colección, eliminar un objeto existente o acceder a uno de los objetos contenidos en la misma.

En este capítulo, vamos a examinar algunas de las estructuras de datos fundamentales y sus aplicaciones. Utilizando un protocolo de alto nivel, escribiremos operaciones típicas que normalmente están soportadas por las estructuras de datos y describiremos también brevemente sus usos. Siempre que sea posible, daremos una estimación del coste de implementar de manera

eficiente estas operaciones. Esta estimación se basará a menudo en la analogía con aplicaciones no informáticas de la estructura de datos. Nuestro protocolo de alto nivel usualmente soporta tan solo un conjunto fundamental de operaciones básicas. Posteriormente, cuando describamos los fundamentos de cómo pueden implementarse las estructuras de datos (en general, existirán múltiples ideas que compiten entre sí), podremos centrarnos más fácilmente en los detalles algorítmicos independientes del lenguaje si restringimos el conjunto de operaciones a un conjunto mínimo fundamental.

Como ejemplo, la Figura 6.1 ilustra un protocolo genérico que muchas estructuras de datos tienden a seguir. No vamos a utilizar realmente este protocolo de forma directa en ningún código. Sin embargo, una jerarquía de estructuras de datos basada en la herencia podría utilizar esta clase como punto de partida.

Después, proporcionaremos una descripción de la interfaz que la API de Colecciones proporciona para estas estructuras de datos. No queremos decir en modo alguno que la API de Colecciones represente la mejor forma de hacer las cosas. Sin embargo, lo que sí representa es una librería de estructuras de datos y algoritmos cuya disponibilidad está garantizada. Su uso ilustra también algunos de los problemas fundamentales que tendremos que resolver en cuanto empecemos a tener en cuenta los aspectos teóricos.

Retrasaremos la consideración de cómo implementar las estructuras de datos de manera eficiente a la Parte Cuatro. En ese punto proporcionaremos, como parte del paquete `weiss.nonstandard`, algunas implementaciones alternativas de estructuras de datos que se ajustan a los protocolos simples desarrollados en este capítulo. También desarrollaremos una implementación para los componentes básicos de la API de Colecciones descritos en el capítulo, dentro del paquete `weiss.util`. Por tanto, estaremos separando la interfaz de la API de Colecciones (es decir, lo que hace, que es lo que describimos en este capítulo) de su implementación (es decir, cómo se hace, lo que abordaremos en la Parte Cuatro). Esta técnica –la separación de la interfaz de la implementación– forma parte del paradigma de orientación a objetos. El usuario de la estructura de datos solo necesita ver las operaciones disponibles, no la implementación. Recuerde que estos son los conceptos de encapsulación y ocultamiento de la información que se utilizan en la programación orientada a objetos.

El resto de este capítulo se organiza de la forma siguiente: en primer lugar, expondremos los fundamentos del *patrón iterador*, que se usa a lo largo de toda la API de Colecciones. Después, explicaremos la interfaz de los contenedores e iteradores de la API de Colecciones. A continuación describiremos algunos algoritmos de la API de Colecciones y, finalmente, examinaremos algunas otras estructura de datos, muchas de las cuales están soportadas en la API de Colecciones.

```
1 package weiss.nonstandard;
2
3 // protocolo SimpleContainer
4 public interface SimpleContainer<AnyType>
5 {
6     void insert( AnyType x );
7     void remove( AnyType x );
8     AnyType find( AnyType x );
9
10    boolean isEmpty();
11    void makeEmpty();
12 }
```

Figura 6.1 Protocolo genérico para muchas estructuras de datos.

## 6.2 El patrón iterador

La API de Colecciones hace uso de una técnica común conocida con el nombre de *patrón iterador*. Así que, antes de comenzar con las explicaciones sobre la API de Colecciones, vamos a examinar las ideas que subyacen al patrón iterador.

Considere el problema de imprimir los elementos de una colección. Tipicamente, la colección será una matriz, por lo que si asumimos que el objeto *v* es una matriz, podemos imprimir fácilmente su contenido con un código como el siguiente:<sup>1</sup>

```
for( int i = 0; i < v.length; i++ )
System.out.println( v[ i ] );
```

En este bucle, *i* es un objeto iterador, porque es el objeto que se emplea para controlar la iteración. Sin embargo, utilizar el entero *i* como iterador restringe el diseño: solo podemos almacenar la colección en una estructura de tipo matricial. Una alternativa más flexible consiste en diseñar una clase iteradora que encapsule una posición dentro de una colección. La clase iteradora proporciona métodos para recorrer la colección.

La clave es el concepto de programación de acuerdo con una interfaz: queremos que el código que realice el acceso al contenedor sea lo más independiente posible del tipo de contenedor. Esto se lleva a cabo empleando únicamente métodos que sean comunes para todos los contenedores y sus iteradores.

Hay muchos posibles diseños distintos de iterador. Si sustituimos *int i* por *IteratorType itr*, entonces el bucle anterior nos quedaría

Un objeto iterador controla la iteración a través de una colección.

Cuando programamos de acuerdo con una interfaz, escribimos código que utilice los métodos más abstractos posibles. Estos métodos se aplicarán a los tipos concretos actuales.

<sup>1</sup> El bucle *for* avanzado añadido en Java 5 constituye simplemente una sintaxis adicional. El compilador expande el bucle *for* avanzado para obtener el código que se muestra aquí.

```
for( itr = v.first( ); itr.isValid( ); itr.advance( ) )
    System.out.println( itr.getData( ) );
```

donde `first` nos da el primer elemento de la colección, `isValid` nos dice si una posición es válida, `advance` hace avanzar el iterador y `getData` nos da los datos correspondientes a la posición actual. Esto sugiere utilizar una clase iteradora que contenga métodos similares a `isValid`, `advance`, `getData`, etc.

Vamos a describir dos diseños, fuera de la API de Colecciones, que conducen al diseño de iterador de la API de Colecciones. Hablaremos de los detalles específicos de los iteradores de la API de Colecciones en la Sección 6.3.2, dejando las implementaciones para la Parte Cuatro.

### 6.2.1 Diseño básico de un iterador

**iterator** devuelve un iterador apropiado para la colección.

El primer diseño de iterador utiliza solo tres métodos. Se exige a la clase contenedora que proporcione un método `iterator`. Este método `iterator` devuelve un método apropiado para la colección. La clase iteradora solo tiene dos métodos, `hasNext` y `next`. `hasNext` devuelve `true` si la iteración no ha terminado aun. `next` devuelve el siguiente elemento de la colección (y en el proceso hace avanzar la posición actual). Esta interfaz de iterador es similar a la interfaz proporcionada en la API de Colecciones..

Para ilustrar la implementación de este diseño, vamos a esbozar la clase de colección utilizada y a proporcionar una clase iteradora; estas clases serán `MyContainer` y `MyContainerIterator`, respectivamente. Su uso se ilustra en la Figura 6.2. Los miembros de datos y el método `iterator` de `MyContainer` se describen en la Figura 6.3. Para simplificar las cosas vamos a omitir los constructores, y también los métodos tales como `add`, `size`, etc. La clase `ArrayList` de los capítulos anteriores puede utilizarse para proporcionar una implementación de estos métodos. También evitaremos el uso de genéricos por el momento.

El método `iterator` de la clase `MyContainer` simplemente devuelve un nuevo iterador; observe que el iterador debe tener información acerca del contenedor en el que está iterando. Por ello, el iterador se construye con una referencia a `MyContainer`.

```
1  public static void main( String [ ] args )
2  {
3      MyContainer v = new MyContainer( );
4
5      v.add( "3" );
6      v.add( "2" );
7
8      System.out.println( "Container contents: " );
9      MyContainerIterator itr = v.iterator( );
10     while( itr.hasNext( ) )
11         System.out.println( itr.next( ) );
12 }
```

Figura 6.2 Un método `main`, para ilustrar el diseño 1 de iterador.

```

1 package weiss.ds;
2
3 public class MyContainer
4 {
5     Object [ ] items;
6     int size;
7
8     public MyContainerIterator iterator( )
9         [ return new MyContainerIterator( this ); ]
10
11    // Otros métodos
12 }

```

Figura 6.3 La clase MyContainer, diseño 1.

La Figura 6.4 muestra `MyContainerIterator`. El iterador mantiene una variable (`current`) que representa la posición actual dentro del contenedor, así como una referencia al propio contenedor. La implementación del constructor y de los dos métodos es bastante simple. El constructor inicializa la referencia al contenedor, `hasNext` compara simplemente la posición actual con el tamaño del contenedor y `next` utiliza la posición actual para indexar la matriz (y luego hace avanzar la posición actual).

El iterador se construye con una referencia al contenedor a través del cual está iterando.

```

1 // Una clase iteradora que recorre un objeto MyContainer.
2
3 package weiss.ds;
4
5 public class MyContainerIterator
6 {
7     private int current = 0;
8     private MyContainer container;
9
10    MyContainerIterator( MyContainer c )
11        { container = c; }
12
13    public boolean hasNext( )
14        { return current < container.size; }
15
16    public Object next( )
17        { return container.items[ current++ ]; }
18 }

```

Figura 6.4 Implementación de MyContainerIterator, diseño 1.

Sería mejor como diseño incluir más funcionalidad dentro del iterador.

Una limitación en el diseño de este iterador es la interfaz relativamente limitada. Observe que es imposible hacer que el iterador vuelva al principio, y observe también que el método `next` acopla el acceso a un elemento y a la operación de avance del iterador. Este diseño de `next`, `hasNext` es lo que se utiliza en la API de Colecciones de Java; muchas personas creen que la API debería haber proporcionado un iterador más flexible. Ciertamente, es posible poner más funcionalidad en el iterador sin necesidad de modificar la implementación de la clase `MyContainer`. Por otro lado, el hacer esto no nos sirve para ilustrar ningún nuevo principio.

Observe que en la implementación de `MyContainer`, los miembros de datos `items` y `size` tienen visibilidad de paquete, en lugar de ser privados. Esta desafortunada relajación de las normas usuales de privacidad de los miembros de datos es necesaria, porque `MyContainerIterator` tiene que acceder a estos miembros de datos. De forma similar, el constructor de `MyContainerIterator` tiene visibilidad de paquete, para que pueda ser invocado por `MyContainer`.

### 6.2.2 Factorías e iteradores basados en herencia

El iterador diseñado hasta ahora consigue abstraer el concepto de iteración dentro de una clase iteradora. Esto es bastante conveniente, porque quiere decir que si la colección varía, pasando de ser una colección basada en matriz a alguna otra cosa, el código básico, como por ejemplo las líneas 10 y 11 de la Figura 6.2, no necesita cambiar.

Un esquema de iteración basado en herencia define una interfaz iteradora. Los clientes programan de acuerdo con esta interfaz.

Aunque esto constituye una mejora significativa, si cambiamos de una colección basada en matriz a alguna otra cosa, tendremos que modificar todas las declaraciones del iterador. Por ejemplo, en la Figura 6.2, tendríamos que cambiar la línea 9. En esta sección vamos a ver una alternativa.

Nuestra idea básica consiste en definir una interfaz `Iterator`. Correspondiéndose con cada tipo diferente de contenedor habrá un iterador que implemente el protocolo `Iterator`. En nuestro ejemplo, esto nos da tres clases: `MyContainer`, `Iterator` y `MyContainerIterator`. La relación existente es que `MyContainerIterator` *ES-UN Iterator*. La razón por la que hacemos esto es que cada contenedor puede ahora crear un iterador apropiado, pero pasarlo de vuelta en forma de un `Iterator` abstracto.

La Figura 6.5 muestra `MyContainer`. En la versión revisada de `MyContainer`, el método `iterator` devuelve una referencia a un objeto `Iterator`; el tipo real resultará ser un `MyContainerIterator`. Puesto que `MyContainerIterator` *ES-UN Iterator*, podemos hacer esto con total seguridad.

Un método `factoría` crea una nueva instancia concreta, pero la devuelve utilizando una referencia al tipo de interfaz.

Puesto que `iterator` crea y devuelve un nuevo objeto `Iterator`, cuyo tipo real es desconocido, se suele conocer con el nombre de *método factoría*. La interfaz iteradora, que sirve simplemente para acceder al protocolo mediante el que se puede acceder a todas las subclases de `Iterator`, se muestra en la Figura 6.6. Hay solo dos cambios en la implementación de `MyContainerIterator`, mostrada en la Figura 6.7, y ambos cambios afectan a la línea 5. En primer lugar, se ha añadido la cláusula `implements`. En segundo lugar, `MyContainerIterator` ya no necesita ser una clase pública.

En ningún lugar de `main` existe ninguna mención real del tipo del iterador.

La Figura 6.8 ilustra cómo se utilizan los iteradores basados en herencia. En la línea 9, vemos la declaración de `itr`: ahora es una referencia a un `Iterator`. En ninguna parte de `main` podemos encontrar ninguna mención del

```

1 package weiss.ds;
2
3 public class MyContainer
4 {
5     Object [ ] items;
6     int size;
7
8     public Iterator iterator( )
9         { return new MyContainerIterator( this ); }
10
11    // Otros métodos no mostrados.
12 }
```

**Figura 6.5** La clase MyContainer, diseño 2.

```

1 package weiss.ds;
2
3 public interface Iterator
4 {
5     boolean hasNext( );
6     Object next( );
7 }
```

**Figura 6.6** La interfaz Iterator, diseño 2.

```

1 // Una clase iteradora que recorre un objeto MyContainer.
2
3 package weiss.ds;
4
5 class MyContainerIterator implements Iterator
6 {
7     private int current = 0;
8     private MyContainer container;
9
10    MyContainerIterator( MyContainer c )
11        { container = c; }
12
13    public boolean hasNext( )
14        { return current < container.size; }
15
16    public Object next( )
17        { return container.items[ current++ ]; }
18 }
```

**Figura 6.7** Implementación de MyContainerIterator, diseño 2.

```

1  public static void main( String [ ] args )
2  {
3      MyContainer v = new MyContainer( );
4
5      v.add( "3" );
6      v.add( "2" );
7
8      System.out.println( "Container contents: " );
9      Iterator itr = v.iterator( );
10     while( itr.hasNext( ) )
11         System.out.println( itr.next( ) );
12 }

```

Figura 6.8 Un método `main`, para ilustrar el diseño 2 de iterador.

tipo real `MyContainerIterator`. El hecho de que exista un `MyContainerIterator` no es utilizado por ninguno de los clientes de la clase `MyContainer`. Este es un diseño muy adecuado e ilustra bastante bien la idea de ocultar una implementación y *programar de acuerdo con una interfaz*. La implementación puede hacerse todavía mejor utilizando clases anidadas, junto con una característica de Java conocida con el nombre de *clases internas*. Dejaremos esos detalles de implementación hasta el Capítulo 15.

## 6.3 La API de Colecciones: contenedores e iteradores

Esta sección describe los fundamentos básicos de los iteradores de la API de Colecciones y explica cómo interactúan con los contenedores. Sabemos que un iterador es un objeto que se utiliza para recorrer una colección de objetos. En la API de Colecciones, dicho tipo de colección está abstraído mediante la interfaz `Collection` y el iterador está abstraído mediante la interfaz `Iterator`.

Los iteradores de la API de Colecciones son en cierto modo poco flexibles, en el sentido de que solo proporcionan unas cuantas operaciones. Estos iteradores emplean un modelo de herencia que está descrito en la Sección 6.2.2.

### 6.3.1 La interfaz Collection

La interfaz `Collection` representa un grupo de objetos, conocidos con el nombre de *elementos*.

La interfaz `Collection` representa un grupo de objetos, conocidos con el nombre de *elementos*. Algunas implementaciones, como las listas, están desordenadas; otras, como los conjuntos y los mapas pueden estar ordenadas. Algunas implementaciones permiten duplicados, otras no. A partir de Java 5, la interfaz `Collection` y toda la API de Colecciones hacen uso de los genéricos. Todos los contenedores soportan las siguientes operaciones:

`boolean isEmpty( )`

Devuelve `true` si el contenedor no contiene ningún elemento y `false` en caso contrario.

```
int size( )
```

Devuelve el número de elementos del contenedor.

```
boolean add( AnyType x )
```

Añade el elemento *x* al contenedor. Devuelve *true* si esta operación tiene éxito y *false* en caso contrario (por ejemplo, si el contenedor no admite duplicados y *x* ya se encuentra dentro del contenedor).

```
boolean contains( Object x )
```

Devuelve *true* si *x* está en el contenedor y *false* en caso contrario.

```
boolean remove( Object x )
```

Elimina el elemento *x* del contenedor. Devuelve *true* si *x* se ha eliminado y *false* en caso contrario.

```
void clear( )
```

Hace que el contenedor pase a estar vacío.

```
Object [ ] toArray( )
```

```
<OtherType> OtherType [ ] toArray( OtherType [ ] arr )
```

Devuelve una matriz que contiene referencias a todos los elementos del contenedor.

```
java.util.Iterator<AnyType> iterator( )
```

Devuelve un *Iterator* que puede utilizarse para comenzar a recorrer todas las posiciones del contenedor.

Puesto que *Collection* es genérica, solo permite incluir en la colección objetos de un tipo específico (*AnyType*). Por tanto, el parámetro que hay que añadir es *AnyType*. El parámetro de *contains* y *remove* también debería ser de tipo *AnyType*; sin embargo, por razones de compatibilidad descendente es de tipo *Object*. Ciertamente, si se invocan *contains* o *remove* con un parámetro que no sea de tipo *AnyType*, el valor de retorno será *false*.

El método *toArray* devuelve una matriz que contiene referencias a los elementos de la colección. En algunos casos, puede ser más rápido manipular esta matriz que emplear un iterador para manipular la colección; sin embargo, el coste de hacer las cosas así es que se consume un espacio adicional. La ocasión más común en la que sería útil utilizar la matriz es cuando se está accediendo a la colección varias veces o mediante bucles anidados. Si se está accediendo a la matriz una única vez, secuencialmente, es poco probable que la utilización de *toArray* permita acelerar el procesamiento; en muchos casos, puede hacer que las cosas se ejecuten más lentamente, además de obligarnos a utilizar un espacio adicional.

Una versión de *toArray* devuelve la matriz en un tipo que es *Object[]*. La otra versión permite al usuario especificar el tipo de la matriz pasando un parámetro que contenga la matriz (y evitando así el coste de transformar los tipos durante la manipulación subsiguiente). Si la matriz no es lo suficientemente grande, se devuelve en su lugar una matriz del tamaño suficiente; sin embargo, esto no debería llegar nunca a ser necesario. El siguiente fragmento de código muestra cómo obtener una matriz a partir de una colección *Collection<String> coll*:

```
String [ ] theStrings = new String[ coll.size( ) ];
coll.toArray( theStrings );
```

A partir de este punto, la matriz puede manipularse mediante los mecanismos normales de indexación de matrices. Generalmente, la versión de `toArray` que resulta preferible es la de un parámetro, porque se evita el coste en términos de tiempo de ejecución asociado a las transformaciones de tipos. Finalmente, el método `iterator` devuelve un `Iterator<AnyType>`, que puede utilizarse para recorrer la colección.

La Figura 6.9 ilustra una especificación de la interfaz `Collection`. La interfaz `Collection` real en `java.util` contiene algunos métodos adicionales, pero nosotros nos conformaremos con este subconjunto. Por convenio, todas las implementaciones suministran tanto un constructor de cero parámetros que crea una colección vacía, como un constructor que crea una colección que hace referencia a los mismos elementos que otra colección. Esto es básicamente una operación de copia de una colección. Sin embargo, no existe ninguna sintaxis en el lenguaje que obligue a esta implementación en los constructores.

La interfaz `Collection` amplía `Iterable`, lo que quiere decir que se la puede aplicar el bucle `for` avanzado. Recuerde que la interfaz `Iterable` requiere la implementación de un método `iterator` que devuelve un `java.util.Iterator`. El compilador expandirá el bucle `for` avanzado, con las apropiadas llamadas a métodos de `java.util.Iterator`. En la línea 41, vemos el método `iterator` requerido por la interfaz `Iterable`. Sin embargo, hay que recalcar que estamos aprovechándonos de los tipos de retorno covariantes (Sección 4.1.11), porque el tipo de retorno del método `iterator` en la línea 41 es en realidad `weiss.util.Iterator`, que es nuestra propia clase que amplía `java.util.Iterator` y que se muestra en la Sección 6.3.2.

La API de Colecciones también codifica la noción de un *método de interfaz opcional*. Por ejemplo, suponga que queremos una colección inmutable: una vez construida, su estado no debería cambiar nunca. Las colecciones inmutables parecen ser incompatibles con `Collection`, ya que `add` y `remove` no tienen sentido para las colecciones inmutables.

Sin embargo, existe una solución: aunque el implementador de la colección inmutable debe implementar `add` y `remove`, no hay ninguna regla que diga que estos métodos deban hacer nada. En lugar de ello, el implementador puede simplemente generar una excepción de tiempo de ejecución `UnsupportedOperationException`. Al hacer esto, el implementador habrá implementado técnicamente la interfaz, al mismo tiempo que no proporciona en realidad `add` y `remove`.

Por convenio, los métodos de interfaz que documentan que son *opcionales* pueden implementarse de esta forma. Si la implementación decide no implementar un método opcional, entonces debería documentar ese hecho. Es responsabilidad del usuario cliente de la API verificar que el método esté implementado, consultando la documentación; y si el cliente ignora la documentación e invoca el método de todas formas, se genera la excepción de tiempo de ejecución `UnsupportedOperationException`, lo que indica un error de programación.

```

1 package weiss.util;
2
3 /**
4 * Interfaz Collection; la raíz de todas las colecciones 1.5.
5 */
6 public interface Collection<AnyType> extends Iterable<AnyType>, java.io.Serializable
7 {

```

Continúa

Figura 6.9 Una especificación de ejemplo de la interfaz `Collection`.

```
8  /**
9   * Devuelve el número de elementos de esta colección.
10  */
11 int size( );
12
13 /**
14  * Comprueba si esta colección está vacía.
15  */
16 boolean isEmpty( );
17
18 /**
19  * Comprueba si un cierto elemento se encuentra en la colección.
20  */
21 boolean contains( Object x );
22
23 /**
24  * Añade un elemento a esta colección.
25  */
26 boolean add( AnyType x );
27
28 /**
29  * Elimina un elemento de esta colección.
30  */
31 boolean remove( Object x );
32
33 /**
34  * Cambia el tamaño de esta colección, que pasa a ser cero.
35  */
36 void clear( );
37
38 /**
39  * Obtiene un objeto Iterator utilizado para recorrer la colección.
40  */
41 Iterator<AnyType> iterator( );
42
43 /**
44  * Obtiene una vista de la colección en forma de matriz primitiva.
45  */
46 Object [ ] toArray( );
47
48 /**
49  * Obtiene una vista de la colección en forma de matriz primitiva.
50  */
51 <OtherType> OtherType [ ] toArray( OtherType [ ] arr );
52 }
```

Figura 6.9 (Continuación).

Los métodos opcionales son hasta cierto punto controvertidos, pero no representan ninguna nueva adición al lenguaje. Se trata simplemente de un convenio.

Nosotros terminaremos implementando todos los métodos. Lo más interesante de esto es `Iterator`, que es un método factoría que crea y devuelve un objeto `Iterator`. Las operaciones que peden ser realizadas por un `Iterator` se describen en la Sección 6.3.2.

### 6.3.2 La interfaz Iterator

Un `Iterator` es un objeto que nos permite iterar a través de todos los objetos de una colección.

Como se describe en la Sección 6.2, un *iterador* es un objeto que permite iterar a través de los objetos de una colección. La técnica de utilización de una clase iteradora se explicó en el contexto de los vectores de solo lectura en la Sección 6.2.

La interfaz `Iterator` de la API de Colecciones es pequeña y solo contiene tres métodos:

`boolean hasNext()`

Devuelve `true` si hay más elementos que ver en esta iteración.

`AnyType next()`

Devuelve una referencia al siguiente objeto que todavía no haya sido visto por este iterador. El objeto pasará a considerarse visto y se hará avanzar, por tanto, el iterador.

`void remove()`

Elimina el último elemento visualizado mediante `next`. Este método solo puede invocarse una vez entre llamadas sucesivas a `next`.

La interfaz `Iterator` solo contiene tres métodos:  
`next`, `hasNext` y `remove`.

Cada colección define su propia implementación de la interfaz `Iterator`, en una clase que es invisible para los usuarios del paquete `java.util`.

Los iteradores también esperan que se utilice un contenedor estable. Un problema importante que surge en el diseño de contenedores e iteradores es el de decidir qué sucede si se modifica el estado de un contenedor mientras que está en marcha una iteración. La API de Colecciones adopta una posición estricta: cualquier modificación estructural externa del contenedor (adiciones, eliminaciones, etc.) provocará la generación de una excepción `ConcurrentModificationException` por parte de los métodos del iterador cuando se invoque uno de esos métodos. En otras palabras, si tenemos un iterador y luego se añade un objeto al contenedor, y posteriormente invocamos el método `next` del iterador, el iterador detectará que es inválido y `next` generará una excepción.

Esto significa que es imposible eliminar un objeto de un contenedor después de haberlo visualizado mediante un iterador, sin invalidar el iterador. Esta es una de las razones por las que

existe un método `remove` en la clase iteradora. Invocar el método `remove` del iterador hace que se elimine del contenedor el último objeto que hayamos visualizado. Esto hará que se invaliden todos los demás iteradores que estén visualizando este contenedor, pero no el iterador que haya realizado la operación `remove`. También es bastante probable que este método sea más eficiente que el método `remove` del contenedor, al menos para algunas

Los métodos de `Iterator` generan una excepción si su contenedor sufre alguna modificación estructural.

colecciones. Sin embargo, `remove` no puede invocarse dos veces seguidas. Además, `remove` preserva la semántica de `next` y `hasNext`, porque el siguiente elemento no visualizado de la iteración continuará siendo el mismo.

Esta versión de `remove` se enumera como un método opcional, por lo que el programador tendrá que comprobar si está implementado. El diseño de `remove` ha sido criticado por muchas personas como un diseño no demasiado elegante, pero nosotros lo vamos a emplear en un cierto punto del libro.

La Figura 6.10 proporciona una especificación de ejemplo de la interfaz `Iterator`. (Nuestra clase iteradora amplía la versión estándar contenida en `java.util`, para que el bucle `for` avanzado pueda funcionar.) Como ejemplo de utilización del `Iterator`, las rutinas de la Figura 6.11 imprimen cada uno de los elementos de un contenedor. Si el contenedor es un conjunto ordenado, sus elementos se imprimirán en orden. La primera implementación emplea un iterador directamente, mientras que la segunda implementación utiliza un bucle `for` avanzado. El bucle `for` avanzado es simplemente una sustitución del compilador. El compilador, en la práctica, lo que hace es generar la primera versión (con `java.util.Iterator`) a partir de la segunda.

```
1 package weiss.util;
2
3 /**
4  * Interfaz Iterator.
5 */
6 public interface Iterator<AnyType> extends java.util.Iterator<AnyType>
7 {
8     /**
9      * Comprueba si hay elementos sobre los que todavía no se haya iterado.
10     */
11    boolean hasNext( );
12
13    /**
14     * Obtiene el siguiente elemento (todavía no visualizado) de la colección.
15     */
16    AnyType next( );
17
18    /**
19     * Elimina el último elemento devuelto por next.
20     * Solo se puede invocar una vez después de invocar next.
21     */
22    void remove( );
23 }
```

Figura 6.10 Una especificación de ejemplo de `Iterator`.

```

1 // Imprime el contenido de Collection c (usando el iterador directamente)
2 public static <AnyType> void printCollection( Collection<AnyType> c )
3 {
4     Iterator<AnyType> itr = c.iterator( );
5     while( itr.hasNext( ) )
6         System.out.print( itr.next( ) + " " );
7     System.out.println( );
8 }
9
10 // Imprime el contenido de Collection c (usando el bucle for avanzado)
11 public static <AnyType> void printCollection( Collection<AnyType> c )
12 {
13     for( AnyType val : c )
14         System.out.print( val + " " );
15     System.out.println( );
16 }
```

Figura 6.11 Impresión del contenido de cualquier colección Collection.

## 6.4 Algoritmos genéricos

La API de Colecciones proporciona unos cuantos algoritmos de propósito general que operan sobre todos los contenedores. Se trata de métodos estáticos en la clase `Collections` (observe que se trata

de una clase distinta de la interfaz `Collection`). También hay algunos métodos estáticos en la clase `Arrays` que manipulan matrices (ordenación, búsquedas, etc.). La mayoría de estos métodos están sobrecargados –una versión genérica y otra para cada uno de los tipos primitivos, excepto `boolean`.

Vamos a examinar solo unos cuantos de los algoritmos con la intención de mostrar solo las ideas generales utilizadas en la API de Colecciones, al mismo tiempo que documentamos los algoritmos específicos que se emplearán en la Parte Tres.

Algunos de los algoritmos hacen uso de objetos función. En consecuencia, el material de la Sección 4.8 es un requisito esencial para poder entender esta sección.

La clase `Collections` contiene un conjunto de métodos estáticos que operan sobre objetos `Collection`.

El material de la Sección 4.8 es un prerequisito esencial para poder entender esta sección.

### 6.4.1 Objetos función Comparator

Muchas clases y rutinas de la API de Colecciones requieren la capacidad de ordenar objetos. Hay dos formas de hacer esto. Una posibilidad es que los objetos implementen la interfaz `Comparable` y proporcionen un método `compareTo`. La otra posibilidad es que la función de comparación se incruste como método `compare` en un objeto que implemente la interfaz `Comparator`. `Comparator` está definida en `java.util`; en la Figura 4.39 ya mostramos una implementación de ejemplo, la cual repetimos en la Figura 6.12.

```
1 package weiss.util;
2
3 /**
4  * Interfaz del objeto función Comparator.
5  */
6 public interface Comparator<AnyType>
7 {
8     /**
9      * Devuelve el resultado de comparar lhs y rhs.
10     * @param lhs primer objeto.
11     * @param rhs segundo objeto.
12     * @return < 0 si lhs es menor que rhs,
13     *         0 si lhs es igual que rhs,
14     *         > 0 si lhs es mayor que rhs.
15     * @throws ClassCastException si los objetos no se pueden comparar.
16     */
17     int compare( AnyType lhs, AnyType rhs ) throws ClassCastException;
18 }
```

Figura 6.12 La interfaz `Comparator`, originalmente definida en `java.util` y reescrita para el paquete `weiss.util`.

#### 6.4.2 La clase Collections

Aunque no vamos a hacer uso de la clase `Collections` en este texto, esta tiene dos métodos que ilustran cómo están escritos los algoritmos genéricos de la API de Colecciones. Escribiremos estos métodos en la implementación de la clase `Collections` mostrada en las Figuras 6.13 y 6.14.

La Figura 6.13 comienza ilustrando la técnica común de declarar un constructor privado en aquellas clases que solo contienen métodos estáticos. Esto impide la instantiación de la clase. Se continua proporcionando el método `reverseOrder`. Este es un método factoría que devuelve un `Comparator` que proporciona el inverso de la ordenación natural para objetos `Comparable`. El objeto devuelto, creado en la línea 20, es una instancia de la clase `ReverseComparator` escrita en las líneas 23 a 29. En la clase `ReverseComparator`, utilizamos el método `compareTo`. Este es un ejemplo del tipo de código que podría implementarse mediante una clase anónima. Tenemos otra declaración similar para el comparador predeterminado; puesto que la API estándar no proporciona un método público para devolver esto, declaramos nuestro método con visibilidad de paquete.

La Figura 6.14 ilustra el método `max`, que devuelve el elemento más grande dentro de cualquier colección `Collection`. La llamada al método de un solo parámetro `max` invoca al método `max` de dos parámetros suministrando el comparador predeterminado. La sintaxis extraña en la lista de parámetros de tipo se emplea para garantizar que el mecanismo de borrado de tipos en `max` genere `Object` (en lugar de `Comparable`). Esto es importante porque las versiones anteriores de Java empleaban `Object` como tipo de retorno, y queremos garantizarnos la compatibilidad descendente. El método `max` de dos parámetros combina el patrón iterador con el patrón de objeto función para recorrer la colección, y en la línea 75 utiliza llamadas al objeto función para actualizar el elemento máximo.

```
1 package weiss.util;
2
3 /**
4 * Clase sin instancias que contiene métodos estáticos que operan sobre colecciones.
5 */
6 public class Collections
7 {
8     private Collections( )
9     {
10    }
11
12    /*
13     * Devuelve un comparador que impone el inverso de la ordenación
14     * predeterminada a una colección de objetos que
15     * implementa la interfaz Comparable.
16     * @return el comparador.
17    */
18    public static <AnyType> Comparator<AnyType> reverseOrder( )
19    {
20        return new ReverseComparator<AnyType>();
21    }
22
23    private static class ReverseComparator<AnyType> implements Comparator<AnyType>
24    {
25        public int compare( AnyType lhs, AnyType rhs )
26        {
27            return - ((Comparable)lhs).compareTo( rhs );
28        }
29    }
30
31    static class DefaultComparator<AnyType extends Comparable<? super AnyType>>
32        implements Comparator<AnyType>
33    {
34        public int compare( AnyType lhs, AnyType rhs )
35        {
36            return lhs.compareTo( rhs );
37        }
38    }
}
```

---

Figura 6.13 La clase Collections, (parte 1): constructor privado y reverseOrder.

```

39  /**
40   * Devuelve el objeto máximo de la colección,
41   * utilizando la ordenación predeterminada
42   * @param coll la colección.
43   * @return el objeto máximo.
44   * @throws NoSuchElementException si coll está vacía.
45   * @throws ClassCastException si los objetos de la colección
46   * no pueden compararse.
47   */
48 public static <AnyType extends Object & Comparable<? super AnyType>>
49 AnyType max( Collection<? extends AnyType> coll )
50 {
51     return max( coll, new DefaultComparator<AnyType>() );
52 }
53
54 /**
55  * Devuelve el objeto máximo de la colección.
56  * @param coll la colección.
57  * @param cmp el comparador.
58  * @return el objeto máximo.
59  * @throws NoSuchElementException si coll está vacía.
60  * @throws ClassCastException si los objetos de la colección
61  * no pueden compararse.
62  */
63 public static <AnyType>
64 AnyType max( Collection<? extends AnyType> coll, Comparator<? super AnyType> cmp)
65 {
66     if( coll.size( ) == 0 )
67         throw new NoSuchElementException( );
68
69     Iterator<? extends AnyType> itr = coll.iterator( );
70     AnyType maxValue = itr.next( );
71
72     while( itr.hasNext( ) )
73     {
74         AnyType current = itr.next( );
75         if( cmp.compare( current, maxValue ) > 0 )
76             maxValue = current;
77     }
78
79     return maxValue;
80 }
81 }

```

Figura 6.14 La clase Collections (parte 2): max.

### 6.4.3 Búsqueda binaria

La implementación de la búsqueda binaria en la API de Colecciones es el método estático `Arrays.binarySearch`. Actualmente existen siete versiones sobrecargadas –una por cada uno de los tipos primitivos salvo `boolean`, más otras dos versiones sobrecargadas que funcionan sobre objetos de tipo `Object` (una funciona con un comparador, mientras que la otra utiliza el comparador predeterminado). Implementaremos las versiones `Object` (utilizando genéricos); las otras siete versiones se pueden realizar cortando y pegando.

`binarySearch` utiliza una búsqueda binaria y devuelve el índice del elemento para el que se ha encontrado una correspondencia, o un número negativo si el elemento no ha podido ser encontrado.

Como siempre, para poder utilizar la búsqueda binaria la matriz debe estar ordenada; si no lo está, los resultados no estarán definidos (verificar que la matriz está ordenada destruiría la cota de tiempo logarítmico de la operación).

Si la búsqueda del elemento tiene éxito, se devuelve el índice de la correspondencia encontrada. Si la búsqueda no tiene éxito, determinamos la primera posición que contiene un elemento mayor, añadimos 1 a esta posición y luego devolvemos el negado de dicho valor. Por tanto, el valor de retorno será siempre negativo, porque será como máximo  $-1$  (que se obtendrá si el elemento que estamos buscando es más pequeño que todos los restantes elementos) y será como mínimo  $-a.length - 1$  (que se obtendrá si el elemento que estamos buscando es mayor que todos los restantes elementos).

La implementación se muestra en la Figura 6.15. Como sucedía en las rutinas `max`, el método `binarySearch` de dos parámetros invoca el método `binarySearch` de tres parámetros (véanse las líneas 17 y 18). La rutina de búsqueda binaria de tres parámetros se asemeja a la implementación de la Figura 5.12. En Java 5, la versión de dos parámetros no utiliza genéricos. En lugar de ello, todos los tipos son `Object`. Pero nuestra implementación genérica parece tener más sentido. La versión de tres parámetros es genérica en Java 5. Utilizaremos el método `binarySearch` en la Sección 10.1.

### 6.4.4 Ordenación

La clase `Arrays` contiene un conjunto de métodos estáticos que operan sobre matrices.

La API de Colecciones proporciona un conjunto de métodos `sort` sobre cargados en la clase `Arrays`. Basta con pasarles una matriz de valores primitivos o una matriz de `Object` que implemente `Comparable`, o una matriz de `Object` y un `Comparator`. No hemos proporcionado ningún método `sort` en nuestra clase `Arrays`.

```
void sort( Object [ ] arr )
```

Recoloca los elementos de la matriz para que estén ordenados, utilizando la ordenación natural.

```
void sort( Object [ ] arr, Comparator cmp )
```

Recoloca los elementos de la matriz para que estén ordenados, utilizando la ordenación especificada por el comparador.

En Java 5, estos métodos se han escrito como métodos genéricos. Se exige que los algoritmos genéricos de ordenación se ejecuten en un tiempo  $O(N \log N)$ .

```
1 package weiss.util;
2
3 /**
4  * Clase sin instancias que contiene métodos
5  * estáticos para manipular matrices.
6 */
7 public class Arrays
8 {
9     private Arrays( ) { }
10
11    /**
12     * Buscar en la matriz ordenada arr utilizando el comparador predeterminado.
13     */
14    public static <AnyType extends Comparable<AnyType>> int
15        binarySearch( AnyType [ ] arr, AnyType x )
16    {
17        return binarySearch( arr, x,
18            new Collections.DefaultComparator<AnyType>( ) );
19    }
20
21    /**
22     * Realiza una búsqueda en la matriz ordenada arr usando un comparador.
23     * Si arr no está ordenada, los resultados no están definidos.
24     * @param arr la matriz en la que hay que buscar.
25     * @param x el objeto que hay que buscar.
26     * @param cmp el comparador.
27     * @return si se encuentra x, devuelve el índice en el que se ha
28     * encontrado. En caso contrario, el valor de retorno es un número
29     * negativo igual a -( p + 1 ), donde p es la primera posición mayor
30     * que x. Este valor puede ir desde -1 hasta -(arr.length+1).
31     * @throws ClassCastException si los elementos no son comparables.
32     */
33    public static <AnyType> int
34        binarySearch( AnyType [ ] arr, AnyType x, Comparator<? super AnyType> cmp )
35    {
36        int low = 0, mid = 0;
37        int high = arr.length;
38
39        while( low < high )
40        {
41            mid = ( low + high ) / 2;
42            if( cmp.compare( x, arr[ mid ] ) > 0 )
43                low = mid + 1;
44            else
45                high = mid;
46        }
47        if( low == arr.length || cmp.compare( x, arr[ low ] ) != 0 )
48            return - ( low + 1 );
49        return low;
50    }
51 }
```

Figura 6.15 Implementación del método binarySearch en la clase Arrays.

## 6.5 La interfaz List

Una *lista* es una colección de elementos en la que los elementos tienen una posición.

La interfaz List amplía la interfaz Collection y abstrae la noción de posición.

Una *lista* es una colección de elementos en la que los elementos tienen una posición. El ejemplo más obvio de lista es una matriz. En una matriz, los elementos se colocan en las posiciones 0, 1, etc.

La interfaz List amplia la interfaz Collection y abstrae la noción de posición. La interfaz en java.util añade numerosos métodos a la interfaz Collection. Nosotros nos conformaremos con añadir los tres mostrados en la Figura 6.16.

Los primeros dos métodos son get y set, que son similares a los métodos que ya hemos visto en ArrayList. El tercer método devuelve un iterador más flexible, el ListIterator.

### 6.5.1 La interfaz ListIterator

Como se muestra en la Figura 6.17, ListIterator es simplemente como un Iterator, salvo porque es bidireccional. Por tanto, podemos tanto avanzar como retroceder. Debido a esto, hay que proporcionar al método factoría listIterator que lo crea un valor que sea lógicamente igual al

```
1 package weiss.util;
2
3 /**
4  * Interfaz List. Contiene mucho menos que la de java.util
5  */
6 public interface List<AnyType> extends Collection<AnyType>
7 {
8     AnyType get( int idx );
9     AnyType set( int idx, AnyType newVal );
10
11    /**
12     * Obtiene un objeto ListIterator que se usa para recorrer
13     * la colección bidireccionalmente.
14     * @return un iterador posicionado
15     * antes del elemento solicitado.
16     * @param pos el índice para iniciar el iterador.
17     * Utilice size() para realizar un recorrido inverso completo.
18     * Utilice 0 para realizar un recorrido completo en dirección normal.
19     * @throws IndexOutOfBoundsException si pos no está
20     * entre 0 y size(), ambos incluidos.
21     */
22     ListIterator<AnyType> listIterator( int pos );
23 }
```

---

Figura 6.16 Una interfaz List de ejemplo.

número de elementos que ya hayan sido visitados en la dirección normal de avance. Si este valor es cero, el `ListIterator` se inicializa en la primera posición, igual que un `Iterator`. Si este valor tiene el tamaño del objeto `List`, el iterador se inicializa como si ya hubiera procesado todos los elementos en la dirección normal de avance. Por tanto, en este estado, `hasNext` devuelve `false`, pero podemos utilizar `hasPrevious` y `previous` para recorrer la lista en sentido inverso.

`ListIterator` es una versión bidireccional de `Iterator`.

La Figura 6.18 ilustra que podemos utilizar `itr1` para recorrer una lista en la dirección normal de avance y, después de alcanzar el final podemos volver a recorrer la lista hacia atrás. También ilustra `itr2`, que se coloca al final y que simplemente procesa la lista `ArrayList` en sentido inverso. Finalmente, se ilustra el bucle `for` avanzado.

Una dificultad con `ListIterator` es que la semántica de `remove` debe ser modificada ligeramente. La nueva semántica es que `remove` elimina del objeto `List` el último objeto devuelto como resultado de la invocación de `next` o de `previous`, pudiéndose invocar `remove` una única vez entre

```

1 package weiss.util;
2
3 /**
4  * Interfaz ListIterator para la interfaz List.
5  */
6 public interface ListIterator<AnyType> extends Iterator<AnyType>
7 {
8     /**
9      * Comprueba si hay más elementos en la colección al realizar
10     * la iteración en sentido inverso.
11     * @return true si hay más elementos en la colección al
12     * recorrerla en sentido inverso.
13     */
14     boolean hasPrevious();
15
16     /**
17      * Obtiene el elemento anterior de la colección.
18      * @return el elemento anterior (todavía no visualizado) de la colección
19      * al recorrerla en sentido inverso.
20     */
21     AnyType previous();
22
23     /**
24      * Elimina el último elemento devuelto por next o previous.
25      * Solo puede invocarse una única vez después de next o previous.
26     */
27     void remove();
28 }
```

Figura 6.17 Interfaz `ListIterator` de ejemplo.

llamadas sucesivas a `next` o `previous`. Para sustituir la salida *javadoc* generada para `remove`, el método `remove` se incluye en la interfaz `ListIterator`.

La interfaz de la Figura 6.17 es solo una interfaz parcial. Hay algunos otros métodos adicionales en `ListIterator` que no vamos a explicar en este libro, pero que se emplean como ejercicios. Entre estos métodos se incluyen `add` y `set`, que permiten al usuario realizar cambios en la lista `List` en la posición actualmente ocupada por el iterador.

```
1 import java.util.ArrayList;
2 import java.util.ListIterator;
3
4 class TestArrayList
5 {
6     public static void main( String [ ] args )
7     {
8         ArrayList<Integer> lst = new ArrayList<Integer>( );
9         lst.add( 2 ); lst.add( 4 );
10        ListIterator<Integer> itr1 = lst.listIterator( 0 );
11        ListIterator<Integer> itr2 = lst.listIterator( lst.size( ) );
12
13        System.out.print( "Forward: " );
14        while( itr1.hasNext( ) )
15            System.out.print( itr1.next( ) + " " );
16        System.out.println( );
17
18        System.out.print( "Backward: " );
19        while( itr1.hasPrevious( ) )
20            System.out.print( itr1.previous( ) + " " );
21        System.out.println( );
22
23        System.out.print( "Backward: " );
24        while( itr2.hasPrevious( ) )
25            System.out.print( itr2.previous( ) + " " );
26        System.out.println( );
27
28        System.out.print( "Forward: " );
29        for( Integer x : lst )
30            System.out.print( x + " " );
31        System.out.println( );
32    }
33 }
```

---

Figura 6.18 Un programa de ejemplo que ilustra la iteración bidireccional.

### 6.5.2 La clase LinkedList

Hay dos implementaciones básicas de `List` en la API de Colecciones. Una implementación es `ArrayList`, que ya hemos visto anteriormente. La otra es la lista enlazada `LinkedList`, que almacena internamente los elementos de una forma distinta de la que emplea `ArrayList`, lo que conduce a una serie de compromisos en lo que se refiere al rendimiento. Una tercera versión es `Vector`, que es como `ArrayList`, pero proviene de una librería más antigua y que está presente fundamentalmente por compatibilidad con el código *heredado* (antiguo). La utilización de `Vector` ya no es común.

`ArrayList` puede resultar apropiada si las inserciones se efectúan solo al final de la matriz (utilizando `add`), por las razones explicadas en la Sección 2.4.3. `ArrayList` duplica la capacidad interna de la matriz si una inserción al final de la misma hace que se exceda la capacidad interna. Aunque esto nos da un buen rendimiento O mayúscula, especialmente si añadimos un constructor que permita al llamarante sugerir la capacidad de la matriz interna, `ArrayList` es una elección poco adecuada si las inserciones no se hacen al final, porque entonces nos veremos obligados a desplazar elementos para hacer sitio a los elementos nuevos.

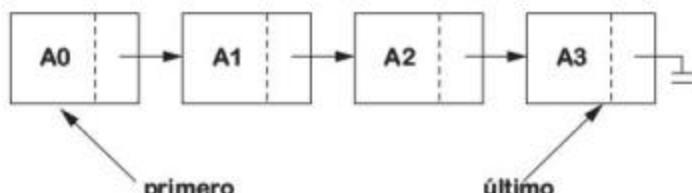
En una lista enlazada, los elementos se almacenan de forma no contigua, en lugar de emplearse la matriz contigua usual. Para hacer esto, almacenamos cada objeto en un nodo que contiene el objeto y una referencia al siguiente nodo de la lista, como se muestra en la Figura 6.19. En este escenario, lo que hacemos es mantener referencias al primer y último nodo de la lista.

Para ser concretos, un nodo típico tendría el aspecto siguiente:

```
class ListNode
{
    Object data; // Algun elemento
    ListNode next;
}
```

En cualquier punto, podemos añadir un nuevo elemento `x` al final haciendo lo siguiente:

```
last.next = new ListNode(); // Añadir un nuevo ListNode
last = last.next;           // Ajustar la referencia al ultimo nodo
last.data = x;              // Colocar x en el nodo
last.next = null;           // Es el ultimo; ajustar la ref. al siguiente.
```



La clase `LinkedList`  
implementa una lista  
enlazada.

La lista enlazada se utiliza  
para evitar tener que  
mover grandes cantidades  
de datos. Almacena los  
elementos con un gasto  
adicional de una referencia  
por cada elemento.

Figura 6.19 Una lista enlazada simple.

Ahora ya no podemos encontrar un único elemento arbitrario con un único acceso. En lugar de ello, tendremos que recorrer la lista. Esto es similar a la diferencia entre acceder a un elemento que se encuentra en un disco compacto (un acceso) o en una cinta (un acceso secuencial). Aunque parezca que esto hace que las listas enlazadas sean menos atractivas que las matrices, siguen teniendo ventajas. En primer lugar, una inserción en mitad de la lista no exige mover todos los elementos situados después del punto de inserción. Los movimientos de datos son muy caros en la práctica, y la lista enlazada permite realizar una operación de inserción con solo un número constante de instrucciones de asignación.

El compromiso básico entre `ArrayList` y `LinkedList` es que `get` no es eficiente para `LinkedList`, mientras que la inserción y eliminación en mitad de un contenedor están soportadas de manera más eficiente por `LinkedList`.

El acceso a la lista se realiza a través de una clase iteradora.

Comparando `ArrayList` y `LinkedList`, vemos que las inserciones y borrados en la mitad de la secuencia son ineficientes en `ArrayList` pero pueden ser eficientes en una lista `LinkedList`. Sin embargo, un `ArrayList` permite un acceso directo utilizando un índice, mientras que una `LinkedList` no debería permitirlo. En realidad, lo cierto es que en la API de Colecciones, `get` y `set` forman parte de la interfaz `List`, por lo que `LinkedList` soporta estas operaciones, aunque lo hace de una forma muy lenta. Por tanto, `LinkedList` puede utilizarse siempre, a menos que haga falta una indexación eficiente. `ArrayList` puede seguir siendo una mejor elección si las inserciones solo pueden producirse al final de la lista.

Para acceder a los elementos de la lista, necesitamos una referencia al nodo correspondiente en lugar de un índice. La referencia al nodo estará normalmente oculta dentro de una clase iteradora.

Puesto que `LinkedList` realiza las operaciones de adición y borrado de forma más eficiente, tiene más operaciones que `ArrayList`. Algunas de las operaciones adicionales disponibles para `LinkedList` son las siguientes:

`void addLast( AnyType element )`

Añade `element` al final de esta `LinkedList`.

`void addFirst( AnyType element )`

Añade `element` al principio de esta `LinkedList`.

`AnyType getFirst( )`

`AnyType element( )`

Devuelve el primer elemento de esta `LinkedList`. `element` se añadió en Java 5.

`AnyType getLast( )`

Devuelve el último elemento de esta `LinkedList`.

`AnyType removeFirst( )`

`AnyType remove( )`

Elimina y devuelve el primer elemento de esta `LinkedList`. `remove` se añadió en Java 5.

`AnyType removeLast( )`

Elimina y devuelve el último elemento de esta `LinkedList`.

Implementaremos la clase `LinkedList` en la Parte Cuatro.

### 6.5.3 Tiempo de ejecución para las distintas listas

En la Sección 6.5.2 vimos que, para algunas operaciones, `ArrayList` representa una mejor elección que `LinkedList`, mientras que para otras operaciones sucede justo lo contrario. En esta sección, en lugar de analizar los tiempos de ejecución de manera informal, vamos a analizarlos en términos de  $O$  mayúscula. Inicialmente, nos concentraremos en el siguiente subconjunto de operaciones:

- `add` (al final)
- `add` (al principio)
- `remove` (al final)
- `remove` (al principio)
- `get` y `set`
- `contains`

#### Costes de `ArrayList`

Para `ArrayList`, la adición al final significa simplemente colocar un elemento en la siguiente posición de la matriz, e incrementar el tamaño actual. Ocasionalmente, tendremos que redimensionar la capacidad de la matriz, pero como esta es una operación extremadamente rara, se puede argumentar con cierta razón que no afecta al tiempo de ejecución. Por tanto, el coste de añadir al final de una lista `ArrayList` no depende del número de elementos almacenados en la lista y es, por tanto,  $O(1)$ .

De forma similar, eliminar del final del `ArrayList` implica simplemente reducir el tamaño actual y es también  $O(1)$ , `get` y `set` en `ArrayList` se convierten en operaciones de indexación de la matriz, que normalmente se asume que requieren un tiempo constante y son, por tanto, operaciones  $O(1)$ .

No hace falta decir que, cuando hablamos del coste de una única operación sobre una colección, resulta difícil concebir algo que sea mejor que  $O(1)$  (es decir, un tiempo constante) por cada operación. Para poder obtener un rendimiento mejor, haría falta que las operaciones fueran cada vez más rápidas a medida que la colección aumentara de tamaño, lo que sería bastante extraño.

Sin embargo, no todas las operaciones son  $O(1)$  en un `ArrayList`. Como hemos visto, si añadimos al principio del `ArrayList`, entonces cada elemento de la lista debe ser desplazado una posición de índice hacia arriba. Por tanto, si hubiera  $N$  elementos en el `ArrayList`, la adición de un elemento al principio sería una operación  $O(N)$ . De forma similar, eliminar un elemento del principio del `ArrayList` requiere desplazar todos los elementos una posición de índice hacia abajo, lo que también es una operación  $O(N)$ . Y una comprobación `contains` sobre un `ArrayList` es una operación  $O(N)$ , porque potencialmente tendremos que examinar cada elemento del `ArrayList`.

No hace falta decir que  $O(N)$  por cada operación no es tan conveniente como  $O(1)$  por cada operación. De hecho, cuando consideramos que la operación `contains` es  $O(N)$  que consiste básicamente en una búsqueda exhaustiva, podríamos argumentar que  $O(N)$  por cada operación para una operación básica con una colección parece el peor resultado posible.

#### Costes de `LinkedList`

Si examinamos las operaciones de `LinkedList`, podemos ver que añadir un elemento al principio o al final es una operación  $O(1)$ . Para añadir al principio, simplemente creamos un nuevo nodo y lo añadimos al principio actualizando `first`. Esta operación no depende de conocer cuántos nodos

posteriores contiene la lista. Para añadir un elemento al final, simplemente creamos un nuevo nodo y lo añadimos al final, ajustando `last`.

Eliminar el primer elemento de la lista enlazada, es una operación  $O(1)$ , porque nos limitamos a hacer avanzar `first` al siguiente nodo de la lista. Eliminar el último elemento de la lista enlazada parece ser también  $O(1)$ , ya que necesitamos mover `last` al último nodo y actualizar en este nodo el enlace que apunta al nodo siguiente. Sin embargo, llegar hasta el penúltimo nodo no es tan fácil en una lista enlazada, como se deduce de la Figura 6.19.

En una lista enlazada clásica, en la que cada nodo almacena un enlace al nodo siguiente, el disponer de un enlace al último nodo no nos proporciona ninguna información acerca del penúltimo nodo. La idea obvia de mantener un enlace al penúltimo nodo no funciona, porque ese tercer enlace también tendría que ser actualizado durante una operación de borrado. En lugar de ello, lo que haremos será obligar a cada nodo a mantener un enlace al nodo anterior de la lista. Esto se muestra en la Figura 6.20 y se conoce con el nombre de *lista doblemente enlazada*.

En una lista doblemente enlazada, las operaciones `add` y `remove` en cualquiera de los dos extremos requieren un tiempo  $O(1)$ . Como sabemos, existe un compromiso, sin embargo, porque `get` y `set` no son eficientes con este tipo de lista. En lugar de acceder directamente a través de una matriz tenemos que seguir una serie de enlaces. En algunos casos, podremos optimizar ese recorrido comenzando por el final en lugar de por el principio, pero si la operación `get` o `set` se refiere a un elemento que está situado cerca de la parte central de la lista, necesitará un tiempo  $O(N)$ .

`contains` en una lista enlazada es igual que en `ArrayList`: el algoritmo básico es una búsqueda secuencial que termina examinando potencialmente cada uno de los elementos, y por tanto se trata de una operación  $O(N)$ .

## Comparación de los costes de `ArrayList` y `LinkedList`

La Figura 6.21 compara los tiempos de ejecución de las operaciones simples en `ArrayList` y `LinkedList`.

Para ver la diferencia entre la utilización de `ArrayList` y `LinkedList` en una rutina de mayor tamaño, vamos a echar un vistazo a algunos métodos que operan sobre un objeto `List`. En primer lugar, suponga que construimos un objeto `List` añadiendo elementos al final del mismo.

```
public static void makeList1( List<Integer> lst, int N )
{
    lst.clear();
    for( int i = 0; i < N; i++ )
        lst.add( i );
}
```

Independientemente de si se pasa como parámetro un `ArrayList` o una `LinkedList`, el tiempo de ejecución de `makeList1` es  $O(N)$ , porque cada llamada a `add`, estando situados al final de la

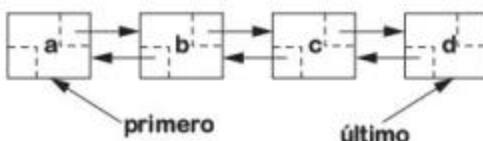


Figura 6.20 Una lista doblemente enlazada.

	ArrayList	LinkedList
add/remove al final	$O(1)$	$O(1)$
add/remove al principio	$O(N)$	$O(1)$
get/set	$O(1)$	$O(N)$
contains	$O(N)$	$O(N)$

Figura 6.21 Costes de cada una de las operaciones sencillas para ArrayList y LinkedList.

lista, requiere un tiempo constante. Por otro lado, si construimos una List añadiendo elementos al principio,

```
public static void makeList2( List<Integer> lst, int N )
{
    lst.clear();
    for( int i = 0; i < N; i++ )
        lst.add( 0, i );
}
```

el tiempo de ejecución será  $O(N)$  para una LinkedList, pero  $O(N^2)$  para un ArrayList, porque en un ArrayList, la operación de adición de elementos al principio de la lista es una operación  $O(N)$ .

La siguiente rutina trata de calcular la suma de los números de una lista:

```
public static int sum( List<Integer> lst )
{
    int total = 0;
    for( int i = 0; i < N; i++ )
        total += lst.get( i );
}
```

Aquí, el tiempo de ejecución es  $O(N)$  para un ArrayList, pero  $O(N^2)$  para una LinkedList, ya que en una LinkedList, las llamadas a get son operaciones  $O(N)$ . En lugar de ello, utilice un bucle for avanzado, que hará que el tiempo de ejecución sea  $O(N)$  para cualquier List, porque el iterador podrá avanzar de manera eficiente de un elemento al siguiente.

#### 6.5.4 Eliminación y adición de elementos en mitad de una colección List

La interfaz List contiene dos operaciones:

```
void add( int idx, AnyType x );
void remove( int idx );
```

que permiten la adición de un elemento en un índice especificado y la eliminación de un elemento en un índice especificado. Para un ArrayList, estas operaciones son en general  $O(N)$ , debido al desplazamiento de elementos requerido.

Para una `LinkedList`, en principio cabe esperar que, si sabemos dónde se está realizando el cambio, entonces deberíamos poder llevarlo a cabo de forma eficiente dividiendo los enlaces de la lista enlazada. Por ejemplo, es fácil ver que, en principio, eliminar un único nodo en una lista doblemente enlazada requiere modificar algunos enlaces en los nodos posterior y anterior. Sin embargo, estas operaciones siguen siendo  $O(N)$  en una `LinkedList` porque hace falta un tiempo  $O(N)$  para encontrar el nodo.

Esta es precisamente la razón por la que `Iterator` proporciona un método `remove`. La idea es que, a menudo, un elemento solo se elimina después de haberlo examinado y de haber decidido que podemos descartarlo. Esto es similar a la idea de recoger elementos del suelo: a medida que buscamos por el suelo, si vemos un elemento, lo tomamos inmediatamente, porque ya estamos ahí.

Como ejemplo, vamos a proporcionar una rutina que elimine todos los elementos con valor par de una lista. Por tanto, si la lista contiene 6, 5, 1, 4, 2, entonces después de invocar el método contendrá 5, 1.

Hay varias posibles ideas para un algoritmo que elimine elementos de la lista a medida que los va encontrando. Por supuesto, una idea consiste en construir una nueva lista que contenga todos los números impares y luego borrar la lista original y copiar de nuevo en ella esos números impares. Pero nos interesa más escribir una versión limpia que evite hacer una copia y que lo que haga sea borrar los elementos de la lista a medida que los va encontrando.

Esta estrategia resulta, casi con seguridad, incorrecta para un `ArrayList`, porque el eliminar un elemento de un lugar arbitrario en un `ArrayList` es caro. (Es posible diseñar un algoritmo diferente para `ArrayList` que funcione mejor, pero ahora no nos vamos a preocupar por esto.) En una `LinkedList`, podemos tener algo más de esperanza, ya que, como sabemos, eliminar un elemento de una posición conocida puede hacerse de forma eficiente reordenando algunos enlaces.

La Figura 6.22 muestra el primer intento. En un `ArrayList`, como era de esperar, la operación `remove` no es eficiente, por lo que la rutina requiere un tiempo cuadrático. Una `LinkedList` manifiesta dos problemas. En primer lugar, la llamada a `get` no es eficiente, por lo que la rutina requiere un tiempo cuadrático. Además, la llamada a `remove` es igualmente ineficiente, porque como ya hemos visto, resulta muy caro llegar hasta la posición `i`.

La Figura 6.23 muestra un intento de rectificar el problema. En lugar de utilizar `get`, empleamos un iterador para recorrer la lista. Esto es bastante eficiente, pero cuando empleamos el método `remove` de `Collection` para eliminar un valor par, la operación no es eficiente, porque el método `remove` tiene que buscar de nuevo el elemento, lo que requiere un tiempo lineal. Pero si ejecutamos

```
1 public static void removeEvensVer1( List<Integer> lst )
2 {
3     int i = 0;
4     while( i < lst.size( ) )
5         if( lst.get( i ) % 2 == 0 )
6             lst.remove( i );
7         else
8             i++;
9 }
```

---

Figura 6.22 Elimina los números pares de una lista; cuadrático para todos los tipos de listas.

```

1 public static void removeEvensVer2( List<Integer> lst )
2 {
3     for( Integer x : lst )
4         if( x % 2 == 0 )
5             lst.remove( x );
6 }

```

**Figura 6.23** Elimina los números pares de una lista; no funciona debido a `ConcurrentModificationException`.

el código, vemos que la situación es todavía peor: el programa genera una excepción `ConcurrentModificationException` porque, al eliminar un elemento, el iterador subyacente empleado por el bucle `for` avanzado queda invalidado. (El código de la Figura 6.22 explica por qué: no podemos esperar que el bucle `for` avanzado entienda que solo debe avanzar si no se está eliminando un elemento.)

La Figura 6.24 muestra una idea que sí funciona: después de que el iterador encuentre un elemento con valor par, podemos emplear el iterador para eliminar el valor que acaba de encontrar. Para una `LinkedList`, la llamada al método `remove` del iterador solo requiere un tiempo constante, porque el iterador se encuentra en el nodo que hay que eliminar (o cerca de él). Por tanto, para una `LinkedList`, la rutina completa requiere un tiempo lineal, en lugar de un tiempo cuadrático.

Para un `ArrayList`, incluso aunque el iterador se encuentre en el punto que hay que eliminar, la operación `remove` sigue siendo cara, porque habrá que desplazar los elementos de la matriz; así que, como cabía esperar, toda la rutina sigue requiriendo un tiempo cuadrático para un `ArrayList`.

Si ejecutamos el código de la Figura 6.24, pasándole un `LinkedList<Integer>`, necesita 0,015 segundos para una `LinkedList` de 400.000 elementos y 0,031 segundos para una `LinkedList` de 800.000 elementos, por lo que se trata claramente de una rutina con tiempo lineal, dado que el tiempo de ejecución se multiplica por el mismo factor que el tamaño de la entrada. Cuando pasamos un `ArrayList<Integer>`, la rutina tarda aproximadamente 1,25 minutos para un `ArrayList` de 400.000 elementos y cerca de cinco minutos para un `ArrayList` de 800.000 elementos; la multiplicación por cuatro del tiempo de ejecución cuando la entrada se multiplica solo por un factor de dos es coherente con el comportamiento cuadrático.

Para la operación de adición de elementos se produce una situación similar. La interfaz `Iterator` no proporciona un método `add`, pero `ListIterator` sí que lo hace. No hemos mostrado dicho método en la Figura 6.17; en el Ejercicio 6.24 le pediremos que lo utilice.

```

1 public static void removeEvensVer3( List<Integer> lst )
2 {
3     Iterator<Integer> itr = lst.iterator( );
4
5     while( itr.hasNext( ) )
6         if( itr.next( ) % 2 == 0 )
7             itr.remove( );
8 }

```

**Figura 6.24** Elimina los números pares de una lista; cuadrático en `ArrayList`, pero requiere solo un tiempo lineal para `LinkedList`.

## 6.6 Pilas y colas

En esta sección describiremos dos contenedores: la pila y la cola. En principio, ambos tienen interfaces muy simples (aunque no en la API de Colecciones) e implementaciones muy eficientes. Aun así, como veremos, se trata de estructuras de datos muy útiles.

### 6.6.1 Pilas

Una pila restringe el acceso, limitándolo al elemento más recientemente insertado.

Una *pila* es una estructura de datos en la que el acceso está restringido al elemento más recientemente insertado. Se comporta en buena medida como cualquier pila común de papeles, de platos o de periódicos. El último elemento añadido a la pila se coloca en la parte superior y es fácilmente accesible, mientras que resulta más difícil acceder a los elementos que han estado en la pila durante un cierto tiempo. Por tanto, la pila es apropiada si esperamos acceder solo al elemento superior; todos los restantes elementos son inaccesibles.

En una pila, las tres operaciones naturales `insert`, `remove` y `find` se denominan `push`, `pop` y `top`. Estas operaciones básicas se ilustran en la Figura 6.25.

La interfaz mostrada en la Figura 6.26 ilustra el protocolo típico, que es similar al protocolo que hemos visto anteriormente en la Figura 6.1. Introduciendo elementos en la pila y luego extrayéndolos, podemos utilizar la pila para invertir el orden de las cosas.

Las operaciones con la pila consumen un tiempo constante.

Cada operación con la pila debería consumir un tiempo constante, independientemente del número de elementos que la pila contenga. Por analogía, encontrar el periódico del día en una pila de periódicos es siempre rápido independientemente de la profundidad que tenga la pila. Sin embargo, el acceso arbitrario a una pila no está soportado de manera eficiente, así que no incluimos ese tipo de acceso como opción dentro del protocolo.

Lo que hace que la pila sea útil son las numerosas aplicaciones en las que lo único que necesitamos es acceder al elemento más recientemente insertado. Un uso importante de las pilas es en el diseño de compiladores.

### 6.6.2 Pilas y lenguajes informáticos

Los compiladores comprueban los programas para detectar errores sintácticos. Sin embargo, a menudo, la falta de un solo símbolo (por ejemplo, un símbolo de fin de comentario `*/` que falte o

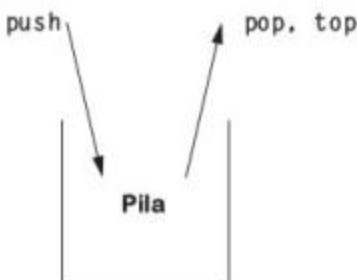


Figura 6.25 El modelo de pila: la introducción de datos en la pila se realiza mediante `push`, la lectura mediante `top` y el borrado mediante `pop`.

```
1 // Protocolo para Stack.  
2  
3 package weiss.nonstandard;  
4  
5 public interface Stack<AnyType>  
6 {  
7     void push( AnyType x ); // insertar  
8     void pop( );           // eliminar  
9     AnyType top( );       // encontrar  
10    AnyType topAndPop( ); // encontrar + eliminar  
11  
12    boolean isEmpty( );  
13    void makeEmpty( );  
14 }
```

Figura 6.26 Protocolo para la pila.

una llave de cierre ) hace que el compilador genere centenares de líneas de diagnóstico sin llegar a identificar el error real; esto es especialmente cierto cuando se emplean clases anónimas.

Una herramienta útil en esta situación es un programa que compruebe si todo está equilibrado, es decir, si cada llave de apertura, (, se corresponde con una de cierre, ), si cada corchete de apertura, [, tiene su correspondiente ], etc. La secuencia [(())] es legal pero [(]) no lo es –así que contar simplemente el número de veces que aparece cada símbolo es insuficiente. (Suponga por ahora que estamos procesando simplemente una secuencia de símbolos sintácticos, por lo que no nos vamos a preocupar de problemas tales como que la constante de caracteres '{' no necesita que exista otra constante de caracteres '}' correspondiente.)

Una pila resulta útil para comprobar si existen símbolos no equilibrados, porque sabemos que al encontrarnos con un símbolo de cierre como ), este deberá corresponderse con el símbolo de apertura ( más reciente que todavía no esté cerrado. Por tanto, introduciendo los símbolos de apertura en una pila, podemos comprobar fácilmente si un símbolo de cierre tiene sentido. Específicamente, tendremos el siguiente algoritmo:

1. Crear una pila vacía.
2. Leer los símbolos hasta el final del archivo.
  - a. Si el elemento sintáctico es un símbolo de apertura, introducirlo en la pila.
  - b. Si se trata de un símbolo de cierre y la pila está vacía, informar de que se ha producido un error.
  - c. En caso contrario, sacar un elemento de la pila. Si el símbolo extraído no es el símbolo de apertura correspondiente, informar de un error.
3. Al final del archivo, si la pila no está vacía, informar de un error.

Puede utilizarse una pila para comprobar que existen símbolos no equilibrados.

En la Sección 11.1 desarrollaremos este algoritmo para que funcione para (casi) todos los programas Java. Tendremos en cuenta todos los detalles relevantes, incluyendo los informes de

La pila se emplea para implementar llamadas a métodos en la mayoría de los lenguajes de programación.

errores y el procesamiento de comentarios, cadenas de caracteres y constantes de caracteres, así como de las secuencias de escape.

El algoritmo para comprobar que los símbolos están equilibrados sugiere una manera de implementar las llamadas a métodos. El problema es que, cuando se hace una llamada a un nuevo método, todas las variables locales del método que realiza la invocación tienen que ser guardadas por el sistema; en caso contrario, el nuevo método sobreescribiría las variables de la rutina llamante. Además, hay que guardar la posición actual de la rutina llamante para que el nuevo método sepa a dónde tiene que volver una vez que termine. La razón de que este problema sea similar al del equilibrado de símbolos es porque una llamada a método y una vuelta de un método son, esencialmente, lo mismo que un paréntesis de apertura y otro de cierre, por lo que podemos aplicar las mismas ideas. Esto es, en efecto, lo que sucede. Como se explica en la Sección 7.3, la pila se utiliza para implementar las llamadas a métodos en la mayoría de los lenguajes de programación.

El algoritmo de análisis de precedencia de operadores utiliza una pila para evaluar expresiones.

Una aplicación final importante de la pila es la evaluación de expresiones en los lenguajes informáticos. En la expresión  $1+2*3$ , vemos que en el punto en que nos encontramos con  $*$ , ya hemos leído el operador  $+$  y los operandos 1 y 2. ¿El operador  $*$  opera sobre 2, o sobre  $1+2$ ? Las reglas de precedencia nos dicen que  $*$  opera sobre 2, que es el operando más recientemente encontrado. Después de encontrar el 3, podemos evaluar  $2*3$  para obtener 6 y luego aplicar el operador  $+$ . Este proceso sugiere que los operandos y los resultados intermedios deberían guardarse en una pila. También sugiere que los operadores se guarden en la pila (ya que el  $+$  se conserva hasta después de evaluar  $*$ , que tiene mayor precedencia). Un algoritmo que utiliza esta estrategia es el *análisis de precedencia de operadores*, que se describe en la Sección 11.2.

### 6.6.3 Colas

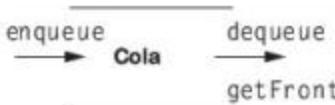
La cola restringe el acceso, limitándolo al elemento insertado menos recientemente.

Otra estructura de datos simple es la *cola*, la cual restringe el acceso limitándolo al elemento menos recientemente insertado. En muchos casos, ser capaz de encontrar y/o eliminar el elemento más recientemente insertado es importante. Pero en otra serie de casos, no solo no tiene ninguna importancia sino que es precisamente lo que no tenemos que hacer. En un sistema multiproceso, por ejemplo, cuando se envían trabajos a una impresora, esperamos que se imprima en primer lugar el trabajo menos reciente o más antiguo. Este orden no solo resulta justo, sino que también es obligatorio para garantizar que el primer trabajo de impresión no esté esperando indefinidamente. Esa es la razón por la que podemos esperar encontrarnos colas de impresión en todos los sistemas de gran tamaño.

Las operaciones básicas soportadas por las colas son las siguientes:

- `enqueue`, o inserción en la parte final de la cola.
- `dequeue`, o eliminación del elemento situado al principio de la cola.
- `getFront`, o acceso al elemento situado al principio de la cola.

La Figura 6.27 ilustra estas operaciones con las colas. Históricamente, `dequeue` y `getFront` han estado combinadas en una sola operación. Esto se hace haciendo que `dequeue` devuelva una referencia al elemento que ha eliminado.



**Figura 6.27** Modelo de cola: la introducción de datos se realiza mediante enqueue, la consulta mediante getFront y el borrado mediante dequeue.

Puesto que las operaciones con una cola y las operaciones con una pila están restringidas de forma similar, cabe esperar que también necesitan un tiempo constante por cada consulta, y efectivamente es así. Todas las operaciones básicas con la cola tardan un tiempo  $O(1)$ . Veremos varias aplicaciones de las colas en los casos de estudio.

Las operaciones con una cola consumen un tiempo constante.

#### 6.6.4 Pilas y colas en la API de Colecciones

La API de Colecciones proporciona una clase `Stack` para la pilas, pero no proporciona ninguna clase para las colas. Los métodos de `Stack` son `push`, `pop` y `peek`. Sin embargo, la clase `Stack` amplía `Vector` y es más lenta de lo necesario; como `Vector`, su uso ya no se recomienda y puede sustituirse por operaciones con `List`. Antes de Java 1.4, el único soporte de `java.util` para las operaciones con colas era utilizar una `LinkedList` (por ejemplo, `addLast`, `removeFirst` y `getFirst`). Java 5 añade una interfaz `Queue` para las colas, parte de la cual se muestra en la Figura 6.28. Sin embargo, necesitamos seguir empleando métodos de `LinkedList`. Los nuevos métodos son `add`, `remove` y `element`.

La API de colecciones proporciona una clase `Stack`, pero no proporciona ninguna clase para colas. Java 5 añade una interfaz `Queue`.

## 6.7 Conjuntos

Un conjunto `Set` es un contenedor que no contiene duplicados. Soporta todos los métodos de `Collection`. Lo más importante es que, tal como explicamos en la Sección 6.5.3, `contains` para una lista `List` es ineficiente, independientemente de si la `List` es un `ArrayList` o una `LinkedList`. Por el contrario, una implementación de librería de `Set` debe soportar de manera eficiente la operación `contains`. De forma similar, el método `remove` de `Collection` (que tiene como parámetro un objeto especificado, no un índice especificado) es ineficiente para una `List`, porque está implícito que lo primero que tiene que hacer `remove` es encontrar el elemento que hay que eliminar; esencialmente, esto hace que `remove` sea al menos tan difícil como `contains`. Para un `Set`, también se espera que `remove` esté implementado eficientemente. Y finalmente, se espera asimismo que `add` tenga una implementación eficiente. No hay sintaxis en Java que pueda utilizarse para especificar que una operación deba satisfacer una determinada restricción de tiempo de ejecución, o para indicar que una cierta colección no contenga duplicados; por ello, la Figura 6.29 ilustra que la interfaz `Set` hace poco más que declarar un tipo.

Un conjunto `Set` no contiene duplicados.

Un `SortedSet` es un `Set` que mantiene (internamente) sus elementos ordenados. Los objetos añadidos al `SortedSet` deben ser comparables, o sino habrá que proporcionar un `Comparator` en

**SortedSet** es un contenedor ordenado. No permite duplicados.

el momento de instanciar un contenedor. Un **SortedSet** soporta todos los métodos de **Set**, pero se garantiza que su iterador recorra todos los elementos por orden. El **SortedSet** también nos permite encontrar los elementos mayor y menor. La interfaz para nuestro subconjunto de **SortedSet** se muestra en la Figura 6.30.

```

1 package weiss.util;
2
3 /**
4  * Interfaz Queue.
5  */
6 public interface Queue<AnyType> extends Collection<AnyType>
7 {
8     /**
9      * Devuelve pero no elimina el elemento situado al "principio"
10     * de la cola.
11     * @return el elemento inicial o null si la cola está vacía.
12     * @throws NoSuchElementException si la cola está vacía.
13     */
14     AnyType element( );
15
16     /**
17      * Devuelve pero no elimina el elemento situado al "principio"
18      * de la cola.
19      * @return el elemento inicial.
20      * @throws NoSuchElementException si la cola está vacía.
21      */
22     AnyType remove( );
23 }
```

Figura 6.28 Posible interfaz Queue.

```

1 package weiss.util;
2
3 /**
4  * Interfaz Set.
5  */
6 public interface Set<AnyType> extends Collection<AnyType>
7 {
8 }
```

Figura 6.29 Posible interfaz Set.

```
1 package weiss.util;
2
3 /**
4  * Interfaz SortedSet.
5  */
6 public interface SortedSet<AnyType> extends Set<AnyType>
7 {
8     /**
9      * Devuelve el comparador utilizado por este SortedSet.
10     * @return el comparador o null si se utiliza
11     * el comparador predeterminado.
12     */
13    Comparator<? super AnyType> comparator( );
14
15    /**
16     * Encuentra el elemento más pequeño del conjunto.
17     * @return el elemento más pequeño.
18     * @throws NoSuchElementException si el conjunto está vacío.
19     */
20    AnyType first( );
21
22    /**
23     * Encuentra el elemento más grande del conjunto.
24     * @return el elemento más grande.
25     * @throws NoSuchElementException si el conjunto está vacío.
26     */
27    AnyType last( );
28 }
```

Figura 6.30 Posible interfaz SortedSet.

### 6.7.1 La clase TreeSet

El `SortedSet` está implementado mediante un `TreeSet`. La implementación subyacente del `TreeSet` es un árbol equilibrado de búsqueda binaria y se explica en el Capítulo 19.

Si no se especifica lo contrario, la ordenación utiliza el comparador predeterminado. Podemos especificar una ordenación alternativa proporcionando un comparador al constructor. Como ejemplo, la Figura 6.31 ilustra cómo construir un `SortedSet` que almacene cadenas de caracteres. La llamada a `printCollection` permitirá imprimir los elementos en orden decreciente.

TreeSet es una  
implementación de  
SortedSet.

El `TreeSet`, como todos los conjuntos `Set`, no permite duplicados. Dos elementos se consideran iguales si el método `compare` del comparador devuelve 0.

```

1  public static void main( String [] args )
2  {
3      Set<String> s = new TreeSet<String>( Collections.reverseOrder( ) );
4      s.add( "joe" );
5      s.add( "bob" );
6      s.add( "hal" );
7      printCollection( s ); // Figura 6.11
8  }

```

Figura 6.31 Ilustración de TreeSet, utilizando ordenación inversa.

En la Sección 5.6 hemos examinado el problema de la búsqueda estática y vimos que si se nos presentan los elementos ordenados, podemos soportar la operación `find` en un tiempo logarítmico de caso peor. Esto es una búsqueda estática, porque después de que se nos presenten los elementos, no podemos añadir ni eliminar ningún elemento. El `SortedSet` sí que nos permite añadir y eliminar elementos.

Esperamos que el coste de caso peor de las operaciones `contains`, `add` y `remove` sea  $O(\log N)$ , porque eso se correspondería con la cota obtenida para la búsqueda binaria estática. Lamentablemente, para la implementación más simple de `TreeSet`, esto no se cumple. El caso medio es logarítmico, pero el caso peor es  $O(N)$  y se presenta de forma bastante frecuente. Sin embargo, aplicando algunos trucos algorítmicos, podemos obtener una estructura más compleja que sí que tenga un coste de  $O(\log N)$  por operación. Se garantiza que el `TreeSet` de la API de Colecciones tenga este rendimiento, y en el Capítulo 19 explicaremos cómo obtenerlo utilizando el *árbol de búsqueda binaria* y sus variantes, y proporcionaremos una implementación de `TreeSet` con un iterador.

Podemos también utilizar un árbol de búsqueda binaria para acceder al  $K$ -ésimo elemento más pequeño en un tiempo logarítmico.

Mencionemos, para terminar, que aunque es posible encontrar los elementos mayor y menor en un `SortedSet` en un tiempo  $O(\log N)$ , encontrar el  $K$ -ésimo elemento más pequeño, donde  $K$  sea un parámetro, no está soportado en la API de Colecciones. Y sin embargo, es posible realizar esta operación en un tiempo  $O(\log N)$ , mientras se preserva el tiempo de ejecución de las restantes operaciones, aunque para ello hace falta algo más de trabajo.

### 6.7.2 La clase HashSet

El HashSet implementa la interfaz Set. No requiere un comparador.

Además del `TreeSet`, la API de Colecciones proporciona una clase `HashSet` que implementa la interfaz `Set`. El `HashSet` difiere de `TreeSet` en que no puede utilizarse para enumerar elementos por orden, ni tampoco para obtener los elementos mayor y menor. De hecho, los elementos del `HashSet` no tienen por qué ser comparables en modo alguno. Esto significa que el `HashSet` es menos potente que el `TreeSet`. Si no es importante tener que enumerar los elementos de un `Set` en orden, entonces a menudo es preferible utilizar un `HashSet`, porque al no tener que mantener la ordenación, el `HashSet` permite obtener un mejor rendimiento. Para poder hacer esto, los elementos incluidos en el `HashSet` deben proporcionar una serie de pistas a los algoritmos de `HashSet`. Esto se

```

1  public static void main( String [] args )
2  {
3      Set<String> s = new HashSet<String>();
4      s.add( "joe" );
5      s.add( "bob" );
6      s.add( "hal" );
7      printCollection( s ); // Figura 6.11
8  }

```

**Figura 6.32** Una ilustración de `HashSet`, en la que los elementos se imprimen en un orden desconocido.

lleva a cabo haciendo que cada elemento implemente un método `hashCode` especial; describiremos este método más adelante dentro de esta subsección.

La Figura 6.32 ilustra el uso del `HashSet`. Se garantiza que, si iteramos a través de todo el `HashSet`, visualizaremos cada elemento una sola vez, pero el orden en el que se recorrerán los elementos no está definido. Es casi seguro que dicho orden no coincidirá con el orden de inserción, ni tampoco con ningún otro criterio de ordenación.

Como todos los conjuntos `Set`, el `HashSet` no permite duplicados. Dos elementos se consideran iguales, si el método `equals` dice que lo son. Por tanto, cualquier objeto que se inserte en el `HashSet` debe tener un método `equals` apropiadamente definido.

Recuerde que, en la Sección 4.9, explicamos que es esencial sustituir la definición de `equals` (proporcionando una nueva versión de `equals` que admita un `Object` como parámetro), en lugar de sobrecargar dicho método.

### Implementación de `equals` y `hashCode`

La sustitución de `equals` es bastante complicada cuando están implicados los mecanismos de herencia. La especificación de `equals` establece que si `p` y `q` no son `null`, `p.equals(q)` debe devolver el mismo valor que `q.equals(p)`. Esto no es así en la Figura 6.33. En ese ejemplo, claramente `b.equals(c)` devuelve `true`, como cabía esperar. La expresión `a.equals(b)` también devuelve `true`, porque se utiliza el método `equals` de `BaseClass`, y ese método solo compara los componentes `x`. Sin embargo, `b.equals(a)` devuelve `false`, porque se utiliza el método `equals` de `DerivedClass` y, en la línea 29, la comprobación `instanceof` fallará (`a` no es una instancia de `DerivedClass`).

`equals` debe ser simétrica. Esto puede resultar problemático cuando está implicada la herencia.

Hay dos soluciones estándar para este problema. Una consiste en hacer que el método `equals` sea final en `BaseClass`. Esto evita el problema de tener métodos `equals` que entran en conflicto. La otra solución consiste en forzar la comprobación `equals` para exigir que los tipos sean idénticos y no simplemente compatibles, ya que es la compatibilidad unidireccional lo que hace que deje de funcionar `equals`. En este ejemplo, un objeto de tipo `BaseClass` y otro de tipo `DerivedClass` nunca se declararían como iguales. La Figura 6.34 muestra una implementación correcta. La línea 8 contiene la comprobación fundamental, `getClass` devuelve un objeto especial de tipo `Class` (observe la `C` mayúscula) que contiene información acerca de la clase de un objeto. `getClass` es un método final de la clase `Object`. Si lo invocas

La solución 1 consiste en no sustituir `equals` por debajo de la clase base. La solución 2 consiste en exigir que se comparan objetos de tipo idéntico utilizando para ello `getClass`.

```
1 class BaseClass
2 {
3     public BaseClass( int i )
4     { x = i; }
5
6     public boolean equals( Object rhs )
7     {
8         // Esta es una comprobación errónea (ok para una clase final)
9         if( !( rhs instanceof BaseClass ) )
10            return false;
11
12        return x == ( (BaseClass) rhs ).x;
13    }
14
15    private int x;
16 }
17
18 class DerivedClass extends BaseClass
19 {
20     public DerivedClass( int i, int j )
21     {
22         super( i );
23         y = j;
24     }
25
26     public boolean equals( Object rhs )
27     {
28         // Esta es una comprobación errónea.
29         if( !( rhs instanceof DerivedClass ) )
30             return false;
31
32         return super.equals( rhs ) &&
33             y == ( (DerivedClass) rhs ).y;
34     }
35
36     private int y;
37 }
38
39 public class EqualsWithInheritance
40 {
41     public static void main( String [ ] args )
42     {
43         BaseClass a = new BaseClass( 5 );
44         DerivedClass b = new DerivedClass( 5, 8 );
45         DerivedClass c = new DerivedClass( 5, 8 );
46
47         System.out.println( "b.equals(c): " + b.equals( c ) );
48         System.out.println( "a.equals(b): " + a.equals( b ) );
49         System.out.println( "b.equals(a): " + b.equals( a ) );
50     }
51 }
```

---

Figura 6.33 Una ilustración de una implementación incorrecta de `equals`.

```

1 class BaseClass
2 {
3     public BaseClass( int i )
4         { x = i; }
5
6     public boolean equals( Object rhs )
7     {
8         if( rhs == null || getClass( ) != rhs.getClass( ) )
9             return false;
10
11        return x == ( (BaseClass) rhs ).x;
12    }
13
14    private int x;
15 }
16
17 class DerivedClass extends BaseClass
18 {
19     public DerivedClass( int i, int j )
20     {
21         super( i );
22         y = j;
23     }
24
25     public boolean equals( Object rhs )
26     {
27         // Comprobación de clase no necesaria; getClass() se invoca
28         // en el método equals de la superclase.
29         return super.equals( rhs ) &&
30             y == ( (DerivedClass) rhs ).y;
31     }
32
33    private int y;
34 }

```

Figura 6.34 Implementación correcta de `equals`.

con dos objetos diferentes devuelve la misma instancia de `Class`, entonces los dos objetos tienen tipos idénticos.

Al utilizar un `HashSet`, también tenemos que sustituir el método especial `hashCode` especificado en `Object`; `hashCode` devuelve un valor `int`. Piense en `hashCode` como en algo que proporcionara una pista de confianza acerca del lugar en el que los elementos están almacenados. Si la pista es errónea, no podrá encontrarse el elemento, así que si dos elementos son iguales deberían proporcionar

Si se sustituye `equals` por `hashCode`, hay que sustituir también el método `hashCode`, porque sino el `HashSet` no funcionará.

pistas idénticas. La especificación de `hashCode` dice que si dos objetos son declarados iguales por el método `equals`, entonces el método `hashCode` debe devolver el mismo valor para ellos, y si se viola esta especificación, el `HashSet` no podrá encontrar los objetos, aun cuando `equals` declare que hay una correspondencia. Si `equals` declara que los objetos no son iguales, el método `hashCode` debería devolver un valor distinto para ellos, aunque esto no es obligatorio. Sin embargo, aunque no sea obligatorio sí que es muy beneficioso para el rendimiento de `HashSet` que `hashCode` solo proporcione resultados idénticos para objetos desiguales en raras ocasiones. En el Capítulo 20 explicaremos cómo interactúan `hashCode` y `HashSet`.

La Figura 6.35 ilustra una clase `SimpleStudent` en la que dos objetos `SimpleStudent` son iguales si ambos tienen el mismo nombre (y ambos son de tipo `SimpleStudent`). Esto podría sustituirse utilizando las técnicas de la Figura 6.34 de la forma necesaria, o bien declarando este método como `final`. Si se le declara `final`, entonces la comprobación utilizada solo permitirá declarar como iguales a dos objetos `SimpleStudent` que sean de tipo idéntico. Si, teniendo un `equals` final, sustituimos la comprobación de la línea 40 por una comprobación `instanceof`, entonces cualesquiera dos objetos de la jerarquía podrán ser declarados iguales si sus nombres se corresponden.

El método `hashCode` de las líneas 47 y 48 simplemente utiliza el `hashCode` del campo `name`. Así, si dos objetos `SimpleStudent` tienen el mismo nombre (tal como lo declare `equals`), tendrán también el mismo `hashCode`, ya que, presumiblemente, los implementadores de `String` habrán respetado la especificación de `hashCode`.

El programa de prueba de acompañamiento forma parte de una prueba de mayor tamaño que ilustra todos los contenedores básicos. Observe que si el `hashCode` no se ha implementado, se añadirán al `HashSet` los tres objetos `SimpleStudent`, porque no se detectará el duplicado.

Resulta que, como promedio, las operaciones `HashSet` puede realizarse en un tiempo constante. Esto parece un resultado bastante sorprendente, porque implica que el coste de una única operación `HashSet` no depende de si el `HashSet` contiene 10 elementos o 10.000. La teoría que subyace al concepto de `HashSet` es fascinante y se describe en el Capítulo 20.

## 6.8 Mapas

Un `Map` se utiliza para almacenar una colección de entradas formadas por sus *claves* y sus *valores*. El mapa asigna claves a valores.

Un `Map` se utiliza para almacenar una colección de entradas formadas por sus *claves* y sus *valores*. El `Map` asigna claves a valores. Las claves deben ser diferentes, pero pueden asignarse varias claves a un mismo valor. Por tanto, los que no necesitan ser diferentes son los valores. Existe una interfaz `SortedMap` que mantiene el mapa ordenado, desde el punto de vista lógico, según las claves.

No es sorprendente que existan dos implementaciones: `HashMap` y `TreeMap`. `HashMap` no mantiene las claves en orden, mientras que `TreeMap` sí que lo hace. Por simplicidad, no vamos a implementar la interfaz `SortedMap`, pero sí que implementaremos `HashMap` y `TreeMap`.

El `Map` puede implementarse como un `Set` instanciado con un `par` (véase la Sección 3.9), cuyo comparador o implementación `equals/hashCode` solo hace referencia a la clave. La interfaz `Map` no amplía `Collection`; por el contrario, es independiente. En las Figuras 6.36 y 6.37 se muestra una interfaz de ejemplo que contiene los métodos más importantes.

```

1 /**
2  * Programa de pruebas para HashSet.
3 */
4 class IteratorTest
5 {
6     public static void main( String [ ] args )
7     {
8         List<SimpleStudent> stud1 = new ArrayList<SimpleStudent>();
9         stud1.add( new SimpleStudent( "Bob", 0 ) );
10        stud1.add( new SimpleStudent( "Joe", 1 ) );
11        stud1.add( new SimpleStudent( "Bob", 2 ) ); // duplicate
12
13        // Si hashCode está implementado, solo tendrá 2 elementos.
14        // En caso contrario, tendrá 3 porque
15        // no se detectará el duplicado.
16        Set<SimpleStudent> stud2 = new HashSet<SimpleStudent>( stud1 );
17
18        printCollection( stud1 ); // Bob Joe Bob (orden no especificado)
19        printCollection( stud2 ); // Dos elementos en orden no especificado
20    }
21 }
22
23 /**
24  * Ilustra el uso de hashCode>equals para una clase definida por el usuario.
25  * Los estudiantes se ordenan basándose solo en el nombre.
26 */
27 class SimpleStudent implements Comparable<SimpleStudent>
28 {
29     String name;
30     int id;
31
32     public SimpleStudent( String n, int i )
33     {
34         name = n; id = i;
35     }
36
37     public String toString( )
38     {
39         return name + " " + id;
40     }
41
42     public boolean equals( Object rhs )
43     {
44         if( rhs == null || getClass( ) != rhs.getClass( ) )
45             return false;
46
47         SimpleStudent other = (SimpleStudent) rhs;
48         return name.equals( other.name );
49     }
50
51     public int hashCode( )
52     {
53         return name.hashCode( );
54     }
55 }
```

Figura 6.35 Ilustra los métodos `equals` y `hashCode` que se emplean en `HashSet`.

```
1 package weiss.util;
2
3 /**
4  * Interfaz Map.
5  * Un mapa almacena parejas clave/valor.
6  * En nuestra implementación, no están permitidas las claves duplicadas.
7 */
8 public interface Map<KeyType,ValueType> extends java.io.Serializable
9 {
10    /**
11     * Devuelve el número de claves en este mapa.
12     */
13    int size( );
14
15    /**
16     * Comprueba si este mapa está vacío.
17     */
18    boolean isEmpty( );
19
20    /**
21     * Comprueba si este mapa contiene una clave especificada.
22     */
23    boolean containsKey( KeyType key );
24
25    /**
26     * Devuelve el valor que se corresponde con la clave o null
27     * si no se encuentra la clave. Dado que están permitidos los valores
28     * null, comprobar si el valor de retorno es null puede no ser una forma
29     * segura de determinar si la clave está presente en el mapa.
30     */
31    ValueType get( KeyType key );
32
33    /**
34     * Añade la pareja clave/valor al mapa, sustituyendo el valor
35     * original en caso de que la clave ya estuviera presente.
36     * Devuelve el antiguo valor asociado con la clave o
37     * null si la clave no estaba presente antes de esta llamada.
38 */
39    ValueType put( KeyType key, ValueType value );
40
41    /**
42     * Elimina la clave y su valor del mapa.
43     * Devuelve el valor previamente asociado con la clave
44     * o null si la clave no estaba presente antes de esta llamada.
45     */
46    ValueType remove( KeyType key );
```

---

Figura 6.36 Una interfaz Map de ejemplo (parte 1).

```
47  /**
48   * Elimina todas las parejas clave/valor del mapa.
49   */
50 void clear( );
51
52 /**
53  * Devuelve las claves del mapa.
54  */
55 Set<KeyType> keySet( );
56
57 /**
58  * Devuelve los valores del mapa. Puede haber duplicados.
59  */
60 Collection<ValueType> values( );
61
62 /**
63  * Devuelve un conjunto de objetos Map.Entry correspondientes
64  * a las parejas clave/valor contenidas en el mapa.
65  */
66 Set<Entry<KeyType,ValueType>> entrySet( );
67
68 /**
69  * Interfaz utilizada para acceder a las parejas clave/valor de un mapa.
70  * A partir de un mapa, utilice entrySet().iterator para obtener un
71  * iterador sobre un Set de parejas. El método next() de este iterador
72  * proporciona objetos de tipo Map.Entry<KeyType,ValueType>.
73  */
74 public interface Entry<KeyType,ValueType> extends java.io.Serializable
75 {
76     /**
77      * Devuelve la clave de esta pareja.
78      */
79     KeyType getKey( );
80
81     /**
82      * Devuelve el valor de esta pareja.
83      */
84     ValueType getValue( );
85
86     /**
87      * Cambia el valor de esta pareja.
88      * @return el valor antiguo asociado con esta pareja.
89      */
90     ValueType setValue( ValueType newValue );
91 }
92 }
```

Figura 6.37 Una interfaz Map de ejemplo (parte 2).

La mayoría de los métodos tienen un semántica intuitiva. El método `put` se utiliza para añadir un par de clave/valor, `remove` se emplea para eliminar una par de clave/valor (solo se especifica la clave) y `get` devuelve el valor asociado con una clave. Están permitidos valores `null`, lo que complica las cosas para `get`, ya que el valor de retorno proporcionado por `get` no permitirá distinguir entre una búsqueda fallida y una búsqueda que haya tenido éxito y que devuelva `null` como valor. Si se sabe que existen valores `null` en el mapa, puede emplearse `containsKey`.

La interfaz `Map` no proporciona un método `iterator` ni una clase iteradora. En lugar de ello, devuelve una `Collection` que puede utilizarse para visualizar los contenidos del mapa.

El método `keySet` proporciona una `Collection` que contiene todas las claves. Puesto que no están permitidas las claves duplicadas, el resultado de `keySet` es un `Set`, para el cual podemos obtener un iterador. Si el `Map` es un `SortedMap`, el `Set` es un `SortedSet`.

De forma similar, el método `values` devuelve una `Collection` que contiene todos los valores. En este caso se trata realmente de una `Collection`, ya que los valores duplicados están permitidos.

`Map.Entry` abstracta la  
noción de una pareja dentro  
del mapa.

Finalmente, el método `entrySet` devuelve una `Collection` de parejas clave/valor. De nuevo, se trata de un `Set`, porque las parejas deben tener claves distintas. Los objetos del `Set` devueltos por el `entrySet` son parejas; debe haber un tipo que represente a esas parejas clave/valor. Este tipo es especificado por la interfaz `Entry` anidada dentro de la interfaz `Map`. Por tanto, el tipo de objeto almacenado en el `entrySet` es `Map.Entry`.

La Figura 6.38 ilustra el uso del `Map` con `TreeMap`. En la línea 23 se crea un mapa vacío y luego se rellena con una serie de llamadas a `put` en las líneas 25 a 29. La última llamada a `put` simplemente sustituye un valor con "unlisted". Las líneas 31 y 32 imprimen el resultado de una llamada a `get`, que se utiliza para obtener el valor correspondiente a la clave "Jane Doe". Más interesante es la rutina `printMap` que abarca las líneas 8 a 19.

En `printMap`, en la línea 12, obtenemos un `Set` que contiene parejas `Map.Entry`. A partir del `Set`, podemos utilizar un bucle `for` avanzado para visualizar las entradas `Map.Entry` y podemos obtener la información de clave y valor utilizando `getKey` y `getValue`, como se muestra en las líneas 16 y 17.

`keySet`, `values` y  
`entrySet` devuelven  
vistas.

Volviendo a `main`, vemos que `keySet` devuelve un conjunto de claves (en la línea 37) que pueden imprimirse en la línea 38 llamando a `printCollection` (en la Figura 6.11); de forma similar, en las líneas 41 y 42, `values` devuelve una colección de valores que se puede imprimir. Más interesante resulta el que el conjunto de claves y la colección de valores son *vistas* del mapa,

por lo que los cambios en el mapa se ven inmediatamente reflejados en el conjunto de claves y la colección de valores, y las eliminaciones que efectuemos en el conjunto de claves o en el de valores se convierten en eliminaciones en el mapa subyacente. Por tanto, la línea 44 no solo elimina la clave del conjunto de claves, sino también la entrada asociada en el mapa. De forma similar, la línea 45 elimina una entrada del mapa. Por tanto, la operación de impresión en la línea 49 refleja un mapa en el que se han eliminado dos entradas.

Las vistas en sí mismas constituyen un concepto interesante y explicaremos los detalles específicos acerca de cómo se implementan más adelante, cuando implementemos las clases de mapas. En la Sección 6.10 se exponen algunos ejemplos adicionales de vistas.

La Figura 6.39 ilustra otro uso del mapa, en un método que devuelve los elementos de una lista que aparecen más de una vez. En este código, se está utilizando internamente un mapa para agrupar

```
1 import java.util.Map;
2 import java.util.TreeMap;
3 import java.util.Set;
4 import java.util.Collection;
5
6 public class MapDemo
7 {
8     public static <KeyType,ValueType>
9     void printMap( String msg, Map<KeyType,ValueType> m )
10    {
11        System.out.println( msg + ":" );
12        Set<Map.Entry<KeyType,ValueType>> entries = m.entrySet( );
13
14        for( Map.Entry<KeyType,ValueType> thisPair : entries )
15        {
16            System.out.print( thisPair.getKey( ) + ": " );
17            System.out.println( thisPair.getValue( ) );
18        }
19    }
20
21    public static void main( String [ ] args )
22    {
23        Map<String,String> phonel = new TreeMap<String,String>( );
24
25        phonel.put( "John Doe", "212-555-1212" );
26        phonel.put( "Jane Doe", "312-555-1212" );
27        phonel.put( "Holly Doe", "213-555-1212" );
28        phonel.put( "Susan Doe", "617-555-1212" );
29        phonel.put( "Jane Doe", "unlisted" );
30
31        System.out.println( "phonel.get(\"Jane Doe\"): " +
32                            phonel.get( "Jane Doe" ) );
33        System.out.println( "\nThe map is: " );
34        printMap( "phonel", phonel );
35
36        System.out.println( "\nThe keys are: " );
37        Set<String> keys = phonel.keySet( );
38        printCollection( keys );
39
40        System.out.println( "\nThe values are: " );
41        Collection<String> values = phonel.values( );
42        printCollection( values );
43
44        keys.remove( "John Doe" );
45        values.remove( "unlisted" );
46
47        System.out.println( "After John Doe and 1 unlisted are removed" );
48        System.out.println( "\nThe map is: " );
49        printMap( "phonel", phonel );
50    }
51 }
```

Figura 6.38 Una ilustración de cómo se utiliza la interfaz Map.

```

1 public static List<String> listDuplicates( List<String> coll )
2 {
3     Map<String, Integer> count = new TreeMap<String, Integer>();
4     List<String> result = new ArrayList<String>();
5
6     for( String word : coll )
7     {
8         Integer occurs = count.get( word );
9         if( occurs == null )
10            count.put( word, 1 );
11        else
12            count.put( word, occurs + 1 );
13    }
14
15    for( Map.Entry<String, Integer> e : count.entrySet() )
16        if( e.getValue() >= 2 )
17            result.add( e.getKey() );
18
19    return result;
20 }

```

Figura 6.39 Un uso típico de un mapa.

los duplicados: la clave del mapa es un elemento y el valor es el número de veces que el elemento ha aparecido. Las líneas 8-12 ilustran la idea típica que podemos ver a la hora de construir un mapa de esta forma. Si el elemento nunca ha sido insertado en el mapa, lo hacemos con un recuento igual a 1. En caso contrario, actualizamos el recuento. Observe el juicioso uso de los mecanismos de *autoboxing* y *unboxing*. Después, en las líneas 15-17 utilizamos un iterador para recorrer el conjunto de entradas, obteniendo las claves que aparezcan con un recuento mayor o igual que dos en el mapa.

## 6.9 Colas con prioridad

La cola con prioridad solo permite acceder al elemento mínimo.

Aunque los trabajos de impresión enviados a una impresora suelen colocarse en una cola, esa política puede no ser siempre la más adecuada. Por ejemplo, un trabajo de impresión podría ser particularmente importante, por lo que desearíamos poder permitir que ese trabajo se imprima en cuanto la impresora esté disponible. A la inversa, cuando la impresora finalice un trabajo y haya varios trabajos de 1 página y uno de 100 páginas esperando, puede ser razonable imprimir el último trabajo al final, aun cuando no sea el último trabajo enviado. (Lamentablemente, la mayoría de los sistemas no hacen esto, lo cual puede resultar particularmente molesto en ciertas ocasiones.)

De forma similar, en un entorno multiusuario, el planificador del sistema operativo debe decidir cuál de entre varios procesos ejecutar. En general, a cada proceso solo se le permite ejecutarse durante un periodo de tiempo fijo. Un algoritmo no muy adecuado para implementar este tipo de

procedimiento implicaría la utilización de una cola. Los trabajos se colocarían inicialmente al final de la cola. El planificador tomaría una y otra vez el primer trabajo de la cola, lo ejecutaría hasta que finalizara o se consumiera su límite de tiempo y lo volvería a colocar al final de la cola si no lo ha terminado. Generalmente, esta estrategia no es apropiada porque los trabajos cortos deben esperar, lo que hace que parezca que necesitan mucho tiempo para ejecutarse. Obviamente, los usuarios que estén ejecutando un editor no deberían experimentar un retardo visible a la hora de visualizar en pantalla los caracteres que escriben. Por tanto, los trabajos cortos (es decir, los que emplean menos recursos) deberían tener precedencia sobre los trabajos que ya hayan consumido una gran cantidad de recursos. Además, algunos trabajos que hacen un uso intensivo de los recursos, como por ejemplo los trabajos ejecutados por el administrador del sistema, pueden ser importantes y también deberían tener precedencia.

Si asignamos a cada trabajo un número para medir su prioridad, entonces el número más pequeño (páginas impresas, recursos utilizados) tenderá a indicar una mayor importancia. Por tanto, desearemos poder acceder al elemento más pequeño de una colección de elementos y eliminarlo de la misma. Para hacer esto utilizamos las operaciones `findMin` y `deleteMin`. La estructura de datos que soporta estas operaciones es la denominada *cola con prioridad* y soporta únicamente el acceso al elemento mínimo. La Figura 6.40 ilustra la operaciones básicas en un cola con prioridad.

Aunque la cola con prioridad es una estructura de datos fundamental, antes de Java 5 no existía ninguna implementación de ella en la API de Colecciones. Un `SortedSet` no era suficiente, porque es importante para una cola con prioridad el permitir elementos duplicados.

En Java 5, `PriorityQueue` es una clase que implementa la interfaz `Queue`. Así, `insert`, `findMin` y `deleteMin` se expresan mediante llamadas a `add`, `element` y `remove`. `PriorityQueue` puede construirse sin ningún parámetro, con un comparador o con otra colección compatible. A lo largo del texto, a menudo emplearemos los términos `insert`, `findMin` y `deleteMin` para describir los métodos de la cola con prioridad. La Figura 6.41 ilustra el uso de este tipo de colas.

Ya que la cola con prioridad solo soporta las operaciones `deleteMin` y `findMin`, cabría esperar que su rendimiento fuera un compromiso entre la cola de tiempo constante y el conjunto de tiempo logarítmico. Y de hecho, efectivamente es así. La cola básica con prioridad soporta todas las operaciones en un tiempo logarítmico de caso peor, utiliza solo una matriz, soporta la inserción en un tiempo promedio constante, es simple de implementar y se conoce con el nombre de *montículo binario*. Esta estructura es una de las estructuras de datos más elegantes que se conocen. En el Capítulo 21 proporcionaremos detalles sobre la implementación del montículo binario. Una implementación alternativa que soporta una operación adicional es el denominado *montículo de emparejamiento*, descrito en el Capítulo 23. Puesto que hay muchas implementaciones eficientes de las colas con prioridad, resulta bastante

El *montículo binario* implementa la cola con prioridad en un tiempo logarítmico por cada operación, utilizando solo algo de espacio adicional.



Figura 6.40 Modelo de cola con prioridad: solo es accesible el elemento mínimo.

```

1 import java.util.PriorityQueue;
2
3 public class PriorityQueueDemo
4 {
5     public static <AnyType extends Comparable<? super AnyType>>
6         void dumpPQ( String msg, PriorityQueue<AnyType> pq )
7     {
8         System.out.println( msg + ":" );
9         while( !pq.isEmpty( ) )
10            System.out.println( pq.remove( ) );
11    }
12
13 // Realizar algunas inserciones y eliminaciones (hechas en dumpPQ).
14 public static void main( String [ ] args )
15 {
16     PriorityQueue<Integer> minPQ = new PriorityQueue<Integer>();
17
18     minPQ.add( 4 );
19     minPQ.add( 3 );
20     minPQ.add( 5 );
21
22     dumpPQ( "minPQ", minPQ );
23 }
24 }
```

**Figura 6.41** Una rutina para ilustrar la clase `PriorityQueue`.

desafortunado que los diseñadores de la librería no decidieran hacer de `PriorityQueue` una interfaz. De todos modos, la implementación de `PriorityQueue` en Java 5 es suficiente para la mayoría de las aplicaciones de colas con prioridad.

Un uso importante de las colas con prioridad es en la simulación dirigida por sucesos.

Una aplicación importante de las colas con prioridad es en la *simulación dirigida por sucesos*. Considere por un ejemplo un sistema, como por ejemplo un banco, en el que los clientes llegan y esperan en la cola hasta que está disponible uno de  $K$  posibles cajeros. La llegada de los clientes está gobernada por una función de distribución de probabilidad, como también lo está el tiempo de servicio (la cantidad de tiempo que un cajero necesita para proporcionar un servicio completo a un cliente). Nos interesa obtener estadísticas tales como cuánto tiempo tiene que esperar como media un cliente, o cuál será como media la longitud de una cola.

Con ciertas distribuciones de probabilidad y ciertos valores de  $K$ , podemos calcular estas estadísticas de manera exacta. Sin embargo, a medida que va creciendo el valor de  $K$ , el análisis se hace considerablemente más difícil, por lo que resulta atractiva la idea de utilizar una computadora para simular el funcionamiento del banco. De esta forma, los directivos del banco pueden determinar cuántos cajeros hacen falta para garantizar un servicio razonable. Una simulación dirigida por

sucesos consiste en el procesamiento de una serie de sucesos consecutivos. Los dos tipos de sucesos aquí son: (1) la llegada de un cliente y (2) la marcha de un cliente, liberando a un cajero. En cualquier momento, tendremos una colección de sucesos esperando a tener lugar. Para ejecutar la simulación, necesitaremos determinar cuál es el suceso siguiente; dicho suceso será aquel cuyo instante de materialización sea mínimo. Por tanto, usaremos una cola con prioridad que extraiga el suceso con un tiempo mínimo para procesar la lista de sucesos de manera eficiente. En la Sección 13.2 presentaremos una explicación completa y una implementación de una simulación dirigida por sucesos.

## 6.10 Vistas en la API de Colecciones

En la Sección 6.8 hemos visto un ejemplo de métodos que devuelven "vistas" de un mapa. Específicamente, `keySet` devuelve una vista que representa un `Set` de todas las claves del mapa; `values` devuelve una vista que representa una `Collection` de todos los valores del mapa; `entrySet` devuelve una vista que representa un `Set` de todas las entradas del mapa. Los cambios efectuados en el mapa se verán reflejados en cualquiera de las vistas, y los cambios efectuados en cualquier vista se verán reflejados en el mapa y también en las restantes vistas. Este comportamiento se ha ilustrado en la Figura 6.38 utilizando el método `remove` aplicado al conjunto de claves y a la colección de valores.

Existen muchos otros ejemplos de vistas en la API de Colecciones. En esta sección, vamos a ver dos de dichos usos del concepto de vista.

### 6.10.1 El método `subList` para objetos `List`

El método `subList` admite dos parámetros que representan índices de lista y devuelve una vista de una colección `List` cuyo rango incluye el primer índice y excluye el último índice. Así,

```
System.out.println( theList.subList( 3, 8 ) );
```

imprimirá los cinco elementos de la sublista. Puesto que `subList` es una lista, los cambios no estructurales en la sublista se reflejarán en el original y viceversa. Sin embargo, al igual que ocurre con los iteradores, una modificación estructural en la lista original invalidará la sublista. Finalmente, y quizás sea este el detalle más importante, como la sublista es una lista, y no una copia de una parte de la lista original, el coste de invocar `subList` es  $O(1)$  y las operaciones efectuadas sobre la sublista conservan su eficiencia inherente.

### 6.10.2 Los métodos `headSet`, `subSet` y `tailSet` para conjuntos `SortedSet`

La clase `SortedSet` tiene métodos que devuelven vistas del `Set`:

```
SortedSet<AnyType> subSet(AnyType fromElement, AnyType toElement);  
SortedSet<AnyType> headSet(AnyType toElement);  
SortedSet<AnyType> tailSet(AnyType fromElement);
```

`fromElement` y `toElement` lo que hacen es particionar en la práctica `SortedSet` en tres subconjuntos: el `headSet` (parte inicial), el `subSet` (parte central) y el `tailSet` (parte final). La Figura 6.42 ilustra esto partiendo una línea de números.

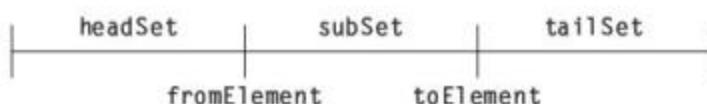


Figura 6.42 `headSet`, `subset` y `tailSet` posicionados sobre una línea de números.

En estos métodos, `toElement` no se incluye en ningún rango, mientras que `fromElement` sí se incluye. En Java 6, existen sobrecargas adicionales de estos métodos que permiten al llamante controlar si `fromElement` y `toElement` tienen que ser incluidos en cualquier rango concreto.

Por ejemplo, en un conjunto `Set<String>`, el número de palabras que empiezan por la letra 'x' está dado por:

```
words.subSet("x", true, "y", false).size()
```

El rango de un valor `val` en cualquier conjunto `Set s` (es decir, si `val` es el tercer valor mayor, tendrá rango 3) está dado por

```
s.tailSet(val).size()
```

Puesto que los subconjuntos son vistas, los cambios realizados en los subconjuntos se reflejarán en el original y viceversa, y las modificaciones estructurales realizadas en cualquiera de ellos se verán reflejadas en el otro. Como sucede con las listas, puesto que los subconjuntos son vistas y no copias de una parte del conjunto original, el coste de invocar `headSet`, `tailSet` o `subSet` es comparable con cualquiera de las otras operaciones con conjuntos, que será  $O(\log N)$ , y las operaciones `add`, `contains` y `remove` (pero no `size!`) sobre el subconjunto mantendrán su eficiencia inherente.

## Resumen

---

En este capítulo hemos examinado las estructuras básicas de datos que se utilizarán a lo largo del libro. Hemos proporcionado protocolos genéricos y hemos explicado cuál debería ser el tiempo de ejecución para cada estructura de datos. También hemos descrito la interfaz proporcionada por la API de Colecciones. En capítulos posteriores, mostraremos cómo se utilizan estas estructuras de datos y llegaremos a dar una implementación de cada estructura de datos que satisfaga las cotas de tiempo de ejecución que hemos expuesto aquí. La Figura 6.43 resume los resultados que se obtendrían para la secuencia de operaciones genéricas `insert`, `find` y `remove`.

El Capítulo 7 describe una importante herramienta de solución de problemas conocida con el nombre de *recursión*. La recursión permite resolver de forma eficiente muchos problemas utilizando algoritmos compactos y es crucial para la eficiente implementación de algoritmos de ordenación y de diversas estructuras de datos.

Estructura de datos	Acceso	Comentarios
Pila	Solo más reciente, <code>pop</code> , $O(1)$	Muy rápida
Cola	Solo el menos reciente, <code>dequeue</code> , $O(1)$	Muy rápida
Lista	Cualquier elemento	$O(N)$
TreeSet	Cualquier elemento por nombre o rango, $O(\log N)$	Caso promedio fácil de hacer; el caso peor requiere un cierto esfuerzo.
HashSet	Cualquier elemento por nombre, $O(1)$	Caso promedio.
Cola con prioridad	<code>findMin</code> , $O(1)$ , <code>deleteMin</code> , $O(\log N)$	<code>insert</code> es $O(1)$ como promedio, $O(\log N)$ en caso peor.

Figura 6.43 Resumen de algunas estructuras de datos.



## Conceptos clave

**análisis de precedencia de operadores** Un algoritmo que utiliza una pila para evaluar expresiones. (256)

**árbol de búsqueda binaria** Una estructura de datos que soporta la inserción, eliminación y búsqueda. También se puede utilizar para acceder al  $K$ -ésimo elemento más pequeño. El coste es un tiempo logarítmico de caso promedio para una implementación simple y un tiempo logarítmico de caso peor para una implementación más cuidadosa. (260)

**Arrays** Contiene un conjunto de métodos estáticos que operan sobre matrices. (242)

**cola** Una estructura de datos que restringe el acceso, limitándolo al elemento menos recientemente insertado. (256)

**cola con prioridad** Una estructura de datos que soporta únicamente el acceso al elemento mínimo. (270)

**Collection** Una interfaz que representa un grupo de objetos, a los que se conoce con el nombre de elementos. (232)

**Collections** Una clase que contiene un conjunto de métodos estáticos que opera sobre objetos `Collection`. (238)

**estructura de datos** Una representación de datos y las operaciones permitidas con esos datos, lo que hace que resulte posible la reutilización de componentes. (226)

**hashCode** Un método utilizado por `HashSet` que debe ser sustituido para los objetos si se sustituye el método `equals` del objeto. (264)

**HashMap** La implementación de la API de Colecciones de un `Map` con claves desordenadas. (264)

**hashSet** La implementación de la API de Colecciones de un `Set` desordenado. (260)

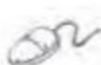
**iterador** Un objeto que permite acceder a los elementos de un contenedor. (236)

- Iterator** Interfaz de la API de Colecciones que especifica el protocolo para un iterador unidireccional. (236)
- LinkedList** Clase de la API de Colecciones que implementa una lista enlazada. (247)
- List** Interfaz de la API de Colecciones que especifica el protocolo para una lista. (244)
- lista** Una colección de elementos en la que los elementos tienen una posición. (244)
- lista enlazada** Una estructura de datos que se utiliza para evitar tener que desplazar grandes cantidades de datos. Utiliza una pequeña cantidad de espacio adicional por cada elemento. (247)
- ListIterator** Interfaz de la API de Colecciones que proporciona una iteración bidireccional. (244)
- Map** Interfaz de la API de Colecciones que abstrae una colección de parejas compuestas por claves y valores, y que asigna valores a las claves. (264)
- Map.Entry** Abstacta la idea de una pareja en un mapa. (268)
- método factoría** Un método que crea nuevas instancias concretas pero las devuelve utilizando una referencia a una clase abstracta. (230)
- montículo binario** Implementa la cola con prioridad en tiempo logarítmico por operación, utilizando una matriz. (271)
- pila** Una estructura de datos que restringe el acceso, limitándolo al elemento más recientemente insertado. (254)
- programar de acuerdo con una interfaz** La técnica de utilizar clases escribiendo el código en términos de la interfaz más abstracta. Trata incluso de ocultar el nombre de la clase concreta con la que se está trabajando. (232)
- Set** Interfaz de la API de Colecciones que abstrae una colección sin duplicados. (257)
- SortedSet** Interfaz de la API de Colecciones que abstrae un conjunto ordenado sin duplicados. (259)
- TreeMap** Implementación en la API de Colecciones de un Map con claves ordenadas. (264)
- TreeSet** Implementación en la API de Colecciones de un SortedSet. (259)



## Errores comunes

1. No se preocupe por las optimizaciones de bajo nivel hasta después haberse concentrado en las cuestiones básicas de diseño y de carácter algorítmico.
2. Cuando envíe un objeto función como parámetro, debe enviar un objeto construido, y no simplemente el nombre de la clase.
3. Al utilizar un Map, si no está seguro de si una clave se encuentra en el mapa, es posible que necesite utilizar containsKey en lugar de comprobar el resultado de get.
4. Una cola con prioridad no es una cola. Simplemente parece serlo.



## Internet

Hay una gran cantidad de código en este capítulo. El código de prueba se encuentra en el directorio raíz, los protocolos no estándar están en el paquete `weiss.nonstandard` y todo lo demás se encuentra en el paquete `weiss.util`.

<b>Collection.java</b>	Contiene el código de la Figura 6.9.
<b>Iterator.java</b>	Contiene el código de la Figura 6.10.
<b>Collections.java</b>	Contiene el código de las Figuras 6.13 y 6.14.
<b>Arrays.java</b>	Contiene el código de la Figura 6.15.
<b>List.java</b>	Contiene el código de la Figura 6.16.
<b>ListIterator.java</b>	Contiene el código de la Figura 6.17.
<b>TestArrayList.java</b>	Ilustra <code>ArrayList</code> , como en la Figura 6.18.
<b>Set.java</b>	Contiene el código de la Figura 6.29. El código en línea contiene un método extra que no forma parte de Java 5.
 <b>Stack.java</b>	Contiene el protocolo no estándar de la Figura 6.26.
 <b>UnderflowException.java</b>	Contiene una excepción no estándar.
<b>Queue.java</b>	Contiene la interfaz estándar de la Figura 6.28.
<b>SortedSet.java</b>	Contiene el código de la Figura 6.30.
<b>TreeSetDemo.java</b>	Contiene el código de las Figuras 6.11 y 6.31.
<b>IteratorTest.java</b>	Contiene el código que ilustra todos los iteradores, incluyendo el código de las Figuras 6.11, 6.32 y 6.35.
 <b>EqualsWithInheritance.java</b>	Contiene el código de las Figuras 6.33 y 6.34, combinadas en uno solo fragmento de código.
<b>Map.java</b>	Contiene el código de las Figuras 6.36 y 6.37.
<b>MapDemo.java</b>	Contiene el código de la Figura 6.38.
<b>DuplicateFinder.java</b>	Contiene el código de la Figura 6.39.
<b>PriorityQueueDemo.java</b>	Contiene el código de la Figura 6.41.



## Ejercicios

### EN RESUMEN

- 6.1** Muestre los resultados de la siguiente secuencia: `add(2)`, `add(7)`, `add(0)`, `add(4)`, `remove()` y `remove()` cuando las operaciones `add` y `remove` se corresponden con las operaciones básicas de las siguientes estructuras de datos:
- Pila.

- b. Cola
- c. Cola con prioridad

## EN TEORÍA

- 6.2** Una cola de doble frente soporta las inserciones y borrados tanto al final como al principio de la cola. ¿Cuál es el tiempo de ejecución por cada operación?
- 6.3** ¿Cuáles de las estructuras de datos de la Figura 6.43 permitirían obtener algoritmos de ordenación que se ejecuten en un tiempo inferior al cuadrático (insertando todos los elementos en la estructura de datos y luego eliminándolos por orden)?
- 6.4** ¿Pueden soportarse en tiempo logarítmico todas las operaciones siguientes: `insert`, `deleteMin`, `deleteMax`, `findMin` y `findMax`?
- 6.5** Considere el siguiente método, cuya implementación no se muestra:

```
// Precondición: la colección c representa una Collection de
// otras colecciones.
// c no es null; ninguna de las colecciones es null
// str no es null
// Postcondición: devuelve el número de apariciones del
// objeto String str en c.
public static
int count( Collection<Collection<String>> c, String str )
```

- Proporcione una implementación de `count`.
- Asuma que la `Collection` `c` contiene  $N$  colecciones y que cada una de esas colecciones contiene  $N$  objetos. ¿Cuál es el tiempo de ejecución de `count`, tal como lo ha escrito en el apartado (a)?
- Suponga que se tarda 2 milisegundos en ejecutar `count` cuando  $N$  (especificada) es 100. Asumiendo que los términos de menor orden son despreciables, ¿cuánto se tardaría en ejecutar `count` cuando  $N$  sea 500?

## EN LA PRÁCTICA

- 6.6** Demuestre que las siguientes operaciones pueden soportarse simultáneamente en tiempo constante: `push`, `pop` y `findMin`. Observe que `deleteMin` no forma parte del repertorio. *Pista:* mantenga dos pilas –una para almacenar los elementos y otra para almacenar los mínimos a medida que aparezcan.
- 6.7** Demuestre cómo implementar una cola eficientemente utilizando una `List` como miembro de datos.
- 6.8** Escriba una rutina que emplee la API de Colecciones para imprimir los elementos de cualquier `Collection` en orden inverso. No utilice un `ListIterator`.
- 6.9** Demuestre cómo implementar una pila de manera eficiente utilizando una `List` como miembro de datos.

- 610** `hasSpecial`, mostrado a continuación, devuelve un valor verdadero si hay dos números distintos de la lista cuya suma es igual a un tercer número de la lista. Suponga que  $N$  es el tamaño de la lista.

```
// Devuelve true si la suma de dos números de c es igual a
// un tercer número de c.
public static boolean hasSpecial( List<Integer> c )
{
    for( int i = 0; i < c.size( ); i++ )
        for( int j = i + 1; j < c.size( ); j++ )
            for( int k = 0; k < c.size( ); k++ )
                if( c.get( i ) + c.get( j ) == c.get( k ) )
                    return true;

    return false;
}
```

- ¿Cuál es el tiempo de ejecución de `hasSpecial` cuando la lista es un `ArrayList`?
- ¿Cuál es el tiempo de ejecución de `hasSpecial` cuando la lista es una `LinkedList`?

- 611** `equals`, mostrado a continuación, devuelve un valor verdadero si las dos listas tienen el mismo tamaño y contienen los mismos elementos en el mismo orden. Suponga que  $N$  es el tamaño de ambas listas.

```
public boolean equals( List<Integer> lhs, List<Integer> rhs )
{
    if( lhs.size( ) != rhs.size( ) )
        return false;

    for( int i = 0; i < lhs.size( ); i++ )
        if( !lhs.get( i ).equals( rhs.get( i ) ) )
            return false;

    return true;
}
```

- ¿Cuál es el tiempo de ejecución de `equals` cuando la lista es un `ArrayList`?
- ¿Cuál es el tiempo de ejecución de `equals` cuando la lista es una `LinkedList`?

- 612** `containsAll` devuelve un valor verdadero si la primera lista contiene todos los elementos de la segunda lista. Suponga que las listas tienen aproximadamente el mismo tamaño y que constan de unos  $N$  elementos cada una.

```
public static boolean containsAll( List<Integer> bigger,
                                 List<Integer> items )
{
    outer:
    for( int i = 0; i < bigger.size( ); i++ )
    {
```

```

        Integer itemToFind = bigger.get( i );

        for( int j = 0; j < items.size( ); j++ )
            if( items.get( j ).equals( itemToFind ) ) // match
                continue outer;

        // Si llegamos aquí, no hay ninguna entrada de items
        // que se corresponda con bigger.get(i)
        return false;
    }
    return true;
}

```

- Suponga que se tarda 10 segundos en ejecutar `containsAll` con dos listas `ArrayList` de 1.000 elementos idénticos. ¿Cuánto se tardará en ejecutar `containsAll` con dos listas `ArrayList` de 2.000 elementos idénticos?
- Explique en una frase cómo hacer el algoritmo eficiente para todos los tipos de listas.

**6.13** `intersect`, mostrado a continuación, devuelve el número de elementos que se encuentran en ambas listas. Suponga que ambas listas tienen  $N$  elementos.

```

// Devuelve el número de elementos que están a la vez en c1 y c2
// Asume que no hay duplicados en ninguna lista.
public static int intersect ( List<Integer> c1, List<Integer> c2 )
{
    int count = 0;

    for( int i = 0; i < c1.size( ); i++ )
    {
        int item1 = c1.get( i );
        for( int j = 0; j < c2.size( ); j++ )
        {
            if( c2.get( j ) == item1 )
            {
                count++;
                break;
            }
        }
    }
    return count;
}

```

- ¿Cuál es el tiempo de ejecución de `intersect` cuando la lista es un `ArrayList`?
- ¿Cuáles es el tiempo de ejecución de `intersect` cuando la lista es una `LinkedList`?

- 6.14** Considere la siguiente implementación del método `clear` (que vacía cualquier colección).

```
public abstract class AbstractCollection<AnyType>
    implements Collection<AnyType>

{
    public void clear( )
    {
        Iterator<AnyType> itr = this.iterator( );
        while( itr.hasNext( ) )
        {
            itr.next( );
            itr.remove( );
        }
    }
    ...
}
```

- Suponga que `LinkedList` amplía `AbstractCollection` y no sustituye `clear`. ¿Cuál es el tiempo de ejecución de `clear`?
- Suponga que `ArrayList` amplía `AbstractCollection` y no sustituye `clear`. ¿Cuál es el tiempo de ejecución de `clear`?
- Suponga que se tardan 4 segundos en ejecutar `clear` sobre un `ArrayList` de 10.000 elementos. ¿Cuánto se tardará en ejecutar `clear` sobre un `ArrayList` de 500.000 elementos?
- Describa lo más claramente posible el comportamiento de esta implementación alternativa de `clear`:

```
public void clear( )
{
    for( AnyType item : this )
        this.remove( item );
}
```

- 6.15** El método estático `removeHalf` elimina la primera mitad de una `List` (si hay un número impar de elementos, entonces se elimina algo menos de la mitad de la lista). Una posible implementación de `removeHalf` es la que se muestra a continuación:

```
public static void removeHalf( List<?> lst )
{
    int size = lst.size( );

    for( int i = 0; i < size / 2; i++ )
        lst.remove( 0 );
}
a. ¿Por qué no podemos utilizar lst.size( )/2 como comprobación en el bucle for?
```

- b. ¿Cuál es el tiempo de ejecución O mayúscula si `lst` es un `ArrayList`?
- c. ¿Cuál es el tiempo de ejecución O mayúscula si `lst` es un `LinkedList`?

- 616** `containsSum`, mostrado a continuación, devuelve un valor verdadero si hay dos números de la lista cuya suma sea igual a  $K$ . Suponga que  $N$  es el tamaño de la lista.

```
public static boolean containsSum( List<Integer> lst, int K )
{
    for( int i = 0; i < lst.size( ); i++ )
        for( int j = i + 1; j < lst.size( ); j++ )
            if( lst.get( i ) + lst.get( j ) == K )
                return true;

    return false;
}
```

- a. ¿Cuál es el tiempo de ejecución de `containsSum` cuando la lista es un `ArrayList`?
- b. ¿Cuál es el tiempo de ejecución de `containsSum` cuando la lista es una `LinkedList`?
- c. Suponga que se tardan 2 segundos en ejecutar `containsSum` sobre un `ArrayList` de 1.000 elementos, ¿cuánto se tardará en ejecutar `containsSum` sobre un `ArrayList` de 3.000 elementos? Puede suponer que `containsSum` devuelve valor falso en ambos casos.

- 617** Escriba un programa de prueba para ver cuál de las siguientes llamadas consigue borrar con éxito todos los elementos de una `LinkedList` de java.

```
c.removeAll( c );
c.removeAll( c.subList( 0, c.size( ) ) );
```

- 618** El método estático `removeEveryOtherItem` elimina los elementos en las posiciones pares (0, 2, 4, etc.) de una lista `List`. Una posible implementación de `removeEveryOtherItem` es la que se muestra a continuación.

```
public static void removeEveryOtherItem( List<?> lst )
{
    for( int i = 0; i < lst.size( ); i++ )
        lst.remove( i );
}
```

- a. ¿Cuál es el tiempo de ejecución O mayúscula si `lst` es un `ArrayList`?
- b. ¿Cuál es el tiempo de ejecución O mayúscula si `lst` es un `LinkedList`?

- 619** Considere la siguiente implementación del método `removeAll` (que elimina todas las apariciones de todos los elementos en la colección que se le pasa como parámetro).

```
public abstract class AbstractCollection<AnyType>
    implements Collection<AnyType>
{
```

```
public boolean removeAll ( Collection<? extends AnyType> c )
{
    Iterator<AnyType> itr = this.iterator();
    boolean wasChanged = false;

    while( itr.hasNext() )
    {
        if( c.contains( itr.next() ) )
        {
            itr.remove();
            wasChanged = true;
        }
    }
    return wasChanged;
}
...
}
```

- Suponga que `LinkedList` amplía `AbstractCollection` y no sustituye `removeAll`. ¿Cuál es el tiempo de ejecución de `removeAll` cuando `c` es una `List`?
  - Suponga que `LinkedList` amplía `AbstractCollection` y no sustituye `removeAll`. ¿Cuál es el tiempo de ejecución de `removeAll` cuando `c` es un `TreeSet`?
  - Suponga que `ArrayList` amplía `AbstractCollection` y no sustituye `removeAll`. ¿Cuál es el tiempo de ejecución de `removeAll` cuando `c` es una `List`?
- 6.20** El método `changeList` sustituye cada `String` de la lista por sus equivalentes en mayúscula y minúscula. Así, si la lista original contenía `[Hello,NewYork]`, entonces la nueva lista contendrá `[hello, HELLO, newyork, NEWYORK]`. Utilice los métodos `add` y `remove` de `ListIterator` para escribir una implementación eficiente del método `changeList` para listas enlazadas

```
public static void changeList( LinkedList<String> theList )
```

- 6.21** El método `set` de `ListIterator` permite al llamante cambiar el valor del último elemento visualizado. Implemente el método `toUpperCase` (que pasa la lista a mayúsculas) mostrado a continuación utilizando un `ListIterator`:

```
public static void toUpper( List<String> theList )
```

## PROYECTOS DE PROGRAMACIÓN

- 6.22** La interfaz `RandomAccess` no contiene ningún método, sino que pretende servir de marcador: una clase `List` implementa la interfaz solo si sus métodos `get` y `set` son muy eficientes. De la misma forma, `ArrayList` implementa la interfaz `RandomAccess`. Implemente el método estático `removeEveryOtherItem`, descrito en el Ejercicio 6.18. Si `list` implementa `RandomAccess` (utilice una comprobación

`instanceof`), entonces emplee `get` y `set` para reposicionar los elementos en la mitad inicial de la lista. En caso contrario, use un iterador que sea eficiente para listas enlazadas.

- 6.23** Escriba, en el menor número de líneas posible, un código que elimine todas las entradas de un `Map` cuyo valor sea `null`.
- 6.24** Las operaciones soportadas por `SortedSet` también pueden implementarse utilizando una matriz y manteniendo el tamaño actual. Los elementos de la matriz se almacenan ordenados en posiciones consecutivas de la matriz. Así, `contains` puede implementarse mediante una búsqueda binaria. Haga lo siguiente:
- Describa los algoritmos para `add` y `remove`.
  - ¿Cuál es el tiempo de ejecución de estos algoritmos?
  - Escriba una implementación que use estos algoritmos, utilizando el protocolo de la Figura 6.1.
  - Escriba una implementación que use estos algoritmos, utilizando el protocolo estándar `SortedSet`.
- 6.25** Puede implementarse una cola con prioridad almacenando los elementos en una matriz desordenada e insertando los elementos en la siguiente ubicación disponible. Haga lo siguiente:
- Describa los algoritmos para `findMin`, `deleteMin` e `insert`.
  - ¿Cuál será el tiempo de ejecución O mayúscula para `findMin`, `deleteMin` e `insert` utilizando estos algoritmos?
  - Escriba una implementación que utilice estos algoritmos.
- 6.26** Una cola se puede implementar utilizando una matriz y manteniendo el tamaño actual. Los elementos de la cola se almacenan en posiciones consecutivas de la matriz, estando siempre el elemento inicial en la posición 0. Observe que este no es el método más eficiente. Haga los siguientes:
- Describa los algoritmos para `getFront`, `enqueue` y `dequeue`.
  - ¿Cuál será el tiempo de ejecución O mayúscula para `getFront`, `enqueue` y `dequeue` utilizando estos algoritmos?
  - Escriba una implementación que utilice estos algoritmos utilizando el protocolo de la Figura 6.28.
- 6.27** Una cola con prioridad de doble extremo permite acceder tanto al elemento mínimo como al máximo. En otras palabras, están soportadas las siguientes operaciones: `findMin`, `deleteMin`, `findMax` y `deleteMax`. Haga lo siguiente:
- Describa los algoritmos para `insert`, `findMin`, `deleteMin`, `findMax` y `deleteMax`.
  - ¿Cuál será el tiempo de ejecución O mayúscula para `findMin`, `deleteMin`, `findMax`, `deleteMax` e `insert` utilizando estos algoritmos?
  - Escriba una implementación que emplee estos algoritmos.

- 6.28** Asegurándose de que los elementos de la cola con prioridad estén ordenados de forma decreciente (es decir, que el mayor elemento sea el primero y el menor sea el último), pueden implementarse tanto `findMin` con `deleteMin` en tiempo constante. Sin embargo, `insert` es una operación cara. Haga lo siguiente:
- Describa los algoritmos para `insert`, `findMin` y `deleteMin`.
  - ¿Cuál será el tiempo de ejecución O mayúscula para `insert`?
  - Escriba una implementación que utilice estos algoritmos.

- 6.29** Un `MultiSet` es como un `Set`, pero permite duplicados. Considere la siguiente interfaz para un `MultiSet`:

```
public interface MultiSet<AnyType>
{
    void add( AnyType x );
    boolean contains( AnyType x );
    int count( AnyType x );
    boolean removeOne( AnyType x );
    boolean removeAll( AnyType x );
    void toArray( AnyType [] arr );
}
```

Hay muchas formas de implementar la interfaz `MultiSet`. Un `TreeMultiSet` almacena los elementos ordenados. La representación de datos puede ser un `TreeMap`, en el que la clave será un elemento perteneciente al multiconjunto y el valor representará el número de veces que el elemento esté almacenado. Implemente el `TreeMultiSet` y asegúrese de proporcionar un método `toString`.

- 6.30** La estructura de datos denominada montículo intermedial soporta las siguientes operaciones: `insert`, `findKth` y `removeKth`. Las dos últimas se encargan de encontrar y eliminar, respectivamente, el  $k$ -ésimo elemento más pequeño (donde  $k$  es un parámetro). La implementación más simple mantiene los datos ordenados. Haga lo siguiente:
- Describa los algoritmos que se pueden utilizar para soportar las operaciones del montículo intermedial.
  - ¿Cuál será el tiempo de ejecución O mayúscula para cada una de las operaciones básicas, utilizando estos algoritmos?
  - Escriba una implementación que utilice estos algoritmos.

- 6.31** Escriba un método que elimine uno de cada dos elementos en una `List`. La rutina debe ejecutarse en tiempo lineal y emplear un espacio adicional constante si la lista `List` es una `LinkedList`.

- 6.32** `Collections.fill` toma una lista `List` y un valor `value`, e introduce `value` en todas las posiciones de la lista. Implemente `fill`.

- 6.33** Escriba un método que tome como parámetro un `Map<String, String>` y devuelva un nuevo `Map<String, String>` en el que las claves y los valores estén

intercambiados. Genere una excepción si hay valores duplicados en el mapa que se pasa como parámetro.

- 6.34** `Collections.reverse` toma una lista `List` e invierte su contenido. Implemente `reverse`.
- 6.35** Escriba un programa que lea cadenas de caracteres de entrada y las imprima ordenadas según su longitud, comenzando por la cadena de caracteres más corta. Si hay un subconjunto de cadenas de entrada que tienen todas ellas la misma longitud, el programa debe imprimirlas por orden alfabético.



## Referencias

Las referencias para la teoría que subyace a estas estructuras de datos se proporcionan en la Parte Cuatro. La API de Colecciones se describe en los libros Java más recientes (véanse las referencias del Capítulo 1).

# Recursión

A un método que esté parcialmente definido en términos de sí mismo se le denomina *recursivo*. Al igual que otros muchos lenguajes, Java soporta los métodos recursivos. La recursión, que es el uso de métodos recursivos, es una potente herramienta de programación que en muchos casos puede proporcionar algoritmos que son, a la vez, cortos y eficientes. En este capítulo vamos a explorar cómo funciona la recursión, proporcionando así ideas acerca de sus variaciones, limitaciones y uso. Comenzaremos nuestras explicaciones sobre la recursión examinando el principio matemático en el que se basa esta técnica: la *inducción matemática*. Después proporcionaremos ejemplos de métodos recursivos simples y demostraremos que esos métodos generan las respuestas correctas.

En este capítulo, vamos a ver

- Las cuatro reglas básicas de la recursión.
- Aplicaciones numéricas de la recursión, que conducen a la implementación de un algoritmo de cifrado.
- Una técnica general denominada *divide y vencerás*.
- Una técnica general denominada *programación dinámica*, que es similar a la recursión, pero que utiliza tablas en lugar de emplear llamadas a métodos recursivos.
- Una técnica general denominada *retroceso*, que equivale a una cuidadosa búsqueda exhaustiva.

## 7.1 ¿Qué es la recursión?

Un *método recursivo* es un método que hace, directa o indirectamente, una llamada a sí mismo. Esta acción puede parecer similar a los razonamientos circulares: ¿cómo puede un método *F* resolver un problema llamándose a sí mismo? La clave está en que el método *F* se llama a sí mismo dentro de un contexto diferente, que generalmente es más simple que el anterior. He aquí algunos ejemplos.

Un *método recursivo* es un método que hace, directa o indirectamente, una llamada a sí mismo.

- Los archivos de una computadora suelen estar almacenados en directorios. Los usuarios pueden crear subdirectorios que almacenen más archivos y directorios. Suponga que deseamos examinar cada archivo de un directorio *D* incluyendo todos los archivos de todos los

subdirectorios (y de sus sub-subdirectorios, etc.). Podemos hacerlo examinando recursivamente cada archivo de cada subdirectorio y luego examinando todos los archivos del directorio  $D$  (como se explica en el Capítulo 18).

- Suponga que tenemos un diccionario de gran tamaño. Las palabras de los diccionarios se definen utilizando otras palabras. Cuando buscamos el significado de un término puede que, en ocasiones, no entendamos la definición y que tengamos que buscar el significado de algunas de las palabras que esa definición contiene. De forma similar, podríamos no entender algunas de las definiciones de esas otras palabras, lo que nos llevaría a tener que continuar con nuestra búsqueda durante un cierto tiempo. Como el diccionario es finito, terminaremos llegando a un punto en el que comprenderemos todas las palabras de una cierta definición (y por tanto entenderemos la propia definición gracias a las otras definiciones), o en el que encontraremos que las definiciones son circulares y estamos en un callejón sin salida o en el que comprobemos que alguna palabra que necesitamos comprender no está definida en el diccionario. Nuestra estrategia recursiva para comprender las palabras es la siguiente: si sabemos el significado de una palabra, habremos terminado; en caso contrario, la buscamos en el diccionario. Si comprendemos todas las palabras de la definición habremos terminado. En caso contrario, intentamos averiguar qué significa la definición buscando recursivamente aquellas palabras que no conoczamos. Este procedimiento terminará siempre si el diccionario está bien definido, aunque podríamos entrar en un bucle infinito si una cierta palabra estuviera definida de forma circular.
- Los lenguajes informáticos suelen definirse frecuentemente de manera recursiva. Por ejemplo, una expresión aritmética es un objeto o una expresión entre paréntesis, dos expresiones sumadas, etc.

La recursión es una potente herramienta de resolución de problemas. La manera más fácil de expresar muchos algoritmos es mediante una formulación recursiva. Además, las soluciones más eficientes a muchos problemas están basadas en esta formulación recursiva natural. Pero debemos tener cuidado de no crear una lógica circular que termine haciéndonos entrar en un bucle infinito.

En este capítulo, vamos a explicar las condiciones generales que deben satisfacer los algoritmos recursivos y vamos a proporcionar varios ejemplos prácticos. Veremos que en ocasiones existen algoritmos que se expresan naturalmente de forma recursiva, pero que deben ser reescritos sin utilizar la recursión.

## 7.2 Fundamentos: demostraciones por inducción matemática

La *inducción* es una importante técnica de demostración que se utiliza para demostrar teoremas que se cumplen para enteros positivos.

En esta sección vamos a hablar del concepto de demostración por *inducción* matemática. (A lo largo del capítulo omitiremos la palabra matemática a la hora de describir esta técnica.) La *inducción* suele emplearse para demostrar teoremas que se cumplen para enteros positivos. Vamos a comenzar demostrando un teorema simple, el Teorema 7.1. Este teorema concreto puede demostrarse fácilmente mediante otros métodos, pero a menudo la demostración por inducción resulta ser el mecanismo de demostración más simple.

**Teorema 7.1**

Para cualquier entero  $N \geq 1$ , la suma de los  $N$  primeros enteros, dada por  $\sum_{i=1}^N i = 1+2+\dots+N$ , es igual a  $N(N+1)/2$ .

Obviamente, el teorema es cierto para  $N = 1$  porque tanto el lado izquierdo como el derecho de la igualdad tienen el valor 1. Una comprobación adicional muestra que también es cierto para  $2 \leq N \leq 10$ . Sin embargo, el hecho de que el teorema se cumpla para todos los valores  $N$  que se pueden comprobar fácilmente a mano, no implica que se cumpla para todo posible valor de  $N$ . Considere, por ejemplo, los números de la forma  $2^{2^k} + 1$ . Los primeros cinco números (correspondientes a  $0 \leq k \leq 4$ ) son 3, 5, 17, 257 y 65.537. Todos estos números son primos. De hecho, hubo una época en la que los matemáticos plantearon la conjetura de que todos los números de esta forma son primos. Sin embargo, resulta que no es cierto. Podemos demostrarlo fácilmente, comprobando mediante una computadora que  $2^{2^5} + 1 = 641 \times 6.700.417$ . De hecho, no se conoce ningún otro número primo de la forma  $2^{2^k} + 1$ , aparte de los ya citados.

Una demostración por inducción se lleva a cabo en dos etapas. En primer lugar, como acabamos de hacer, demostramos que el teorema es cierto para los valores más pequeños, luego demostramos que si el teorema es cierto para los primeros casos, puede ampliarse para incluir el caso siguiente. Por ejemplo, demostramos que si un teorema es cierto para todo  $1 \leq N \leq k$ , entonces también debe ser cierto para  $1 \leq N \leq k + 1$ . Una vez que hayamos demostrado cómo ampliar el rango de casos para los que el teorema se cumple, habremos demostrado que se cumple para todos los casos. La razón es, obviamente, que podemos ampliar indefinidamente el rango de casos para los que el teorema se cumple. Vamos a utilizar esta técnica para demostrar el Teorema 7.1.

Una demostración por inducción demuestra que el teorema se cumple para algunos casos simples y luego demuestra cómo ampliar indefinidamente el rango de casos para los que el teorema es cierto.

**Demostración del Teorema 7.1**

Evidentemente, el teorema es cierto para  $N = 1$ . Suponga que el teorema es cierto para todo  $1 \leq N \leq k$ . Entonces

$$\sum_{i=1}^{k+1} i = (k+1) + \sum_{i=1}^k i \quad (7.1)$$

Por hipótesis, el teorema es cierto para  $k$ , así que podemos sustituir el sumatorio del lado derecho de la Ecuación 7.1 por  $k(k+1)/2$ , obteniendo

$$\sum_{i=1}^{k+1} i = (k+1) + k(k+1)/2 \quad (7.2)$$

Una manipulación algebraica del lado derecho de la Ecuación 7.2 nos da

$$\sum_{i=1}^{k+1} i = (k+1)(k+2)/2$$

Este resultado confirma el teorema para el caso  $k + 1$ . Por tanto, por inducción, el teorema es cierto para todos los enteros  $N \geq 1$ .

¿Por qué esto constituye una demostración? En primer lugar, el teorema es cierto para  $N = 1$ , lo que se denomina la *base*. Podemos contemplar este hecho como si fuera la base de nuestra creencia de que el teorema es cierto con carácter general. En una demostración por inducción, la base es el caso fácil que puede demostrarse a mano. Una vez que hayamos establecido la base, utilizamos la *hipótesis inductiva* para asumir que el teorema es cierto para un valor  $k$

En una demostración por inducción, la base es el caso sencillo que puede ser demostrado a mano.

**La hipótesis inductiva** supone que el teorema es cierto para un caso arbitrario y que, bajo esa suposición, también es cierto para el caso siguiente.

arbitrario y luego, con esa suposición, demostramos que si el teorema es cierto para  $k$ , entonces es cierto para  $k + 1$ . En nuestro caso, sabemos que el teorema es cierto para la base  $N = 1$ , así que también es cierto para  $N = 2$ . Puesto que es cierto para  $N = 2$ , tiene que ser cierto para  $N = 3$ . Y como es cierto para  $N = 3$ , tiene que serlo para  $N = 4$ . Ampliando esta manera de razonar, sabemos que el teorema es cierto para todo entero positivo a partir de  $N = 1$ .

Aplicaremos la demostración por inducción a un segundo problema, no tan simple como el primero. En primer lugar, examinemos 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 de los primeros  $N$  cuadrados, con signos alternativos. El valor de esta secuencia será 1, 3, 6, 10 y 15. Por tanto, en general, la suma parece ser igual a la suma de los  $N$  primeros enteros, la cual es igual, como ya sabemos por el Teorema 7.1, a  $N(N + 1)/2$ . El Teorema 7.2 demuestra este resultado.

### Teorema 7.2

La suma  $\sum_{i=1}^N (-1)^{N-i} i^2 = N^2 - (N-1)^2 + (N-2)^2 - \dots$  es  $N(N+1)/2$ .

#### Demostración

La demostración es por inducción.

Base: evidentemente, el teorema es cierto para  $N = 1$ .

Hipótesis inductiva: en primer lugar, suponemos que el teorema es cierto para  $k$ :

$$\sum_{i=k}^1 (-1)^{k-i} i^2 = \frac{k(k+1)}{2}$$

A continuación deberemos demostrar que es cierto para  $k + 1$ ,

$$\sum_{i=k+1}^1 (-1)^{k+1-i} i^2 = \frac{(k+1)(k+2)}{2}$$

Escribimos

$$\sum_{i=k+1}^1 (-1)^{k+1-i} i^2 = (k+1)^2 - k^2 + (k-1)^2 - \dots \quad (7.3)$$

Si volvemos a escribir el lado derecho de la ecuación de la Ecuación 7.3, obtenemos

$$\sum_{i=k+1}^1 (-1)^{k+1-i} i^2 = (k+1)^2 - (k^2 - (k-1)^2 + \dots)$$

y sustituyendo obtenemos

$$\sum_{i=k+1}^1 (-1)^{k+1-i} i^2 = (k+1)^2 - \left( \sum_{i=k}^1 (-1)^{k-i} i^2 \right) \quad (7.4)$$

Si aplicamos la hipótesis inductiva, entonces podemos sustituir el sumatorio del lado derecho de la Ecuación 7.4, obteniendo

$$\sum_{i=k+1}^1 (-1)^{k+1-i} i^2 = (k+1)^2 - k(k+1)/2 \quad (7.5)$$

Una simple manipulación algebraica en el lado derecho de la Ecuación 7.5 nos da

$$\sum_{i=k+1}^1 (-1)^{k+1-i} i^2 = (k+1)(k+2)/2$$

lo que demuestra el teorema para  $N = k + 1$ . Por tanto, por inducción, el teorema es cierto para todo  $N \geq 1$ .

## 7.3 Recursión básica

Las demostraciones por inducción nos muestran que si sabemos que una afirmación es cierta para el caso más pequeño y podemos demostrar que un caso implica el siguiente, entonces sabemos que la afirmación es cierta para todos los casos.

En ocasiones, hay funciones matemáticas que se definen de forma recursiva. Por ejemplo, sea  $S(N)$  la suma de los  $N$  primeros enteros. Entonces,  $S(1) = 1$  y podemos escribir  $S(N) = S(N - 1) + N$ . Aquí hemos definido la función  $S$  en términos de una instancia más pequeña de sí misma. La definición recursiva de  $S(N)$  es prácticamente idéntica a la forma explícita  $S(N) = N(N + 1)/2$ , con la única excepción de que la definición recursiva solo está definida para enteros positivos y se puede calcular de una forma menos directa.

Un método recursivo se define en términos de una instancia más pequeña de sí mismo. Debe existir algún caso base que pueda calcularse sin necesidad de usar la recursión.

En ocasiones, escribir una fórmula recursivamente es más fácil que escribirla en forma explícita. La Figura 7.1 muestra una implementación directa de la función recursiva. Si  $N = 1$ , tendremos la base, para la cual sabemos que  $S(1) = 1$ . Tenemos este caso en las líneas 4 y 5. En caso contrario, aplicamos la definición recursiva  $S(N) = S(N - 1) + N$  precisamente en la línea 7. Es complicado imaginar que se pudiera implementar el método recursivo de una forma más simple que esta, por lo que la cuestión natural que se plantea es: ¿funciona realmente el método?

La respuesta, salvo por lo que en breve comentaremos, es que esta rutina funciona perfectamente. Examinemos cómo se evaluaría la llamada a `s(4)`. Cuando se hace la llamada a `s(4)`, la comprobación de la línea 4 falla. A continuación ejecutamos la línea 7, en la que se evalúa `s(3)`. Como sucede con cualquier otro método, esta evaluación requiere una llamada a `s`. En dicha llamada, llegamos a la línea 4, donde la comprobación falla; por tanto, pasamos a la línea 7. En ese punto, llamamos a `s(2)`. De nuevo, llamamos a `s` y ahora `n` será 2. La comprobación de la línea 4 sigue fallando, así que invocamos `s(1)` en la línea 7. Ahora tenemos que `n` es igual a 1, por lo que `s(1)` devolverá 1. En este punto, `s(2)` puede continuar, sumando el valor de retorno de `s(1)` a 2; por tanto, `s(2)` devuelve 3. Ahora `s(3)` continuará su ejecución, sumando el valor 3 devuelto por `s(2)` a `n`, lo que es 3; por tanto `s(3)` devolverá 6. Este resultado permite completar la llamada a `s(4)`, que termina por devolver el valor 10.

Observe que, aunque `s` parece estar llamándose a sí mismo, en realidad está llamando a un clon de sí mismo. Ese clon es simplemente otro método con parámetros distintos. En cualquier instante determinado solo habrá un clon activo; el resto de los clones tendrán su ejecución pendiente. Es misión de la computadora, no nuestra, encargarse de gestionar todas esas llamadas. Si esas tareas

```

1 // Evaluar la suma de los n primeros enteros.
2 public static long s( int n )
3 {
4     if( n == 1 )
5         return 1;
6     else
7         return s( n - 1 ) + n;
8 }
```

Figura 7.1 Evaluación recursiva de la suma de los  $N$  primeros enteros.

El caso base es una instancia que se puede resolver sin necesidad de recursión. Toda llamada recursiva debe progresar hacia un caso base.

de gestión implicaran una carga excesiva para la computadora, entonces ya sí que tendríamos que preocuparnos. Hablaremos de estos detalles más adelante en el capítulo.

Un *caso base* es una instancia que se puede resolver sin necesidad de recursión. Toda llamada recursiva debe progresar hacia el caso base, si queremos que la ejecución termine en algún momento. Con esto podemos plantear nuestras dos primeras (de un total de cuatro) *reglas de recursión* fundamentales.

1. *Caso base*: siempre tiene que haber al menos un caso que se pueda resolver sin utilizar recursión.
2. *Progresión*: toda llamada recursiva debe progresar hacia un caso base.

Nuestra rutina de evaluación recursiva presenta, de todos modos, algunos problemas. Uno de ellos es la llamada a `s(0)`, para la que el método no se comporta adecuadamente.<sup>1</sup> Este comportamiento es natural porque la definición recursiva de  $S(N)$  no permite que  $N < 1$ . Podemos resolver este problema ampliando la definición de  $S(N)$  para incluir  $N = 0$ . Puesto que no hay ningún número que sumar en este caso, un valor natural para  $S(0)$  sería 0. Este valor tiene sentido, porque la definición recursiva puede seguir aplicándose a  $S(1)$ , ya que  $S(0) + 1$  es 1. Para implementar este cambio, simplemente sustituimos 1 por 0 en las líneas 4 y 5. Los valores negativos de  $N$  también hacen que se produzcan errores, pero este problema se puede resolver de forma similar y se deja como tarea para el lector en el Ejercicio 7.6.

Un segundo problema es que, si el parámetro  $n$  es grande, pero no tan grande que la respuesta no pueda caber en un `int`, el programa puede fallar o quedarse colgado. Nuestro sistema informático, por ejemplo, no puede manejar los casos de  $N \geq 8.882$ . La razón es que, como ya hemos comentado, la implementación de la recursión requiere que el sistema lleve la cuenta de las llamadas recursivas pendientes, y para cadenas de recursión suficientemente grandes, la computadora simplemente se queda sin memoria. Explicaremos esta condición con más detalle posteriormente en el capítulo. Esta rutina, asimismo, requiere algo más de tiempo de ejecución que la utilización de un bucle equivalente, debido a que ese control de las llamadas recursivas pendientes también requiere un cierto tiempo.

No hace falta decir que este ejemplo concreto no ilustra el mejor uso de la recursión, ya que se trata de un problema que se puede resolver muy fácilmente sin utilizar esta técnica. La mayoría de las buenas aplicaciones de la recursión no agotan la memoria de la computadora y solo consumen un tiempo ligeramente mayor que las implementaciones no recursivas. A cambio de ese ligero incremento en el tiempo de ejecución, la recursión casi siempre permite obtener un código más compacto.

### 7.3.1 Impresión de números en cualquier base

Un buen ejemplo de cómo la recursión simplifica la codificación de rutinas es la impresión de números. Suponga que queremos imprimir un número no negativo  $N$  en forma decimal y que no tenemos disponible una función de impresión de números. Sin embargo, imagine que sí que

<sup>1</sup> Se realiza una llamada a `s(-1)` y el programa termina fallando debido a que se alcanza un punto en el que habrá demasiadas llamadas recursivas pendientes. Las llamadas recursivas no irán progresando hacia un caso base.

podemos imprimir un dígito cada vez. Considere, por ejemplo, en esas circunstancias, cómo podríamos imprimir el número 1369. En primer lugar, tendríamos que imprimir 1, luego 3, después 6 y luego 9. El problema es que obtener el primer dígito es algo engorroso; dado un número  $n$ , necesitamos un bucle para determinar el primer dígito de  $n$ . Por el contrario, el último dígito siempre está disponible de forma inmediata mediante  $n \% 10$  (que es  $n$  para  $n$  menor que 10).

La recursión proporciona una solución elegante. Para imprimir 1369, imprimimos 136, seguido del último dígito, 9. Como ya hemos mencionado, imprimir el último dígito empleando el operador `%` es sencillo. Imprimir el número que resulta de eliminar el último dígito también es sencillo, porque es el mismo problema que imprimir  $n / 10$ . Por tanto, podemos realizar esa tarea mediante una llamada recursiva.

El código mostrado en la Figura 7.2 implementa esta rutina de impresión. Si  $n$  es menor de 10, la línea 6 no se ejecuta y solo se imprime el dígito  $n \% 10$ ; en caso contrario, se imprimen recursivamente todos los dígitos menos el último y después se imprime el último dígito.

Observe que tenemos un caso base (que  $n$  sea un entero de un solo dígito) y como el problema recursivo tiene un dígito menos, todas las llamadas recursivas irán progresando hacia el caso base. Por tanto, habremos satisfecho las dos primeras reglas fundamentales de la recursión.

Para que nuestra rutina de impresión sea útil, podemos ampliarla para que nos permita imprimir en cualquier base comprendida entre 2 y 16.<sup>2</sup> Esta modificación se muestra en la Figura 7.3.

```

1 // Imprimir n en base 10, de forma recursiva.
2 // Precondición: n >= 0.
3 public static void printDecimal( long n )
4 {
5     if( n >= 10 )
6         printDecimal( n / 10 );
7     System.out.print( (char) ('0' + ( n % 10 ) ) );
8 }
```

Figura 7.2 Una rutina recursiva para imprimir  $N$  en forma decimal.

```

1 private static final String DIGIT_TABLE = "0123456789abcdef";
2
3 // Imprimir n en cualquier base de forma recursiva.
4 // Precondición: n >= 0, base es válida.
5 public static void printInt( long n, int base )
6 {
7     if( n >= base )
8         printInt( n / base, base );
9     System.out.print( DIGIT_TABLE.charAt( (int) ( n % base ) ) );
10 }
```

Figura 7.3 Una rutina recursiva para imprimir  $N$  en cualquier base.

<sup>2</sup> El método `toString` de Java admite cualquier base, pero muchos lenguajes no tienen incorporada esta capacidad.

Hemos introducido una constante `String` para hacer que la impresión de `a` a `f` sea más fácil. Cada dígito se imprime ahora indexando la cadena de caracteres `DIGIT_TABLE`. La rutina `printInt` no es robusta. Si `base` es mayor que 16, el índice para acceder a `DIGIT_TABLE` podría salirse del rango. Si `base` es igual a 0, se producirá un error aritmético cuando se intente hacer una división por 0 en la línea 8.

**El fallo a la hora de progresar hacia el caso base implica que el programa no funciona.**

Una rutina de preparación comprueba la validez de la primera llamada y luego invoca la rutina recursiva.

El error más interesante ocurre cuando `base` es 1. Entonces la llamada recursiva de la línea 8 no consigue progresar hacia el caso base, porque los dos parámetros de la llamada recursiva son idénticos a los de la llamada original. Por tanto, el sistema realizará llamadas recursivas hasta que termine quedándose sin espacio de memoria (y termine de forma poco grácil).

Podemos hacer la rutina más robusta añadiendo una comprobación explícita para `base`. El problema con esta estrategia es que la comprobación se realizaría en cada una de las llamadas recursivas a `printInt`, no solo durante la primera llamada. Pero si `base` era válida en la primera llamada, volver a comprobarla es absurdo, porque no va a cambiar durante el curso de la recursión y por tanto seguirá siendo válida. Una forma de evitar esta inefficiencia consiste en programar una rutina de preparación. La *rutina de preparación* comprueba la validez de `base` y luego invoca a la rutina recursiva, como se muestra en la Figura 7.4. El uso de rutinas de preparación para los programas recursivos es una técnica bastante común.

### 7.3.2 Por qué funciona

En el Teorema 7.3 vamos a demostrar, de forma en cierto modo rigurosa, que el algoritmo `printDecimal` funciona. Nuestro objetivo es verificar que el algoritmo es correcto, por lo que la demostración se basa en la suposición de que no hemos cometido ningún error sintáctico.

Se puede demostrar la corrección de los algoritmos recursivos utilizando la inducción matemática.

La demostración del Teorema 7.3 ilustra un principio importante. A la hora de diseñar un algoritmo recursivo, siempre podemos asumir que las llamadas recursivas funcionan (si van progresando hacia el caso base) porque, cuando se desarrolla una demostración, esta suposición se emplea como hipótesis inductiva.

A primera vista, dicha suposición parece extraña. Sin embargo, recuerde que siempre asumimos que las llamadas a métodos funcionan, y por tanto la suposición de que la llamada recursiva funciona no es en realidad distinta. Al igual que con cualquier otro método, una rutina recursiva necesita combinar soluciones obtenidas a partir de llamadas a otros métodos, para obtener una solución final. No obstante, nada impide que los otros métodos puedan incluir instancias más simples del método original.

#### Teorema 7.3

El algoritmo `printDecimal` mostrado en la Figura 7.2 imprime correctamente `n` en base 10.

#### Demostración

Sea  $k$  el número de dígitos en `n`. La demostración es por inducción sobre  $k$ .

Base: si  $k = 1$ , entonces no se realiza ninguna llamada recursiva y la línea 7 imprime correctamente el único dígito de `n`.

Continúa

**Demostración  
(cont.)**

**Hipótesis inductiva:** suponga que printDecimal funciona correctamente para todos los enteros con un número de dígitos  $k \geq 1$ . Demostraremos que esta suposición implica la corrección para cualquier entero  $n$  de  $k + 1$  dígitos. Puesto que  $k \geq 1$ , la instrucción if de la línea 5 se satisface para un entero de  $k + 1$  dígitos. Por la hipótesis inductiva, la llamada recursiva de la línea 6 imprime los primeros  $k$  dígitos de  $n$ . Entonces, la línea 7 imprime el dígito final. Por tanto, si puede imprimirse cualquier entero de  $k$  dígitos, entonces también podrá imprimirse un entero de  $k + 1$  dígitos. Por inducción, concluimos que printDecimal funciona para todo valor de  $k$  y por tanto para todo valor  $n$ .

```

1 public final class PrintInt
2 {
3     private static final String DIGIT_TABLE = "0123456789abcdef";
4     private static final int MAX_BASE = DIGIT_TABLE.length();
5
6     // Imprimir n en cualquier base, de forma recursiva
7     // Precondición: n >= 0, 2 <= base <= MAX_BASE
8     private static void printIntRec( long n, int base )
9     {
10         if( n >= base )
11             printIntRec( n / base, base );
12         System.out.print( DIGIT_TABLE.charAt( (int) ( n % base ) ) );
13     }
14
15     // Rutina de preparación
16     public static void printInt( long n, int base )
17     {
18         if( base <= 1 || base > MAX_BASE )
19             System.err.println( "Cannot print in base " + base );
20         else
21         {
22             if( n < 0 )
23             {
24                 n = -n;
25                 System.out.print( "-" );
26             }
27             printIntRec( n, base );
28         }
29     }
30 }
```

Figura 7.4 Un programa robusto de impresión de números.

Esta observación nos lleva a la tercera regla fundamental de la recursión.

3. *"Es necesario creer"*: asuma siempre que la llamada recursiva funciona.

La tercera regla fundamental de la recursión: asuma siempre que la llamada recursiva funciona. Utilice esta regla para diseñar sus algoritmos.

La regla 3 nos dice que a la hora de diseñar un método recursivo, no tenemos por qué intentar trazar la posiblemente larga serie de llamadas recursivas. Como hemos visto anteriormente, esta tarea puede ser muy complicada y tiende a dificultar el diseño y la verificación. Cualquier buen uso de la recursión hace que esa tarea de trazado sea casi imposible de comprender. Intuitivamente, lo que estamos haciendo es dejar que la computadora se encargue de realizar toda la gestión que, en caso de que nos encargáramos nosotros de hacerla, daría como resultado un código mucho más largo.

Este principio es tan importante que debemos formularlo de nuevo: *asuma siempre que la llamada recursiva funciona*.

### 7.3.3 Cómo funciona

Recuerde que la implementación de la recursión requiere una serie de tareas adicionales de gestión por parte de la computadora. Dicho de otra forma, la implementación de cualquier método requiere una cierta gestión, y las llamadas recursivas no tienen nada de especial a este respecto (salvo porque pueden agotar las capacidades de gestión de la computadora al invocarse una rutina recursiva a sí misma demasiadas veces).

Para ver cómo puede una computadora gestionar la recursión o de modo más general, cualquier secuencia de llamadas a métodos, considere la manera en que una persona cualquiera podría gestionar un día muy atareado. Imagine que estamos editando un archivo en nuestra computadora y suena el teléfono. Al sonar el teléfono de nuestro domicilio tenemos que dejar de editar el archivo para atender la llamada. Puede que queramos escribir en un papel lo que estábamos haciendo en el archivo por si acaso la llamada telefónica dura mucho tiempo y luego no nos recordamos. Ahora imagine que, mientras que se encuentra hablando por teléfono con su esposa, le suena el teléfono móvil. Entonces deja a su esposa en espera, depositando el teléfono sobre la mesa. Puede anotar en un papel que ha dejado el teléfono descolgado sobre la mesa. Mientras está hablando por el teléfono móvil, alguien llama a la puerta. Entonces puede decirle a su interlocutor que espere mientras va a abrir la puerta para ver quién es. Así que deja el teléfono móvil sobre otro mueble, escribe en otro papel que ha dejado el móvil en un cierto lugar con una llamada en espera y abre la puerta. Llegados a este punto, habrá escrito tres notas para sí mismo, siendo la correspondiente al teléfono móvil la más reciente. Al abrir la puerta, se dispara la alarma antirrobo, porque se le ha olvidado desactivarla, así que tiene que decirle a la persona que está en la puerta que se espere. Entonces escribe otra nota para sí mismo, recordándole que hay alguien en la puerta, mientras desactiva la alarma antirrobo. Aunque está abrumado, ahora puede terminar de gestionar todas las tareas que había iniciado, en orden inverso: tratando primero con la persona de la puerta, terminado luego la conversación por el teléfono móvil, concluyendo luego la conversación con su esposa a través del teléfono fijo y finalizando después con la edición del archivo. Simplemente tiene que retroceder a través de la pila de anotaciones que ha ido haciendo para sí mismo. Observe la palabra que hemos utilizado: lo que habrá estado haciendo durante este tiempo es gestionar una "pila" de anotaciones.

Java, al igual que otros lenguajes como C++, implementa los métodos utilizando una pila interna de registros de activación. Cada *registro de activación* contiene información relevante

acerca del método, incluyendo, por ejemplo, los valores de sus parámetros y de sus variables locales. El contenido concreto de un registro de activación dependerá del sistema con el que estemos trabajando.

La pila de registros de activación se utiliza porque los métodos vuelven en orden inverso a su invocación. Recuerde que las pilas son muy útiles a la hora de invertir el orden de las cosas. En el escenario más popular, la cima de la pila almacena el registro de activación correspondiente al método actualmente activo. Cuando se invoca el método *G*, se inserta en la pila un registro de activación para *G*, lo que hace que *G* sea el método actualmente activo. Cuando un método vuelve, se extrae de la pila un registro y el registro de activación que pasa a estar en la parte superior de la pila contendrá los valores restaurados.

Por ejemplo, la Figura 7.5 muestra una pila de registros de activación que se forma durante el proceso de evaluación de *s(4)*. En este punto, tendremos las llamadas a *main*, *s(4)* y *s(3)* pendientes y estaremos procesando activamente *s(2)*.

El gasto de espacio en el que incurrimos será la memoria utilizada para almacenar un registro de activación por cada método actualmente activo. Así, en nuestro ejemplo anterior en el que *s(8883)* fallaba, el sistema dispone de aproximadamente 8.883 registros de activación. (Observe que *main* genera por sí mismo un registro de activación.) La tarea de insertar y extraer de la pila interna un registro de activación representa también el gasto adicional correspondiente a la ejecución de una llamada a método.

La estrecha relación existente entre la recursión y las pilas nos sugiere que los programas recursivos siempre pueden implementarse de forma iterativa con una pila explícita. Presumiblemente, nuestra pila almacenará elementos más pequeños que un registro de activación, por lo que podemos confiar razonablemente en que emplearemos menos espacio. El resultado de utilizar una pila explícita es un código ligeramente más largo, pero más rápido. Los compiladores modernos con optimización de compilación han reducido los costes asociados con la recursión a un grado tal, que en lo que respecta a la velocidad, raramente merece la pena eliminar la recursión de una aplicación que la esté utilizando correctamente.

La gestión de invocaciones de métodos en un lenguaje procedural y orientado a objetos se lleva a cabo utilizando una pila de registros de activación. La recursión es un subproducto natural de este sistema de trabajo.

Las secuencias de llamadas a métodos y de retornos de métodos son operaciones con una pila.

La recursión siempre puede ser eliminada utilizando una pila. Esto es necesario en ocasiones para ahorrar espacio.

### 7.3.4 Demasiada recursión puede ser peligrosa

En este texto vamos a proporcionar muchos ejemplos de la potencia de la recursión. Sin embargo, antes de examinar esos ejemplos, es preciso tener en cuenta que la recursión no siempre resulta adecuada. Por ejemplo, el uso de la recursión en la Figura 7.1 es desaconsejable, porque un bucle

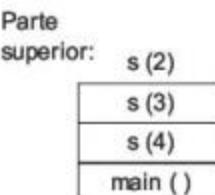


Figura 7.5 Una pila de registros de activación.

No utilice la recursión como sustituto de un bucle sencillo.

El número de Fibonacci  $i$ -ésimo es igual a la suma de los dos números de Fibonacci anteriores.

No lleve a cabo trabajo redundante de manera recursiva, el programa será terriblemente inefficiente.

permitiría resolver el problema igualmente. Una consideración práctica a este respecto es que la gestión requerida por las llamadas recursivas consume un cierto tiempo y limita los valores de  $n$  para los que el programa es correcto. Una regla heurística adecuada es que jamás debe utilizarse la recursión como sustituto de un bucle sencillo.

Otro problema bastante más grave es el que se pone de manifiesto cuando intentamos calcular de manera recursiva los números de Fibonacci. Los *números de Fibonacci*  $F_0, F_1, \dots, F_i$  se definen de la forma siguiente:  $F_0 = 0$  y  $F_1 = 1$ ; el  $i$ -ésimo número de Fibonacci es igual a la suma de los números de Fibonacci ( $i$ -ésimo - 1) y ( $i$ -ésimo - 2); por tanto  $F_i = F_{i-1} + F_{i-2}$ . A partir de esta definición, podemos determinar que la serie de los números de Fibonacci continua de la forma siguiente: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Los números de Fibonacci tienen una increíble serie de propiedades, que parece estar siempre incrementándose. De hecho, hay una revista, *The Fibonacci Quarterly*, creada con el único propósito de publicar teoremas relativos a los números de Fibonacci. Por ejemplo, la suma de los cuadrados de dos números de Fibonacci consecutivos es otro número de Fibonacci. La suma de los  $N$  primeros números de Fibonacci es siempre una unidad inferior a  $F_{N+2}$  (consulte el Ejercicio 7.11 para ver algunas otras igualdades interesantes).

Puesto que los números de Fibonacci se definen de manera recursiva, parece natural escribir una rutina recursiva con el fin de determinar  $F_N$ . Esta rutina recursiva, mostrada en la Figura 7.6, funciona, pero presenta un problema grave. En nuestra máquina relativamente rápida, se tarda aproximadamente un minuto en calcular  $F_{40}$ , lo cual es una cantidad absurda de tiempo, si tenemos en cuenta que el cálculo básico solo requiere 39 sumas.

El problema subyacente es que esta rutina recursiva realiza cálculos redundantes. Para calcular  $\text{fib}(n)$ , calculamos recursivamente  $\text{fib}(n-1)$ . Cuando termina la llamada recursiva, calculamos  $\text{fib}(n-2)$  utilizando otra llamada recursiva. Pero ya habíamos calculado  $\text{fib}(n-2)$  durante el proceso de cálculo de  $\text{fib}(n-1)$ , así que la llamada a  $\text{fib}(n-2)$  es un cálculo redundante, y por tanto un desperdicio de recursos. De hecho, hemos hecho dos llamadas a  $\text{fib}(n-2)$  en lugar de solo una.

Normalmente, hacer dos llamadas a métodos en lugar de una representaría duplicar el tiempo de ejecución del programa. Sin embargo, en este caso, las cosas son aún peores: cada llamada a

```

1 // Calcular el n-ésimo número de Fibonacci.
2 // 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 Una rutina recursiva para calcular los números de Fibonacci: no es una buena idea.

`fib(n-1)` y cada llamada a `fib(n-2)` hacen una llamada a `fib(n-3)`; por tanto, habrá en la práctica tres llamadas a `fib(n-3)`. De hecho, el tema va empeorando según avanzamos, cada llamada a `fib(n-2)` o `fib(n-3)` provoca una llamada a `fib(n-4)`, así que habrá cinco llamadas a `fib(n-4)`. Es decir, obtenemos un efecto de composición: cada llamada recursiva va haciendo cada vez más trabajo redundante.

Sea  $C(N)$  el número de llamadas a `fib` realizadas durante la evaluación de `fib(n)`. Claramente,  $C(0) = C(1) = 1$  llamada. Para  $N \geq 2$ , llamamos a `fib(n)`, más todas las llamadas necesarias para evaluar `fib(n-1)` y `fib(n-2)` de forma recursiva e independiente. Por tanto,  $C(N) = C(N-1) + C(N-2) + 1$ . Por inducción, podemos verificar fácilmente que para  $N \geq 3$  la solución a esta recurrencia es  $C(N) = F_{N+2} + F_{N-1} - 1$ . Por tanto, el número de llamadas recursivas es mayor que el número de Fibonacci que estamos tratando de calcular, y crece exponencialmente. Para  $N = 40$ ,  $F_{40} = 102.334.155$ , y el número total de llamadas recursivas supera los 300.000.000. No resulta extraño, por tanto, que el programa se eternice. El aumento explosivo de llamadas recursivas se ilustra en la Figura 7.7.

Este ejemplo ilustra la cuarta y última regla básica de la recursión.

4. *Regla del interés compuesto:* nunca duplique el trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

La rutina recursiva `fib` es exponencial.

La cuarta regla fundamental de la recursión: nunca duplique el trabajo que hay que realizar, resolviendo la misma instancia de un problema mediante llamadas recursivas separadas.

### 7.3.5 Previsualización de árboles

El *árbol* es una estructura fundamental en las Ciencias de la computación. Casi todos los sistemas operativos almacenan los archivos en árboles o estructuras similares al árbol. Los árboles se emplean también en el diseño de compiladores, el procesamiento de textos y los algoritmos de búsqueda. Hablaremos en detalle de los árboles en los Capítulos 18 y 19. También haremos uso de los árboles en las Secciones 11.2.4 (árboles de expresión) y 12.1 (códigos de Huffman).

Hay una definición de árbol que es recursiva: o bien un árbol está vacío o bien consta de una raíz y de cero o más subárboles no vacíos  $T_1, T_2, \dots, T_k$  cada una de cuyas raíces están conectadas por una arista a la raíz, como se ilustra en la Figura 7.8. En algunos casos (y especialmente en los *árboles binarios* de los que hablaremos en el Capítulo 18), podemos permitir que algunos de los subárboles estén vacíos.

Utilizando una definición no recursiva, un *árbol* está compuesto por un conjunto de nodos y un conjunto de aristas dirigidas que conectan parejas de nodos. A lo largo de este texto solo vamos a tomar en consideración los árboles que tienen raíz. Un árbol que tiene raíz presenta las siguientes propiedades:

Un árbol consta de un conjunto de nodos y de un conjunto de aristas dirigidas que los unen.

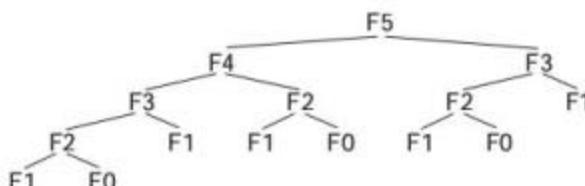


Figura 7.7 Una traza del cálculo recursivo de los números de Fibonacci.

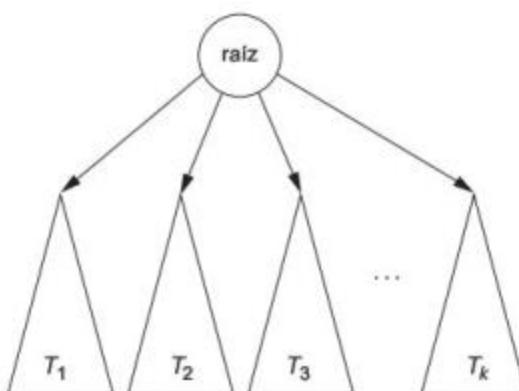


Figura 7.8 Una visión recursiva de un árbol.

- Uno de los nodos está designado como raíz.
- Todo nodo  $c$ , salvo la raíz, tiene una arista dirigida que entra en él y proviene de un nodo  $p$ . El nodo  $p$  es el *padre* de  $c$  y  $c$  es uno de los *hijos* de  $p$ .
- Existe un camino único que recorre el árbol desde la raíz a cada nodo. El número de aristas que hay que recorrer es la *longitud de ese camino*.

Los padres y los hijos se definen de manera natural. Una arista dirigida conecta al *padre* con el *hijo*.

Una hoja no tiene ningún hijo.

Los padres y los hijos se definen de forma natural. Una arista dirigida conecta al *padre* con el *hijo*.

La Figura 7.9 muestra un árbol. El nodo raíz es  $A$ : los hijos de  $A$  son  $B$ ,  $C$ ,  $D$  y  $E$ . Puesto que  $A$  es el nodo raíz, no tiene padre; todos los restantes nodos sí tienen padre. Por ejemplo, el padre de  $B$  es  $A$ . Un nodo que no tiene ningún hijo se denomina *hoja*. Las hojas de este árbol son  $C$ ,  $F$ ,  $G$ ,  $H$ ,  $I$  y  $K$ . La longitud del camino que va desde  $A$  hasta  $K$  es 3 (aristas); la longitud del camino que va desde  $A$  a  $A$  es 0 (aristas).

### 7.3.6 Ejemplos adicionales

Quizá la mejor forma de comprender la recursión es analizando una serie de ejemplos. En esta sección, vamos a ver cuatro ejemplos adicionales de recursión. Los dos primeros se pueden implementar fácilmente de forma no recursiva, pero los dos últimos muestran parte de la potencia que proporciona la técnica de recursión.

#### Factoriales

Recuerde que  $M$  es el producto de los  $N$  primeros enteros. Por tanto, podemos expresar  $M$  como  $N$  veces  $(N - 1)!$  Combinando este hecho con el caso base,  $1! = 1$ , esta información nos proporciona inmediatamente todo lo que necesitamos para una implementación recursiva. Dicha implementación se muestra en la Figura 7.10.

#### Búsqueda binaria

En la Sección 5.6.2 hemos descrito la búsqueda binaria. Recuerde que en una búsqueda binaria, realizamos una búsqueda dentro de una matriz ordenada  $A$  examinando el elemento medio. Si

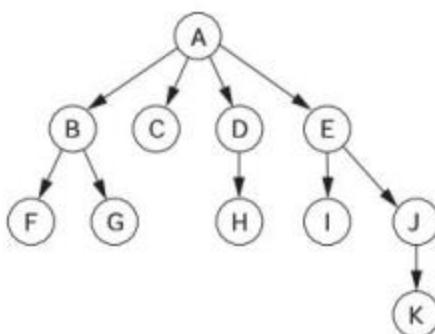


Figura 7.9 Un árbol.

encontramos una correspondencia, habremos terminado. En caso contrario, si el elemento que estamos buscando es más pequeño que el elemento medio, buscamos en la submatriz situada a la izquierda del elemento medio; en caso contrario, buscamos en la submatriz situada a la derecha del elemento medio. Este procedimiento asume que la submatriz no está vacía; si lo está, entonces el elemento no habrá podido ser encontrado.

Esta descripción se refleja directamente en el método recursivo mostrado en la Figura 7.11. El código ilustra una técnica temática en la que la rutina pública de preparación hace una llamada a una rutina recursiva y devuelve el valor de retorno de esta. Aquí, la rutina de preparación establece los puntos mínimo y máximo de la submatriz que son 0 y `a.length - 1`.

En el método recursivo, el caso base de las líneas 18 y 19 se encarga de procesar la situación en la que tengamos una submatriz vacía. En caso contrario, seguimos la descripción dada anteriormente efectuando una llamada recursiva con la submatriz apropiada (líneas 24 a 26) si no se ha detectado una correspondencia. Cuando se detecta una correspondencia, se devuelve el índice respectivo en la línea 28.

Observe que el tiempo de ejecución en términos de O mayúscula, no cambia aquí con respecto a la implementación no recursiva, porque estamos realizando el mismo trabajo. En la práctica, el tiempo de ejecución sería ligeramente mayor, debido a los costes ocultos asociados con la recursión.

```

1 // Evaluar 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 7.10 Implementación recursiva del método `factorial`.

```
1  /**
2   * Realiza la búsqueda binaria estándar utilizando dos comparaciones
3   * por nivel. Esta rutina de preparación llama al método recursivo.
4   * @return el índice del elemento encontrado o NOT_FOUND si no se encuentra.
5   */
6  public static <AnyType extends Comparable<? super AnyType>>
7      int binarySearch( AnyType [ ] a, AnyType x )
8  {
9      return binarySearch( a, x, 0, a.length -1 );
10 }
11
12 /**
13  * Rutina recursiva oculta.
14  */
15 private static <AnyType extends Comparable<? super AnyType>>
16     int binarySearch( AnyType [ ] a, AnyType x, int low, int high )
17 {
18     if( low > high )
19         return NOT_FOUND;
20
21     int mid = ( low + high ) / 2;
22
23     if( a[ mid ].compareTo( x ) < 0 )
24         return binarySearch( a, x, mid + 1, high );
25     else if( a[ mid ].compareTo( x ) > 0 )
26         return binarySearch( a, x, low, mid - 1 );
27     else
28         return mid;
29 }
```

---

Figura 7.11 Una rutina de búsqueda binaria utilizando recursión.

## Dibujo de una regla

La Figura 7.12 muestra el resultado de ejecutar un programa Java que dibuja las marcas de una regla. Aquí, consideramos el problema de marcar una pulgada. En el medio se encuentra la marca de mayor tamaño. En la Figura 7.12, a la izquierda del punto central hay una versión en miniatura de la regla y a la derecha del mismo hay una segunda versión también en miniatura de la regla. Este resultado sugiere utilizar un algoritmo recursivo que dibuje primero la línea central y luego las mitades izquierda y derecha.

No tiene que comprender los detalles de cómo se dibujan líneas y formas en Java para entender este programa. Le basta con saber que un objeto `Graphics` es algo sobre lo que se dibuja. El método `drawRuler` de la Figura 7.13 es nuestra rutina recursiva. Utiliza el método `drawLine`, que forma parte de la clase `Graphics`. El método `drawLine` dibuja una línea desde un punto de coordenadas



Figura 7.12 Una regla dibujada de forma recursiva.

$(x, y)$  a otro punto de coordenadas  $(x, y)$ , donde las coordenadas representan distancias con respecto a la esquina superior izquierda.

Nuestra rutina dibuja marcas con un número `level` de diferentes alturas; cada llamada recursiva es un nivel más profundo que la anterior (en la Figura 7.12 hay ocho niveles). Primero trata con el caso base en las líneas 4 y 5. Después se dibuja la marca central en la línea 9. Finalmente, las dos miniaturas se dibujan recursivamente en las líneas 11 y 12. En el código en línea, hemos incluido código adicional para hacer más lento el proceso de dibujo. De esta forma, se puede ver el orden en el que el algoritmo recursivo va dibujando las líneas.

### Estrella fractal

En la Figura 7.14(a) se muestra un patrón aparentemente complejo denominado *estrella fractal*, el cual podemos dibujar fácilmente utilizando recursión. Todo el lienzo está inicialmente pintado de color gris (no mostrado); el patrón se forma dibujando cuadrados de color blanco sobre el fondo gris. El último cuadrado dibujado se encuentra en el centro. La Figura 7.14(b) muestra el dibujo resultante justo después de dibujar el último cuadrado. Por tanto, antes de que se dibujara el último cuadrado, se habían dibujado cuatro versiones en miniatura, una en cada uno de los cuatro cuadrantes disponibles. Este patrón nos proporciona la información necesaria para deducir el algoritmo recursivo que hay que utilizar.

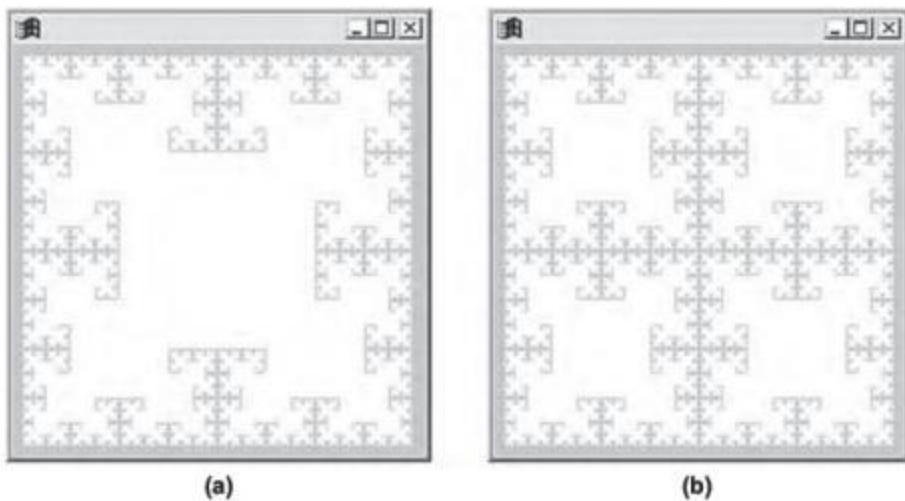
Como en el ejemplo anterior, el método `drawFractal` utiliza una rutina de la librería Java. En este caso, `fillRect` dibuja un rectángulo; es preciso especificar su esquina superior izquierda y sus dimensiones. El código se muestra en la Figura 7.15. Los parámetros de `drawFractal` incluyen el centro del fractal y la dimensión global. A partir de estos datos, podemos calcular en la línea 5,

```

1 // Código Java para dibujar la Figura 7.12.
2 void drawRuler( Graphics g, int left, int right, int level )
3 {
4     if( level < 1 )
5         return;
6
7     int mid = ( left + right ) / 2;
8
9     g.drawLine( mid, 80, mid, 80 - level * 5 );
10
11    drawRuler( g, left, mid - 1, level - 1 );
12    drawRuler( g, mid + 1, right, level - 1 );
13 }

```

Figura 7.13 Un método recursivo para dibujar una regla.



**Figura 7.14** (a) Una estrella fractal dibujada mediante el código de la Figura 7.15. (b) La misma estrella inmediatamente después de añadir el último cuadrado.

```

1 // Dibujar la imagen de la Figura 7.14.
2 void drawFractal( Graphics g, int xCenter,
3                     int yCenter, int boundingDim )
4 {
5     int side = boundingDim / 2;
6
7     if( side < 1 )
8         return;
9
10    // Calcular las esquinas.
11    int left = xCenter - side / 2;
12    int top = yCenter - side / 2;
13    int right = xCenter + side / 2;
14    int bottom = yCenter + side / 2;
15
16    // Dibujar recursivamente cuatro cuadrantes.
17    drawFractal( g, left, top, boundingDim / 2 );
18    drawFractal( g, left, bottom, boundingDim / 2 );
19    drawFractal( g, right, top, boundingDim / 2 );
20    drawFractal( g, right, bottom, boundingDim / 2 );
21
22    // Dibujar el cuadrado central, donde se solapan los cuadrantes.
23    g.fillRect( left, top, right - left, bottom - top );
24 }
```

**Figura 7.15** Código para dibujar la estrella fractal mostrada en la Figura 7.14.

el tamaño del gran cuadrado central. Después de gestionar el caso base en las líneas 7 y 8, calculamos las fronteras del rectángulo central. A continuación, podemos dibujar los cuatro fractales en miniatura en las líneas 17 a 20. Por último, dibujamos el cuadrado central en la línea 23. Observe que este cuadrado debe ser dibujado después de las llamadas recursivas. En caso contrario, obtendríamos una imagen distinta (en el Ejercicio 7.33, le pediremos que describa la diferencia).

## 7.4 Aplicaciones numéricas

En esta sección, vamos a echar vistazo a tres problemas extraídos principalmente de la teoría de números. La teoría de números se solía considerar una rama interesante, pero inútil de las matemáticas. Sin embargo, en los últimos 30 años, ha surgido una importante aplicación de la teoría de números: la seguridad de los datos. Comenzaremos nuestra exposición con unas pequeñas notas sobre fundamentos matemáticos y luego mostraremos algoritmos recursivos para resolver tres problemas. Podemos combinar estas rutinas con un cuarto algoritmo más complejo (descrito en el Capítulo 9), para implementar un algoritmo que puede emplearse para codificar y decodificar mensajes. Hasta la fecha, nadie ha sido capaz de demostrar que el esquema de cifrado aquí descrito no sea seguro.

He aquí los cuatro problemas que vamos a examinar.

1. *Exponenciación modular*: calcular  $X^N \pmod{P}$ .
2. *Máximo común divisor*: calcular  $\gcd(A, B)$ .
3. *Inversa multiplicativa*: calcular  $X$  a partir de la relación de equivalencia  $AX \equiv 1 \pmod{P}$ .
4. *Prueba de primalidad*: determinar si  $N$  es primo (dejaremos este problema para el Capítulo 9).

Los enteros con los que vamos a tratar son todos ellos de gran tamaño, formados por al menos 100 dígitos cada uno. Por tanto, debemos disponer de una forma de representar enteros de gran tamaño junto con un conjunto completo de algoritmos para las operaciones básicas de suma, resta, multiplicación, división, etc. Java proporciona la clase `BigInteger` con este propósito. Implementar esa clase de manera eficiente no es una tarea trivial y de hecho existe una amplia literatura técnica sobre dicha materia.

Utilizaremos números `long` para simplificar la presentación de nuestro código. Los algoritmos descritos aquí funcionan con objetos de gran tamaño, ejecutándose en una cantidad de tiempo razonable.

### 7.4.1 Aritmética modular

Los problemas de esta sección, así como la implementación de la estructura de datos de tabla hash (Capítulo 20), requiere el uso del operador `%` de Java. El operador `%`, al que designaremos `operator%`, calcula el resto de la división de dos tipos enteros. Por ejemplo, `13%10` da como resultado 3, igual que `3%10` y `23%10`. Cuando calculamos el resto de una división por 10, el rango de posibles resultados va de 0 a 9.<sup>3</sup> Este rango hace que `operator%` sea útil para generar enteros de pequeño tamaño.

<sup>3</sup> Si  $n$  es negativo,  $n%10$  va de 0 a -9.

Si dos números  $A$  y  $B$  nos dan el mismo resto al dividirlos por  $N$ , decimos que son congruentes módulo  $N$ , lo que se expresa como  $A \equiv B \pmod{N}$ . En este caso, debe ser obligatoriamente cierto que  $N$  sea divisor de  $A - B$ . Además, la inversa también es cierta: si  $N$  es divisor de  $A - B$ , entonces  $A \equiv B \pmod{N}$ . Puesto que solo hay  $N$  posibles restos ( $0, 1, \dots, N - 1$ ) decimos que el conjunto de los enteros se divide en clases de congruencia de módulo  $N$ . En otras palabras, cada entero puede ser asignado a una de las  $N$  clases, y los que pertenecen a una misma clase son congruentes entre sí, módulo  $N$ .

Utilizaremos tres importantes teoremas en nuestros algoritmos (dejamos la demostración de estos teoremas para el Ejercicio 7.8).

1. Si  $A \equiv B \pmod{N}$ , entonces para todo  $C$ ,  $A + C \equiv B + C \pmod{N}$ .
2. Si  $A \equiv B \pmod{N}$ , entonces para todo  $D$ ,  $AD \equiv BD \pmod{N}$ .
3. Si  $A \equiv B \pmod{N}$ , entonces para cualquier entero positivo  $P$ ,  $A^P \equiv B^P \pmod{N}$ .

Estos teoremas permiten realizar ciertos cálculos con menos esfuerzo. Por ejemplo, suponga que queremos saber cuál es el último dígito de  $3333^{5555}$ . Puesto que este número tiene más de 15.000 dígitos, resultaría muy costoso calcular la respuesta directamente. Sin embargo, lo que queremos es determinar  $3333^{5555} \pmod{10}$ . Como  $3333 \equiv 3 \pmod{10}$ , solo necesitamos calcular  $3^{5555} \pmod{10}$ . Puesto que  $3^4 = 81$ , sabemos que  $3^4 \equiv 1 \pmod{10}$ , y elevando ambos lados a la potencia 1388 vemos que  $3^{5552} \equiv 1 \pmod{10}$ . Si multiplicamos ahora ambos lados por  $3^3 = 27$ , obtenemos  $3^{5555} \equiv 27 \equiv 7 \pmod{10}$ , completando así el cálculo.

#### 7.4.2 Exponenciación modular

En esta sección vamos a demostrar cómo calcular  $X^N \pmod{P}$  de manera eficiente. Podemos hacerlo inicializando `result` a 1 y luego multiplicando repetidamente `result` por  $X$ , aplicando el operador `%` después de cada multiplicación. Utilizar `operator%` de esta forma para sustituir el resultado de la última multiplicación, hace que cada multiplicación sea más fácil, porque se mantiene `result` con un valor pequeño.

Después de  $N$  multiplicaciones, `result` será la respuesta que estamos buscando. Sin embargo, realizar  $N$  multiplicaciones será poco práctico si  $N$  es un `BigInteger` de 100 dígitos. De hecho, si  $N$  es 1.000.000.000, resulta poco práctico en todas las máquinas, salvo 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  $N$  es impar, entonces

$$X^N = X \cdot X^{N-1} = X \cdot (X \cdot X)^{\lfloor N/2 \rfloor}$$

(Recuerde que  $|X|$  es el mayor entero menor o igual que  $X$ ). Como antes, para realizar la exponenciación modular aplicamos un `%` después de cada multiplicación.

El algoritmo recursivo mostrado en la Figura 7.16 representa una implementación directa de esta estrategia. Las líneas 8 y 9 se encargan de tratar el caso base:  $X^0$  es 1, por definición.<sup>4</sup> En la línea

<sup>4</sup> Definimos  $0^0 = 1$  para los propósitos de este algoritmo. También suponemos que  $N$  es no negativo y que  $P$  es positivo.

```

1  /**
2   * Devuelve  $x^n \pmod p$ 
3   * Asume que  $x, n \geq 0, p > 0, x < p, 0^0 = 1$ 
4   * Puede producirse desbordamiento si  $p > 31$  bits.
5   */
6  public static long power( long x, long n, long p )
7  {
8      if( n == 0 )
9          return 1;
10     long tmp = power( ( x * x ) % p, n / 2, p );
11     if( n % 2 != 0 )
12         tmp = ( tmp * x ) % p;
13     return tmp;
14 }
15
16
17 }
```

Figura 7.16 Una rutina de exponentiación modular.

11, hacemos una llamada recursiva basada en la identidad establecida en el párrafo anterior. Si  $N$  es par, esta llamada calcula la respuesta deseada; si  $N$  es impar, necesitamos hacer una multiplicación adicional por  $X$  (y utilizar `operator%`).

Este algoritmo es más rápido que el algoritmo simple propuesto anteriormente. Si  $M(N)$  es el número de multiplicaciones utilizadas por `power`, tenemos que  $M(N) \leq M(\lfloor N/2 \rfloor) + 2$ . La razón es que si  $N$  es par, realizamos una multiplicación más las que se efectúan de modo recursivo, mientras que si  $N$  es impar, realizamos dos multiplicaciones más las que se llevan a cabo recursivamente. Puesto que  $M(0) = 0$ , podemos demostrar que  $M(N) < 2 \log N$ . El factor logarítmico se puede obtener sin realizar cálculos directos, aplicando el principio de división por la mitad (véase la Sección 5.5), que nos da el número de invocaciones recursivas de `power`. Además, un valor medio de  $M(N)$  será  $(3/2)\log N$ , ya que en cada paso recursivo,  $N$  tiene la misma probabilidad de ser par o impar. Si  $N$  es un número de 100 dígitos, en el caso peor solo tendremos que hacer unas 665 multiplicaciones (y como promedio solo necesitaremos unas 500).

La exponentiación puede llevarse a cabo mediante un número logarítmico de exponentiaciones.

### 7.4.3 Máximo común divisor e inversas multiplicativas

Dados dos enteros no negativos  $A$  y  $B$ , su máximo común divisor,  $\gcd(A, B)$ , es el mayor entero  $D$  que es tanto divisor de  $A$  como de  $B$ . Por ejemplo,  $\gcd(70, 25)$  es 5. En otras palabras, el *máximo común divisor* (*mcd*, o en inglés *greatest common divisor*, *gcd*) es el mayor entero que es divisor de dos enteros dados.

El *máximo común divisor* de dos enteros es el mayor entero que es divisor de ambos.

Podemos verificar fácilmente que  $\gcd(A, B) = \gcd(A - B, B)$ . Si  $D$  es divisor de  $A$  y  $B$ , también debe ser divisor de  $A - B$ , y si  $D$  es divisor tanto de  $A - B$  como de  $B$ , entonces también debe ser divisor de  $A$ .

Esta observación nos conduce a un algoritmo sencillo, en el que vamos restando sucesivamente  $B$  de  $A$ , transformando el problema en otro más pequeño. Llegará un momento en que  $A$  tendrá un valor menor que  $B$ , en cuyo caso podemos intercambiar los papeles de  $A$  y  $B$  y continuar con el algoritmo. En un cierto momento,  $B$  pasará a tener el valor 0. Entonces, sabremos que  $\gcd(A, 0) = A$  y, ya que cada transformación preserva el máximo común divisor de los valores originales  $A$  y  $B$ , habremos conseguido obtener la respuesta que buscábamos. Este algoritmo se denomina *algoritmo de Euclides* y fue descrito por primera vez hace algo más de 2.000 años. Aunque es un algoritmo correcto, es inútil para números muy grandes, porque es probable que en ese caso haga falta un enorme número de restas.

Una modificación del algoritmo, más eficiente desde el punto de vista computacional, es la que resulta de observar que restar repetidamente  $B$  de  $A$  hasta que  $A$  sea más pequeño que  $B$ , es equivalente a convertir  $A$  en precisamente  $A \bmod B$ . Por tanto,  $\gcd(A, B) = \gcd(B, A \bmod B)$ . Esta definición recursiva, junto con el caso base en que  $B = 0$ , se utiliza para obtener directamente la rutina mostrada en la Figura 7.17. Para visualizar cómo funciona el algoritmo, observe que en el ejemplo anterior hemos empleado la siguiente secuencia de llamadas recursivas para deducir que el máximo común divisor de 70 y 25 es 5:  $\gcd(70, 25) \Rightarrow \gcd(25, 20) \Rightarrow \gcd(20, 5) \Rightarrow \gcd(5, 0) \Rightarrow 5$ .

El número de llamadas recursivas utilizadas es proporcional al logaritmo de  $A$ , que tiene el mismo orden de magnitud que las otras rutinas que hemos presentado en esta sección. La razón es que con dos llamadas recursivas, el problema se reduce al menos a la mitad. La demostración de este hecho lo dejamos como para tarea para el lector en el Ejercicio 7.13.

El máximo común divisor y la inversa multiplicativa pueden también calcularse en un tiempo logarítmico utilizando una variante del algoritmo de Euclides.

El algoritmo gcd se utiliza implicitamente para resolver un problema matemático similar. La solución  $1 \leq X < N$  de la ecuación  $AX \equiv 1 \pmod{N}$  se denomina inversa multiplicativa de  $A$  mod  $N$ . Suponga también que  $1 \leq A < N$ . Por ejemplo, la inversa de 3, mod 13 es 9; es decir,  $3 \cdot 9 \bmod 13$  da como resultado 1.

La capacidad de calcular inversas multiplicativas es importante porque ecuaciones como  $3i \equiv 7 \pmod{13}$  pueden resolverse fácilmente si conocemos la inversa multiplicativa. Estas ecuaciones surgen en muchas aplicaciones, incluyendo el algoritmo de cifrado explicado al final de esta sección. En este ejemplo, si multiplicamos por la inversa de 3 (es decir, 9), obtenemos  $i \equiv 63 \pmod{13}$ , por lo que  $i = 11$  es una solución. Si

$$AX \equiv 1 \pmod{N}, \text{ entonces } AX + NY = 1 \pmod{N}$$

es cierto para todo  $Y$ . Para algún  $Y$ , el lado izquierdo debe ser exactamente igual a 1. Por tanto, la ecuación

$$AX + NY = 1$$

es resoluble si y solo si  $A$  tiene una inversa multiplicativa.

Dadas  $A$  y  $B$ , vamos a ver cómo calcular  $X$  e  $Y$  tales que satisfagan

$$AX + BY = 1$$

Suponemos que  $0 \leq |B| < |A|$  y luego ampliaremos el algoritmo gcd para calcular  $X$  e  $Y$ .

En primer lugar, consideraremos el caso base,  $B \equiv 0$ . En este caso, tenemos que resolver  $AX = 1$ , lo que implica que tanto  $A$  como  $X$  son 1. De hecho, si  $A$  no es 1, no existe una inversa multiplicativa. Por tanto,  $A$  tiene una inversa multiplicativa módulo  $N$  solo si  $\gcd(A, N) = 1$ .

```

1  /**
2   * Devolver el máximo común divisor.
3   */
4  public static long gcd( long a, long b )
5  {
6      if( b == 0 )
7          return a;
8      else
9          return gcd( b, a % b );
10 }

```

Figura 7.17 Cálculo del máximo común divisor.

Por el contrario,  $B$  es distinto de cero. Recuerde que  $\text{gcd}(A, B) \equiv \text{gcd}(B, A \bmod B)$ . Por tanto, hacemos  $A = BQ + R$ . Aquí  $Q$  es el cociente y  $R$  es el resto, y por tanto la llamada recursiva será  $\text{gcd}(B, R)$ . Suponga que podemos resolver recursivamente

$$BX_1 + RY_1 = 1$$

Dado que  $R = A - BQ$ , tenemos

$$BX_1 + (A - BQ)Y_1 = 1$$

lo que significa que

$$AY_1 + B(X_1 - QY_1) = 1$$

Por tanto,  $X = Y_1$  e  $Y = X_1 - \lfloor A/B \rfloor Y_1$  será una solución de  $AX + BY = 1$ . Hemos codificado esta observación directamente como `fullGcd` en la Figura 7.18. El método `inverse` simplemente llama a `fullGcd`, donde  $X$  e  $Y$  son variables estáticas de clase. El único detalle que queda es que el valor especificado para  $X$  puede ser negativo. Si es así, la línea 35 de `inverse` lo trasformará en positivo. Dejamos al lector que demuestre este hecho en el Ejercicio 7.16. La demostración se puede hacer por inducción.

#### 7.4.4 El criptosistema RSA

Durante siglos, se creía que la teoría de números era una rama de las matemáticas carente por completo de aplicaciones prácticas. Recientemente, sin embargo, ha demostrado ser un campo de gran importancia, debido a su aplicación a la criptografía.

El problema que vamos a considerar tiene dos partes. Suponga que Alice quiere enviar un mensaje a Bob, pero está preocupada porque la transmisión pueda ser interceptada. Por ejemplo, si la transmisión se hace a través de una línea telefónica y alguien pincha la línea, alguna otra persona podría leer el mensaje. Vamos a asumir que, aunque la línea pueda estar siendo interceptada, no existe ningún acto destructivo (es decir, la señal no sufre ningún daño); es decir, Bob recibe todo lo que Alice envíe.

Una solución a este problema consiste en utilizar el *cifrado*, un esquema de comunicación para transmitir mensajes de modo que no puedan ser leídos por terceros. El cifrado está compuesto de

La teoría de números se emplea en criptografía porque la factorización parece ser un proceso mucho más complejo que la multiplicación.

```

1 // Variables internas para fullGcd
2 private static long x;
3 private static long y;
4
5 /**
6 * Aplica a la inversa el algoritmo de Euclides para determinar
7 * x e y tales que si gcd(a,b) = 1,
8 * ax + by = 1.
9 */
10 private static void fullGcd( long a, long b )
11 {
12     long x1, y1;
13
14     if( b == 0 )
15     {
16         x = 1;
17         y = 0;
18     }
19     else
20     {
21         fullGcd( b, a % b );
22         x1 = x; y1 = y;
23         x = y1;
24         y = x1 - ( a / b ) * y1;
25     }
26 }
27
28 /**
29 * Resolver ax == 1 (mod n), asumiendo que gcd( a, n ) = 1.
30 * @return x.
31 */
32 public static long inverse( long a, long n )
33 {
34     fullGcd( a, n );
35     return x > 0 ? x : x + n;
36 }

```

**Figura 7.18** Una rutina para determinar la inversa multiplicativa.

dos partes. En primer lugar, Alice *cifra* el mensaje y envía el resultado, que ya no estará compuesto por texto legible. Cuando Bob recibe la transmisión procedente de Alice, lo *descifra* para obtener el original. La seguridad del algoritmo se basa en el hecho de que nadie, excepto Bob, debería poder realizar el descifrado, incluyendo a la propia Alice (si es que ella no conservara el mensaje original).

Por tanto, Bob debe proporcionar a Alice un método de cifrado que solo él sepa cómo invertir. Este problema plantea un enorme desafío. Muchos algoritmos propuestos pueden ser reventados mediante sutiles técnicas de descifrado de códigos. Uno de los métodos, descrito aquí, es el *criptosistema RSA* (denominado así por las iniciales de sus autores), que constituye una elegante implementación de una estrategia de cifrado.

Aquí solo vamos a proporcionar una panorámica de alto nivel del cifrado, mostrando cómo interactúan de una manera práctica los métodos escritos en esta sección. Las referencias proporcionadas al final del capítulo contienen indicaciones para localizar descripciones más detalladas, así como demostraciones de las principales propiedades del algoritmo.

Sin embargo, en primer lugar, observe que un mensaje está compuesto por una secuencia de caracteres y que cada carácter está formado simplemente por una secuencia de bits. Por tanto, todo mensaje es una secuencia de bits. Si descomponemos el mensaje en bloques de  $B$  bits, podemos interpretar el mensaje como una serie de números de muy gran tamaño. Por tanto, el problema básico se reduce a cifrar un número muy grande y luego a descifrar el resultado.

### Cálculo de las constantes RSA

El algoritmo RSA comienza obligando al receptor a determinar una serie de constantes. En primer lugar, se eligen de modo aleatorio dos números primos de gran tamaño  $p$  y  $q$ . Típicamente, estos números deberán tener al menos unos 100 dígitos cada uno. Por lo que a este ejemplo respecta, suponga que  $p = 127$  y  $q = 211$ . Observe que Bob es el receptor y que es, por tanto, quien está llevando a cabo estos cálculos. Observe también que tenemos a nuestra disposición una enorme cantidad de números primos. Bob puede continuar comprobando distintos números aleatorios hasta que dos de ellos pasen las pruebas de primalidad (de las que hablaremos en el Capítulo 9).

A continuación, Bob calcula  $N = pq$  y  $N' = (p - 1)(q - 1)$ , que para este ejemplo nos da  $N = 26.797$  y  $N' = 26.460$ . Bob continúa después seleccionando cualquier valor tal que  $\gcd(e, N') = 1$ . En términos matemáticos, lo que selecciona es un valor  $e$  que sea relativamente primo a  $N'$ . Bob puede continuar probando diferentes valores de  $e$  utilizando la rutina mostrada en la Figura 7.17 hasta encontrar uno que satisfaga dicha propiedad. Cualquier valor primo  $e$  nos vale, así que encontrar  $e$  es tan fácil como encontrar un número primo. En este caso,  $e = 13.379$  es una de las muchas posibilidades. A continuación, se calcula  $d$ , la inversa multiplicativa de  $e$ , mod  $N'$  utilizando la rutina mostrada en la Figura 7.18. En este ejemplo,  $d = 11.099$ .

Una vez que Bob ha calculado todas estas constantes, lo que hace es lo siguiente: en primer lugar, destruye  $p$ ,  $q$  y  $N'$ . La seguridad del sistema se vería comprometida si cualquiera de estos valores fuera descubierto. A continuación, Bob entrega los valores de  $e$  y  $N$  a todos aquellos que deseen enviarle mensajes cifrados, pero mantiene el valor de  $d$  en secreto.

### Algoritmos de cifrado y descifrado

Para cifrar un entero  $M$ , el emisor calcula  $M^e \pmod{N}$  y envía el resultado. En nuestro caso, si  $M = 10.237$ , el valor enviado será 8.422. Cuando recibe un entero cifrado  $R$ , todo lo que Bob tiene que hacer es calcular  $R^d \pmod{N}$ . Para  $R = 8.422$ , obtiene como resultado el valor original  $M = 10.237$  (lo que evidentemente no es por casualidad). Por consiguiente, tanto el cifrado como el descifrado pueden llevarse a cabo utilizando la rutina de exponentiación modular dada en la Figura 7.16.

El cifrado se utiliza para transmitir mensajes de modo que no puedan ser leídos por terceros.

El criptosistema RSA es un método de cifrado muy popular.

El algoritmo funciona porque la elección de los valores  $e$ ,  $d$  y  $N$  garantiza (gracias a una demostración de la teoría de números que cae fuera del alcance de este texto) que  $M^d \equiv M \pmod{N}$ , siempre y cuando  $M$  y  $N$  no tengan ningún factor común. Puesto que los únicos factores de  $N$  son dos números primos de 100 dígitos, es prácticamente imposible que eso suceda.<sup>5</sup> Por tanto, el descifrado de texto cifrado nos permite obtener de nuevo el texto original.

Lo que hace que este esquema parezca seguro es que aparentemente se requiere conocer  $d$  para poder efectuar la decodificación. Ahora bien,  $N$  y  $e$  determinan de manera unívoca el valor de  $d$ . Por ejemplo, si conseguimos factorizar  $N$ , obtenemos  $p$  y  $q$  y podremos reconstruir el valor de  $d$ . Lo que pasa es que la factorización es aparentemente muy difícil de realizar para números de gran tamaño. Por tanto, la seguridad del sistema RSA se basa en la creencia de que factorizar números de gran tamaño es intrínsecamente muy difícil. Hasta el momento, esa creencia se ha demostrado correcta.

Este esquema general se conoce con el nombre de *criptografía de clave pública*, que permite que cualquiera que desee recibir mensajes publique una información de cifrado que cualquiera pueda utilizar al mismo tiempo que mantiene el código de descifrado en secreto. En el sistema RSA, los valores de  $e$  y  $N$  serían calculados una sola vez por cada persona y se darían a conocer en un lugar de pública lectura.

El algoritmo RSA se utiliza ampliamente para implementar sistemas seguros de correo electrónico, así como sistemas seguros de transacciones por Internet. Cuando accedemos a una página web a través del protocolo https, se realiza una transacción segura utilizando medios criptográficos. El método que se emplea en realidad es más complejo que el que aquí hemos descrito. Uno de los problemas que obligan a emplear variantes distintas de la que aquí se ha expuesto es que el algoritmo RSA es algo lento a la hora de enviar mensajes de gran tamaño.

Un método más rápido se denomina *DES*. A diferencia del algoritmo RSA, DES es un algoritmo de una sola clave, lo que quiere decir que se emplea la misma clave para cifrar y para descifrar. Es algo parecido a la llave que utilizariamos para la puerta de nuestro domicilio. El problema con los algoritmos de una sola clave es que ambos interlocutores necesitan compartir esa única clave. ¿Cómo puede uno de los interlocutores cerciorarse de que el otro disponga de esa clave? Este problema puede resolverse utilizando el

algoritmo RSA. Una solución típica consistiría en que, por ejemplo, Alice generara aleatoriamente una única clave para el cifrado DES. Después, cifraría su mensaje utilizando DES, que es mucho más rápido que RSA. A continuación, transmitiría el mensaje cifrado a Bob. Para que Bob pueda decodificar el mensaje cifrado, necesita obtener la clave DES que Alice ha utilizado. Una clave DES es relativamente corta, por lo que Alice puede utilizar RSA para cifrar la clave DES y luego enviársela a Bob en una segunda transmisión. Bob descifraría, a continuación, esa segunda transmisión de Alice, obteniendo la clave DES, con lo que podrá descifrar el mensaje original. Estos tipos de protocolos con una serie de mejoras forman la base de la mayoría de las implementaciones prácticas de los sistemas de cifrado.

En la *criptografía de clave pública*, cada participante publica el código que otras personas pueden utilizar para enviarle mensajes cifrados, pero mantiene el código de descifrado en secreto.

En la práctica, RSA se emplea para cifrar la clave utilizada por algún algoritmo de cifrado de una sola clave, como DES.

<sup>5</sup> Es más probable ganar tres veces seguidas a la lotería. Sin embargo, si  $M$  y  $N$  tienen un factor común, el sistema se verá comprometido, porque el máximo común divisor será un factor de  $N$ .

## 7.5 Algoritmos de tipo divide y vencerás

Una importante técnica de resolución de problemas que hace uso de la recursión es la conocida como técnica de divide y vencerás. Un *algoritmo de divide y vencerás* es un eficiente algoritmo recursivo que está compuesto de dos partes:

- *Divide*, durante la cual se resuelven recursivamente problemas sucesivamente más pequeños (excepto, por supuesto, los casos base, que se resuelven directamente).
- *Vencerás*, en la que se forma la solución al problema original componiendo las soluciones a los subproblemas.

Un algoritmo de tipo divide y vencerás es un algoritmo recursivo que, por regla general, es muy eficiente.

En la técnica de divide y vencerás, la recursión es el divide y el procesamiento adicional es el vencerás.

Tradicionalmente, las rutinas en las que el algoritmo contiene al menos dos llamadas recursivas se denominan algoritmos de tipo divide y vencerás, mientras que las rutinas cuyo texto contiene solo una llamada recursiva no se clasifican dentro de esa categoría. En consecuencia, las rutinas recursivas presentadas hasta ahora en este capítulo no son algoritmos de tipo divide y vencerás. Asimismo, los subproblemas usualmente deben ser disjuntos (es decir, esencialmente no solapados), para evitar los excesos de costes que ya hemos podido ver en los cálculos recursivos de ejemplo de los números de Fibonacci.

En esta sección proporcionamos un ejemplo del paradigma divide y vencerás. En primer lugar, veremos cómo utilizar la recursión para resolver el problema de la suma máxima de subsecuencia. Después, proporcionaremos un análisis para demostrar que el tiempo de ejecución es  $O(N \log N)$ . Aunque ya hemos utilizado un algoritmo lineal para este problema, la solución es bastante ilustrativa de un conjunto de soluciones que tienen un amplio rango de aplicaciones, incluyendo los algoritmos de ordenación, como el de ordenación por mezcla y la ordenación rápida, que veremos en el Capítulo 8. En consecuencia, aprender la técnica es importante. Finalmente, mostraremos la forma general para el cálculo del tiempo de ejecución de una amplia gama de algoritmos de tipo divide y vencerás.

### 7.5.1 El problema de la suma máxima de subsecuencia contigua

En la Sección 5.3 hemos hablado del problema de encontrar, dentro de una secuencia de números, una subsecuencia contigua cuya suma sea máxima. Por comodidad, vamos a volver a enunciar el problema aquí.

#### Problema de la suma máxima de una subsecuencia contigua

Dada una serie de enteros (posiblemente negativos)  $A_1, A_2, \dots, A_N$ , encontrar el valor máximo de  $\sum_{k=1}^j A_k$  e identificar la secuencia correspondiente. La suma máxima de una secuencia contigua es cero si todos los enteros son negativos.

Allí presentamos tres algoritmos de diferente complejidad. Uno era un algoritmo basado en una búsqueda exhaustiva: calculábamos la suma de cada subsecuencia posible y luego seleccionábamos el máximo. También describimos una mejora cuadrática que aprovechaba el hecho de que cada subsecuencia puede calcularse en un tiempo constante a partir de una subsecuencia anterior. Puesto

El problema de la suma máxima de una subsecuencia contigua puede resolver con un algoritmo de tipo divide y vencerás.

que tenemos  $O(N^2)$  subsecuencias, esta cota es la mejor que se puede obtener cuando se emplea una técnica en la que se examinan directamente todas las subsecuencias. También proporcionamos un algoritmo de tiempo lineal que examinaba solo unas pocas subsecuencias. Sin embargo, la corrección de ese algoritmo no era obvia.

Vamos a considerar un algoritmo de tipo divide y vencerás. Suponga que la secuencia de entrada de ejemplo es  $\{4, -3, 5, -2, -1, 2, 6, -2\}$ . Vamos a dividir esta entrada en dos mitades como se muestra en la Figura 7.19. Entonces, la suma máxima de subsecuencia contigua puede producirse de tres formas distintas:

- *Caso 1:* reside enteramente en la primera mitad.
- *Caso 2:* reside enteramente en la segunda mitad.
- *Caso 3:* comienza en la primera mitad y termina en la segunda mitad.

Vamos a ver cómo encontrar los máximos de cada uno de estos tres casos de manera más eficiente que utilizando una búsqueda exhaustiva.

Comencemos examinando el caso 3. Queremos evitar el bucle anidado que resultaría de considerar todos los  $N/2$  puntos iniciales y los  $N/2$  puntos finales independientemente. Podemos evitar esa situación sustituyendo dos bucles anidados por dos bucles consecutivos. Los bucles consecutivos, cada uno de ellos de tamaño  $N/2$ , se combinan para dar lugar a un procesamiento de carácter simplemente lineal. Podemos hacer esta sustitución porque cualquier subsecuencia contigua que comience en la primera mitad y termine en la segunda mitad debe incluir tanto el último elemento de la primera mitad como el primer elemento de la segunda mitad.

La Figura 7.19 muestra que, para cada elemento de la primera mitad, podemos calcular la suma de subsecuencia contigua que termina en el elemento situado más a la derecha. Podemos hacer esto con una exploración que vaya de derecha a izquierda, comenzando por la línea divisoria entre las dos mitades. De forma similar, podemos calcular la suma de subsecuencia contigua para todas las subsecuencias que comiencen con el primer elemento de la segunda mitad. Después, podemos combinar estas dos subsecuencias para formar la máxima subsecuencia contigua que abarque la frontera divisoria. En este ejemplo, la secuencia resultante va desde el primer elemento de la primera mitad al penúltimo elemento de la segunda mitad. La suma total es la suma de las dos subsecuencias, es decir  $4 + 7 = 11$ .

Este análisis muestra que el caso 3 se puede resolver en un tiempo lineal. ¿Pero qué sucede con los casos 1 y 2? Puesto que hay  $N/2$  elementos en cada mitad, una búsqueda exhaustiva aplicada a cada mitad seguirá requiriendo un tiempo cuadrático; específicamente, lo único que habremos hecho es eliminar la mitad del trabajo, pero la mitad de un tiempo cuadrático sigue siendo cuadrático.

Primera mitad	Segunda mitad	
4    -3    5    -2	-1    2    6    -2	Valores
4*    0    3    -2	-1    1    7*    5	Suma acumulada
Suma acumulada a partir del centro (*indica el máximo de cada una de las mitades).		

Figura 7.19 División del problema de la subsecuencia máxima contigua en dos mitades.

En consecuencia, en los casos 1 y 2 lo que hacemos es aplicar la misma estrategia –dividir en más mitades. Podemos continuar dividiendo esas mitades una y otra vez hasta que la división sea imposible. Este enfoque se puede enunciar sucintamente de la forma siguiente: *resolver los casos 1 y 2 recursivamente*. Como demostraremos más adelante, al hacer esto se reduce el tiempo de ejecución por debajo de la cota cuadrática, porque los ahorros se van acumulando a lo largo del algoritmo. A continuación se muestra un resumen de la parte principal del algoritmo.

1. Calcular recursivamente la suma máxima de las subsecuencias contiguas que residan enteramente en la primera mitad.
2. Calcular recursivamente la suma máxima de las subsecuencias contiguas que residan enteramente en la segunda mitad.
3. Calcular, mediante dos bucles consecutivos, la suma máxima de las subsecuencias contiguas que comiencen en la primera mitad y terminen en la segunda mitad.
4. Seleccionar el máximo de las tres sumas.

Todo algoritmo recursivo requiere que se especifique un caso base. Cuando el tamaño del problema sea de solo un elemento, no utilizaremos recursión. El método Java resultante se muestra en la Figura 7.20.

La forma general de la llamada recursiva consiste en pasar la matriz de entrada junto con los límites derecho e izquierdo, que delimitan la parte de la matriz sobre la que se está operando. Una rutina de preparación de una única línea se encarga de preparar el terreno, pasando como parámetros los límites 0 y  $N - 1$  junto con la matriz.

Las líneas 12 y 13 se encargan del caso base. Si `left == right`, habrá un solo elemento, y será la subsecuencia máxima contigua en caso de que el elemento sea no negativo (en caso contrario, la secuencia vacía de suma 0 será la máxima). Las líneas 15 y 16 realizan las dos llamadas recursivas. Estas llamadas siempre se aplican a un problema más pequeño que el problema original; por tanto, vamos progresando hacia el caso base. Las líneas 18 a 23 y 25 a 30 calculan las sumas máximas de las secuencias que comienzan en el borde central. La suma de estos dos valores será la suma máxima de las secuencias que abarquen ambas mitades. La rutina `max3` (no mostrada aquí) devuelve el valor máximo de las tres posibilidades disponibles.

### 7.5.2 Análisis de una recurrencia básica de tipo divide y vencerás

El algoritmo recursivo de cálculo de la suma máxima de subsecuencia contigua funciona realizando un procesamiento lineal para calcular una suma que incluye el borde central, y realizando luego dos llamadas recursivas. Estas llamadas, en conjunto, calculan una suma que incluye el borde central, efectúan llamadas recursivas adicionales, etc. El trabajo total realizado por el algoritmo será entonces proporcional al trabajo de exploración realizado en todas las llamadas recursivas.

La Figura 7.21 ilustra gráficamente cómo funciona el algoritmo para  $N = 8$  elementos. Cada rectángulo representa una llamada a `maxSumRec` y la longitud del rectángulo es proporcional al tamaño de la submatriz (y por tanto al coste de explorar la submatriz con la que se está operando en

Análisis intuitivo del algoritmo de tipo divide y vencerás para el cálculo de la suma máxima de subsecuencia contigua: invertimos  $O(N)$  por cada nivel.

```

1  /**
2  * Algoritmo recursivo de suma máxima de subsecuencia contigua.
3  * Encuentra la suma máxima en una submatriz que abarca a[left..right].
4  * No trata de mantener la mejor secuencia actual.
5  */
6  private static int maxSumRec( int [ ] a, int left, int right )
7  {
8      int maxLeftBorderSum = 0, maxRightBorderSum = 0;
9      int leftBorderSum = 0, rightBorderSum = 0;
10     int center = ( left + right ) / 2;
11
12     if( left == right ) // Caso base
13         return a[ left ] > 0 ? a[ left ] : 0;
14
15     int maxLeftSum = maxSumRec( a, left, center );
16     int maxRightSum = maxSumRec( a, center + 1, right );
17
18     for( int i = center; i >= left; i-- )
19     {
20         leftBorderSum += a[ i ];
21         if( leftBorderSum > maxLeftBorderSum )
22             maxLeftBorderSum = leftBorderSum;
23     }
24
25     for( int i = center + 1; i <= right; i++ )
26     {
27         rightBorderSum += a[ i ];
28         if( rightBorderSum > maxRightBorderSum )
29             maxRightBorderSum = rightBorderSum;
30     }
31
32     return max3( maxLeftSum, maxRightSum,
33                 maxLeftBorderSum + maxRightBorderSum );
34 }
35
36 /**
37 * Rutina de preparación para el algoritmo de tipo divide y vencerás
38 * para el cálculo de la suma máxima de subsecuencia contigua.
39 */
40 public static int maxSubsequenceSum( int [ ] a )
41 {
42     return a.length > 0 ? maxSumRec( a, 0, a.length - 1 ) : 0;
43 }

```

**Figura 7.20** Un algoritmo de tipo divide y vencerás para el problema del cálculo de la suma máxima de subsecuencia contigua.

esa invocación). La llamada inicial se muestra en la primera línea; el tamaño de la submatriz es  $N$ , que representa el coste de exploración asociado con el tercer caso. La llamada inicial hace entonces dos llamadas recursivas, lo que nos da dos submatrices de tamaño  $N/2$ . El coste de cada exploración en el caso 3 será la mitad del coste original, pero como hay dos llamadas recursivas, el coste combinado de dichas llamadas será también  $N$ . Cada una de estas dos instancias recursivas hace a su vez otras dos llamadas recursivas, lo que nos da cuatro subproblemas que tienen un tamaño igual a un cuarto del tamaño original. Por tanto, el total de los costes asociados con el caso 3 es también  $N$ .

Al final, terminaremos por llegar al caso base. Cada caso base tiene un tamaño igual a 1 y hay  $N$  de esos casos. Por supuesto, en este caso no hay ningún coste asociado con el caso 3, pero calculamos una unidad de coste por realizar la comprobación que determina si el único elemento restante es positivo o negativo. El coste total es, por tanto, como se ilustra en la Figura 7.21,  $N$  por cada nivel de recursión. Cada nivel divide a la mitad el tamaño del problema básico, por lo que el principio de división por la mitad nos dice que habrá aproximadamente  $\log N$  niveles. De hecho, el número de niveles es  $1 + \lceil \log N \rceil$  (que será igual a 4 cuando  $N$  sea igual a 8). Por tanto, cabe esperar que el tiempo total de ejecución sea  $O(N \log N)$ .

Este análisis nos da una explicación intuitiva de por qué el tiempo de ejecución es  $O(N \log N)$ . Sin embargo, en general, expandir un algoritmo recursivo para examinar su comportamiento no es buena idea; viola la tercera regla de la recursión. A continuación, vamos a considerar un tratamiento matemático más formal.

Sea  $T(N)$  el tiempo requerido para resolver un problema de suma máxima de subsecuencia contigua de tamaño  $N$ . Si  $N = 1$ , el programa requiere una cierta cantidad constante de tiempo para ejecutar las líneas 12 a 13, la cual consideraremos que es igual a 1 unidad. Por tanto,  $T(1) = 1$ . En caso contrario, el programa debe efectuar dos llamadas recursivas y el trabajo lineal implicado en calcular la suma máxima para el caso 3. El procesamiento constante adicional es absorbido por el término  $O(N)$ . ¿Cuánto tardan las dos llamadas recursivas? Puesto que resuelve problemas de tamaño  $N/2$ , sabemos que cada una debe requerir  $T(N/2)$  unidades de tiempo; en consecuencia, el trabajo recursivo total es  $2T(N/2)$ . Este análisis nos da las ecuaciones

$$\begin{aligned} T(1) &= 1 \\ T(N) &= 2T(N/2) + O(N) \end{aligned}$$

Por supuesto, para que la segunda ecuación tenga sentido,  $N$  debe ser una potencia de 2. En caso contrario, en algún punto  $N/2$  no sería par. Una ecuación más precisa sería

$$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N)$$

Para simplificar los cálculos, vamos a asumir que  $N$  es una potencia de 2 y a sustituir el término  $O(N)$  por  $N$ . Estas suposiciones no tienen demasiada importancia y no afectan al resultado  $O$  mayúscula. En consecuencia, necesitamos obtener una solución explícita para  $T(N)$  a partir de

$$T(1) = 1 \quad \text{y} \quad T(N) = 2T(N/2) + N \tag{7.6}$$

Esta ecuación se ilustra en la Figura 7.21, así que sabemos que la respuesta será  $N \log N + N$ . Podemos verificar fácilmente el resultado examinando unos cuantos valores:  $T(1) = 1$ ,  $T(2) = 4$ ,  $T(4) = 12$ ,  $T(8) = 32$  y  $T(16) = 80$ . Ahora vamos a demostrar este análisis matemáticamente en el Teorema 7.4, utilizando dos métodos diferentes.

Observe que el análisis más formal es aplicable a todas las clases de algoritmos que resuelvan recursivamente dos mitades y utilicen un trabajo adicional lineal.

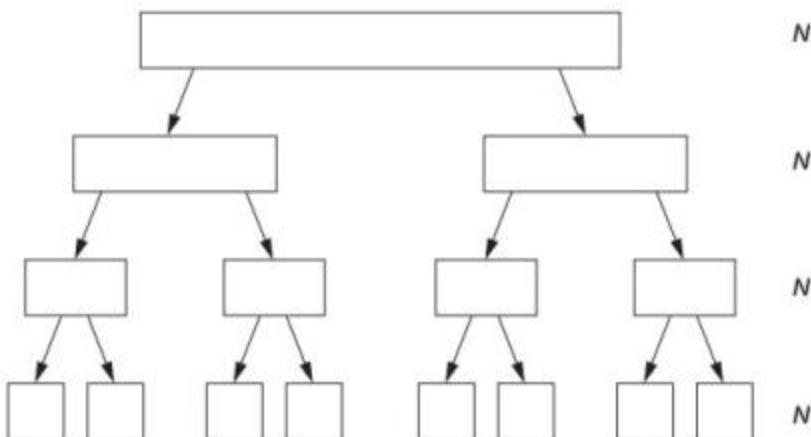


Figura 7.21 Traza de las llamadas recursivas en el algoritmo recursivo de cálculo de la suma máxima de subsecuencia contigua, en el caso de  $N = 8$  elementos.

#### Teorema 7.4

Suponiendo que  $N$  es una potencia de 2, la solución a la ecuación  $T(N) = 2T(N/2) + N$  con la condición inicial  $T(1) = 1$ , es  $T(N) = N\log N + N$ .

#### Demostración (método 1)

Para un valor de  $N$  suficientemente grande, tenemos que  $T(N/2) = 2T(N/4) + N/2$ , porque podemos utilizar la Ecuación 7.6 con  $N/2$  en lugar de  $N$ . En consecuencia, tenemos

$$2T(N/2) = 4T(N/4) + N$$

Sustituyendo esto en la Ecuación 7.6, obtenemos

$$T(N) = 4T(N/4) + 2N \quad (7.7)$$

Si empleamos la Ecuación 7.6 para  $N/4$  y multiplicamos por 4, obtenemos

$$4T(N/4) = 8T(N/8) + N$$

que podemos utilizar para sustituirlo en el lado derecho de la Ecuación 7.7, obteniendo

$$T(N) = 8T(N/8) + 3N$$

Continuando de esta forma, obtenemos

$$T(N) = 2^k T(N/2^k) + kN$$

Finalmente, utilizando  $k = \log N$  (lo que tiene sentido, porque entonces  $2^k = N$ ), obtenemos

$$T(N) = NT(1) + N\log N = N\log N + N$$

Aunque este método de demostración parece funcionar bien, puede ser difícil de aplicar en algunos otros casos más complicados, porque tiende a dar ecuaciones muy largas. A continuación se muestra un segundo método que parece ser más fácil, porque genera ecuaciones verticalmente que resultan más fáciles de manipular.

### Demostración del Teorema 7.4 (método 2)

Dividimos la Ecuación 7.6 entre  $N$ , lo que nos una nueva ecuación:

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

Esta ecuación ahora es válida para cualquier  $N$  que sea una potencia de 2, por lo que podemos escribir las siguientes ecuaciones

$$\begin{aligned} \frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + 1 \\ \frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + 1 \\ \frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + 1 \\ &\quad \dots \\ \frac{T(2)}{2} &= \frac{T(1)}{1} + 1 \end{aligned} \tag{7.8}$$

Una suma telescopica genera un gran número de términos que se cancelan.

Ahora sumamos todas las ecuaciones de la Ecuación 7.8. Es decir, sumamos todos los términos situados en el lado izquierdo e igualamos el resultado a la suma de todos los términos del lado derecho. El término  $T(N/2) / (N/2)$  aparece en ambos lados y por tanto se cancela. De hecho, casi todos los términos aparecen en ambos lados y se cancelan. Esto es lo que se denomina *suma telescopica*. Después de sumarlo 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, por lo que todos los 1 que aparecen en estas ecuaciones suman  $\log N$ . Multiplicando por  $N$  obtenemos la respuesta final que coincide con la obtenida anteriormente.

Observe que, si no hubiéramos dividido por  $N$  al principio de la solución, no se podría haber formado la suma telescopica. Decidir cuál es la división necesaria para garantizar la obtención de una suma telescopica requiere algo de experiencia, y hace que el método sea un poco más difícil de aplicar que la primera alternativa. Sin embargo, una vez que se ha encontrado el divisor correcto, la segunda alternativa tiende a producir una serie de cálculos que encajan mejor en una hoja de papel, lo que conduce a que se produzca un menor número de errores matemáticos. Por el contrario, el primer método utiliza una técnica que está más basada en la fuerza bruta.

Observe que, siempre que tengamos un algoritmo de tipo divide y vencerás que resuelva dos problemas de tamaño mitad con un trabajo lineal adicional, siempre obtendremos un tiempo de ejecución  $O(N \log N)$ .

### 7.5.3 Una cota superior general para el tiempo de ejecución de los algoritmos divide y vencerás

La fórmula general dada en esta sección permite que el número de subproblemas, el tamaño de los subproblemas y la cantidad de trabajo adicional estén representados por fórmulas generales. El resultado puede utilizarse sin necesidad de comprender la demostración.

El análisis de la Sección 7.5.2 muestra que, cuando se divide un problema en dos mitades iguales que se resuelve de forma recursiva con un coste adicional  $O(N)$ , el resultado es un algoritmo  $O(N \log N)$ . ¿Qué pasa si dividimos un problema en tres problemas de tamaño mitad con un coste adicional lineal, o en siete problemas de tamaño mitad con un coste cuadrático adicional? (véase el Ejercicio 7.18). En esta sección vamos a proporcionar una fórmula general para calcular el tiempo de ejecución de un algoritmo de tipo divide y vencerás. La fórmula requiere tres parámetros:

- $A$ , que es el número de subproblemas.
- $B$ , que es el tamaño relativo de los subproblemas (por ejemplo,  $B = 2$  representa subproblemas de tamaño mitad).
- $k$ , que es representativo del hecho de que el coste adicional es  $\Theta(N^k)$ .

La fórmula y su demostración se presentan en el Teorema 7.5. La demostración de la fórmula requiere estar familiarizado con las sumas geométricas. Sin embargo, el conocimiento de la demostración no es necesario para poder utilizar la fórmula.

#### Teorema 7.5

La solución a la ecuación  $T(N) = AT(N/B) + O(N^k)$ , donde  $A \geq 1$  y  $B > 1$ , es

$$T(N) = \begin{cases} O(N^{\log_B A}) & \text{para } A > B^k \\ O(N^k \log N) & \text{para } A = B^k \\ O(N^k) & \text{para } A < B^k \end{cases}$$

Antes de demostrar el Teorema 7.5, vamos a examinar algunas aplicaciones. Para el problema de suma máxima de subsecuencia contigua, tenemos dos problemas, dos mitades y un coste adicional de tipo lineal. Los valores aplicables son  $A = 2$ ,  $B = 2$  y  $k = 1$ . Por tanto, se aplica el segundo caso del Teorema 7.5 y obtenemos  $O(N \log N)$ , lo que concuerda con nuestro cálculo anterior. Si resolvemos de forma recursiva tres problemas de tamaño mitad con un coste lineal adicional, tendremos  $A = 3$ ,  $B = 2$  y  $k = 1$ , y se aplicará el primer caso. El resultado será  $O(N^{\log_2 3}) = O(N^{1.59})$ . Aquí, el coste adicional no contribuye al coste total del algoritmo. Cualquier coste adicional inferior a  $O(N^{1.59})$  nos daría el mismo tiempo de ejecución para el algoritmo recursivo. Un algoritmo que resolviera tres problemas de tamaño mitad, pero requiriera un coste adicional cuadrático, tendría un tiempo de ejecución  $O(N^2)$  porque se aplicaría el tercer caso. De hecho, el coste adicional domina en cuanto excede del umbral  $O(N^{1.59})$ . En el umbral, el coste adicional es el factor logarítmico mostrado en el segundo caso. Ahora, procedamos a demostrar el Teorema 7.5.

### Demostración del Teorema 7.5

Siguiendo el método esbozado en la segunda demostración del Teorema 7.4, vamos a suponer que  $N$  es una potencia de  $B$  y que  $N = B^M$ . Entonces  $N/B = B^{M-1}$  y  $N^k = (B^M)^k = (B^k)^M$ . Suponemos que  $T(1) = 1$  e ignoramos el factor constante  $O(N^k)$ . Entonces tendremos la siguiente ecuación básica

$$T(B^M) = AT(B^{M-1}) + (B^k)^M$$

Si dividimos entre  $A^M$  obtenemos la nueva ecuación básica

$$\frac{T(B^M)}{A^M} = \frac{T(B^{M-1})}{A^{M-1}} + \left(\frac{B^k}{A}\right)^M$$

Ahora podemos escribir esta ecuación para todo  $M$ , obteniendo

$$\begin{aligned} \frac{T(B^M)}{A^M} &= \frac{T(B^{M-1})}{A^{M-1}} + \left(\frac{B^k}{A}\right)^M \\ \frac{T(B^{M-1})}{A^{M-1}} &= \frac{T(B^{M-2})}{A^{M-2}} + \left(\frac{B^k}{A}\right)^{M-1} \\ \frac{T(B^{M-2})}{A^{M-2}} &= \frac{T(B^{M-3})}{A^{M-3}} + \left(\frac{B^k}{A}\right)^{M-2} \\ &\dots \\ \frac{T(B^1)}{A^1} &= \frac{T(B^0)}{A^0} + \left(\frac{B^k}{A}\right)^1 \end{aligned} \tag{7.9}$$

Si sumamos el conjunto de ecuaciones representado por la Ecuación 7.9, de nuevo casi todos los términos del lado izquierdo se cancelan con los primeros términos del lado derecho, lo que nos da

$$\begin{aligned} \frac{T(B^M)}{A^M} &= 1 + \sum_{i=1}^M \left(\frac{B^k}{A}\right)^i \\ &= \sum_{i=0}^M \left(\frac{B^k}{A}\right)^i \end{aligned}$$

Por tanto

$$T(N) = T(B^M) = A^M \sum_{i=0}^M \left(\frac{B^k}{A}\right)^i \tag{7.10}$$

Si  $A > B^k$ , entonces la suma es una serie geométrica con una razón inferior a 1. Puesto que la suma de una serie infinita converge a una constante, esta suma finita también está acotada por una constante. Por tanto, obtenemos

$$T(N) = O(A^M) = O(N^{\log_B A}) \tag{7.11}$$

**Demostración  
del  
Teorema 7.5  
(cont.)**

Si  $A = B^k$ , entonces 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  $A^M = N^k$ .

$$T(N) = O(A^M \log_B N) = O(N^k \log_B N) = O(N^k \log N)$$

Por último, si  $A < B^k$ , entonces los términos de la serie geométrica con mayores que 1. Podemos calcular la suma utilizando una fórmula estándar, obteniendo

$$T(N) = A^M \frac{\left(\frac{B^k}{A}\right)^{M+1} - 1}{\frac{B^k}{A} - 1} = O\left(A^M \left(\frac{B^k}{A}\right)^M\right) = O((B^k)^M) = O(N^k)$$

lo que demuestra el último caso del Teorema 7.5.

## 7.6 Programación dinámica

La programación dinámica resuelve subproblemas de manera no recursiva anotando las respuestas en una tabla.

Un problema que pueda ser expresado matemáticamente de forma recursiva también puede ser expresado como un algoritmo recursivo. En muchos casos, el hacer esto nos proporciona una significativa mejora en el rendimiento, por comparación con las búsquedas exhaustivas más simples. Cualquier fórmula matemática recursiva podría traducirse directamente a un algoritmo recursivo, pero a menudo el compilador no hará justicia al algoritmo recursivo y el resultado será un programa inefficiente. Este es el caso por ejemplo del cálculo recursivo de los números de Fibonacci que hemos descrito en la Sección 7.3.4. Para evitar esta explosión recursiva, podemos utilizar la *programación dinámica* para reescribir el algoritmo recursivo en forma de un algoritmo no recursivo, que vaya anotando sistemáticamente en una tabla las respuestas a los subproblemas. Ilustraremos esta técnica con el siguiente problema.

### Problema del cambio de moneda

Para una moneda nacional en la que haya monedas físicas de  $C_1, C_2, \dots, C_N$  (céntimos), ¿cuál es el número mínimo de monedas necesarias para obtener  $K$  céntimos de cambio?

Los algoritmos voraces toman decisiones localmente óptimas en cada paso. Esta es la cosa más sencilla que podemos hacer, pero no siempre es la correcta.

La moneda nacional de Estados Unidos tiene monedas de 1, 5, 10 y 25 centavos (ignore la moneda de 50 centavos que se usa de forma menos frecuente). Podemos obtener 63 centavos utilizando dos monedas de 25 centavos, una moneda de 10 centavos y tres monedas de 1 centavo, lo que nos da un total de seis monedas. El obtener el cambio necesario con este conjunto de monedas es relativamente sencillo: basta con utilizar repetidamente la moneda de mayor valor que tengamos disponible. Se puede demostrar que para el caso de los Estados Unidos, esta técnica siempre minimiza el número total de monedas utilizadas, lo que es un ejemplo de lo que se denomina algoritmos voraces. En un *algoritmo voraz*, se toma durante cada fase una decisión que parece ser óptima sin ocuparse de las consecuencias futuras. Esta estrategia de "aprovechar ahora lo que puedas" es lo que da el nombre a

este tipo algoritmos. Cuando un problema se puede resolver mediante un algoritmo voraz, solemos mostrarnos satisfechos: los algoritmos voraces suelen casar con nuestra intuición y hacen que la codificación de los programas sea relativamente sencilla. Sin embargo, los algoritmos voraces no siempre funcionan. Si en Estados Unidos se empleara también una moneda de 21 centavos, el algoritmo voraz seguiría dándonos una solución en la que se emplearan seis monedas, pero la solución óptima utilizaría solo tres (tres monedas de 21 centavos).

La cuestión, entonces, es cómo resolver el problema para un conjunto de monedas arbitrario. Vamos a suponer que siempre hay una moneda de 1 céntimo, de modo que la solución siempre existe. Una estrategia simple para seleccionar  $K$  céntimos en monedas utiliza la recursión de la manera siguiente:

1. Si podemos obtener el valor deseado utilizando exactamente una moneda, ese será el mínimo.
2. En caso contrario, para cada posible valor  $i$  calculamos el número mínimo de monedas necesario para obtener  $i$  céntimos y  $K - i$  céntimos independientemente. A continuación, seleccionamos el valor de  $i$  que minimice esta suma.

Por ejemplo, veamos cómo podemos obtener 63 centavos de cambio. Claramente, una moneda no será suficiente. Podemos calcular el número de monedas requeridas para obtener 1 moneda de 1 centavo de cambio y 62 centavos de cambio independientemente (se necesitarán una y cuatro monedas, respectivamente). Obtenemos estos resultados recursivamente, así que deben ser considerados como óptimos (sucede que los 62 centavos se obtienen mediante dos monedas de 21 centavos y dos monedas de 10 centavos). Por tanto, tenemos un método que utiliza cinco monedas. Si dividimos el problema en 2 centavos y 61 centavos, las soluciones recursivas nos dan 2 y 4, respectivamente, lo que hace un total de seis monedas. Continuamos probando todas las posibilidades, algunas de las cuales se muestran en la Figura 7.22. Al final, encontraremos una división en 21 centavos y 42 centavos, que se pueden conseguir con una y dos monedas, respectivamente, lo que a su vez permite obtener el resultado final con solo tres monedas. La última división que necesitaremos probar es 31 centavos y 32 centavos. Podemos obtener 31 centavos con dos monedas y 32 centavos con tres monedas, lo que da un total de cinco; pero el mínimo continuará siendo tres monedas.

Un algoritmo recursivo simple para el cálculo del número de monedas es fácil de escribir, pero resulta poco eficiente.

De nuevo, resolvemos cada uno de estos subproblemas recursivamente, lo que nos lleva al algoritmo natural mostrado en la Figura 7.23. Si ejecutamos el algoritmo para valores totales pequeños, funciona perfectamente, pero como sucede con los cálculos de Fibonacci, este algoritmo

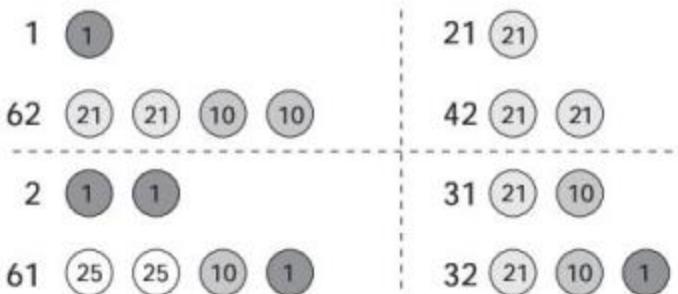


Figura 7.22 Algunos de los subproblemas resueltos recursivamente en la Figura 7.23.

```

1  // Devuelve el número mínimo de monedas necesarias para obtener cierto
2  // valor. Algoritmo recursivo simple muy ineficiente.
3  public static int makeChange( int [ ] coins, int change )
4  {
5      int minCoins = change;
6
7      for( int i = 0; i < coins.length; i++ )
8          if( coins[ i ] == change )
9              return 1;
10
11     // No se corresponde; resolver recursivamente.
12     for( int j = 1; j <= change / 2; j++ )
13     {
14         int thisCoins = makeChange( coins, j )
15                     + makeChange( coins, change - j );
16
17         if( thisCoins < minCoins )
18             minCoins = thisCoins;
19     }
20
21     return minCoins;
22 }
```

**Figura 7.23** Un procedimiento recursivo simple pero ineficiente para resolver el problema del cálculo del número de monedas de cambio.

Nuestro algoritmo recursivo alternativo para el cálculo del número de monedas sigue siendo ineficiente.

demasiado trabajo redundante, y no terminará en un plazo de tiempo razonable para el caso de los 63 centavos.

Un algoritmo alternativo consistiría en reducir el problema recursivamente especificando una de las monedas. Por ejemplo, para 63 centavos, podemos calcular el número de monedas de las siguientes formas, como se muestra en la Figura 7.24.

- Una moneda de 1 centavo más 62 centavos distribuidos recursivamente.
- Una moneda de 5 centavos más 58 centavos distribuidos recursivamente.
- Una moneda de 10 centavos más 53 centavos distribuidos recursivamente.
- Una moneda de 21 centavos más 42 centavos distribuidos recursivamente.
- Una moneda de 25 centavos más 38 centavos distribuidos recursivamente.

En lugar de resolver 62 problemas recursivos, como en la Figura 7.22, solo necesitamos cinco llamadas recursivas, una para cada tipo distinto de moneda. De nuevo, esta implementación

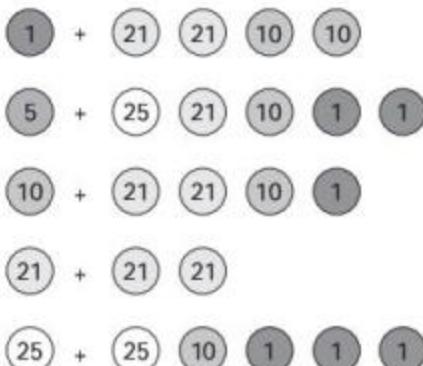


Figura 7.24 Un algoritmo recursivo alternativo para el problema del cálculo de monedas.

recursiva tan ingenua resulta muy inefficiente, porque calcula varias veces ciertas respuestas. Por ejemplo, en el primer caso nos queda el problema de obtener 62 centavos en monedas. En este subproblema, una de las llamadas recursivas selecciona una moneda de 10 centavos y luego resuelve recursivamente el caso correspondiente a 52 centavos. En el tercer caso, nos quedan 53 centavos. Una de sus llamadas recursivas elimina la moneda de 1 centavo y luego resuelve recursivamente el caso de 52 centavos. Este trabajo redundante vuelve a hacer que el tiempo de ejecución sea excesivo. Sin embargo, si somos cuidadosos podemos hacer que el algoritmo se ejecute de forma razonablemente rápida.

El truco consiste en guardar en una matriz las respuestas a los distintos subproblemas. Esta técnica de programación dinámica forma la base de muchos algoritmos. Una respuesta para un valor de gran tamaño depende solo de respuestas para valores menores, así que podemos calcular la forma óptima de cambiar 1 centavo, luego 2 centavos, luego 3 centavos, etc. Esta estrategia se muestra en el método de la Figura 7.25.

En primer lugar, en la línea 8 observamos que 0 centavos pueden cambiarse utilizando cero monedas. Empleamos la matriz `lastCoin` para ver qué moneda se utilizó en último lugar para obtener el cambio óptimo. En caso contrario, tratamos de obtener un cambio igual a `cents`, donde `cents` va de 1 hasta el valor final `maxChange`. Para obtener un cambio igual a `cents`, probamos sucesivamente cada moneda como se indica en la instrucción `for` que comienza en la línea 15. Si el valor de la moneda es mayor que el cambio que estamos intentando obtener, no hay nada que hacer. En caso contrario, comprobamos en la línea 19 si el número de monedas utilizadas para resolver el subproblema más la moneda adicional se combinan para dar un número total de monedas menor que el número mínimo de monedas empleado hasta ahora. En caso afirmativo, se realiza una actualización en las líneas 21 y 22. Cuando el bucle termina para el número actual de `cents`, los mínimos pueden insertarse en las matrices, lo cual se hace en las líneas 26 y 27.

Al final del algoritmo, `coinsUsed[1]` representa el número mínimo de monedas necesarias para obtener `i` centavos de cambio (`i==maxChange` es la solución particular que estamos buscando). Efectuando una traza hacia atrás de `lastCoin`, podemos averiguar las monedas necesarias para obtener la solución. El tiempo de ejecución es el que corresponde a dos bucles `for` anidados y será, por tanto,  $O(NK)$ , donde  $N$  es el número de diferentes tipos de monedas y  $K$  es el valor que estamos intentando obtener.

```

1 // Algoritmo de progr. dinámica para resolver el problema de obtención de
2 // cambio. Como resultado, la matriz coinsUsed se rellena con el nº mín.
3 // de monedas necesarias para obtener todos los valores de 0 -> maxChange
4 // y lastCoin contiene una de las monedas necesarias para esa solución.
5 public static void makeChange( int [ ] coins, int differentCoins,
6     int maxChange, int [ ] coinsUsed, int [ ] lastCoin )
7 {
8     coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;
9
10    for( int cents = 1; cents <= maxChange; cents++ )
11    {
12        int minCoins = cents;
13        int newCoin = 1;
14
15        for( int j = 0; j < differentCoins; j++ )
16        {
17            if( coins[ j ] > cents ) // No se puede usar la moneda j
18                continue;
19            if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins )
20            {
21                minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
22                newCoin = coins[ j ];
23            }
24        }
25
26        coinsUsed[ cents ] = minCoins;
27        lastCoin[ cents ] = newCoin;
28    }
29 }

```

**Figura 7.25** Un algoritmo de programación dinámica para resolver el problema de obtención de cambio calculando el número óptimo de monedas para todas las cantidades que van de 0 a maxChange y manteniendo la información necesaria para construir la secuencia de monedas necesaria.

## 7.7 Retroceso

Un algoritmo de retroceso utiliza la recursión para probar todas las posibilidades.

En esta sección vamos a explicar la última aplicación de la recursión. Demostraremos cómo escribir una rutina para hacer que la computadora seleccione un movimiento en el juego de tres en raya (que en inglés se denomina Tic-Tac-Toe). La clase Best, mostrada en la Figura 7.26, se utiliza para almacenar el movimiento óptimo devuelto por el algoritmo de selección de movimientos. El esqueleto para una clase TicTacToe se muestra en la

```

1 final class Best
2 {
3     int row;
4     int column;
5     int val;
6
7     public Best( int v )
8         { this( v, 0, 0 ); }
9
10    public Best( int v, int r, int c )
11        { val = v; row = r; column = c; }
12 }
```

Figura 7.26 Una clase para almacenar un movimiento evaluado.

Figura 7.27. La clase tiene un objeto de datos `board` que representa la posición actual del juego.<sup>6</sup> Se especifican diversos métodos triviales, incluyendo rutinas para borrar el tablero, para comprobar si un cuadrado está ocupado, para colocar algo en un cuadrado y para comprobar si alguien ha conseguido la victoria. Los detalles de implementación se proporcionan en el código en línea.

El desafío consiste en decidir, para cualquier posición dada, cuál es el mejor movimiento. La rutina utilizada es `chooseMove`. La estrategia general implica el uso de un algoritmo de retroceso. Un *algoritmo de retroceso* utiliza la recursión para comprobar exhaustivamente todas las posibilidades.

La base para tomar esta decisión es `positionValue`, que se muestra en la Figura 7.28. El método `positionValue` devuelve `HUMAN_WIN`, `DRAW`, `COMPUTER_WIN` o `UNCLEAR`, dependiendo de si el tablero representa una victoria del jugador humano, un empate, una victoria de la computadora o una situación que no está clara.

La estrategia empleada es la *estrategia minimax*, que está basada en la suposición de que ambos jugadores juegan de forma óptima. El valor de una posición será `COMPUTER_WIN` si la computadora puede forzar una victoria. Si la computadora puede forzar un empate pero no una victoria, el valor es `DRAW`; si el jugador humano puede forzar una victoria, el valor es `HUMAN_WIN`. Queremos que la computadora gane, así que `HUMAN_WIN < DRAW < COMPUTER_WIN`.

La estrategia minimax se utiliza para el juego de tres en raya. Está basada en la suposición de que ambos jugadores juegan de forma óptima.

Para la computadora, el valor de la posición es el máximo de todos los valores de las posiciones que pueden resultar de hacer un movimiento. Suponga que un movimiento conduce a una posición ganadora, que dos movimientos conducen a un empate y que seis movimientos conducen a una posición perdedora. Entonces, la posición de partida será una posición ganadora, porque la computadora puede forzar la victoria. Además, la jugada que habrá que hacer será precisamente el movimiento que conduce a la posición ganadora. Para el jugador humano, utilizamos el mínimo en lugar del máximo.

<sup>6</sup> El juego de tres en raya utiliza un tablero de tres por tres. Dos jugadores van colocando alternativamente sus fichas en los cuadrados. El primero en conseguir tres cuadrados alineados en una fila, en una columna o en una de las dos diagonales gana.

```

1 class TicTacToe
2 {
3     public static final int HUMAN = 0;
4     public static final int COMPUTER = 1;
5     public static final int EMPTY = 2;
6
7     public static final int HUMAN_WIN = 0;
8     public static final int DRAW = 1;
9     public static final int UNCLEAR = 2;
10    public static final int COMPUTER_WIN = 3;
11
12    // Constructor
13    public TicTacToe( )
14    { clearBoard( ); }
15
16    // Encontrar el movimiento óptimo
17    public Best chooseMove( int side )
18    { /* Implementación en la Figura 7.29 */ }
19
20    // Calcular el valor estático de la posición actual (ganar, empate, etc.)
21    private int positionValue( )
22    { /* Implementación en la Figura 7.28 */ }
23
24    // Hacer una jugada, incluyendo la comprobación de validez
25    public boolean playMove( int side, int row, int column )
26    { /* Implementación en el código en línea*/ }
27
28    // Dejar el tablero vacío
29    public void clearBoard( )
30    { /* Implementación en el código en línea */ }
31
32    // Devolver true si el tablero está lleno
33    public boolean boardIsFull( )
34    { /* Implementación en el código en línea */ }
35
36    // Devolver true si el tablero muestra una victoria
37    public boolean isAWin( int side )
38    { /* Implementación en el código en línea */ }
39
40    // Hacer una jugada, posiblemente borrando un cuadrado
41    private void place( int row, int column, int piece )
42    { board[ row ][ column ] = piece; }
43
44    // Comprobar si un cuadrado está vacío
45    private boolean squareIsEmpty( int row, int column )
46    { return board[ row ][ column ] == EMPTY; }
47
48    private int [ ] [ ] board = new int[ 3 ][ 3 ];
49 }

```

---

**Figura 7.27** Esqueleto de la clase TicTacToe.

Este enfoque sugiere un algoritmo recursivo para determinar el valor de una posición. Llevar la cuenta de cuál es el mejor movimiento posible es una simple cuestión de mantener los correspondientes valores, una vez que hayamos escrito el algoritmo básico para determinar el valor de cada posición. Si la posición es una posición terminal (es decir, si podemos ver directamente que se han conseguido tres en raya o que el tablero está lleno sin que nadie haya conseguido tres en raya), el valor de la posición es inmediato. En caso contrario, comprobamos recursivamente todos los movimientos, calculando el valor de cada posición resultante, después de lo cual calculamos el valor máximo. La llamada recursiva requiere entonces que el jugador humano evalúe el valor de la posición. Para el jugador humano, el valor es el mínimo de todos los posibles movimientos siguientes, porque dicho jugador está tratando de forzar una derrota de la computadora. Por tanto, el método recursivo `chooseMove`, mostrado en la Figura 7.29, admite un parámetro `side`, que indica a quién le corresponde mover.

Las líneas 12 y 13 se encargan de tratar el caso base de la recursión. Si tenemos una respuesta inmediata, podemos volver. En caso contrario, configuramos algunos valores en las líneas 15 a 22, dependiendo de a qué jugador le corresponda mover. El código de las líneas 28 a 38 se ejecuta una vez por cada uno de los movimientos disponibles. Probamos el movimiento en la línea 28, evaluamos recursivamente el movimiento en la línea 29 (guardando el valor) y luego deshacemos el movimiento en la línea 30. Las líneas 33 y 34 comprueban si este movimiento es el mejor que se ha encontrado hasta el momento. En caso afirmativo, se ajusta `value` en la línea 36 y se anota el movimiento en la línea 37. En la línea 41 devolvemos el valor de la posición en un objeto `Best`.

Aunque la rutina mostrada en la Figura 7.29 resuelve de forma óptima el juego de tres en raya, realiza una considerable tarea de búsqueda. Específicamente, para seleccionar el primer movimiento en un tablero vacío realiza 549.946 llamadas recursivas (este número se obtiene ejecutando el programa). Utilizando algunos trucos algorítmicos, podemos calcular la misma información con una cantidad menor de búsquedas. Una de dichas técnicas de optimización se conoce con el nombre de *poda alfa-beta*, que es una mejora del algoritmo minimax. Describiremos esta técnica en detalle en el Capítulo 10. La aplicación de la poda alfa-beta reduce el número de llamadas a solo 18.297.

La poda alfa-beta es una mejora del algoritmo minimax.

```
1 // Calcular el valor estático de la posición actual (ganar, empate, etc.)  
2 private int positionValue( )  
3 {  
4     return isAWin( COMPUTER ) ? COMPUTER_WIN :  
5         isAWin( HUMAN ) ? HUMAN_WIN :  
6             boardIsFull( ) ? DRAW : UNCLEAR;  
7 }
```

Figura 7.28 Rutina de soporte para la evaluación de posiciones.

```
1 // Encontrar el movimiento óptimo
2 public Best chooseMove( int side )
3 {
4     int opp;           // El otro jugador
5     Best reply;        // Mejor respuesta del oponente
6     int dc;            // Variable auxiliar
7     int simpleEval;   // Resultado de una evaluación inmediata
8     int bestRow = 0;
9     int bestColumn = 0;
10    int value;
11
12    if( ( simpleEval = positionValue( ) ) != UNCLEAR )
13        return new Best( simpleEval );
14
15    if( side == COMPUTER )
16    {
17        opp = HUMAN; value = HUMAN_WIN;
18    }
19    else
20    {
21        opp = COMPUTER; value = COMPUTER_WIN;
22    }
23
24    for( int row = 0; row < 3; row++ )
25        for( int column = 0; column < 3; column++ )
26            if( squareIsEmpty( row, column ) )
27            {
28                place( row, column, side );
29                reply = chooseMove( opp );
30                place( row, column, EMPTY );
31
32                // Actualizar si el jugador obtiene una mejor posición
33                if( side == COMPUTER && reply.val > value
34                    || side == HUMAN && reply.val < value )
35                {
36                    value = reply.val;
37                    bestRow = row; bestColumn = column;
38                }
39            }
40
41    return new Best( value, bestRow, bestColumn );
42 }
```

---

Figura 7.29 Una rutina recursiva para encontrar un movimiento óptimo en el tres en raya.

## Resumen

En este capítulo hemos examinado la recursión y hemos demostrado que se trata de una potente herramienta de resolución de problemas. A continuación se exponen sus reglas fundamentales, de las que no debería olvidarse nunca.

1. *Caso base*: siempre tiene que haber al menos un caso que se pueda resolver sin utilizar recursión.
2. *Progresión*: toda llamada recursiva debe progresar hacia un caso base.
3. "*Es necesario creer*": asuma siempre que la llamada recursiva funciona.
4. *Regla del Interés compuesta*: nunca duplique el trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

La recursión tiene muchas aplicaciones, algunas de las cuales han sido descritas en este capítulo. Tres importantes técnicas de diseño de algoritmos basadas en la recursión son las técnicas de divide y vencerás, de programación dinámica y de retrocesos.

En el Capítulo 8 examinaremos el tema de la ordenación. El algoritmo de ordenación más rápido conocido tiene carácter recursivo.



## Conceptos clave

**algoritmo de tipo divide y vencerás** Un tipo de algoritmo recursivo que generalmente es muy eficiente. La recursión es la parte de *divide*, mientras que la combinación de las soluciones recursivas es la parte de *vencerás*. (313)

**algoritmo voraz** Es un algoritmo que toma decisiones localmente óptimas en cada paso –una forma simple, pero no siempre correcta de proceder. (322)

**árbol** Una estructura de datos ampliamente utilizada que está compuesta por un conjunto de nodos y un conjunto de aristas que conectan parejas de nodos. A lo largo del libro, asumiremos que el árbol posee una raíz. (299)

**base** En una demostración por inducción, el caso sencillo que se puede demostrar a mano. (289)

**caso base** Una instancia que puede resolverse sin necesidad de recursión. Toda llamada recursiva debe progresar hacia un caso base. (292)

**cifrado** Un esquema de codificación utilizado en la transmisión de mensajes de modo que no puedan ser leídos por terceros. (311)

**criptografía de clave pública** Un tipo de criptografía en la que cada participante publica el código que otros pueden utilizar para enviarle mensajes cifrados, al mismo tiempo que mantiene el código de descifrado en secreto. (312)

**criptosistema RSA** Un popular método de cifrado. (311)

**estrategia minimax** Una estrategia utilizada para el tres en raya y otros juegos de estrategia, basada en la suposición de que ambos jugadores juegan de forma óptima. (327)

**hipótesis inductiva** La hipótesis de que un teorema es cierto para un caso arbitrario y de que, bajo esta suposición, también es cierto para el siguiente caso. (290)

**hoja** En un árbol, un nodo que no tiene ningún hijo. (300)

**inducción** Una técnica de demostración utilizada para demostrar teoremas que son válidos para enteros positivos. (288)

**inversa multiplicativa** La solución  $1 \leq X < N$  a la ecuación  $AX \equiv 1 \pmod{N}$ . (308)

**máximo común divisor** El máximo común divisor de dos enteros es el mayor entero que es divisor de ambos. (307)

**método recursivo** Un método que realiza, directa o indirectamente, una llamada a sí mismo. (291)

**números de Fibonacci** Una secuencia de números en la que el  $i$ -ésimo número es la suma de los dos números anteriores. (298)

**poda alfa-beta** Una mejora del algoritmo minimax. (329)

**programación dinámica** Una técnica que evita la explosión recursiva, anotando las respuestas en una tabla. (322)

**registro de activación** El método mediante el que se lleva a cabo la gestión del procesamiento en un lenguaje procedimental. Se utiliza una pila de registros de activación. (297)

**reglas de la recursión** 1. *Caso base*: siempre tiene que haber al menos un caso que se pueda resolver sin utilizar recursión. (292); 2. *Progresión*: toda llamada recursiva debe progresar hacia un caso base. (292); 3. “*Es necesario creer*”: asuma siempre que la llamada recursiva funciona. (296); 4. *Regla del interés compuesto*: nunca duplique el trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas. (299)

**retroceso** Un algoritmo que utiliza la recursión para probar todas las posibilidades. (326)

**rutina de preparación** Una rutina que comprueba la validez del primer caso y luego invoca la rutina recursiva. (294)

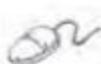
**suma telescopica** Un procedimiento que genera grandes cantidades de términos que se cancelan entre sí. (319)



## Errores comunes

1. El error más común en el uso de la recursión es olvidarse de incluir un caso base.
2. Asegúrese de que cada llamada recursiva progresiona hacia un caso base. En caso contrario, la recursión será incorrecta.
3. Deben evitarse las llamadas recursivas solapadas, porque tienden a dar algoritmos con tiempo de procesamiento exponencial.

- 4 No es un buen estilo de programación utilizar la recursión en lugar de un bucle sencillo.
- 5 Los algoritmos recursivos se analizan utilizando una fórmula recursiva. No dé por supuesto que una llamada recursiva consume un tiempo lineal.



## Internet

Se proporciona la mayor parte del código del capítulo, incluyendo un programa del juego de tres en raya. En el Capítulo 10 se expondrá una versión mejorada del algoritmo del juego de tres en raya, que utiliza estructuras de datos más sofisticadas. A continuación se indican los nombres de archivo.

### **RecSum.java**

La rutina mostrada en la Figura 7.1 con un método `main` simple.

### **PrintInt.java**

La rutina mostrada en la Figura 7.4 para imprimir un número en cualquier base más un método `main`.

### **Factorial.java**

La rutina mostrada en la Figura 7.10, para calcular factoriales.

### **BinarySearchRecursive.java**

Prácticamente igual que **BinarySearch.java** (en el Capítulo 6), pero con la rutina `binarySearch` mostrada en la Figura 7.11.

### **Ruler.java**

La rutina mostrada en la Figura 7.13, lista para ejecutarse. Contiene código que obliga a que el dibujo se realice lentamente.

### **FractalStar.java**

La rutina mostrada en la Figura 7.15, lista para ejecutarse. Contiene código que obliga a que el dibujo se realice lentamente.

### **Numerical.java**

Las rutinas matemáticas presentadas en la Sección 7.4, la rutina de prueba de primalidad y un método `main` en **RSA.java** que ilustra los cálculos RSA.

### **MaxSumTest.java**

Las cuatro rutinas de suma máxima de subsecuencia contigua.

### **MakeChange.java**

La rutina mostrada en la Figura 7.25, con un método `main` simple.

### **TicTacSlow.java**

El algoritmo para el juego de tres en raya con una rutina `main` primitiva. Véase también **Best.java**.



## Ejercicios

### EN RESUMEN

- 7.1** Calcule el máximo común divisor  $\text{gcd}(1995, 1494)$ .
- 7.2** A continuación se muestran cuatro alternativas para la línea 11 de la rutina `power` (en la Figura 7.16). ¿Por qué es errónea cada alternativa?

```
long tmp = power( x * x, n/2, p );
long tmp = power( power( x, 2, p ), n/2, p );
long tmp = power( power( x, n/2, p ), 2, p );
long tmp = power( x, n/2, p ) * power( x, n/2, p ) % p;
```

- 7.3** Demuestre que el algoritmo voraz de cálculo del número de monedas falla si no existen monedas de cinco centavos.
- 7.4** ¿Cuáles son las cuatro reglas fundamentales de la recursión?
- 7.5** Bob selecciona sendos valores  $p$  y  $q$  iguales a 31 y 41, respectivamente. Determine valores aceptables para los restantes parámetros del algoritmo RSA.
- 7.6** Modifique el programa dado en la Figura 7.1 de modo que se devuelva cero para todo valor de  $n$  negativo. Haga el menor número necesario de cambios.

### EN TEORÍA

- 7.7** Muestre cómo se procesan las llamadas recursivas en el cálculo  $2^{63} \bmod 37$ .
- 7.8** Demuestre que si  $A \equiv B \pmod{N}$ , entonces para cualquier  $C, D$  y  $P$ , los siguientes enunciados son ciertos.
- $A + C \equiv B + C \pmod{N}$
  - $AD \equiv BD \pmod{N}$
  - $A^P \equiv B^P \pmod{N}$
- 7.9** Demuestre por inducción la fórmula del número de llamadas al método recursivo `fib` de la Sección 7.3.4.
- 7.10** Demuestre por inducción que si  $A > B \geq 0$  y la invocación `gcd(a, b)` realiza  $k \geq 1$  llamadas recursivas, entonces  $A \geq F_{k+2}$  y  $B \geq F_{k+1}$ .
- 7.11** Demuestre las siguientes identidades en relación con los números de Fibonacci.
- $F_1 + F_2 + \dots + F_N = F_{N+2} - 1$
  - $F_1 + F_3 + \dots + F_{2N-1} = F_{2N}$
  - $F_0 + F_2 + \dots + F_{2N} = F_{2N+1} - 1$
  - $F_{N-1} F_{N+1} = (-1)^N + F_N^2$
- 7.12** Demuestre por inducción la fórmula

$$F_N = \frac{1}{\sqrt{5}} \left( \left( \frac{1+\sqrt{5}}{2} \right)^N - \left( \frac{1-\sqrt{5}}{2} \right)^N \right)$$

- 7.13** Demuestre que si  $A \geq B$ , entonces  $A \bmod B < A/2$ . (*Pista:* considere los casos  $B \leq A/2$  y  $B > A/2$  por separado.) ¿Cómo demuestra este resultado que el tiempo de ejecución de gcd es logarítmico?
- 7.14** Resuelva la siguiente ecuación. Suponga que  $A \geq 1$ ,  $B > 1$  y  $P \geq 0$ .
- $$T(N) = AT(N/B) + O(N^P \log^P N)$$
- 7.15** Resuelva las siguientes recurrencias, que en todos los casos tienen  $T(0) = T(1) = 1$ . Basta con proporcionar una respuesta O mayúscula.
- $T(N) = T(N/2) + 1$
  - $T(N) = T(N/2) + N$
  - $T(N) = T(N/2) + N^2$
- 7.16** Demuestre por inducción que en el algoritmo gcd ampliado,  $|X| < B$  y  $|Y| < A$ .
- 7.17** Escriba un algoritmo gcd alternativo, basado en las siguientes observaciones (suponga que  $A > B$ ).
- $\gcd(A, B) = 2 \gcd(A/2, B/2)$  si  $A$  y  $B$  son ambos pares.
  - $\gcd(A, B) = \gcd(A/2, B)$  si  $A$  es par y  $B$  es impar.
  - $\gcd(A, B) = \gcd(A, B/2)$  si  $A$  es impar y  $B$  es par.
- 7.18** El algoritmo de Strassen para la multiplicación de matrices multiplica dos matrices  $N \times N$  realizando siete llamadas recursivas para multiplicar dos matrices  $N/2 \times N/2$ . El coste de procesamiento adicional es de tipo cuadrático. ¿Cuál es el tiempo de ejecución del algoritmo de Strassen?
- 7.19** Resuelva las siguientes recurrencias, que en todos los casos tienen  $T(0) = T(1) = 1$ . Basta con proporcionar una respuesta O mayúscula.
- $T(N) = T(N/2) + \log N$
  - $T(N) = T(N/2) + N \log N$
  - $T(N) = T(N/2) + N^2 \log N$
- 7.20** Resuelva las siguientes recurrencias, que en todos los casos tienen  $T(0) = 1$ . Basta con proporcionar una respuesta O mayúscula.
- $T(N) = T(N-1) + 1$
  - $T(N) = T(N-1) + \log N$
  - $T(N) = T(N-1) + N$

## EN LA PRÁCTICA

- 7.21** El problema de la suma de un subconjunto se define de la forma siguiente: dados  $N$  enteros  $A_1, A_2, \dots, A_N$  y un entero  $K$ , ¿existe algún grupo de enteros que sume exactamente  $K$ ? Proporcione un algoritmo  $O(NK)$  para resolver el problema.
- 7.22** Para el problema del cálculo del número de monedas, proporcione un algoritmo que calcule el número de formas distintas de obtener exactamente  $K$  centavos de cambio.
- 7.23** El método `printInt` mostrado en la Figura 7.4 puede tratar incorrectamente el caso en el que  $N = \text{Long.MIN\_VALUE}$ . Explique por qué y corrija el método.

- 7.24** Una formulación alternativa de la solución del cálculo de la suma máxima de subsecuencia contigua consiste en resolver recursivamente los problemas correspondientes a los elementos situados en las posiciones `low` a `mid-1` y luego los de `mid+1` a `high`. Observe que la posición `mid` no se incluye. Demuestre cómo conduce esto a la obtención de un algoritmo  $O(N \log N)$  para el problema completo e implemente el algoritmo, comparando su velocidad con la del algoritmo del texto.
- 7.25** El algoritmo de la suma máxima de subsecuencia contigua proporcionado en la Figura 7.20 no da ninguna indicación de cuál es la secuencia actual. Modifíquelo para que rellene sendos campos estáticos de clase `seqStart` y `seqEnd`, como en la Sección 5.3.
- 7.26** Escriba un método recursivo que devuelva el número de unos en la representación binaria de  $N$ . Utilice el hecho de que este número es igual al número de unos de la representación binaria de  $N/2$ , más 1, en caso de que  $N$  sea impar.
- 7.27** Implemente recursivamente el algoritmo de búsqueda binaria con una comparación por nivel.
- 7.28** La función de Ackerman se define como sigue.

$$A(m, n) = \begin{cases} n+1 & \text{si } m=0 \\ A(m-1, 1) & \text{si } m>0 \text{ y } n=0 \\ A(m-1, A(m, n-1)) & \text{si } m>0 \text{ y } n>0 \end{cases}$$

Implemente la función de Ackerman.

- 7.29** Escriba la rutina con la declaración

```
public static void permute( String str );
```

que imprime todas las permutaciones de los caracteres que componen la cadena `str`. Si `str` es "abc", entonces las cadenas de salida serán `abc`, `acb`, `bac`, `bca`, `cab` y `cba`. Use la recursión.

- 7.30** Repita el Ejercicio 7.29, pero haga que `permute` devuelva una `List<String>` que contenga todas las posibles permutaciones.
- 7.31** El método `allSums` devuelve una `List<Integer>` que contiene todas las posibles sumas que pueden formarse utilizando como máximo una vez cualquiera de los elementos de la matriz de entrada. Por ejemplo, si la entrada contiene 3, 4, 7, entonces `allSums` devuelve [ 0, 3, 4, 7, 7, 10, 11, 13 ]. Observe que los elementos no tienen por qué devolverse en ningún orden concreto, pero observe también que si una suma se puede producir de múltiples formas, tiene que aparecer múltiples veces. Implemente `allSums` de forma recursiva. Si la entrada tiene  $N$  elementos, ¿cuál es el tamaño de la lista devuelta?
- 7.32** Proporcione un algoritmo  $O(2^N)$  para el problema de la suma de un subconjunto descrito en el Ejercicio 7.21. (*Pista:* utilice la recursión.)
- 7.33** Explique lo que sucede si, en la Figura 7.15, dibujamos el cuadrado central antes de realizar las llamadas recursivas.

- 7.34** Considere el siguiente juego para dos jugadores: se colocan  $N$  monedas  $c_1, c_2, \dots, c_N$  (se puede suponer que  $N$  es par) formando una fila sobre una mesa. Los jugadores van jugando alternativamente, y en cada turno un jugador selecciona la primera o la última moneda de la fila, eliminándola y quedándose con ella. Desarrolle un algoritmo que, dada la matriz de monedas, determine la cantidad máxima de dinero que el jugador #1 puede llegar a ganar.
- 7.35** Una *estrella de Koch* se forma partiendo de un triángulo equilátero y luego alterando recursivamente cada segmento lineal de la siguiente forma:
1. Divida el segmento lineal en tres segmentos de igual longitud.
  2. Dibuje un triángulo equilátero que tenga como base el segmento intermedio del paso 1 y que apunte hacia fuera.
  3. Elimine el segmento lineal que forma la base del triángulo formado en el paso 2.
- Las tres primeras iteraciones de este procedimiento se muestran en la Figura 7.30. Escriba un programa Java para dibujar una estrella de Koch.
- 7.36** El método `printReverse` toma como parámetro un objeto `Scanner`, imprime cada línea del flujo de datos `Scanner` y cierra el `Scanner` cuando ha terminado. Sin embargo, las líneas deben ser impresas en el orden inverso a su aparición. En otras palabras, la última línea se imprime primero, mientras que la primera línea es la última en imprimirse. Implemente `printReverse` sin utilizar ninguna clase de la API de Colecciones y sin usar tampoco ningún contenedor escrito por el usuario. Desarrolle la solución utilizando recursión (de modo que se imprima la primera línea DESPUÉS de imprimir recursivamente las líneas subsiguientes en orden inverso).
- 7.37** La función `findMaxAndMin`, definida a continuación, trata de devolver (en una matriz de longitud 2) los elementos máximo y mínimo (si `arr.length` es 1, el máximo y el mínimo coinciden):

```
// Precondición: arr.length >= 1
// Postcondición: el elemento 0 en el valor de retorno es el máximo
//                 el elemento 1 en el valor de retorno es el mínimo
public static double [ ] findMaxAndMin( double [ ] arr )
```

Escriba una rutina recursiva apropiada de tipo `private static` para implementar la rutina de preparación `public static findMaxAndMin` declarada anteriormente.



Figura 7.30 Las tres primeras iteraciones del proceso de formación de una estrella de Koch.

Su rutina recursiva debe dividir un problema en dos mitades aproximadamente iguales, pero nunca debe dividir en dos problemas de tamaño impar (en otras palabras, un problema de tamaño 10, debe dividirse en 4 y 6, en lugar de en 5 y 5).

- 7.38** a. Diseñe un algoritmo recursivo para encontrar la secuencia creciente de números más larga dentro de una matriz rectangular. Por ejemplo, si la matriz contiene

97 47 56 36

35 57 41 13

89 36 98 75

25 45 26 17

entonces la secuencia creciente de números más larga será la secuencia de longitud ocho formada por los valores 17, 26, 36, 41, 47, 56, 57, 97. Observe que no hay duplicados en la secuencia creciente.

- b. Diseñe un algoritmo que resuelva el mismo problema, pero permitiendo secuencias no decrecientes; eso quiere decir que puede haber duplicados en la secuencia creciente.

- 7.39** Utilice la programación dinámica para resolver el problema de la secuencia creciente más larga del Ejercicio 7.38(a). *Pista:* encuentre la mejor secuencia que comienza en cada elemento de la matriz y, para ello, considere los elementos de la matriz en orden decreciente (de modo que el elemento de la matriz que contiene el valor 98 sería el que se analizaría en primer lugar).

## PROYECTOS DE PROGRAMACIÓN

- 7.40** Los coeficientes binomiales  $C(N, k)$  se pueden definir recursivamente como  $C(N, 0) = 1$ ,  $C(N, N) = 1$  y, para  $0 < k < N$ ,  $C(N, k) = C(N - 1, k) + C(N - 1, k - 1)$ . Escriba un método y haga un análisis del tiempo de ejecución necesario para calcular los coeficientes binomiales

- Recursivamente.
- Utilizando programación dinámica.

- 7.41** Escriba una rutina `getAllWords` que tome como parámetro una palabra y devuelva un conjunto `Set` que contenga todas las subcadenas de esa palabra. Las subcadenas no tienen que ser palabras reales ni contiguas, pero las letras que formen las subcadenas deben conservar el mismo orden que tenían en la palabra. Por ejemplo, si la palabra es `cabb`, las palabras devueltas por `getAllWords` podrían ser [ "", "b", "bb", "a", "ab", "abb", "c", "cb", "cbb", "ca", "cab", "cabb" ].

- 7.42** Suponga que tenemos un archivo de datos que contiene líneas compuestas por un único entero o por el nombre de un archivo que contiene más líneas. Observe que un archivo de datos puede hacer referencia a varios otros archivos y que esos otros archivos referenciados pueden a su vez contener el nombre de archivos adicionales, etc. Escriba un método que lea un archivo especificado y devuelva la suma de los enteros contenidos en el archivo y de todos los archivos referenciados. Puede suponer que no se hace referencia a ningún archivo más de una vez.

**7.43** Repita el Ejercicio 7.42, pero añada código que detecte si se está haciendo referencia a un archivo más de una vez. Si se detecta tal situación, las referencias adicionales se ignoran.

**7.44** El método `reverse` mostrado a continuación devuelve la inversa de una cadena de caracteres `String`.

```
String reverse( String str )
```

- Implemente `reverse` recursivamente. No se preocupe por la inefficiencia de la operación de concatenación de cadenas.
- Implemente `reverse` haciendo que `reverse` sea la rutina de preparación para una rutina recursiva privada. `reverse` creará un `StringBuffer` y se lo pasará a la rutina recursiva.

**7.45** El Ejercicio 7.31 describe un método que devuelve todas las sumas que se pueden formar a partir de una colección de elementos. Implemente el método `getOriginalItems`, que toma como parámetro una lista `List` que representa todas las sumas y devuelve la entrada original. Por ejemplo, si el parámetro de `getOriginalItems` es `[ 0, 3, 4, 7, 7, 10, 11, 13 ]`, entonces el valor de retorno será una lista que contenga `[ 3, 4, 7 ]`. Puede asumir que no hay ningún elemento negativo en la salida (y por tanto, tampoco en la entrada).

**7.46** Añada un método `divide` a la clase `Polynomial` del Ejercicio 3.31. Implemente `divide` utilizando recursión.

**7.47** Repita el Ejercicio 7.45, tratando el caso en el que haya elementos negativos. Esta versión del problema es mucho más complicada.

**7.48** Escriba un programa que amplíe (recursivamente) las directivas `#include` de un archivo C++. Hágalo sustituyendo las líneas de la forma

```
#include "filename"
```

por el contenido del archivo `filename`.

**7.49** Implemente el criptosistema RSA con la clase de librería `BigInteger`.

**7.50** Mejore la clase `TicTacToe` haciendo que las rutinas de soporte sean más eficientes.

**7.51** Considere una cuadrícula  $N \times N$  en la que algunos cuadrados están ocupados. Dos cuadrados pertenecen al mismo grupo si comparten una arista común. En la Figura 7.31, hay un grupo de cuatro cuadrados ocupados, tres grupos de dos cuadrados ocupados y dos cuadrados ocupados individuales. Suponga que la cuadrícula está representada por una matriz bidimensional. Escriba un programa que:

- Calcule el tamaño de un grupo cuando se especifique uno de los cuadrados del grupo.
- Calcule el número de grupos diferentes que hay.
- Imprima todos los grupos.

**7.52** Sea  $A$  una secuencia de  $N$  números distintos  $A_1, A_2, \dots, A_N$  ordenados, con  $A_1 = 0$ . Sea  $B$  una secuencia de  $N(N - 1)/2$  números definida por  $B_{ij} = A_j - A_i$  ( $i < j$ ). Sea  $D$  la secuencia obtenida ordenando  $B$ . Tanto  $B$  como  $D$  pueden contener duplicados.

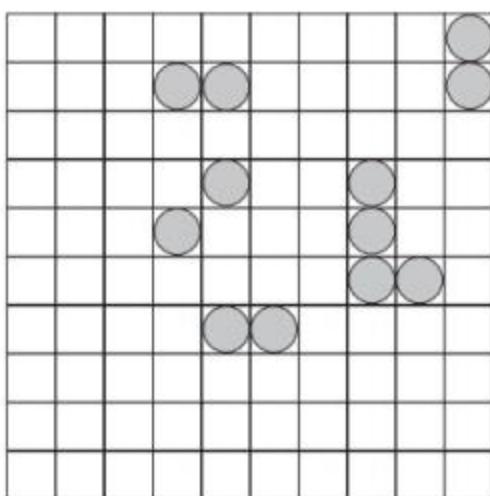


Figura 7.31 Cuadricula para el Ejercicio 7.51.

Ejemplo:  $A = 0, 1, 5, 8$ . Entonces  $D = 1, 3, 4, 5, 7, 8$ . Haga lo siguiente.

- Escriba un programa que construya  $D$  a partir de  $A$ . Esta parte es sencilla.
- Escriba un programa que construya alguna secuencia  $A$  que se corresponda con  $D$ . Observe que  $A$  no tiene por qué ser única. Utilice un algoritmo de retroceso.



## Referencias

Buena parte de este capítulo está basada en las explicaciones de [3]. En [1] se presenta una descripción del algoritmo RSA, junto con una demostración de corrección; esta misma referencia proporciona un capítulo dedicado a la programación dinámica. Los ejemplos de dibujo de formas están adaptados de [2].

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to Algorithms* 3<sup>a</sup> ed., MIT Press, Cambridge, MA, 2010.
2. R. Sedgewick, *Algorithms in Java*, Partes 1–4, 3<sup>a</sup> ed., Addison-Wesley, Reading, MA, 2003.
3. M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice Hall, Upper Saddle River, NJ, 1995.

# Algoritmos de ordenación

La ordenación es una aplicación fundamental dentro del campo de la informática. Buena parte de la salida que cualquier procesamiento genera suele estar ordenada de una u otra forma, y muchos tipos de procesamiento pueden incrementar su eficiencia invocando internamente un proceso de ordenación. Por tanto, la ordenación es quizás la operación más importante y más ampliamente estudiada dentro de la Ciencias de la computación.

En este capítulo, vamos a hablar del problema de ordenar una matriz de elementos. Describiremos y analizaremos los distintos algoritmos de ordenación. Las ordenaciones de este capítulo pueden realizarse enteramente en memoria principal, por lo que el número de elementos es relativamente pequeño (inferior a unos cuantos millones). Las ordenaciones que no se pueden realizar en memoria principal y deben llevarse a cabo en disco o cinta también tienen una gran importancia. Hablaremos de este tipo de ordenación, denominado *ordenación externa*, en la Sección 21.6.

Este análisis de la ordenación es una mezcla de teoría y de práctica. Presentaremos varios algoritmos que tienen un rendimiento distinto, y mostraremos cómo el análisis de las propiedades de rendimiento de un algoritmo nos puede servir de ayuda a la hora de tomar decisiones de implementación que no resultan inmediatamente obvias.

En este capítulo, veremos

- Que las ordenaciones simples se ejecutan en un tiempo cuadrático.
- Cómo codificar la ordenación *Shell*, que es un algoritmo simple y eficiente que se ejecuta en un tiempo subcuadrático.
- Cómo escribir los algoritmos  $O(N \log N)$  ligeramente más complicados, denominados ordenación por mezcla (*mergesort*) y ordenación rápida (*quicksort*).
- Que para cualquier algoritmo de ordenación de propósito general se requieren  $\Omega(N \log N)$  comparaciones.

## 8.1 ¿Por qué es importante la ordenación?

Recuerde de la Sección 5.6 que realizar una búsqueda en una matriz ordenada es mucho más fácil que en una matriz desordenada. Esto es especialmente cierto en el caso de las personas. Es decir, encontrar el nombre de una persona en una guía telefónica es sencillo, por ejemplo, pero encontrar un número telefónico sin conocer el nombre de la persona es prácticamente imposible. Como

resultado, cualquier salida de un cierto tamaño generada por una computadora suele estar ordenada para poder ser interpretada. A continuación se indican algunos otros ejemplos.

- Las palabras de un diccionario están ordenadas (y se ignora la distinción entre mayúsculas y minúsculas).
- Los archivos de un directorio suelen mostrarse con algo tipo de ordenación.
- El índice analítico de un libro se muestra ordenado (ignorándose la distinción entre mayúsculas y minúsculas).
- Las fichas de los libros de una biblioteca están ordenadas tanto por autor como por título.
- La lista de cursos disponibles en una universidad está ordenada primero por departamento y luego por número de curso.
- Muchos bancos proporcionan talonarios cuyos cheques se numeran en orden creciente.
- En un periódico, los calendarios de eventos suelen estar ordenados por fecha.
- Los discos musicales en una tienda de discos suelen estar ordenados según el artista.
- En los programas impresos para las ceremonias de graduación, los departamentos se citan en orden y luego se enumeran en orden los estudiantes de esos departamentos.

Una ordenación inicial de los datos puede incrementar significativamente el rendimiento de un algoritmo.

No es sorprendente que buena parte del trabajo en el campo de la computación implique de un modo u otro tareas de ordenación. Sin embargo, la ordenación también tiene algunos usos indirectos. Por ejemplo, suponga que queremos determinar si alguna matriz almacena algún duplicado. La Figura 8.1 muestra un método simple que requiere un tiempo cuadrático de

caso peor. La ordenación proporciona un algoritmo alternativo. Es decir, si ordenamos una copia de la matriz, entonces todos los duplicados serán adyacentes entre sí y podrán detectarse mediante una única exploración de tiempo lineal de la matriz. El coste de este algoritmo está dominado por el tiempo necesario para realizar la ordenación, así que si podemos llevarla a cabo en un tiempo subcuadrático dispondremos de un algoritmo mejorado. El rendimiento de muchos algoritmos aumenta significativamente cuando ordenamos inicialmente los datos.

```

1 // Devolver true si la matriz tiene algún duplicado y false en otro caso.
2 public static boolean duplicates( Object [ ] a )
3 {
4     for( int i = 0; i < a.length; i++ )
5         for( int j = i + 1; j < a.length; j++ )
6             if( a[ i ].equals( a[ j ] ) )
7                 return true; // Duplicado encontrado
8
9     return false; // No se encontraron duplicados
10 }
```

Figura 8.1 Un algoritmo cuadrático simple para la detección de duplicados.

La amplia mayoría de los proyectos de programación de una cierta envergadura utiliza ordenaciones en un sitio u otro, y en muchos casos el coste de la ordenación determina el tiempo de ejecución. Por tanto, es necesario ser capaces de implementar un método de ordenación que sea rápido y fiable.

## 8.2 Preliminares

Los algoritmos que describimos en este capítulo son todos intercambiables. A cada uno de ellos se le pasa una matriz que contiene los elementos y solo pueden ordenarse objetos que implementen la interfaz `Comparable`.

Las comparaciones son las únicas operaciones permitidas con los datos de entrada. Un algoritmo que toma decisiones de ordenación basándose exclusivamente en las comparaciones se denomina *algoritmo de ordenación basado en comparaciones*.<sup>1</sup> En este capítulo,  $N$  es el número de elementos que se quiere ordenar.

Un algoritmo de ordenación basado en comparaciones toma decisiones de ordenación basándose exclusivamente en comparaciones entre elementos.

## 8.3 Análisis de la ordenación por inserción y de otras ordenaciones simples

La *ordenación por inserción* es un algoritmo de ordenación simple apropiado para pequeños tamaños de entrada. Generalmente, se considera una buena solución si solo hay unos pocos elementos que necesiten ser ordenados, dado que es un algoritmo muy corto y que el tiempo requerido para realizar la ordenación no suele ser un problema. Sin embargo, si estamos tratando con una gran cantidad de datos de entrada, la ordenación por inserción no es una elección correcta porque requiere demasiado tiempo. El código se muestra en la Figura 8.2.

Una ordenación por inserción es cuadrática en los casos peor y promedio. Resulta rápida si la entrada ya ha sido ordenada previamente.

La ordenación por inserción funciona de la forma siguiente. En el estado inicial, el primer elemento, considerado aisladamente, está ordenado. En el estado final, todos los elementos (suponiendo que hay  $N$ ), considerados como un grupo, tienen que estar ordenados. La Figura 8.3 muestra que la acción básica de la ordenación por inserción consiste en ordenar los elementos situados en las posiciones 0 a  $p$  (donde  $p$  va de 1 a  $N - 1$ ). En cada etapa,  $p$  se incrementa en 1. Esto es lo que se encarga de controlar el bucle de la línea 7 de la Figura 8.2.

Cuando entramos en el cuerpo del bucle `for` de la línea 12, se garantiza que todos los elementos de las posiciones 0 a  $p-1$  de la matriz ya han sido ordenados, y que por tanto lo que necesitamos es ampliar la ordenación a las posiciones 0 a  $p$ . La Figura 8.4 nos proporciona una imagen más detallada de lo que hay que hacer, en la que solo se muestra la parte relevante de la matriz. En cada paso, el elemento en negrita tiene que ser añadido a la parte previamente ordenada de la matriz. Podemos hacer eso fácil colocándolo en una variable temporal y desplazando todos los elementos mayores que él una posición a la derecha. Despues, podemos copiar la variable temporal en la

<sup>1</sup> Como se muestra en la Sección 4.8, cambiar la interfaz de ordenación exigiendo un objeto función `Comparator` es muy sencillo.

```

1  /**
2  * Ordenación por inserción simple
3  */
4  public static <AnyType extends Comparable<? super AnyType>>
5  void insertionSort( AnyType [ ] a )
6  {
7      for( int p = 1; p < a.length; p++ )
8      {
9          AnyType tmp = a[ p ];
10         int j = p;
11
12         for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
13             a[ j ] = a[ j - 1 ];
14         a[ j ] = tmp;
15     }
16 }

```

Figura 8.2 Implementación de la ordenación por inserción.

posición anterior del elemento recolocado situado más a la izquierda (indicado mediante un sombreado más claro en la siguiente línea). Mantenemos un contador  $j$ , que es la posición en la que habrá que escribir luego el contenido de la variable temporal. Cada vez que se desplaza un elemento,  $j$  se reduce en una unidad. Las líneas 9-14 implementan este proceso.

Debido a los bucles anidados, cada uno de los cuales tiene  $N$  iteraciones, el algoritmo de ordenación por inserción es  $O(N^2)$ . Además, esta cota puede llegar a alcanzarse, porque si la entrada está ordenada a la inversa, finalmente hará falta un tiempo realmente cuadrático. Un cálculo preciso demuestra que las comprobaciones de la línea 12 de la Figura 8.2 pueden ejecutarse como máximo  $P + 1$  veces para cada valor de  $P$ . Sumando para todo  $P$  se obtiene un tiempo total de

$$\sum_{p=1}^{N-1} (P+1) = \sum_{i=2}^N i = 2 + 3 + 4 + \dots + N = \Theta(N^2)$$

Posición de la matriz	0	1	2	3	4	5
Estado inicial	8	5	9	2	6	3
Después de ordenar a[0..1]	5	8	9	2	6	3
Después de ordenar a[0..2]	5	8	9	2	6	3
Después de ordenar a[0..3]	2	5	8	9	6	3
Después de ordenar a[0..4]	2	5	6	8	9	3
Después de ordenar a[0..5]	2	3	5	6	8	9

Figura 8.3 Acción básica del algoritmo de ordenación por inserción (la parte sombreada está ordenada).

Posición de la matriz	0	1	2	3	4	5
Estado inicial	8	5				
Después de ordenar $a[0..1]$	5	8	9			
Después de ordenar $a[0..2]$	5	8	9	2		
Después de ordenar $a[0..3]$	2	5	8	9	6	
Después de ordenar $a[0..4]$	2	5	6	8	9	3
Después de ordenar $a[0..5]$	2	3	5	6	8	9

Figura 8.4 Un examen más detallado de la acción básica del algoritmo de ordenación por inserción (el sombreado oscuro indica la parte ordenada, el sombreado claro indica dónde se ha colocado el nuevo elemento).

Sin embargo, si la entrada ha sido ordenada previamente, el tiempo de ejecución es  $O(N)$  porque la comprobación al principio del bucle `for` interno siempre fallará de manera inmediata. De hecho, si la entrada está casi ordenada (en seguida definiremos de forma más rigurosa la expresión casi ordenada), la ordenación por inserción se ejecuta con rapidez. Por tanto, el tiempo de ejecución depende no solo de la cantidad de datos de entrada, sino también de la ordenación específica de esa entrada. Debido a esta amplia variación, merece la pena analizar el comportamiento promedio de este algoritmo. El caso promedio resulta ser  $\theta(N^2)$  para la ordenación por inserción, así como para diversos otros algoritmos simples de ordenación.

La ordenación por inserción es cuadrática en los casos peor y promedio. Es rápida si la entrada ya ha sido ordenada.

Una *inversión* es un par de elementos que están desordenados en una matriz. En otras palabras, es cualquier par ordenado  $(i, j)$  que tiene la propiedad de que  $i < j$  pero  $A_i > A_j$ . Por ejemplo, la secuencia  $\{8, 5, 9, 2, 6, 3\}$  tiene 10 inversiones, que se corresponden con los pares  $(8, 5)$ ,  $(8, 2)$ ,  $(8, 6)$ ,  $(8, 3)$ ,  $(5, 2)$ ,  $(5, 3)$ ,  $(9, 2)$ ,  $(9, 6)$ ,  $(9, 3)$  y  $(6, 3)$ . Observe que el número de inversiones es igual al número total de veces que se ejecuta la línea 13 en la Figura 8.2. Esta condición siempre es cierta, porque el efecto de la instrucción de asignación consiste en intercambiar los dos elementos  $a[j]$  y  $a[j-1]$ . (Evitamos tener que realizar en la práctica un número excesivo de intercambios utilizando la variable temporal, pero de todos modos se sigue realizando un intercambio abstracto.) Intercambiar dos elementos que están desordenados hace que se reduzca exactamente en una unidad el número de inversiones, y una matriz ordenada no tiene ninguna inversión. Por tanto, si hay  $I$  inversiones al principio del algoritmo, deberá haber  $I$  intercambios implícitos. Puesto que el algoritmo implica un coste  $O(N)$  adicional, el tiempo de ejecución de la ordenación por inserción será  $O(I + N)$ , donde  $I$  es el número de inversiones en la matriz original. Por tanto, la ordenación por inserción se ejecuta en un tiempo lineal si el número de inversiones es  $O(N)$ .

Una *inversión* mide el grado de desorden.

Podemos calcular cotas precisas para el tiempo de ejecución promedio de la ordenación por inserción, determinando el número promedio de inversiones de una matriz. Sin embargo, definir aquí promedio es difícil. Podemos suponer que no hay elementos duplicados (si permitimos duplicados, ni siquiera está claro cuál es el número promedio de duplicados existentes). También podemos asumir que la entrada consiste en una cierta disposición secuencial de los  $N$  primeros enteros (ya que lo único importante es la ordenación relativa); estas disposiciones secuenciales

se denominan *permutaciones*. También podemos suponer que todas estas permutaciones son igualmente probables. Con todas estas suposiciones, podemos enunciar el Teorema 8.1.

### Teorema 8.1

El número medio de inversiones en una matriz de  $N$  números distintos es  $N(N - 1)/4$ .

#### Demostración

Para cualquier matriz  $A$  de números, considere  $A_i$ , que es la matriz en orden inverso. Por ejemplo, el inverso de la matriz  $1, 5, 4, 2, 6, 3$  es  $3, 6, 2, 4, 5, 1$ . Consideré cualesquiera dos números  $(x, y)$  de la matriz, con  $y > x$ . Este par ordenado representará una inversión o bien en la matriz  $A$  o bien en la matriz  $A_i$ . El número total de estos pares en una matriz  $A$  y en su inversa  $A_i$  es  $N(N - 1)/2$ . Por tanto, una matriz promedio tendrá la mitad de este número, es decir,  $N(N - 1)/4$  inversiones.

El Teorema 8.1 implica que la ordenación por inserción es cuadrática en el caso promedio. También se puede utilizar para proporcionar una cota inferior muy fuerte para cualquier algoritmo que solo intercambie elementos adyacentes. Esta cota inferior se enuncia en el Teorema 8.2.

### Teorema 8.2

Cualquier algoritmo que realice una ordenación intercambiando elementos adyacentes requiere como promedio un tiempo  $\Omega(N^2)$ .

#### Demostración

El número medio de inversiones es inicialmente  $N(N - 1)/4$ . Cada intercambio elimina una única inversión, por lo que hacen falta  $\Omega(N^2)$  intercambios.

La demostración de cota inferior muestra que el rendimiento cuadrático es inherente a cualquier algoritmo que lleve a cabo la ordenación haciendo comparaciones de elementos adyacentes.

Esta demostración es un ejemplo de *demostración de cota inferior*. Es válida no solo para la ordenación por inserción, que realiza intercambios de elementos adyacentes implícitamente, sino también para otros algoritmos simples como la ordenación por burbuja y la ordenación por selección, que no vamos a describir aquí. De hecho, es válido para toda una *clase* de algoritmos, incluyendo algunos todavía no descubiertos, que realizan únicamente intercambios de elementos adyacentes.

Lamentablemente, cualquier confirmación computacional de una demostración aplicable a toda una clase de algoritmos requeriría ejecutar todos los algoritmos de la clase. Esto es imposible, porque hay un número infinito de potenciales algoritmos. Por tanto, cualquier intento de confirmación solo sería aplicable a los algoritmos que se ejecuten. Esta restricción hace que la confirmación de la validez de las demostraciones de cota inferior sean más difíciles que la usual demostración de las cotas superiores de algoritmos a las que estamos acostumbrados. Cualquier procesamiento solo podría demostrar la *falsedad* de una conjectura relativa a una cota inferior, nunca puede demostrarla con carácter general.

Aunque esta demostración de cota inferior es bastante simple, demostrar cuáles son las cotas inferiores es, en general, mucho más complicado que demostrar las cotas superiores. Los argumentos relativos a las cotas inferiores son mucho más abstractos que los correspondientes para las cotas superiores.

Esta cota inferior demuestra que, para que un algoritmo de ordenación pueda ejecutarse en un tiempo subcuadrático o  $o(N^2)$ , debe hacer comparaciones y en particular intercambios entre elementos que estén situados a una cierta distancia. Todo algoritmo de ordenación progres

eliminando inversiones. Para poder ejecutarse de forma eficiente, debe eliminar más de una inversión por cada intercambio.

## 8.4 Ordenación Shell

El primer algoritmo para mejorar sustancialmente la ordenación por inserción fue la ordenación Shell, descubierta en 1959 por Donald Shell. Aunque no es el algoritmo más rápido conocido, la ordenación Shell es un algoritmo subcuadrático cuyo código es solo ligeramente más largo que el de la ordenación por inserción, lo que hace de él el más simple de los algoritmos rápidos.

La idea de Shell era evitar la gran cantidad de desplazamientos de datos, primero comparando elementos situados a una cierta distancia entre sí y luego comparando elementos que estuvieran más próximos, etc., contrayendo gradualmente las operaciones hacia las correspondientes a la ordenación por inserción básica. La ordenación Shell utiliza una secuencia  $h_1, h_2, \dots, h_k$  denominada *secuencia de incrementos*. Cualquier secuencia de incrementos vale, siempre y cuando  $h_1 = 1$ , pero algunas elecciones son mejores que otras. Después de una fase en la que utilizamos un cierto incremento  $h_k$ , tendremos que  $a[i] \leq a[i + h_k]$  para todo  $i$  para el que  $i + h_k$  sea un índice válido; todos los elementos separados una distancia  $h_k$  estarán ordenados. Decimos entonces que la matriz tiene una ordenación de nivel  $h_k$ .

Por ejemplo, la Figura 8.5 muestra una matriz después de varias fases de ordenación Shell. Después de una ordenación de nivel 5, se garantiza que los elementos situados a cinco unidades de distancia estarán en un orden correcto. En la figura, los elementos que están separados cinco unidades tienen un sombreado idéntico, y están ordenados unos respecto de otros. De forma similar, después una ordenación de nivel 3, los elementos situados a tres elementos de distancia estarán ordenados de forma garantizada en relación unos con otros. Una importante propiedad de la ordenación Shell (que enunciaremos sin demostrarla) es que una matriz con ordenación de nivel  $h_k$  que luego se someta a una ordenación de nivel  $h_k - 1$  continuará teniendo una ordenación de nivel  $h_k$ . Si esto no fuera así, el algoritmo probablemente no tendría ninguna utilidad, porque el trabajo realizado en las fases anteriores sería deshecho por las fases siguientes.

En general, una ordenación de nivel  $h_k$  requiere que, para cada posición  $i$  de  $h_k, h_{k+1}, \dots, N - 1$ , coloquemos el elemento en el punto correcto de las posiciones  $i, i - h_k, i - 2h_k$ , etc. Aunque este orden no afecta a la implementación, un cuidadoso examen muestra que una ordenación de nivel  $h_k$  realiza una ordenación por inserción sobre  $h_k$  matrices independientes (mostradas con tonos de sombreado distintos en la Figura 8.5). Por tanto, no resulta sorprendente que, en la Figura 8.7, de la que hablaremos en breve, las líneas 9 a 17 representen una *ordenación por inserción con espaciado*. En una ordenación por inserción con espaciado, después de haber ejecutado el bucle, los elementos que estén separados por una distancia *gap* en la matriz estarán ordenados. Por ejemplo, cuando *gap* es 1, el bucle es idéntico, instrucción por instrucción, a una ordenación por inserción. Es por esto que a la ordenación Shell se la conoce también con el nombre de *ordenación con espaciado decreciente*.

Como hemos visto, cuando *gap* es 1, se garantiza que el bucle interno ordenará la matriz *a*. Si *gap* nunca es 1, siempre habrá alguna entrada para la

La ordenación Shell es un algoritmo subcuadrático que funciona bien en la práctica y es simple de codificar. El rendimiento de la ordenación Shell es altamente dependiente de la secuencia de incrementos y requiere un complicado análisis (que no está completamente resuelto).

La ordenación Shell también se denomina *ordenación con espaciado decreciente*.

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
Después de la ordenación de nivel 5	35	17	11	28	12	41	75	15	96	58	81	94	95
Después de la ordenación de nivel 3	28	12	11	35	15	41	58	17	94	75	81	96	95
Después de la ordenación de nivel 1	11	12	15	17	28	35	41	58	75	81	94	95	96

Figura 8.5 Ordenación Shell después de cada pasada si la secuencia de incrementos es {1, 3, 5}.

La secuencia de incrementos de Shell representa una mejora con respecto a la ordenación por inserción (aunque se conocen otras secuencias mejores).

que la matriz no podrá ser ordenada. Por tanto, la ordenación Shell será capaz de ordenar la matriz si gap termina en algún momento por tener el valor 1. La única cuestión que queda es entonces cómo seleccionar la secuencia de incrementos.

Shell sugirió comenzar con un valor para gap igual a  $N/2$  e ir dividiendo a la mitad hasta alcanzar el valor 1, después de lo cual el programa puede terminar. Utilizando estos incrementos, la ordenación Shell representa una

mejora sustancial con respecto a la ordenación por inserción, a pesar del hecho de que anida tres bucles `for` en lugar de dos, lo que suele ser ineficiente. Alterando la secuencia de espaciados, podemos mejorar aun más el rendimiento del algoritmo. En la Figura 8.6 se muestra un resumen del rendimiento de la ordenación Shell con tres opciones diferentes de secuencias de incrementos.

#### 8.4.1 Rendimiento de la ordenación Shell

El tiempo de ejecución de la ordenación Shell depende fuertemente de la elección de la secuencia de incrementos, y las correspondientes demostraciones pueden ser bastante complicadas. El análisis de caso promedio de la ordenación Shell constituye un problema aun abierto, después de tanto tiempo, salvo por lo que respecta a las secuencias de incrementos más triviales.

Cuando se utilizan los incrementos de Shell, el caso peor es  $O(N^2)$ . Esta cota es alcanzable si  $N$  es exactamente una potencia de 2, si todos los elementos de gran valor están en posiciones con un índice par de la matriz y si todos los elementos pequeños están en posiciones con un índice impar de

$N$	Ordenación por inserción	Ordenación Shell		
		Incrementos de Shell	Solo espaciados impares	División por 2,2
10.000	575	10	11	9
20.000	2.489	23	23	20
40.000	10.635	51	49	41
80.000	42.818	114	105	86
160.000	174.333	270	233	194
320.000	No disponible	665	530	451
640.000	No disponible	1.593	1.161	939

Figura 8.6 Tiempos de ejecución de la ordenación por inserción y de la ordenación Shell para distintas secuencias de incrementos.

la matriz. Al alcanzar la pasada final, todos los elementos de valor grande seguirán estando en las posiciones con índice par de la matriz, y todos los elementos de pequeño valor seguirán estando en las posiciones con índice impar de la matriz. Un cálculo del número de inversiones restantes muestra que la pasada final requerirá un tiempo cuadrático. El hecho de que este sea el caso peor que puede presentarse se deduce, a su vez, del hecho de que una ordenación de nivel  $h_k$  está compuesta por  $h_k$  ordenaciones por inserción de aproximadamente  $N/h_k$  elementos. Por tanto, el coste de cada pasada es  $O(hk(Nh_k)^2)$  o  $O(N^2/h_k)$ . Cuando sumamos este coste para todas las pasadas obtenemos  $O(N^2 \sum 1/h_k)$ . Los incrementos constituyen aproximadamente una serie geométrica, por lo que la suma está acotada por una constante. El resultado es un tiempo de ejecución cuadrático para el caso peor. También podemos demostrar mediante un argumento complejo que, cuando  $N$  es una potencia exacta de 2, el tiempo de ejecución promedio es  $O(N^{3/2})$ . Por tanto, como promedio, los incrementos de Shell proporcionan una mejora significativa respecto a la ordenación por inserción.

Un pequeño cambio en la secuencia de incrementos puede impedir que se produzca el caso peor con tiempo cuadrático. Si dividimos `gap` entre 2 y pasa a ser par, podemos sumarle 1 para hacerle par. Podemos entonces demostrar que el caso peor no es cuadrático sino solo  $O(N^{3/2})$ . Aunque la demostración es complicada, la base de la misma reside en que, con esta nueva secuencia de incrementos, los incrementos consecutivos no comparten ningún factor común (mientras que en la secuencia de incrementos de Shell sí que lo hacen). Cualquier secuencia que satisfaga esta propiedad (y cuyos incrementos se reduzcan de forma aproximadamente geométrica) tendrá un tiempo de ejecución de caso peor de como máximo  $O(N^{3/2})$ .<sup>2</sup> La tasa media del algoritmo con estos nuevos incrementos es desconocido, pero parece ser  $O(N^{5/4})$ , basándose en las simulaciones.

Una tercera secuencia, que funciona muy bien en la práctica pero para la que no existe base teórica, consiste en dividir por 2,2 en lugar de por 2. Este divisor parece conseguir un tiempo de ejecución promedio por debajo de  $O(N^{5/4})$  –quizá  $O(N^{7/8})$ – pero este caso no está resuelto en absoluto. Para entre 100.000 y 1.000.000 de elementos, suele mejorar el rendimiento entre un 25 y 35 por ciento aproximadamente respecto a los incrementos de Shell, aunque nadie sabe por qué. En la Figura 8.7 se muestra una implementación de la ordenación Shell con esta secuencia de incrementos. El complicado código de la línea 8 es necesario para evitar que `gap` pueda tener el valor 0. Si eso sucediera, el algoritmo dejaría de funcionar, porque nunca habría una ordenación de nivel 1. La línea 8 garantiza que se reinicialice `gap` con el valor 1, en caso de que vaya a asignársele el valor 0.

Las entradas de la Figura 8.6 comparan el rendimiento de la ordenación por inserción y de la ordenación Shell, con diversas secuencias de espaciado. Podríamos concluir fácilmente que la ordenación Shell, incluso con la secuencia de espaciado más simple, proporciona una significativa mejora con respecto a la ordenación por inserción, teniendo que pagar por ello el coste de una pequeña complejidad añadida en el código. Un sencillo cambio en la secuencia de incrementos puede mejorar aun más la velocidad. Se pueden hacer más mejoras (véase el Ejercicio 8.26).

En el caso peor, los incrementos de Shell nos dan un comportamiento cuadrático.

Si los incrementos consecutivos son relativamente primos se mejora el rendimiento de la ordenación Shell.

Dividir por 2,2 proporciona un excelente rendimiento en la práctica.

<sup>2</sup> Para darse cuenta de la sutileza del asunto, observe que restar 1 en lugar de sumar 1 no funciona. Por ejemplo, si  $N$  es 186, la secuencia resultante es 93, 45, 21, 9, 3, 1, en la que todos los números comparten el factor común 3.

```

1  /**
2   * Ordenación Shell, utilizando una secuencia sugerida por Gonnet.
3   */
4  public static <AnyType extends Comparable<? super AnyType>>
5  void shellsort( AnyType [ ] a )
6  {
7      for( int gap = a.length / 2; gap > 0;
8          gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) )
9      for( int i = gap; i < a.length; i++ )
10     {
11         AnyType tmp = a[ i ];
12         int j = i;
13
14         for( ; j >= gap && tmp.compareTo( a[j-gap] ) < 0; j -= gap )
15             a[ j ] = a[ j - gap ];
16             a[ j ] = tmp;
17     }
18 }

```

Figura 8.7 Implementación de la ordenación Shell.

Algunas de estas mejoras tienen soporte teórico, pero ninguna secuencia conocida mejora el programa mostrado en la Figura 8.7.

La ordenación Shell es una buena elección para tamaños de entrada moderados.

El rendimiento de la ordenación Shell es bastante aceptable en la práctica, incluso para valores de  $N$  del orden de las decenas de miles. La simplicidad del código hace que sea el algoritmo preferido para ordenar entradas de tamaño moderadamente grande. También constituye un bonito ejemplo de un algoritmo muy simple cuyo análisis es extremadamente complejo.

## 8.5 Ordenación por mezcla

La ordenación por mezcla utiliza la técnica divide y vencerás para obtener un tiempo de ejecución  $O(N \log N)$ .

Recuerde de la Sección 7.5 que se puede utilizar la recursión para desarrollar algoritmos subcuadráticos. Específicamente, un algoritmo de tipo divide y vencerás en el que se resuelvan recursivamente dos problemas de tamaño mitad con un coste adicional  $O(N)$ , da como resultado un algoritmo  $O(N \log N)$ . Uno de esos algoritmos es la *ordenación por mezcla (mergesort)*. Ofrece una cota mejor, al menos teóricamente, que la de la ordenación Shell.

El algoritmo de ordenación por mezcla está compuesto de tres etapas:

1. Si el número de elementos que hay que ordenar es 0 o 1, volver.
2. Ordenar recursivamente y por separado la primera y la segunda mitades.
3. Mezclar las dos mitades ordenadas para obtener un total ordenado.

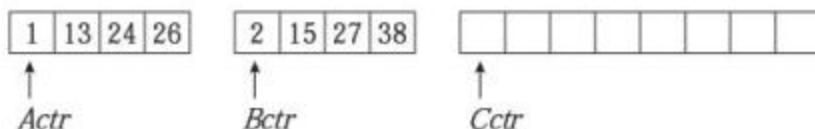
Para demostrar que se trata de un algoritmo  $O(N \log N)$ , solo tenemos que verificar que la mezcla de dos grupos ordenados puede llevarse a cabo en un tiempo lineal. En esta sección vamos a ver cómo mezclar dos matrices de entrada  $A$  y  $B$ , almacenando el resultado en una tercera matriz,  $C$ . Despues proporcionaremos una implementación simple del algoritmo de ordenación por mezcla. La rutina de mezcla es la piedra angular de la mayoría de los algoritmos de ordenación externa, como se demuestra en la Sección 21.6.

La mezcla de matrices ordenadas puede llevarse a cabo en un tiempo lineal.

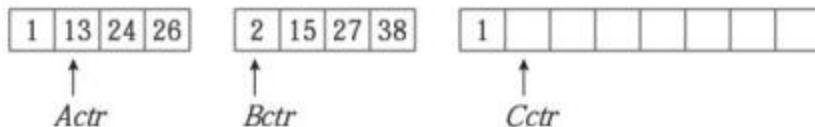
### 8.5.1 Mezcla en tiempo lineal de matrices ordenadas

El algoritmo básico de mezcla toma dos matrices de entrada,  $A$  y  $B$ , una matriz de salida,  $C$ , y tres contadores,  $Actr$ ,  $Bctr$  y  $Cctr$ , que inicialmente se configuran para apuntar al principio de sus respectivas matrices. Se copia entonces el menor de los dos valores  $A[Actr]$  y  $B[Bctr]$  en la siguiente entrada de  $C$  y se hace avanzar a los contadores apropiados. Cuando se agotan los datos de alguna de las dos matrices de entrada, se copia en  $C$  el resto de la otra matriz.

Veamos un ejemplo de cómo funcionaría la rutina de mezcla para los siguientes datos de entrada:



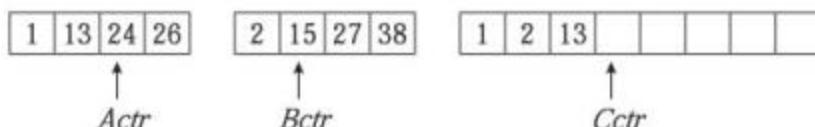
Si la matriz  $A$  contiene 1, 13, 24, 26 y  $B$  contiene 2, 15, 27, 38, el algoritmo se ejecuta de la forma siguiente. En primer lugar, se realiza una comparación entre 1 y 2, se añade 1 a  $C$  y se comparan 13 y 2.



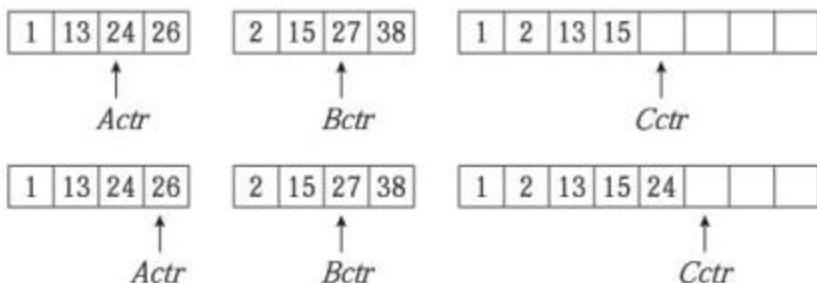
Entonces se añade 2 a  $C$  y se comparan 13 y 15:



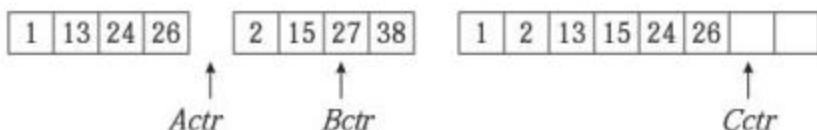
A continuación se añade 13 a  $C$  y se comparan 24 y 15:



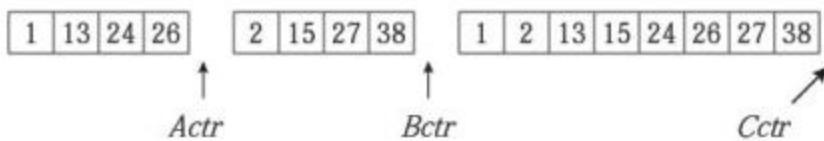
El proceso continúa hasta que se comparan 26 y 27:



Después se añade 26 a *C*, con lo que la matriz *A* se agota:



Finalmente, se copia en *C* el resto de la matriz *B*.



El tiempo necesario para mezclar dos matrices ordenadas es lineal, porque cada comparación hace que se incremente *Cctr* (limitando por tanto el número de comparaciones). Como resultado, un algoritmo de tipo divide y vencerás que utilice un procedimiento de mezcla lineal se ejecutará en un tiempo  $O(N \log N)$  de caso peor. Este tiempo de ejecución también representa los tiempos de caso promedio y de caso mejor, porque el paso de mezcla es siempre lineal.

Un ejemplo del algoritmo de ordenación por mezcla sería la ordenación de la matriz de ocho elementos 24, 13, 26, 1, 2, 27, 38, 15. Después de ordenar recursivamente los primeros cuatro elementos y los cuatro últimos elementos, obtendríamos 1, 13, 24, 26, 2, 15, 27, 38. Después, mezclaríamos las dos mitades obteniendo la matriz final 1, 2, 13, 15, 24, 26, 27, 38.

### 8.5.2 El algoritmo de ordenación por mezcla

La ordenación por mezcla utiliza una memoria adicional lineal, lo que constituye un problema en la práctica.

En la Figura 8.8 se muestra una implementación sencilla de la ordenación por mezcla. La rutina no recursiva `mergeSort` de un único parámetro es una rutina de preparación simple que declara una matriz temporal e invoca recursivamente a `mergeSort` con los límites de la matriz. La rutina `merge` sigue la descripción dada en la Sección 8.5.1. Utiliza la primera mitad de la matriz (indexada desde `left` a `center`) como *A*, la segunda mitad (indexada desde `center+1` a `right`) como *B* y la matriz temporal como *C*. La Figura 8.9 implementa la rutina `merge`. Luego la matriz temporal se copia sobre la matriz original.

Aunque el tiempo de ejecución de la ordenación por mezcla es  $O(N \log N)$ , presenta el significativo problema de que mezclar dos listas ordenadas utiliza una memoria adicional lineal.

El trabajo adicional implicado en copiar la matriz temporal de nuevo sobre la matriz original a lo largo de todo el algoritmo ralentiza la operación considerablemente. Este proceso de copia puede evitarse conmutando juiciosamente los papeles de `a` y `tmpArray` en niveles alternativos de la recursión. Existe una variante de la ordenación por mezcla que también se puede implementar de forma no recursiva.

El tiempo de ejecución de la ordenación por mezcla depende fuertemente de los costes relativos de comparar y mover elementos en la matriz (y en la matriz temporal). En el caso de estar ordenando objetos generales en Java, la comparación de elementos es costosa, porque en un caso general la comparación se realiza mediante objetos función.

Pueden evitarse las copias excesivas con algo más de trabajo, pero no puede prescindirse de la memoria adicional lineal sin un excesivo coste en forma de tiempo adicional de ejecución.

```
1  /**
2   * Algoritmo de ordenación por mezcla.
3   * @param a una matriz de elementos Comparable.
4   */
5  public static <AnyType extends Comparable<? super AnyType>>
6  void mergeSort( AnyType [ ] a )
7  {
8      AnyType [ ] tmpArray = (AnyType [ ]) new Comparable[ a.length ];
9      mergeSort( a, tmpArray, 0, a.length - 1 );
10     }
11
12 /**
13  * Método interno que hace llamadas recursivas.
14  * @param a una matriz de elementos Comparable.
15  * @param tmpArray una matriz para colocar el resultado mezclado.
16  * @param left el índice más a la izquierda de la submatriz.
17  * @param right el índice más a la derecha de la submatriz.
18  */
19 private static <AnyType extends Comparable<? super AnyType>>
20 void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray,
21                 int left, int right )
22 {
23     if( left < right )
24     {
25         int center = ( left + right ) / 2;
26         mergeSort( a, tmpArray, left, center );
27         mergeSort( a, tmpArray, center + 1, right );
28         merge( a, tmpArray, left, center + 1, right );
29     }
30 }
```

Figura 8.8 Rutinas básicas de `mergeSort`.

```

1  /**
2  * Método interno que mezcla dos mitades ordenadas de una submatriz.
3  * @param a una matriz de elementos Comparable.
4  * @param tmpArray una matriz para colocar el resultado mezclado.
5  * @param leftPos el índice más a la izquierda de la submatriz.
6  * @param rightPos el índice del principio de la segunda mitad.
7  * @param rightEnd el índice más a la derecha de la submatriz.
8  */
9 private static <AnyType extends Comparable<? super AnyType>>
10 void merge( AnyType [ ] a, AnyType [ ] tmpArray,
11             int leftPos, int rightPos, int rightEnd )
12 {
13     int leftEnd = rightPos - 1;
14     int tmpPos = leftPos;
15     int numElements = rightEnd - leftPos + 1;
16
17     // Bucle principal
18     while( leftPos <= leftEnd && rightPos <= rightEnd )
19         if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
20             tmpArray[ tmpPos++ ] = a[ leftPos++ ];
21         else
22             tmpArray[ tmpPos++ ] = a[ rightPos++ ];
23
24     while( leftPos <= leftEnd )    // Copiar resto de la primera mitad
25         tmpArray[ tmpPos++ ] = a[ leftPos++ ];
26
27     while( rightPos <= rightEnd ) // Copiar resto de la mitad derecha
28         tmpArray[ tmpPos++ ] = a[ rightPos++ ];
29
30     // Copiar tmpArray en la matriz original
31     for( int i = 0; i < numElements; i++, rightEnd-- )
32         a[ rightEnd ] = tmpArray[ rightEnd ];
33 }

```

---

Figura 8.9 La rutina merge.

Por otro lado, desplazar los elementos no es costoso porque los elementos no se copian; más bien, lo que se produce son cambios en las referencias. La ordenación por mezcla utiliza el menor número de comparaciones de todos los algoritmos de ordenación más populares, y constituye por tanto un buen candidato para la ordenación de propósito general en Java. De hecho, es el algoritmo utilizado en `java.util.Arrays.sort` para ordenar matrices de objetos. Estos costes relativos no son iguales en otros lenguajes, ni tampoco son aplicables a la ordenación de tipos primitivos en Java. Un algoritmo alternativo es la ordenación rápida (*quicksort*), que vamos a describir en la siguiente sección. La ordenación rápida es el algoritmo utilizado en C++ para ordenar todos los tipos y se emplea en `java.util.Arrays.sort` para ordenar matrices de tipos primitivos.

## 8.6 Ordenación rápida

Como su nombre sugiere, la ordenación rápida (*quicksort*) es un rápido algoritmo de tipo divide y vencerás. Su tiempo de ejecución promedio es  $O(N \log N)$ . Su velocidad se debe principalmente a un bucle interno muy preciso y altamente optimizado. Tiene un rendimiento del caso peor de tipo cuadrático, pero con un poco de esfuerzo se puede hacer que ese caso peor sea estadísticamente improbable. Por un lado, el algoritmo de ordenación rápida es relativamente simple de entender y de demostrar su corrección, porque depende de la recursión. Por otro lado, es un algoritmo complicado de implementar, porque pequeños cambios en el código pueden provocar diferencias significativas en el tiempo de ejecución. En primer lugar, vamos a describir el algoritmo en términos generales. A continuación, proporcionaremos un análisis que muestra sus tiempos de ejecución de caso peor y de caso promedio. Emplearemos ese análisis para tomar decisiones acerca de cómo implementar ciertos detalles en Java, como por ejemplo el tratamiento de los elementos duplicados.

Cuando se implementa apropiadamente, la ordenación rápida es un rápido algoritmo de tipo divide y vencerás.

Considere el siguiente algoritmo de ordenación simple para ordenar una lista: seleccionamos arbitrariamente cualquier elemento y luego formamos tres grupos: los elementos más pequeños que el elemento elegido, los que son iguales al elemento elegido y los que son más grandes que el elemento elegido. Ordenamos recursivamente el primer y el tercer grupos y luego concatenamos los tres grupos. Se garantiza que el resultado será una versión ordenada de la lista original (en breve verificaremos esta afirmación). En la Figura 8.10 se muestra una implementación directa de este algoritmo, cuyo rendimiento es, en términos generales, bastante bueno para la mayor parte de listas de datos de entrada. De hecho, si la lista contiene un gran número de duplicados con relativamente pocos elementos distintos, como a veces sucede, entonces el rendimiento es extremadamente bueno.

El algoritmo que hemos descrito forma la base del algoritmo de ordenación clásico, la *ordenación rápida*. Sin embargo, al formar esas listas adicionales y hacerlo además recursivamente, resulta difícil ver en qué sentido hemos mejorado con respecto a la ordenación por mezcla. Realmente, hasta ahora, no lo hemos hecho. Para poder mejorar las cosas, tenemos que evitar utilizar una memoria adicional significativa y disponer de bucles internos que sean limpios. Por ello, la ordenación rápida suele escribirse de una forma que evite crear el segundo grupo (los elementos iguales), y el algoritmo tiene numerosos detalles sutiles que afectan a su velocidad. El resto de la sección describe la implementación más común de la ordenación rápida, que es aquella en la que la entrada es una matriz en la que el algoritmo no crea ninguna matriz adicional.

### 8.6.1 El algoritmo de ordenación rápida

El algoritmo básico de ordenación rápida *Quicksort(S)* consta de los cuatro pasos siguientes:

1. Si el número de elementos de  $S$  es 0 o 1, entonces volver.
2. Seleccionar *cualquier* elemento  $v$  de  $S$ . Dicho elemento se denomina *pivote*.
3. Partitionar  $S - \{v\}$  (los elementos restantes de  $S$ ) en dos grupos disjuntos:  $L = \{x \in S - \{v\} | x \leq v\}$  y  $R = \{x \in S - \{v\} | x \geq v\}$ .
4. Devolver el resultado de *Quicksort(L)* seguido de  $v$  seguido de *Quicksort(R)*.

```

1 public static void sort( List<Integer> items )
2 {
3     if( items.size( ) > 1 )
4     {
5         List<Integer> smaller = new ArrayList<Integer>();
6         List<Integer> same = new ArrayList<Integer>();
7         List<Integer> larger = new ArrayList<Integer>();
8
9         Integer chosenItem = items.get( items.size( ) / 2 );
10        for( Integer i : items )
11        {
12            if( i < chosenItem )
13                smaller.add( i );
14            else if( i > chosenItem )
15                larger.add( i );
16            else
17                same.add( i );
18        }
19
20        sort( smaller ); // Llamada recursiva
21        sort( larger ); // Llamada recursiva
22
23        items.clear( );
24        items.addAll( smaller );
25        items.addAll( same );
26        items.addAll( larger );
27    }
28 }

```

Figura 8.10 Algoritmo de ordenación recursivo simple.

Son varios los puntos que llaman nuestra atención cuando examinamos estos pasos. En primer lugar, el caso múltiple de la recursión incluye la posibilidad de que  $S$  pueda ser un conjunto vacío.

El algoritmo básico de ordenación rápida es recursivo. Los detalles del algoritmo incluyen cómo seleccionar el pivote, cómo decidir la forma de particionar y cómo tratar el problema de los duplicados. Las decisiones incorrectas dan tiempos de ejecución cuadráticos para diversos tipos comunes de matrices de entrada.

Esto es necesario porque las llamadas recursivas podrían generar subconjuntos vacíos. En segundo lugar, el algoritmo permite utilizar cualquier elemento como pivote. El *pivote* divide los elementos de la matriz en dos grupos: los elementos que son menores que el pivote y los elementos que son mayores que el pivote. El análisis que vamos a realizar muestra que algunas elecciones de pivotes son mejores que otras. Por tanto, cuando proporcionemos una implementación real, no nos limitaremos a utilizar cualquier pivote. En lugar de ello, trataremos de hacer una elección juiciosa.

En la etapa de *partición*, cada elemento de  $S$ , excepto el pivote, se coloca bien en  $L$  (que representa la parte izquierda de la matriz) o  $R$  (que representa la parte derecha de la matriz). La intención es que los elementos

menores que el pivote vayan a  $L$  y que los elementos mayores que el pivote vayan a  $R$ . La descripción contenida en el algoritmo, sin embargo, describe de forma ambigua qué hacer con los elementos que son iguales al pivote. Permite que cada instancia de un duplicado vaya a uno cualquiera de los dos subconjuntos, especificando únicamente que debe ir a uno o al otro. Parte de una buena implementación Java consiste, precisamente, en tratar este caso de la forma más eficiente posible. De nuevo, el análisis nos permitirá tomar una decisión razonable.

La Figura 8.11 muestra cómo actúa la ordenación rápida sobre un conjunto de números. El pivote se selecciona (por azar) como el valor 65. Los restantes elementos del conjunto se parten en dos subconjuntos más pequeños. Después, cada grupo de ordena de forma recursiva. Recuerde que, por la tercera regla de la recursión, podemos asumir que este paso funciona. A

El pivote divide los elementos de la matriz en dos grupos: los que son más pequeños que el pivote y los que son grandes que el pivote.

En la etapa de partición se coloca en uno de los dos grupos cada elemento de la matriz salvo el pivote.

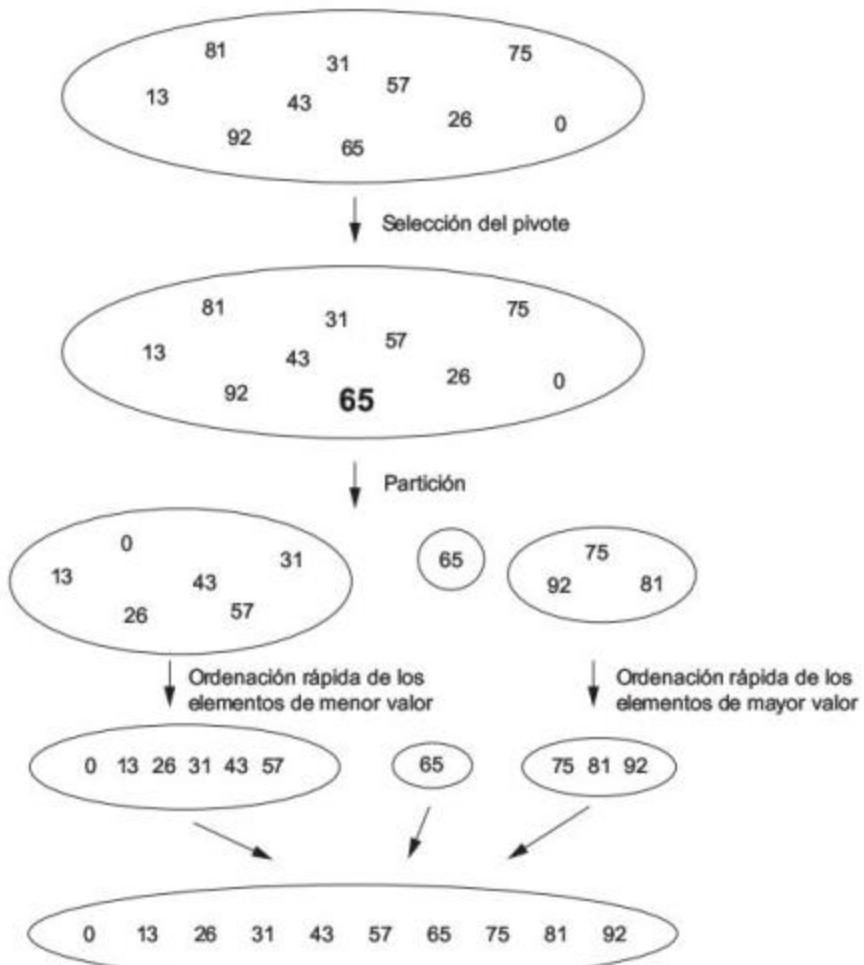


Figura 8.11 Los pasos de la ordenación rápida.

continuación, se obtiene la ordenación del grupo completo de manera trivial. En una implementación Java, los elementos se almacenarían en una parte de una matriz delimitada por `low` y `high`. Después de la etapa de particionamiento, el pivote terminaría estando almacenado en alguna celda `p` de la matriz. Las llamadas recursivas se realizarían entonces con las partes que van desde `low` a `p-1` y luego desde `p+1` a `high`.

Puesto que la recursión nos permite hacer ese enorme acto de fe, la corrección del algoritmo se demuestra de la forma siguiente.

- El grupo de los elementos de pequeño valor se ordena gracias a la recursión.
- El elemento de mayor valor del grupo de los elementos de pequeño valor no es mayor que el pivote, gracias a la partición.
- El pivote no es mayor que el menor de los elementos del grupo de elementos de mayor valor gracias a la partición.
- El grupo de los elementos de mayor valor está ordenado gracias a la recursión.

La ordenación rápida es realmente rápida, porque la etapa de particionamiento puede realizarse rápidamente y sobre la propia matriz.

Aunque puede establecerse fácilmente la corrección del algoritmo, no está claro por qué es más rápido que la ordenación por mezcla. Como hace la ordenación por mezcla, la ordenación rápida resuelve de forma recursiva dos subproblemas y requiere un trabajo adicional lineal (representado por la etapa de particionamiento). Sin embargo, a diferencia de en la ordenación por mezcla, no está garantizado que los subproblemas en la ordenación rápida tengan el mismo tamaño, lo cual afecta negativamente al rendimiento. No obstante, la ordenación rápida puede ser más rápida que la ordenación por mezcla porque la etapa de particionamiento puede realizarse de forma significativamente más rápida que la etapa de mezcla. En particular, la etapa de particionamiento puede efectuarse sin una matriz adicional, y el código para implementarla es muy compacto y eficiente. Esta ventaja compensa el que los subproblemas no tengan igual tamaño.

### 8.6.2 Análisis del algoritmo de ordenación rápida

La descripción del algoritmo nos deja varias cuestiones sin responder: ¿cómo seleccionamos el pivote? ¿Cómo llevamos a cabo la partición? ¿Qué hacemos si vemos un elemento que es igual al pivote? Todas estas cuestiones pueden afectar enormemente al tiempo de ejecución del algoritmo. Vamos a realizar un análisis que nos ayude a decidir cómo implementar los pasos no especificados en una ordenación rápida.

#### Caso mejor

El caso mejor ocurre cuando la operación de partición divide siempre el conjunto en subconjuntos iguales. El tiempo de ejecución es  $O(N \log N)$ .

El caso mejor para la ordenación rápida es que el pivote particione el conjunto en dos subconjuntos de igual tamaño, y que este tipo de particionamiento se produzca en cada etapa de la recursión. Entonces tendremos dos llamadas recursivas con un problema de tamaño mitad más un coste adicional lineal, lo que permite igualar el rendimiento de la ordenación por mezcla. El tiempo de ejecución para este caso es  $O(N \log N)$ . (En realidad no hemos demostrado que este sea el caso mejor, aunque esa demostración es posible, aquí vamos a omitir los detalles.)

## Caso peor

Puesto que los subconjuntos de igual tamaño son buenos para la ordenación rápida, podría esperarse que los subconjuntos de tamaño desigual fueran malos. Efectivamente, es así. Supongamos que, en cada paso de la recursión, el pivote resultara ser el elemento más pequeño, entonces el conjunto de elementos de pequeño valor  $L$  estaría vacío y el conjunto de elementos de mayor valor  $R$  tendría todos los elementos menos el pivote. Entonces tendríamos que llamar recursivamente a la rutina de ordenación rápida para el subconjunto  $R$ . Suponga también que  $T(N)$  es el tiempo de ejecución necesario para ordenar con este algoritmo  $N$  elementos y asumamos que el tiempo para ordenar 0 o 1 elemento es solo 1 unidad de tiempo. Suponga, además, que cargamos  $N$  unidades por la tarea de particionar un conjunto que contiene  $N$  elementos. Entonces, para  $N > 1$ , obtenemos un tiempo de ejecución que satisface

$$T(N) = T(N-1) + N \quad (8.1)$$

En otras palabras, la Ecuación 8.1 establece que el tiempo requerido para realizar una ordenación rápida de  $N$  elementos es igual al tiempo necesario para ordenar recursivamente los  $N-1$  elementos del subconjunto de elementos de mayor valor más las  $N$  unidades de coste necesarias para realizar la partición. Esto asume que, en cada paso de la iteración, somos lo suficientemente desafortunados como para seleccionar como pivote el elemento más pequeño. Para simplificar el análisis, vamos a normalizar, eliminando los factores constantes, y a resolver esta recurrencia haciendo un desarrollo telescopico repetido de la Ecuación 8.1:

$$\begin{aligned} T(N) &= T(N-1) + N \\ T(N-1) &= T(N-2) + (N-1) \\ T(N-2) &= T(N-3) + (N-2) \\ &\dots \\ T(2) &= T(1) + 2 \end{aligned} \quad (8.2)$$

Cuando sumamos todas las fórmulas de la Ecuación 8.2, se produce una cancelación masiva de términos, que nos deja

$$T(N) = T(1) + 2 + 3 + \dots + N = \frac{N(N+1)}{2} = O(N^2) \quad (8.3)$$

Este análisis verifica la intuición de que una división no equilibrada es perjudicial. Invertimos  $N$  unidades de tiempo en particionar y luego tenemos que hacer una llamada recursiva que afecta a  $N-1$  elementos. A continuación, invertimos  $N-1$  unidades para particionar ese grupo, pero tan solo para tener que hacer una llamada recursiva para  $N-2$  elementos. En esa llamada, invertimos  $N-2$  unidades realizando la partición, etc. El coste total de realizar todas las particiones a lo largo de todas las llamadas recursivas se corresponde exactamente con lo obtenido en la Ecuación 8.3. Este resultado nos dice que a la hora de implementar la selección del pivote y la etapa de particionamiento, no queremos hacer nada que pudiera contribuir a que el tamaño de los subconjuntos no esté equilibrado.

El caso peor ocurre cuando la partición genera repentinamente un conjunto vacío. El tiempo de ejecución es  $O(N^2)$ .

## Caso promedio

El caso promedio es  $O(N \log N)$ . Aunque esto parece intuitivo, es necesaria una demostración formal.

Los dos primeros análisis nos dicen que los casos mejor y peor son extremadamente diferentes. Naturalmente, queremos saber lo que sucede en el caso promedio. Esperaríamos que como cada subproblema tiene la mitad de tamaño que el original como promedio, la cota de  $O(N \log N)$  sería también cota del caso promedio. Dicha expectativa, aunque es correcta para la aplicación concreta de la ordenación rápida que estamos examinando aquí, no es una demostración formal. No se puede hablar de los promedios a la ligera. Suponga por ejemplo que tuviéramos un algoritmo de selección de pivote que nos garantizara seleccionar únicamente el mayor o el menor elemento, cada uno de ellos con una probabilidad de  $1/2$ . Entonces el tamaño medio del grupo de elementos de menor valor sería aproximadamente  $N/2$ , al igual que también lo sería el tamaño medio del grupo de elementos de mayor valor (porque cada uno de ellos tiene la misma probabilidad de tener 0 o  $N - 1$  elementos). Pero el tiempo de ejecución de la ordenación rápida con esa estrategia de selección de pivotes siempre sería cuadrático, porque siempre obtendríamos una mala división del grupo de elementos. Por tanto, tenemos que tener cuidado al emplear el calificativo *promedio*. Se puede argumentar que el grupo de elementos de menor valor tiene tanta probabilidad de contener 0, 1, 2, ... o  $N - 1$  elementos, lo cual también es cierto para el grupo de elementos de mayor valor. Con esta suposición, podemos establecer que el tiempo de ejecución del caso promedio es efectivamente  $O(N \log N)$ .

El coste promedio de una llamada recursiva se obtiene promediando los costes para todos los posibles tamaño de subproblema.

Dado que el coste para realizar una ordenación rápida de  $N$  elementos es igual a  $N$  unidades para la etapa de particionamiento más el coste de las dos llamadas recursivas, tenemos que determinar el coste promedio de cada una de las llamadas recursivas. Si  $T(N)$  representa el coste promedio para hacer una ordenación rápida de  $N$  elementos, el coste promedio de cada llamada recursiva será igual a la media –para todos los posibles tamaños de subproblemas– del coste promedio de realizar una llamada recursiva para ese subproblema:

$$T(L) = T(R) = \frac{T(0) + T(1) + T(2) + \dots + T(N-1)}{N} \quad (8.4)$$

La Ecuación 8.4 indica que estamos buscando el coste de cada posible tamaño de subconjunto y realizando el promedio de todos esos costes. Puesto que tenemos dos llamadas recursivas, más un tiempo lineal para realizar la partición, obtenemos

$$T(N) = 2 \left( \frac{T(0) + T(1) + T(2) + \dots + T(N-1)}{N} \right) + N \quad (8.5)$$

El tiempo promedio de ejecución está dado por  $T(N)$ . Resolvemos la Ecuación 8.5 eliminando todos los valores recursivos de  $T$ , salvo el más reciente.

Para resolver la Ecuación 8.5, comenzamos multiplicando ambos lados por  $N$ , obteniendo

$$NT(N) = 2(T(0) + T(1) + T(2) + \dots + T(N-1)) + N^2 \quad (8.6)$$

A continuación escribimos la Ecuación 8.6 para el caso  $N - 1$ , con la idea de poder simplificar enormemente la ecuación mediante una simple resta. Haciendo esto así, nos queda

$$(N-1)T(N-1) = 2(T(0)+T(1)+\dots+T(N-2)) + (N-1)^2 \quad (8.7)$$

Si ahora restamos la Ecuación 8.7 de la Ecuación 8.6 obtenemos

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2N - 1$$

Reordenamos los términos y despreciamos el  $-1$  (que no es significativo) del lado derecho, obteniendo

$$NT(N) = (N+1)T(N-1) + 2N \quad (8.8)$$

Una vez que tenemos  $T(N)$  en función exclusivamente de  $T(N-1)$ , tratamos de formar una suma telescopica.

Ahora tenemos una fórmula para  $T(N)$  en función únicamente de  $T(N-1)$ . De nuevo, la idea consiste en realizar una suma telescopica, pero la Ecuación 8.8 está en una forma incorrecta. 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 formar la suma telescopica:

$$\begin{aligned} \frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2}{N+1} \\ \frac{T(N-1)}{N} &= \frac{T(N-2)}{N-1} + \frac{2}{N} \\ \frac{T(N-2)}{N-1} &= \frac{T(N-3)}{N-2} + \frac{2}{N-1} \\ &\dots \\ \frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2}{3} \end{aligned} \quad (8.9)$$

Si sumamos todas las fórmulas de la Ecuación 8.9, tenemos

$$\begin{aligned} \frac{T(N)}{N+1} &= \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{N} + \frac{1}{N+1}\right) \\ &= 2\left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N+1}\right) - \frac{5}{2} \\ &= O(\log N) \end{aligned} \quad (8.10)$$

La última línea de la Ecuación 8.10 se deduce del Teorema 5.5. Si multiplicamos ambos lados por  $N+1$ , obtenemos el resultado final:

$$T(N) = O(N \log N) \quad (8.11)$$

Utilizamos el hecho de que el  $N$ -ésimo número armónico es  $O(\log N)$ .

### 8.6.3 Selección del pivote

Ahora que hemos establecido que la ordenación rápida podrá ejecutarse en un tiempo medio  $O(N \log N)$ , nuestra preocupación principal consiste en cerciorarnos de que el caso peor no llegue a presentarse nunca. Realizando un complejo análisis, podemos calcular la desviación estándar del tiempo de ejecución de la rutina de ordenación rápida. El resultado es que, si se le presenta al algoritmo una única permutación aleatoria, el tiempo de ejecución utilizado para ordenarla estará, casi con total seguridad, muy próximo a la media. Por tanto, lo único que tenemos que conseguir es que las entradas de tipo degenerado no den como resultado un tiempo de ejecución muy lento. Entre las entradas de tipo degenerado se incluyen los datos que ya han sido ordenados y los datos que solo contienen  $N$  elementos completamente idénticos. En ocasiones, son los casos más sencillos los que ponen en aprietos a los algoritmos.

#### Una solución incorrecta

La selección del pivote es crucial para tener un buen rendimiento. Nunca seleccione el primer elemento como pivote.

La elección más popular, totalmente ingenua, consiste en utilizar el primer elemento (es decir, el elemento en la posición  $low$ ) como pivote. Esta selección es aceptable si la entrada es aleatoria, pero si la entrada ha sido ordenada previamente o está en orden inverso, el pivote proporciona una partición poco adecuada, porque se tratará de uno de los elementos extremos. Además, este comportamiento se replicará recursivamente. Como hemos demostrado anteriormente en el capítulo, terminaríamos con un tiempo de ejecución cuadrático, con lo que no habríamos avanzado nada. Evidentemente, eso sería muy embarazoso. *Nunca utilice el primer elemento como pivote.*

Otra alternativa popular consiste en seleccionar la mayor de las primeras dos claves diferentes<sup>3</sup> como pivote, pero esta selección tiene los mismos efectos perniciosos que elegir la primera clave. Evite utilizar cualquier estrategia que solo examine algunas claves cerca del principio o del final del grupo de datos de entrada.

#### Una elección segura

El elemento central es una elección razonable pero pasiva.

Una elección perfectamente razonable como pivote es el elemento central; es decir, el elemento situado en la celda  $(low+high)/2$  de la matriz. Cuando la entrada ya ha sido ordenada, esta selección nos da el pivote perfecto en cada llamada recursiva. Por supuesto, podríamos construir una secuencia de entrada que forzara un comportamiento cuadrático para esta estrategia (véase el Ejercicio 8.9). Sin embargo, la posibilidad de encontrarnos por azar con un caso que tarde en ejecutarse aunque solo sea el doble que el caso promedio es extremadamente pequeña.

#### Particionamiento basado en la mediana de tres

Seleccionar el elemento central como pivote permite evitar los casos degenerados que se derivan de la utilización de entradas no aleatorias. Sin embargo, observe que esto no es sino una elección de carácter pasivo. Es decir, no tratamos de elegir un buen pivote. En lugar de ello, simplemente evitamos seleccionar un mal pivote. El particionamiento basado en la mediana de tres es el intento de elegir un pivote algo mejor que el pivote típico. En el *particionamiento basado en la mediana*

<sup>3</sup>En un objeto complejo, la clave suele ser la parte del objeto en la que se basa la comparación.

de tres se utiliza como pivote la mediana del primer elemento, del elemento central y del último elemento.

La mediana de un grupo de  $N$  números es el  $\lceil N/2 \rceil$ -ésimo número más pequeño. La mediana es, obviamente, la mejor elección de pivote, porque garantiza una división equitativa de los elementos. Lamentablemente, la mediana es difícil de calcular, lo que ralentizaría considerablemente la ordenación rápida. Por tanto, lo que queremos es obtener una buena estimación de la mediana pero sin invertir demasiado tiempo en dicho cálculo. Podemos obtener dicha estimación mediante *muestreo*, el método clásico utilizado en los sondeos de opinión. Es decir, seleccionamos un subconjunto de los números de entrada y calculamos su mediana. Cuanto mayor sea la muestra, más precisa será la estimación. Sin embargo, cuanto más grande sea la muestra, también necesitaremos más tiempo para evaluar la mediana. Un tamaño muestral de 3 nos da un pequeña mejora en el tiempo medio de ejecución del algoritmo de ordenación rápida y también simplifica el código de particionamiento resultante al eliminar algunos casos especiales. Los grandes tamaños muestrales no mejoran significativamente el rendimiento, así que no merece la pena utilizarlos.

Los tres elementos utilizados en la muestra son el primero, el central y el último. Por ejemplo, con la entrada 8, 1, 4, 9, 6, 3, 5, 2, 7, 0, el elemento situado más a la izquierda es 8, el elemento situado más a la derecha es 0 y el elemento central es 6; por tanto, el pivote sería 6. Observe que para elementos previamente ordenados, conservaremos el elemento central como pivote, y en este caso el pivote será la mediana.

En el particionamiento de la mediana de tres se utiliza como pivote la mediana del primer elemento, del elemento central y del último elemento. Esta técnica simplifica la etapa de particionamiento del algoritmo de ordenación rápida.

#### 8.6.4 Una estrategia de particionamiento

Hay varias estrategias de particionamiento comúnmente utilizadas. La que vamos a describir en esta sección proporciona buenos resultados. La estrategia de particionamiento más sencilla consta de tres pasos. En la Sección 8.6.6 mostraremos las mejoras que se obtienen al utilizar la técnica de selección de pivote basada en la mediana de tres.

El primer paso del algoritmo de particionamiento consiste en quitar de en medio el elemento pivote, intercambiándolo por el último elemento. El resultado para nuestra entrada ejemplo se muestra en la Figura 8.12. El elemento pivote se muestra en sombreado más oscuro al final de la matriz.

Paso 1: intercambiar el pivote con el elemento situado al final.

Por ahora, vamos a asumir que todos los elementos son diferentes, dejando para más adelante la cuestión de qué hacer en presencia de duplicados. Como caso límite, nuestro algoritmo deberá funcionar correctamente cuando *todos* los elementos sean idénticos.

En el paso 2, utilizamos nuestra estrategia de particionamiento para desplazar todos los elementos de pequeño valor hacia la izquierda de la matriz y todos los elementos de valor más grande hacia la derecha. Los términos *pequeño* y *grande* son relativos al pivote. En las Figuras 8.12–8.17, las celdas de color blanco son aquellas que sabemos que están correctamente colocadas. Las celdas con un sombreado claro no están, necesariamente, correctamente colocadas.

Exploramos de izquierda a derecha buscando un elemento de valor grande utilizando para ello un contador  $i$ , inicializado en la posición  $low$ . También exploramos de derecha a izquierda buscando un elemento de valor pequeño, utilizando un contador  $j$  inicializado de forma tal que comience en  $high - 1$ . La Figura 8.13 muestra que la búsqueda de un valor grande se detiene en 8

Paso 2: incrementar  $i$  de izquierda a derecha y  $j$  de derecha a izquierda. Cuando  $i$  encuentra un elemento de valor grande se detiene. Cuando  $j$  encuentra un elemento de valor pequeño, se detiene. Si  $i$  y  $j$  no se han cruzado, intercambiar sus elementos y continuar. En caso contrario, detener este bucle.

y que la búsqueda de un elemento de valor pequeño se detiene en 2. Estas celdas se han marcado con un sombreado claro. Observe que, al saltarnos el valor 7, sabemos que 7 no es un valor pequeño y que está por tanto colocado correctamente. Por ello se presenta también como una celda en blanco. Ahora tenemos un elemento de valor grande, 8, en el lado izquierdo de la matriz y un elemento de valor pequeño, 2, en el lado derecho de la matriz. Tenemos que intercambiar estos dos elementos para colocarlos correctamente, como se muestra en la Figura 8.14.

A medida que continua el algoritmo,  $i$  se detiene en el elemento de valor grande 9 y  $j$  se detiene en el elemento valor pequeño 5. De nuevo, se garantiza que los elementos que  $i$  y  $j$  se han saltado durante la exploración están correctamente colocados. La Figura 8.15 muestra el resultado: los extremos de la matriz (sin contar el pivote) contienen elementos correctamente colocados.

A continuación, intercambiamos los elementos que  $i$  y  $j$  están indexando, como se muestra en la Figura 8.16. La exploración continúa con  $i$  deteniéndose en el elemento de valor grande 9 y  $j$  deteniéndose en el elemento de valor pequeño 3. Sin embargo, llegados a este punto,  $i$  y  $j$  habrán cruzado sus posiciones dentro de la matriz. En consecuencia, un intercambio sería inútil. De aquí que la Figura 8.17 muestre que el elemento al que está accediendo  $j$  ya está correctamente colocado y no debe desplazarse.

La Figura 8.17 muestra que todos los elementos están correctamente colocados, salvo dos. ¿No sería maravilloso si pudiéramos intercambiarlos y con ello dar por finalizada la tarea? Bueno, pues

Paso 3: intercambiar el elemento de la posición  $i$  con el pivote.

efectivamente podemos hacerlo. Todo lo que necesitamos es intercambiar el elemento de la posición  $i$  con el elemento de la última celda (el pivote), como se muestra en la Figura 8.18. El elemento que  $i$  está indexando es obviamente de valor grande, por lo que no hay ningún problema con desplazarlo a la última posición.

8	1	4	9	0	3	5	2	7	6
---	---	---	---	---	---	---	---	---	---

Figura 8.12 Algoritmo de particionamiento: el elemento pivote 6 se coloca al final.

8	1	4	9	0	3	5	2	7	6
---	---	---	---	---	---	---	---	---	---

Figura 8.13 Algoritmo de particionamiento:  $i$  se detiene al llegar al elemento más grande, 8;  $j$  se detiene en 2, el elemento más pequeño.

2	1	4	9	0	3	5	8	7	6
---	---	---	---	---	---	---	---	---	---

Figura 8.14 Algoritmo de particionamiento: se intercambian los elementos desordenados 8 y 2.

2	1	4	9	0	3	5	8	7	6
---	---	---	---	---	---	---	---	---	---

Figura 8.15 Algoritmo de particionamiento:  $i$  se detiene en el elemento de valor grande 9;  $j$  se detiene en el elemento valor pequeño 5.

2	1	4	5	0	3	9	8	7	6
---	---	---	---	---	---	---	---	---	---

Figura 8.16 Algoritmo de particionamiento: se intercambian los elementos 9 y 5, que estaban desordenados.

2	1	4	5	0	3	9	8	7	6
---	---	---	---	---	---	---	---	---	---

Figura 8.17 Algoritmo de particionamiento: i se detiene en el elemento de valor grande 9; j se detiene en 3, el elemento de valor pequeño.

2	1	4	5	0	3	6	8	7	9
---	---	---	---	---	---	---	---	---	---

Figura 8.18 Algoritmo de particionamiento: intercambiar el pivote y el elemento situado en la posición 1.

Observe que el algoritmo de particionamiento no requiere ninguna memoria adicional y que cada elemento se compara exactamente una vez con el pivote. A la hora de escribir el código, este enfoque se traduce en un bucle interno muy compacto.

### 8.6.5 Claves iguales al pivote

Un detalle importante que tenemos que considerar es cómo tratar las claves que sean iguales al pivote. ¿Debería i detenerse cuando encuentre una clave igual al pivote, y debería j hacer lo mismo cuando encuentre una clave igual al pivote? Los contadores i y j deberían hacer lo mismo; en caso contrario, la etapa paso de particionamiento sería tendenciosa. Por ejemplo, si i se detiene y j no lo hace, todas las claves que sean iguales al pivote terminarán en el lado derecho de la matriz.

Consideremos el caso en el que todos los elementos de la matriz sean idénticos. Si tanto i como j se detienen, se producirán muchos intercambios entre elementos idénticos. Aunque estas acciones parecen inútiles, el efecto positivo es que i y j se cruzarán en el centro, por lo que al volver a colocar el pivote en su lugar, la partición creará dos subconjuntos prácticamente iguales. Por ello, se aplicaría el análisis de caso mejor y el tiempo de ejecución sería  $O(N \log N)$ .

Si no se detienen ni i ni j, entonces i terminaría en la última posición (suponiendo, por supuesto, que no se detenga en la posición inmediatamente anterior al pivote), y no se realiza ningún intercambio. Este resultado parece muy positivo hasta que nos damos cuenta de que el pivote estará entonces situado como último elemento, porque esa es la última celda que i llega a tocar. El resultado final son dos subconjuntos muy desequilibrados y un tiempo de ejecución que se corresponde con la cota de caso peor  $O(N^2)$ . El efecto es el mismo que utilizar el primer elemento como pivote en el caso de una entrada previamente ordenada: se requiere un tiempo cuadrático para no hacer nada.

La conclusión es que realizar los intercambios innecesarios y crear subconjuntos desequilibrados es mejor que arriesgarse a tener subconjuntos muy desequilibrados. Por tanto, lo que hacemos es que tanto i como j se detengan en el momento en que se encuentre un elemento igual al pivote. Esta acción resulta ser la única de las cuatro posibilidades que no requiere un tiempo cuadrático para esta entrada.

Los contadores i y j deben detenerse cuando encuentren un elemento igual al pivote para garantizar un buen rendimiento del algoritmo.

A primera vista, preocuparse por una matriz de elementos idénticos puede parecer un poco estúpido. Después de todo, ¿por qué querría alguien ordenar 5.000 elementos idénticos? Sin embargo, recuerde que la ordenación rápida es un algoritmo recursivo. Suponga que hay 100.000 elementos, de los cuales 5.000 son idénticos. Podría llegar a darse el caso de que la ordenación rápida hiciera una llamada recursiva exclusivamente para esos 5.000 elementos idénticos. Por ello, garantizar que los 5.000 elementos idénticos puedan ser ordenados de manera eficiente es realmente importante.

### 8.6.6 Particionamiento basado en la mediana de tres

Calcular la mediana de tres implica ordenar tres elementos. Por tanto, podemos dar algo de ventaja a la etapa de particionamiento, así como dejar de preocuparnos por llegar a traspasar las fronteras de la matriz.

Cuando llevamos a cabo el particionamiento basado en la mediana de tres, podemos recurrir a una optimización sencilla que nos ahorra unas cuantas comparaciones y que simplifica también enormemente el código. La Figura 8.19 muestra la matriz original.

Recuerde que el particionamiento basado en la mediana de tres requiere que determinemos la mediana del primer elemento, el elemento central y del último elemento. La forma más fácil de hacerlo es ordenarlos dentro de la matriz. El resultado se muestra en la Figura 8.20. Observe el sombreado resultante: está garantizado que el elemento que termina en la primera posición es más pequeño (o igual) que el pivote, y también se garantiza que el elemento situado en la última posición es mayor (o igual) que el pivote. Este resultado nos dice cuatro cosas:

- No deberíamos intercambiar el pivote con el elemento situado en la última posición. En su lugar, deberíamos intercambiarlo con el elemento situado en la penúltima posición, como se muestra en la Figura 8.21.
- Podemos hacer que  $i$  comience en  $low+1$  y que  $j$  comience en  $high-2$ .
- Está garantizado que, siempre que  $i$  busque un elemento de valor grande terminará por detenerse, porque en el caso peor se encontrará con el pivote (y hemos impuesto la regla de que el contador debe detenerse en caso de igualdad).
- Está garantizado que, cuando  $j$  esté buscando un elemento de valor pequeño, terminará por detenerse, porque en el caso peor se encontrará con el primer elemento (que tiene que ser como mucho de valor igual al pivote).

Todas estas optimizaciones están incorporadas en el código Java final.

8	1	4	9	6	3	5	2	7	0
---	---	---	---	---	---	---	---	---	---

Figura 8.19 Matriz original.

0	1	4	9	6	3	5	2	7	8
---	---	---	---	---	---	---	---	---	---

Figura 8.20 Resultado de ordenar tres elementos (primero, central y último).

0	1	4	9	7	3	5	2	6	8
---	---	---	---	---	---	---	---	---	---

Figura 8.21 Resultado de intercambiar el pivote con el penúltimo elemento.

### 8.6.7 Matrices de pequeño tamaño

Nuestra optimización final afecta a las matrices de pequeño tamaño. ¿Merece la pena utilizar una rutina potente como la de ordenación rápida cuando solo hay 10 elementos que ordenar? La respuesta es, por supuesto, que no. Una rutina simple como la de ordenación por inserción probablemente sea más rápida para las matrices de pequeño tamaño. La naturaleza recursiva de la ordenación rápida nos dice que nos veríamos obligados a generar muchas llamadas que solo operan sobre subconjuntos de pequeño tamaño. Por tanto, comprobar el tamaño del subconjunto merece la pena. Si es más pequeño que un cierto punto de corte predefinido, aplicamos la ordenación por inserción; en caso contrario, utilizamos la ordenación rápida.

Ordene 10 o menos elementos mediante la ordenación por inserción. Incluya dicha comprobación en la rutina recursiva de ordenación rápida.

Un buen punto de corte son 10 elementos, aunque cualquier punto de corte entre 5 y 20 probablemente permita obtener resultados similares. El punto de corte óptimo depende, en la práctica, de cada máquina. El uso de un punto de corte nos ahorra los casos degenerados. Por ejemplo, determinar la mediana de tres elementos no tiene mucho sentido cuando hay menos de tres elementos.

En el pasado, muchos pensaban que una alternativa todavía mejor era dejar la matriz ligeramente desordenada, no haciendo nada cuando el tamaño del subconjunto estuviera por debajo del punto de corte. Dado que la ordenación por inserción es tan eficiente para matrices prácticamente ordenadas, podríamos demostrar matemáticamente que ejecutar una ordenación por inserción al final, para terminar de ordenar la matriz, es más rápido que ejecutar todas las pequeñas ordenaciones por inserción. El ahorro obtenido equivaldría, aproximadamente, al coste adicional de las llamadas al método de ordenación por inserción.

Actualmente, las llamadas a métodos no son tan costosas como solían ser. Además, resulta costoso realizar una segunda exploración de la matriz completa para la ordenación por inserción. Gracias a una técnica denominada *almacenamiento en caché*, suele ser más conveniente realizar la ordenación por inserción con las matrices de pequeño tamaño. Los accesos a memoria localizados son más rápidos que los no localizados. En muchas máquinas, consultar la memoria dos veces en una exploración es más rápido que consultar la memoria una vez en cada una de dos exploraciones independientes.

La idea de combinar un segundo algoritmo de ordenación cuando las llamadas recursivas al algoritmo de ordenación rápida parecen inapropiadas, también puede utilizarse para garantizar un caso peor  $\mathcal{O}(N \log N)$  para la ordenación rápida. En el Ejercicio 8.22 le pediremos que explore cómo combinar la ordenación rápida y la ordenación por mezcla para obtener el rendimiento de caso promedio de la ordenación rápida casi todas las veces, con un rendimiento de caso peor garantizado equivalente al de ordenación por mezcla. En la práctica, en lugar de la ordenación por mezcla utilizamos otro algoritmo, llamado *ordenación por montículos*, que veremos en la Sección 21.5.

### 8.6.8 Rutina de ordenación rápida en Java

Utilizamos una rutina de preparación para preparar las cosas necesarias para la recursión.

En la Figura 8.22 se muestra una implementación real de la rutina de ordenación rápida. La rutina de un solo parámetro `quicksort`, declarada en las líneas 4 a 8, es simplemente una rutina de preparación que invoca a la rutina recursiva `quicksort`. Por tanto, solo vamos a hablar de la implementación de la versión recursiva de la rutina `quicksort`.

En la línea 17 comprobamos la existencia de submatrices de pequeño tamaño y llamamos a la ordenación por inserción (no mostrada) cuando la instancia del problema está por debajo de un valor especificado dado por la constante `CUTOFF`. En caso contrario, continuamos con el proceso recursivo. Las líneas 21 a 27 ordenan los elementos `low` (menor), `middle` (central) y `high` (mayor), dentro de la propia matriz. Ajustándonos a las explicaciones anteriores, utilizamos el elemento central como pivote y lo intercambiamos con el elemento situado en la penúltima posición en las líneas 30 y 31. Después entramos en la fase de particionamiento, inicializamos los contadores `i` y `j` con un valor situado una posición más allá de sus verdaderos valores iniciales, porque los operadores prefijo de incremento y decremento los ajustarán inmediatamente, antes de los accesos a la matriz en las líneas 37 y 39. Cuando se sale del primer bucle `while` en la línea 37, `i` estará indexando un elemento que es mayor (o posiblemente igual) que el pivote. Del mismo modo, cuando se sale del segundo bucle, `j` estará indexando un elemento que es menor (o posiblemente igual) que el pivote. Si `i` y `j` no se han cruzado, estos elementos se intercambian y continuamos con la exploración. En caso contrario, se termina la exploración y el pivote se restaura en la línea 46. La ordenación se termina cuando se realizan las dos llamadas recursivas en las líneas 48 y 49.

```

1  /**
2   * Algoritmo de ordenación rápida (rutina de preparación)
3   */
4  public static <AnyType extends Comparable<? super AnyType>>
5  void quicksort( AnyType [ ] a )
6  {
7      quicksort( a, 0, a.length - 1 );
8  }
9
10 /**
11  * Método interno de ordenación rápida que hace llamadas recursivas.
12  * Usa particionamiento basado en la mediana de tres y un punto de corte.
13  */
14 private static <AnyType extends Comparable<? super AnyType>>
15 void quicksort( AnyType [ ] a, int low, int high )
16 {
17     if( low + CUTOFF > high )
18         insertionSort( a, low, high );

```

Continúa

Figura 8.22 Rutina de recursión rápida con particionamiento basado en la mediana de tres y punto de corte para matrices de pequeño tamaño.

```

19     else
20         {
21             // Ordenar low, middle, high
22             int middle = ( low + high ) / 2;
23             if( a[ middle ].compareTo( a[ low ] ) < 0 )
24                 swapReferences( a, low, middle );
25             if( a[ high ].compareTo( a[ low ] ) < 0 )
26                 swapReferences( a, low, high );
27             if( a[ high ].compareTo( a[ middle ] ) < 0 )
28                 swapReferences( a, middle, high );
29
30             // Colocar pivot en la posición high - 1
31             swapReferences( a, middle, high - 1 );
32             AnyType pivot = a[ high - 1 ];
33
34             // Comenzar particionamiento
35             int i, j;
36             for( i = low, j = high - 1; ; )
37             {
38                 while( a[ ++i ].compareTo( pivot ) < 0 )
39                     ;
40                 while( pivot.compareTo( a[ --j ] ) < 0 )
41                     ;
42                 if( i >= j )
43                     break;
44                 swapReferences( a, i, j );
45             }
46             // Restaurar pivot
47             swapReferences( a, i, high - 1 );
48             quicksort( a, low, i - 1 ); // Ordenar elementos pequeños
49             quicksort( a, i + 1, high ); // Ordenar elementos grandes
50         }
51     }

```

Figura 8.22 (Continuación).

Las operaciones fundamentales tienen lugar entre las líneas 37 y 40. Las exploraciones están compuestas por operaciones simples: incrementos, accesos a la matriz y comparaciones simples, lo cual es el motivo de que esta ordenación sean tan rápida. Para garantizar que los bucles internos sean compactos y eficientes, tenemos que asegurarnos de que el intercambio de la línea 43 esté formado por las tres asignaciones que esperamos, y no incurra en el coste adicional de efectuar una llamada a método. Por ello, declaramos que `swapReferences` es un método de

El bucle interno de quicksort es muy compacto y eficiente.

La ordenación rápida es un ejemplo clásico de cómo utilizar un análisis como guía para proceder a la implementación de un programa.

tipo `final static` o, en algunos casos, escribimos las tres asignaciones explícitamente (por ejemplo, si el compilador ejerce su derecho de no realizar una optimización en línea).

Aunque el código parece ahora bastante sencillo, eso se debe únicamente al análisis que hemos realizado antes de comenzar con la tarea de codificación. Además, todavía se esconden algunas trampas en el algoritmo (véase el Ejercicio 8.24). La ordenación rápida es un ejemplo clásico de utilización de un análisis como guía para proceder a la implementación de un programa.

## 8.7 Selección rápida

La operación de selección consiste en localizar el  $k$ -ésimo más pequeño de una matriz.

Un problema estrechamente relacionado con el de la ordenación es la *selección*, que es el proceso de localizar el  $k$ -ésimo elemento más pequeño en una matriz de  $N$  elementos. Un caso especial importante es el de encontrar la mediana o el  $N/2$ -ésimo elemento más pequeño. Obviamente, podemos ordenar los elementos, pero como la operación de selección solicita menos información que la de ordenación, cabe esperar que la selección sea un proceso más rápido. Efectivamente, es así. Efectuando un pequeño cambio en la rutina de ordenación rápida podemos resolver el problema de selección en un tiempo lineal como promedio, lo que nos da el algoritmo denominado de selección rápida (*quickselect*). Los pasos de *Quickselect*( $S$ ,  $k$ ) son los siguientes:

1. Si el número de elementos de  $S$  es 1, presumiblemente  $k$  será también 1, por lo que podemos devolver el único elemento de  $S$ .
2. Seleccionar cualquier elemento  $v$  de  $S$ . Será el pivote.
3. Partitionar  $S - \{v\}$  en  $L$  y  $R$ , exactamente igual que para el caso de la ordenación rápida.
4. Si  $k$  es menor o igual que el número de elementos de  $L$ , el elemento que estemos buscando deberá estar en  $L$ . Invocaremos *Quickselect*( $L$ ,  $k$ ) recursivamente. En caso contrario, si  $k$  es exactamente igual a uno más que el número de elementos de  $L$ , el pivote será el  $k$ -ésimo elemento más pequeño, y podemos devolverlo como respuesta. En otro caso, el  $k$ -ésimo elemento más pequeño estará en  $R$  y será el  $(k - |L| - 1)$ -ésimo elemento más pequeño de  $R$ . De nuevo, podemos hacer una llamada recursiva y devolver el resultado.

*Quickselect* se utiliza para realizar una selección. Es similar al algoritmo de ordenación rápida, pero solo realiza una llamada recursiva. El tiempo medio de ejecución es lineal.

*Quickselect* hace una única llamada recursiva, por comparación con las dos que hace la rutina de ordenación rápida. El caso peor del algoritmo de selección rápida es idéntico al de ordenación rápida, que es cuadrático. Se produce cuando una de las llamadas recursivas se realiza sobre un conjunto vacío. En tales casos, el algoritmo de selección rápida no nos permite ahorrar mucho tiempo. Sin embargo, podemos demostrar que el tiempo promedio es lineal, utilizando un análisis similar al empleado para el algoritmo de ordenación rápida (véase el Ejercicio 8.4).

La implementación del algoritmo de selección rápida, mostrada en la Figura 8.23, es más simple que lo que nuestra descripción abstracta parece sugerir. Salvo por el parámetro adicional,  $k$ , y las llamadas recursivas, el algoritmo es idéntico al de ordenación rápida. Cuando termina, el  $k$ -ésimo elemento más pequeño se encontrará en su posición correcta dentro de la matriz. Como la matriz

comienza en el índice 0, el cuarto elemento más pequeño se encontrará en la posición 3. Observe que la ordenación original se verá modificada. Si esto no resulta aceptable, podemos hacer que la rutina de preparación pase como parámetro una copia de la matriz original, en lugar de pasarlá directamente.

La utilización del particionamiento basado en la mediana de tres hace que la probabilidad de encontrarnos en el caso peor sea casi despreciable. Seleccionando cuidadosamente el pivote, podemos garantizar que el caso peor nunca se produzca y que el tiempo de ejecución sea lineal incluso en el caso peor. Sin embargo, el algoritmo resultante tiene exclusivamente un interés teórico, porque la constante que la notación O mayúscula oculta es mucho mayor que la constante obtenida en la implementación normal basada en la mediana de tres.

El algoritmo lineal de caso peor constituye un resultado clásico, aunque resulta poco práctico.

```

1  /**
2   * Método interno de selección que realiza llamadas recursivas.
3   * Usa particionamiento basado en la mediana de tres y un punto de corte.
4   * Coloca el k-ésimo elemento más pequeño en a[k-1].
5   * @param a una matriz de elementos Comparable.
6   * @param low el índice más a la izquierda de la submatriz.
7   * @param high el índice más a la derecha de la submatriz.
8   * @param k el rango deseado (1 es el mínimo) dentro de la matriz global.
9   */
10 private static <AnyType extends Comparable<? super AnyType>>
11 void quickSelect( AnyType [ ] a, int low, int high, int k )
12 {
13     if( low + CUTOFF > high )
14         insertionSort( a, low, high );
15     else
16     {
17         // Ordenar low, middle, high
18         int middle = ( low + high ) / 2;
19         if( a[ middle ].compareTo( a[ low ] ) < 0 )
20             swapReferences( a, low, middle );
21         if( a[ high ].compareTo( a[ low ] ) < 0 )
22             swapReferences( a, low, high );
23         if( a[ high ].compareTo( a[ middle ] ) < 0 )
24             swapReferences( a, middle, high );
25
26         // Colocar pivot en la posición high - 1
27         swapReferences( a, middle, high - 1 );
28         AnyType pivot = a[ high - 1 ];
29

```

Continúa

Figura 8.23 Selección rápida con particionamiento basado en la mediana de tres y punto de corte para matrices de pequeño tamaño.

```

30         // Comenzar el particionamiento
31     int i, j;
32     for( i = low, j = high - 1; ; )
33     {
34         while( a[ ++i ].compareTo( pivot ) < 0 )
35             ;
36         while( pivot.compareTo( a[ --j ] ) < 0 )
37             ;
38         if( i >= j )
39             break;
40         swapReferences( a, i, j );
41     }
42     // Restaurar pivot
43     swapReferences( a, i, high - 1 );
44
45     // Recursión; solo cambia esta parte.
46     if( k <= i )
47         quickSelect( a, low, i - 1, k );
48     else if( k > i + 1 )
49         quickSelect( a, i + 1, high, k );
50     }
51 }

```

Figura 8.23 (Continuación).

## 8.8 Una cota inferior para la ordenación

Aunque disponemos de algoritmos  $O(N \log N)$  para la ordenación, no está claro que esto sea lo mejor que se puede conseguir. En esta sección vamos a demostrar que cualquier algoritmo de ordenación

que solo utilice comparaciones requiere  $\Omega(N \log N)$  comparaciones (y por tanto tiempo) en el caso peor. En otras palabras, *cualquier algoritmo que lleve a cabo la ordenación utilizando comparaciones entre elementos debe al menos emplear aproximadamente  $N \log N$  comparaciones para alguna secuencia de entrada.* Podemos emplear una técnica similar para demostrar que esta condición se cumple en el caso promedio.

Cualquier algoritmo de ordenación basado en comparaciones debe utilizar aproximadamente  $N \log N$  comparaciones como promedio y en el caso peor.

Las demostraciones son abstractas; mostramos cuál es la cota inferior en el caso peor.

¿Debe trabajar todo algoritmo de comparación utilizando comparaciones? La respuesta es no. Sin embargo, los algoritmos que no impliquen el uso de comparaciones generales lo más probable es que solo funcionen para tipos restringidos, como los enteros. Aunque a menudo lo único que necesitaremos ordenar son enteros (véase el Ejercicio 8.19), no podemos realizar esas suposiciones tan generales acerca de la entrada a un algoritmo de ordenación de propósito general. Solo podemos asumir las cosas que sabemos a ciencia cierta; es decir, que debido a que los elementos necesitan ser ordenados, siempre se podrán comparar cualesquiera dos elementos.

A continuación, vamos a demostrar uno de los teoremas más fundamentales en las Ciencias de la computación, el Teorema 8.3. Recuerde primero que el producto de los  $N$  primeros enteros positivos es  $N!$  La demostración es una demostración de existencia, que es bastante abstracta. Demuestra que siempre deberán existir algunas entradas poco adecuadas.

<b>Teorema 8.3</b>	Cualquier algoritmo que realice la ordenación, utilizando exclusivamente comparaciones entre elementos, deberá emplear al menos $\lceil \log(N!) \rceil$ comparaciones para alguna secuencia de entrada.
Demostración	<p>Podemos considerar las posibles entradas como cualquiera de las permutaciones de <math>1, 2, \dots, N</math>, porque solo importa el orden relativo de los elementos de entrada, no sus valores reales. Por tanto, el número de posibles entradas es el número de diferentes permutaciones de <math>N</math> elementos, que es exactamente <math>N!</math> Sea <math>P_i</math> el número de permutaciones que son coherentes con los resultados después de que el algoritmo haya procesado <math>i</math> comparaciones. Sea <math>F</math> el número de comparaciones procesadas cuando la ordenación termine. Sabemos lo siguiente: (a) <math>P_0 = N!</math> porque todas las permutaciones son posibles antes de la primera comparación; (b) <math>P_F = 1</math> porque, si fuera posible más de una permutación, el algoritmo no podría terminar con la seguridad de que ha proporcionado la salida correcta; (c) existe una permutación tal que <math>P_i \geq P_{i-1}/2</math> porque después de una comparación, cada permutación va a uno de dos posibles grupos: el grupo de las permutaciones que siguen siendo posibles y el grupo de las que ya no lo son. El mayor de estos dos grupos deberá tener al menos la mitad de las permutaciones. Por tanto, existe al menos una permutación a la que podemos aplicar esta lógica a lo largo de toda la secuencia de comparaciones. La acción de un algoritmo de ordenación consiste, por tanto, en pasar del estado <math>P_0</math> en el que todas las <math>N!</math> permutaciones son posibles, al estado final <math>P_F</math> en el que solo es posible una permutación, con la restricción de que exista una entrada tal que en cada comparación solo se pueden eliminar la mitad de las permutaciones. Por el principio de división en mitades, sabemos que se requieren al menos <math>\lceil \log(N!) \rceil</math> comparaciones para dicha entrada.</p>

¿Qué valor tiene  $\lceil \log(N!) \rceil$ ? Es aproximadamente  $N \log N - 1,44N$ .

## Resumen

Para la mayoría de las aplicaciones de ordenación interna de carácter general, el método elegido es una ordenación por inserción, una ordenación Shell, una ordenación por mezcla o una ordenación rápida. La decisión respecto a cuál de esos algoritmos utilizar dependerá del tamaño de la entrada y del entorno de computación subyacente.

La ordenación por inserción es apropiada para tamaños de entrada muy pequeños. La ordenación Shell es una buena elección para ordenar cantidades moderadas de entrada. Con una secuencia de incrementos apropiada, proporciona un excelente rendimiento y utiliza solo unas pocas líneas de código. La ordenación por mezcla tiene un rendimiento de caso peor de  $O(N \log N)$ , pero requiere código adicional para evitar parte de las copias adicionales. La ordenación rápida es complicada de codificar. Asintóticamente, tiene casi con seguridad un rendimiento de  $O(N \log N)$  si la imple-

mentación es cuidadosa y hemos demostrado que este resultado es, esencialmente, lo mejor que podemos obtener. En la Sección 21.5 hablaremos de otro tipo muy popular de ordenación interna, la *ordenación por montículos*.

Para probar y comparar las ventajas de los diversos algoritmos de ordenación, tenemos que ser capaces de generar entradas aleatorias. La aleatoriedad es un tema importante en general y hablaremos de él en el Capítulo 9.



## Conceptos clave

**algoritmo de ordenación basado en comparaciones** Un algoritmo que toma decisiones de ordenación basándose exclusivamente en comparaciones. (343)

**demonstración de cota inferior para la ordenación** Confirma que cualquier algoritmo de ordenación basado en comparaciones debe utilizar al menos, aproximadamente,  $N \log N$  comparaciones como promedio y como caso peor. (372)

**inversión** Un par de elementos que están desordenados dentro de una matriz. Este concepto se emplea para medir el grado de desorden. (345)

**ordenación con espaciado decreciente** Otro nombre con el que se designa a la ordenación Shell. (347)

**ordenación por mezcla** Un algoritmo de tipo divide y vencerás que permite realizar una ordenación  $O(N \log N)$ . (350)

**ordenación rápida** Un rápido algoritmo de tipo divide y vencerás, cuando se implementa apropiadamente; en muchas situaciones es el algoritmo de ordenación basado en comparaciones más rápido que se conoce. (355)

**ordenación Shell** Un algoritmo subcuadrático que funciona bien en la práctica y que es fácil de codificar. El rendimiento de la ordenación Shell depende enormemente de la secuencia de incrementos y requiere un análisis bastante complejo (y que todavía no ha sido completamente resuelto). (347)

**partición** El paso de la ordenación rápida que coloca cada elemento, excepto el pivote, en uno de dos grupos, uno compuesto por los elementos menores o iguales que el pivote y otro compuesto por los elementos mayores o iguales que el pivote. (357)

**particionamiento basado en la mediana de tres** Se utiliza como pivote la mediana del primer elemento, del elemento central y del último elemento. Esta técnica simplifica la etapa de particionamiento en el algoritmo de ordenación rápida. (363)

**pivote** En el algoritmo de ordenación rápida un elemento que divide la matriz en dos grupos; uno con los valores más pequeños que el pivote y otro con los valores más grandes que el pivote. (357)

**selección** El proceso de localizar el  $k$ -ésimo elemento más pequeño de una matriz. (370)

**selección rápida** Un algoritmo empleado para realizar una selección y que es similar a la ordenación rápida, pero que solo realiza una llamada recursiva. El tiempo medio de ejecución es lineal. (370)



## Errores comunes

1. Las ordenaciones codificadas en este capítulo comienzan en la posición 0 de la matriz, no en la posición 1.
2. Un error bastante común consiste en utilizar la secuencia de incrementos incorrecta para la ordenación Shell. Asegúrese de que la secuencia de incrementos termine en 1 y evite las secuencias que se sabe que proporcionan un rendimiento inadecuado.
3. La ordenación rápida tiene muchas trampas. Los errores más comunes se refieren a las entradas ordenadas, los elementos duplicados y las particiones degeneradas.
4. Para tamaños de entrada pequeños, la ordenación por inserción resulta apropiada, pero no debe emplearse para grandes tamaños de entrada.



## Internet

Todos los algoritmos de ordenación y una implementación del algoritmo de selección rápida se encuentran en un mismo archivo.

### Duplicate.java

Contiene la rutina de la Figura 8.1 y un programa de prueba.

### Sort.java

Contiene todos los algoritmos de ordenación y el algoritmo de selección.



## Ejercicios

### EN RESUMEN

- 81 Un algoritmo de ordenación es *estable* si los elementos que tienen claves iguales se dejan en el mismo orden que tenían en la entrada. ¿Cuáles de los algoritmos de ordenación de este capítulo son estables y cuáles no? ¿Por qué?
- 82 Explique por qué la versión elaborada de la rutina de ordenación rápida proporcionada en el texto es mejor que permutar aleatoriamente la entrada y seleccionar el elemento central como pivote.
- 83 Ordene la secuencia 6, 0, 9, 3, 0, 5, 7, 3, 1, 5 utilizando
  - a. Ordenación por inserción.
  - b. Ordenación Shell para los incrementos {1, 3, 5}.
  - c. Ordenación por mezcla.
  - d. Ordenación rápida, utilizando el elemento central como pivote y sin punto de corte (muestre todos los pasos).
  - e. Ordenación rápida, utilizando selección de pivote basada en la mediana de tres y un punto de corte de 3.

## EN TEORÍA

- 84** Demuestre que el algoritmo de selección rápida tiene un rendimiento promedio lineal. Hágalo resolviendo la Ecuación 8.5, sustituyendo la constante 2 por 1.
- 85** Demuestre que cualquier algoritmo basado en comparaciones que se utilice para ordenar cuatro elementos requiere al menos cinco comparaciones para alguna entrada. Después, demuestre que existe un algoritmo que ordena cuatro elementos utilizando como mucho cinco comparaciones.
- 86** A la hora de implementar la ordenación rápida, si la matriz contiene muchos duplicados, puede que resulte conveniente realizar una partición en tres subconjuntos (en elementos menores, iguales y mayores que el pivote) y hacer llamadas recursivas más pequeñas. Suponga que se pueden utilizar comparaciones de tres vías.
- Proporcione un algoritmo que realice una partición de tres vías sobre la propia matriz, partiendo de una submatriz de  $N$  elementos utilizando únicamente  $N - 1$  comparaciones de tres vías. Si hay  $d$  elementos iguales al pivote, puede utilizar  $d$  intercambios adicionales Comparable, como adición al algoritmo de particionamiento de dos vías. *Pista:* a medida que  $i$  y  $j$  vayan aproximándose, mantenga los cinco grupos de elementos mostrados.

IGUAL	PEQUEÑO	$i$	DESCONOCIDO	$j$	GRANDE	IGUAL
-------	---------	-----	-------------	-----	--------	-------

- Demuestre que, utilizando el algoritmo del apartado (a), ordenar una matriz de  $N$  elementos que contenga solo  $d$  valores diferentes requiere un tiempo  $\mathcal{O}(dN)$ .
- 87** Cuando la entrada está ordenada en orden inverso, ¿cuál es el tiempo de ejecución de los siguientes algoritmos?
- Ordenación rápida.
  - Ordenación Shell.
  - Ordenación por mezcla.
  - Ordenación por inserción.
- 88** Utilizando la fórmula de Stirling,  $N! \geq (N/e)^N \sqrt{2\pi N}$ , realice una estimación del valor de  $\log(N!)$ .
- 89** Construya una entrada de caso peor para el algoritmo de ordenación rápida con
- El elemento central como pivote.
  - Particionamiento con un pivote basado en la mediana de tres.
- 90** ¿Cuál es el número de comparaciones que la ordenación por mezcla utiliza en el caso peor para ordenar seis números? ¿Resulta esto óptimo?
- 91** La operación de selección se puede resolver con un tiempo lineal de caso peor si se elige el pivote adecuadamente. Suponga que formamos  $N/5$  grupos de 5 elementos y que, para cada grupo, calculamos la mediana. A continuación, utilizamos como pivote la mediana de las  $N/5$  medianas.
- Demuestre que la mediana de 5 elementos se puede obtener con seis comparaciones.

- b. Sea  $T(N)$  el tiempo necesario para resolver el problema de selección con una instancia de  $N$  elementos. ¿Cuál es el tiempo requerido para determinar la mediana de las  $N/5$  medianas? *Pista:* ¿puede utilizarse la recursión para determinar la mediana de las  $N/5$  medianas?
- c. Después de llevar a cabo la etapa de particionamiento, el algoritmo de selección realiza una única llamada recursiva. Demuestre que, si se selecciona como pivote la mediana de las  $N/5$  medianas, el tamaño de la llamada recursiva está limitado a, como mucho,  $7 N/10$ .
- 812** Suponga que ambas matrices  $A$  y  $B$  están ordenadas y contienen  $N$  elementos. Proporcione un algoritmo  $O(\log N)$  para determinar la mediana de  $A \cup B$ .
- 813** Suponga que intercambiamos los elementos  $a[i]$  y  $a[i+k]$  que estaban originalmente desordenados. Demuestre que con ello se eliminan como mínimo 1 inversión y como máximo  $2k - 1$  inversiones.

### EN LA PRÁCTICA

- 814** Cuando todas las claves son iguales, ¿cuál es el tiempo de ejecución de los siguientes algoritmos?
- Ordenación rápida.
  - Ordenación Shell.
  - Ordenación por mezcla.
  - Ordenación por inserción.
- 815** Cuando la entrada está ya ordenada, ¿cuál es el tiempo de ejecución de los siguientes algoritmos?
- Ordenación rápida.
  - Ordenación Shell.
  - Ordenación por mezcla.
  - Ordenación por inserción.
- 816** Una matriz contiene  $N$  números y queremos determinar si dos de los números tienen una suma igual a un cierto número dado  $K$ . Por ejemplo, si la entrada es 8, 4, 1, 6 y  $K$  es 10, la respuesta es sí (4 y 6). Se puede utilizar un mismo número dos veces. Haga lo siguiente:
- Proporcione un algoritmo  $O(N^2)$  para resolver este problema.
  - Proporcione un algoritmo  $O(N \log N)$  para resolver este problema. *Pista:* ordene primero los elementos. Después de eso, puede resolver el problema en tiempo lineal.
- 817** Repita el Ejercicio 8.16 para cuatro números. Trate de diseñar un algoritmo  $O(N^2 \log N)$ . *Pista:* calcule todas las posibles sumas de dos elementos, ordene esas posibles sumas y luego continúe como en el Ejercicio 8.16.
- 818** Repita el Ejercicio 8.16 para la suma de tres números. Trate de diseñar un algoritmo  $O(N^3)$ .

- 8.19** Si tiene más información acerca de los elementos que se están ordenando, podrá ordenarlos en un tiempo lineal. Demuestre que una colección de  $N$  enteros de 16 bits puede ordenarse en un tiempo  $O(N)$ . *Pista:* mantenga una matriz indexada desde 0 a 65.535.
- 8.20** El algoritmo de ordenación rápida proporcionado en el texto utiliza dos llamadas recursivas. Elimine una de las llamadas de la forma siguiente.
- Escriba de nuevo el código de modo que la segunda llamada recursiva sea, incondicionalmente, la última línea de la ordenación rápida. Haga esto invirtiendo el `if/else`, volviendo después de la llamada a `insertionSort`.
  - Elimine la recursión final escribiendo un bucle `while` y alterando `low`.
- 8.21** Continuando con el Ejercicio 8.20, después del apartado (a)
- Realice una comprobación de modo que la submatriz más pequeña sea procesada por la primera llamada recursiva y la submatriz más grande sea procesada por la segunda llamada recursiva.
  - Elimine la recursión final escribiendo un bucle `while` y alterando `low` o `high`, según sea necesario.
- 8.22** Suponga que el algoritmo recursivo de ordenación rápida recibe un parámetro `int`, denominado `depth`, desde la rutina de preparación y cuyo valor inicial es aproximadamente  $2 \log N$ . (`depth` indica la profundidad de recursión.)
- Modifique la rutina recursiva de ordenación rápida para invocar `mergeSort` con la submatriz actual si el nivel de recursión ha alcanzado `depth`. *Pista:* decremente `depth` a medida que vaya haciendo llamadas recursivas; cuando sea 0, cambie a una ordenación por mezcla.
  - Demuestre que el tiempo de ejecución de caso peor de este algoritmo es  $O(N \log N)$ .

## PROYECTOS DE PROGRAMACIÓN

- 8.23** Escriba un método que elimine todos los duplicados de una matriz  $A$  de  $N$  elementos. Devuelve el número de elementos que permanecen en  $A$ . El método debe ejecutarse en un tiempo medio  $O(N \log N)$  –utilice una ordenación rápida como etapa de preprocessamiento– y no debe hacer uso de la API de Colecciones.
- 8.24** Un estudiante modifica la rutina `quicksort` de la Figura 8.22, haciendo los siguientes cambios en las líneas 35 a 40. ¿Es el resultado equivalente a la rutina original?

```

35 for( i = low + 1, j = high - 2; ; )
36 {
37     while( a[ i ] < pivot )
38         i++;
39     while( pivot < a[ j ] )
40         j--;

```

- 8.25** Codifique tanto la ordenación Shell como la ordenación rápida y compare sus tiempos de ejecución. Utilice las mejores implementaciones proporcionadas en el texto y ejecútelas sobre
- Enteros.
  - Números reales de tipo `double`.
  - Cadenas de caracteres.
- 8.26** Compare el rendimiento de la ordenación Shell con diversas secuencias de incremento, de la forma siguiente. Obtenga un tiempo promedio para el tamaño de entrada  $N$ , generando varias secuencias aleatorias de  $N$  elementos. Utilice la misma entrada para todas las secuencias de incremento. En una prueba separada, obtenga el número medio de comparaciones `Comparable` y de asignaciones `Comparable`. Seleccione un número de pruebas repetidas que sea grande, pero que permita completar la tarea en una 1 hora de tiempo de procesador. Las secuencias de incrementos son:
- La secuencia original de Shell (dividir repetidamente por 2).
  - La secuencia original de Shell, sumando 1 si el resultado es par pero distinto de cero.
  - La secuencia de Gonnet mostrada en el texto, con una división repetida por 2,2.
- 8.27** En el algoritmo de ordenación rápida, en lugar de seleccionar tres elementos, como se hace para el particionamiento basado en la mediana de tres, suponga que estamos dispuestos a seleccionar nueve elementos, incluyendo el primero y el último y estando los otros siete equiespaciados dentro la matriz.
- Escriba un código para implementar el particionamiento basado en la mediana de nueve.
  - Considere la siguiente alternativa para la mediana de nueve: agrupe los elementos en tres grupos de tres. Calcule la mediana de cada uno de los tres grupos. A continuación, utilice la mediana de esas medianas. Escriba un código para implementar esta alternativa y compare su rendimiento con el del particionamiento basado en la mediana de nueve.
- 8.28** Escriba una utilidad de ordenación simple, `sort`. El comando `sort` toma como parámetro un nombre de archivo, contenido ese archivo un elemento por línea. De manera predeterminada, las líneas se consideran cadenas de caracteres y están ordenadas de forma lexicográfica normal (diferenciando entre mayúsculas y minúsculas). Añada dos opciones: la opción `-c` significa que la ordenación no debe diferenciar entre mayúsculas y minúsculas; la opción `-in` significa que las líneas deben considerarse enteros, para propósitos de ordenación.
- 8.29** Escriba un programa que lea  $N$  puntos de un plano y proporcione como salida cualquier grupo existente de cuatro o más puntos colineales (es decir, puntos situados sobre una misma línea). El algoritmo obvio de fuerza bruta requeriría un tiempo de  $O(N^4)$ . Sin embargo, hay un algoritmo mejor que hace uso de la ordenación y se ejecuta en un tiempo  $O(N^2 \log N)$ .

- 8.30** En el Ejercicio 8.1 hablábamos de la ordenación estable. Escriba un método que realice una ordenación rápida estable. Para ello, cree una matriz de objetos; cada objeto servirá para almacenar un elemento de datos, junto con su posición inicial en la matriz. (Este es el patrón compuesto; consulte la Sección 3.9.) Después, ordene la matriz; si dos objetos tienen elementos de datos idénticos, utilice la posición inicial para deshacer el empate. Después de haber ordenado la matriz de objetos, reordene la matriz original.
- 8.31** Escriba un método que admita una matriz de valores `String` y devuelva el grupo más grande de palabras que sean anagramas unas de otras. Para hacer esto, ordene la matriz con un `Comparador` que compare las representaciones de las distintas palabras con sus caracteres ordenados. Después de la ordenación, los grupos de palabras que sean anagramas unas de otras, estarán adyacentes en la matriz. Compruebe su método escribiendo un programa que utilice las palabras leídas desde un archivo.
- 8.32** Dos palabras son anagramas si contienen las mismas letras y con la misma frecuencia. Por ejemplo, `stale` y `least` son anagramas una de otra. Una forma simple de comprobarlo es ordenar los caracteres de cada palabra; si se obtiene la misma respuesta (en el ejemplo, se obtendría `aelst`), las palabras son anagramas una de otra. Escriba un método que comprueba si dos palabras son anagramas una de otra.



## Referencias

La referencia clásica para los algoritmos de ordenación es [5]. Otra referencia es [3]. El algoritmo de ordenación Shell apareció por primera vez en [8]. Un estudio empírico de su tiempo de ejecución se realizó en [9]. La ordenación rápida fue descubierta por Hoare [4]; este artículo incluye también el algoritmo de selección rápida y detalla muchos problemas de implementación importantes. Un estudio exhaustivo del algoritmo de ordenación rápida, incluyendo el análisis de la variante basada en la mediana de tres aparece en [7]. En [1] se presenta una implementación C detallada que incluye mejoras adicionales. El Ejercicio 8.22 está basado en [6]. La cota inferior  $\Omega(N \log N)$  para las ordenaciones basadas en comparaciones está tomada de [2]. La presentación de la ordenación Shell está adaptada de [10].

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<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1991.
4. C. A. R. Hoare, "Quicksort", *Computer Journal* 5 (1962), 10–15.

- 5 D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1998.
- 6 D. R. Musser, "Introspective Sorting and Selection Algorithms", *Software—Practice and Experience* 27 (1997), 983–993.
- 7 R. Sedgewick, *Quicksort*, Garland, New York, 1978. (Originalmente presentado como las tesis doctoral del autor, Stanford University, 1975.)
- 8 D. L. Shell, "A High-Speed Sorting Procedure," *Communications of the ACM* 2 7 (1959), 30–32.
- 9 M. A. Weiss, "Empirical Results on the Running Time of Shellsort", *Computer Journal* 34 (1991), 88–91.
- 10 M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice Hall, Upper Saddle River, NJ, 1995.



## Aleatorización

Muchas situaciones en el ámbito de la informática requieren el uso de números aleatorios. Por ejemplo, la moderna criptografía, los sistemas de simulación y, sorprendentemente, incluso los algoritmos de búsqueda y ordenación, dependen de los generadores de números aleatorios. A pesar de ello, los buenos generadores de números aleatorios son difíciles de implementar. En este capítulo vamos a hablar de la generación y uso de números aleatorios.

En este capítulo, veremos

- Cómo se generan los números aleatorios.
- Cómo se generan permutaciones aleatorias.
- Cómo los números aleatorios permiten diseñar algoritmos eficientes, utilizando una técnica general conocida como *algoritmo aleatorizado*.

### 9.1 ¿Por qué necesitamos números aleatorios?

Los números aleatorios se utilizan en muchas aplicaciones. En esta sección vamos a explicar algunas de las más comunes.

Una importante aplicación es en el campo de las pruebas de programas. Suponga, por ejemplo, que queremos probar si uno de los algoritmos de ordenación escritos en el Capítulo 8 funciona realmente. Por supuesto, podríamos proporcionar una pequeña cantidad de datos de entrada, pero si queremos probar esos algoritmos para los grandes conjuntos de datos para los que han sido diseñados, necesitamos una gran cantidad de información de entrada. Proporcionar datos ordenados como entrada permite comprobar un caso, pero sería preferible realizar pruebas más convincentes. Por ejemplo, nos gustaría poder probar el programa ejecutando, quizás, 5.000 ordenaciones para entradas de tamaño 1.000. Para hacer esto, necesitamos una rutina que genere los datos de prueba, lo que a su vez requiere el uso de números aleatorios.

Los números aleatorios tienen usos muy importantes, incluyendo la criptografía, la simulación y la prueba de programas.

Una vez que disponemos de las entradas compuestas por números aleatorios, ¿cómo sabemos si el algoritmo de ordenación funciona? Una prueba consiste en determinar si la ordenación ha colocado la matriz en un orden no decreciente. Obviamente, podemos ejecutar esta prueba mediante una exploración secuencial en un tiempo lineal. ¿Pero cómo sabemos que los elementos presentes después de la ordenación son los mismos que los que había antes de la ordenación? Un método

Una permutación de 1, 2, ...,  $N$  es una secuencia de  $N$  enteros que incluye exactamente una vez cada uno de los valores 1, 2, ...,  $N$ .

consiste en fijar los elementos en una disposición de 1, 2, ...,  $N$ . En otras palabras, comenzamos con una permutación aleatoria de los primeros  $N$  enteros. Una *permutación* de 1, 2, ...,  $N$  es una secuencia de  $N$  enteros que incluye a cada uno de los valores 1, 2, ...,  $N$  exactamente una vez. Entonces, independientemente de la permutación con la que empecemos, el resultado de la ordenación será la secuencia 1, 2, ...,  $N$ , que también se puede comprobar fácilmente.

Además de ayudarnos a generar datos de prueba para verificar la corrección del programa, los números aleatorios son útiles para comparar el rendimiento de diversos algoritmos. La razón es que, de nuevo, pueden emplearse para proporcionar un gran número de entradas.

Otro uso de los números aleatorios es en el campo de las simulaciones. Si queremos conocer el tiempo medio requerido en la prestación de un servicio (por ejemplo, el servicio de cajero en un banco) a la hora de procesar una secuencia de solicitudes, podemos modelar el sistema en una computadora. En esta simulación por computadora generaremos la secuencia de solicitudes mediante números aleatorios.

Otro uso más de los números aleatorios es dentro de la técnica general denominada *algoritmo aleatorizado*, en el que se utiliza un número aleatorio para determinar el siguiente paso ejecutado en el algoritmo. El tipo más común de algoritmo aleatorizado implica seleccionar entre varias alternativas posibles, que son más o menos indistinguibles entre sí. Por ejemplo, en un programa informático comercial para jugar al ajedrez, la computadora selecciona normalmente su primer movimiento de forma aleatoria, en lugar de jugar de manera determinista (en lugar de realizar siempre el mismo movimiento). En este capítulo, vamos a echar un vistazo a varios problemas que pueden resolverse de forma más eficiente utilizando un algoritmo aleatorizado.

## 9.2 Generadores de números aleatorios

Los números pseudoaleatorios tienen muchas propiedades de los números aleatorios. Los generadores de números aleatorios buenos son difíciles de encontrar.

¿Cómo se generan los números aleatorios? La verdadera aleatoriedad es imposible de conseguir en una computadora, porque cualquier número obtenido dependerá del algoritmo empleado para generarlo y, por tanto, no puede llegar a ser verdaderamente aleatorio. Generalmente, basta con producir *números pseudoaleatorios*, es decir, números que *parezcan* ser aleatorios, porque satisfagan muchas de las propiedades de los números aleatorios. Pero producir estos elementos es mucho más difícil que describirlos.

Suponga que necesitamos simular el lanzamiento de una moneda. Una forma de hacerlo consiste en examinar el reloj del sistema. Supuestamente, el reloj del sistema mantiene el número de segundos como parte de la hora actual. Si este número es par, podemos devolver 0 para indicar que ha salido cara; si es impar podemos devolver 1 (para indicar que ha salido cruz). El problema es que esta estrategia no funciona bien si necesitamos generar una secuencia de números aleatorios. Un segundo es mucho tiempo de proceso y el reloj podría no llegar a cambiar en absoluto mientras el programa se está generando, lo que daría como resultado una secuencia de todo ceros o todo unos, lo cual difícilmente puede constituir una secuencia aleatoria. Incluso si registráramos el tiempo en unidades de microsegundos (o más pequeñas) y dejáramos al programa ejecutarse, la secuencia de números generada distaría mucho de ser aleatoria, porque el tiempo entre llamadas sucesivas al generador sería prácticamente idéntico en cada invocación del programa.

Lo que realmente necesitamos es una *secuencia* de números pseudoaleatorios, es decir, una secuencia con las mismas propiedades que una secuencia aleatoria. Suponga que queremos obtener números aleatorios comprendidos entre 0 y 999 y distribuidos uniformemente. En una *distribución uniforme*, todos los números dentro del rango especificado tienen la misma probabilidad de aparecer. También se emplean ampliamente otras distribuciones. El esqueleto de clase mostrado en la Figura 9.1 también da soporte a varias distribuciones distintas y algunos de los métodos básicos son idénticos a los de la clase `java.util.Random`. La mayoría de las distribuciones pueden obtenerse a partir de la distribución uniforme, por lo que es esta la que vamos a considerar en primer lugar. Si la secuencia 0, ..., 999 es una distribución uniforme, se cumplirán las siguientes propiedades:

- El primer número tiene una probabilidad igual de ser 0, 1, 2, ..., 999.
- El  $i$ ésimo número tiene una probabilidad igual de ser 0, 1, 2, ..., 999.
- La media esperada de todos los números generados es 499,5.

Estas propiedades no son particularmente restrictivas. Por ejemplo, podríamos generar el primer número examinando un reloj del sistema que tuviera una precisión de milisegundos y luego empleando el número de milisegundos. Podríamos generar los números posteriores sumando 1 al número anterior, etc. Claramente, después de haber generado 1.000 números, todas las propiedades anteriores se cumplen. Sin embargo, otras propiedades más restrictivas no se cumplen.

Dos propiedades más restrictivas que se cumplirían para números aleatorios uniformemente distribuidos son las siguientes:

- La suma de dos números aleatorios consecutivos tiene la misma probabilidad de ser par o impar.
- Si se generan 1.000 números aleatoriamente, algunos estarán duplicados. (Aproximadamente 368 números no aparecerían.)

Nuestros números no satisfacen estas propiedades. Los números consecutivos siempre dan un resultado impar, y nuestra secuencia está libre de duplicados. Decimos entonces que nuestro generador simple de números pseudoaleatorios ha fallado dos pruebas estadísticas. Todos los generadores de números aleatorios fallan en algunas pruebas estadísticas, pero los buenos generadores fallan en menos pruebas que los que son malos (vea en el Ejercicio 9.17 una prueba estadística común).

En esta sección vamos a describir el generador uniforme más simple que supera un número razonable de pruebas estadísticas. No queremos decir con ello que sea el mejor generador. Sin embargo, resulta adecuado para su uso en aplicaciones en las que sea aceptable disponer de una buena aproximación a una secuencia aleatoria. El método utilizado es el generador de congruencia lineal, que fue descrito por primera vez en 1951. El *generador de congruencia lineal* es un buen algoritmo para generar distribuciones uniformes. Es un generador de números aleatorios en el que se generan números  $X_1, X_2, \dots$  que satisfacen

En una *distribución uniforme*, todos los números dentro del rango especificado tienen la misma probabilidad de aparecer.

Tipicamente, se requiere una secuencia aleatoria, y no solo un número aleatorio.

El generador de congruencia lineal es un buen algoritmo para generar distribuciones uniformes.

```

1 package weiss.util;
2
3 // Clase Random
4 //
5 // CONSTRUCCIÓN: con (a) sin inicializador o (b) un entero
6 // que especifica el estado inicial del generador.
7 // Este generador de números aleatorios solo tiene realmente 31 bits,
8 // por lo que es más débil que el de java.util.
9 //
10 // *****OPERACIONES PÚBLICAS*****
11 // Devuelve un número aleatorio de acuerdo con una cierta distribución:
12 // int nextInt( )                                     --> Uniforme, [1 a 2^31-1)
13 // double nextDouble( )                                --> Uniforme, (0 a 1)
14 // int nextInt( int high )                            --> Uniforme [0..high)
15 // int nextInt( int low, int high )                  --> Uniforme [low..high]
16 // int nextPoisson( double expectedVal )           --> Poisson
17 // double nextNegExp( double expectedVal )          --> Exponencial negativa
18 // void permute( Object [ ] a )                     --> Permutar aleatoriamente
19
20 /**
21 * Clase de números aleatorios utilizando un
22 * generador de congruencia lineal de 31 bits.
23 */
24 public class Random
25 {
26     public Random( )
27     { /* Figura 9.2 */ }
28     public Random( int initialValue )
29     { /* Figura 9.2 */ }
30     public int nextInt( )
31     { /* Figura 9.2 */ }
32     public int nextInt( int high )
33     { /* Implementation in online code. */ }
34     public double nextDouble( )
35     { /* Implementación en el código en linea. */ }
36     public int nextInt( int low, int high )
37     { /* Implementación en el código en linea. */ }
38     public int nextPoisson( double expectedValue )
39     { /* Figura 9.4 */ }
40     public double nextNegExp( double expectedValue )
41     { /* Figura 9.5 */ }
42     public static final void permute( Object [ ] a )
43     { /* Figura 9.6 */ }

```

*Continúa***Figura 9.1** Esqueleto de la clase Random que genera números aleatorios.

```

44     private void swapReferences( Object [ ] a, int i, int j )
45     { /* Implementación en el código en línea. */ }
46
47     private int state;
48 }

```

Figura 9.1 (Continuación).

$$X_{i+1} = AX_i \pmod{M} \quad (9.1)$$

La Ecuación 9.1 establece que podemos obtener el  $(i + 1)$ -ésimo número multiplicando el  $i$ -ésimo número por una constante  $A$  y calculando el resto de dividir el resultado por  $M$ . En Java tendríamos

```
x[ i + 1 ] = A * x[ i ] % M
```

Enseguida especificaremos las constantes  $A$  y  $M$ . Observe que todos los números generados serán menores que  $M$ . Para iniciar la secuencia es necesario proporcionar un cierto valor  $X_0$ . Este valor inicial del generador de números aleatorios se denomina *semilla*. Si  $X_0 = 0$ , la secuencia no es aleatoria, porque genera todo ceros. Pero si se seleccionan cuidadosamente  $A$  y  $M$ , cualquier otra semilla que satisfaga  $1 \leq X_0 < M$  será perfectamente válida. Si  $M$  es primo,  $X_i$  nunca será 0. Por ejemplo, si  $M = 11$ ,  $A = 7$  y la semilla  $X_0 = 1$ , los números generados son

$$7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2, \dots \quad (9.2)$$

La semilla es el valor inicial del generador de números aleatorios.

Generar un número una segunda vez implica que la secuencia se repite. En nuestro caso, la secuencia se repite después de  $M - 1 = 10$  números. La longitud de una secuencia hasta que se repite un número se denomina *periodo* de la secuencia. El periodo obtenido con esta elección de  $A$  es, claramente, el menor posible, porque se generan todos los números menores que  $M$  que son distintos de cero. (Forzosamente, se generará un número repetido en la undécima iteración.)

Si  $M$  es primo, son varios los valores de  $A$  que nos dan un periodo completo de longitud  $M - 1$ , y este tipo de generador de números aleatorios se denomina *generador de congruencia lineal de periodo completo*. Algunos valores de  $A$  no dan un periodo completo. Por ejemplo, si  $A = 5$  y  $X_0 = 1$ , la secuencia tiene un periodo corto de solo cinco números:

$$5, 3, 4, 9, 1, 5, 3, 4, \dots \quad (9.3)$$

La longitud de la secuencia hasta que un número se repite se denomina periodo. Un generador de números aleatorios con periodo  $P$  genera la misma secuencia de números después de  $P$  iteraciones.

Un generador de congruencia lineal de periodo completo tiene un periodo igual a  $M - 1$ .

Si seleccionamos  $M$  de forma que sea un número primo de 31 bits, el periodo debería ser significativamente mayor para muchas aplicaciones. El número primo de 31 bits  $M = 2^{31} - 1 = 2.147.483.647$  es una elección bastante común. Para este número primo,  $A = 48.271$  es uno de los muchos valores que proporciona un generador de congruencia lineal de periodo completo. Su uso ha sido bien estudiado y está recomendado por expertos en la materia. Como veremos posteriormente en el capítulo, tratar de experimentar con los generadores de números aleatorios suele tener como

resultado que se rompa la aleatoriedad, por lo que le recomendamos que se ajuste a esta fórmula a menos que se diga lo contrario.

Implementar esta rutina parece bastante simple. Si `state` representa el último valor calculado por la rutina `nextInt`, el nuevo valor de `state` estará dado por

```
state = ( A * state ) % M; // Incorrecto
```

Debido al desbordamiento,  
debemos reordenar los  
cálculos.

Lamentablemente, si este cálculo se realiza con enteros de 32 bits, la multiplicación provocará un desbordamiento con toda seguridad. Aunque Java proporciona un tipo `long` de 64 bits, utilizarlo es más costoso en tiempo de procesamiento que trabajar con valores `int`; además, no todos los lenguajes soportan operaciones matemáticas con 64 bits, e incluso aunque

soportaran esas operaciones, lo único que significaría es que a alguien se le ocurriría emplear un valor de  $M$  mayor. Posteriormente en esta sección utilizaremos un valor  $M$  de 48 bits, pero por el momento vamos a ceñirnos a la aritmética de 32 bits. Si nos limitamos a utilizar el tipo de dato `int` de 32 bits, podríamos pensar que el resultado del desbordamiento forma parte de la aleatoriedad. Sin embargo, el desbordamiento es inaceptable, porque si se produce, ya no tendríamos la garantía de tener un periodo completo. Una pequeña reordenación nos permite realizar los cálculos sin peligro de desbordamiento. Específicamente, si  $Q$  y  $R$  son el cociente y el resto de  $M/A$ , entonces podemos escribir de nuevo la Ecuación 9.1 como

$$X_{i+1} = A(X_i \bmod Q) - R \lfloor X_i / Q \rfloor + M\delta(X_i) \quad (9.4)$$

cumpliéndose las siguientes condiciones (véase el Ejercicio 9.7).

- El primer término siempre puede evaluarse sin desbordamiento.
- El segundo término se puede evaluar sin desbordamiento si  $R < Q$ .
- $\delta(X_i)$  se evalúa como 0 si el resultado de la resta de los dos primeros términos es positivo y se evalúa como 1 si el resultado de la resta es negativo.

Utilice estos valores  
mientras que no se indique  
lo contrario.

Para los valores indicados de  $M$  y  $A$ , tenemos que  $Q = 44.488$  y  $R = 3.399$ . En consecuencia,  $R < Q$  y una aplicación directa de la fórmula nos permite obtener una implementación de la clase `Random` para generar números aleatorios. El código resultante se muestra en la Figura 9.2. La clase funciona siempre que `int` sea capaz de almacenar el valor  $M$ . La rutina `nextInt` devuelve el valor del estado.

En el esqueleto proporcionado en la Figura 9.1 se incluyen varios otros métodos adicionales. Uno de ellos genera un número aleatorio real en el intervalo abierto que va de 0 a 1, mientras que otro genera un entero aleatorio perteneciente a un intervalo cerrado especificado (véase el código en línea).

Por último, la clase `Random` proporciona un generador para números aleatorios con distribución no uniforme, por si acaso lo necesitamos. En la Sección 9.3 proporcionaremos la implementación de los métodos `nextPoisson` y `nextNegExp`.

Podría parecer que podemos obtener un mejor generador de números aleatorios añadiendo una constante a la ecuación. Por ejemplo, podríamos concluir que

$$X_{i+1} = (48.271X_i + 1) \bmod(2^{31} - 1)$$

```
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   * Construir este objeto Random obteniendo el
8   * reloj inicial a partir del reloj del sistema.
9   */
10 public Random( )
11 {
12     this( (int) ( System.nanoTime( ) % Integer.MAX_VALUE ) );
13 }
14
15 /**
16  * Construir este objeto Random con un
17  * estado inicial especificado
18  * @param initialValue el estado inicial.
19  */
20 public Random( int initialValue )
21 {
22     if( initialValue < 0 )
23     {
24         initialValue += M;
25         initialValue++;
26     }
27
28     state = initialValue;
29     if( state <= 0 )
30         state = 1;
31 }
32
33 /**
34  * Devolver un valor int pseudoaleatorio y cambiar el
35  * estado interno.
36  * @return el valor int pseudoaleatorio.
37  */
38 public int nextInt( )
39 {
40     int tmpState = A * ( state % Q ) - R * ( state / Q );
41     if( tmpState >= 0 )
42         state = tmpState;
43     else
44         state = tmpState + M;
45
46     return state;
47 }
```

Figura 9.2 Generador de números aleatorios que funciona si INT\_MAX es al menos  $2^{31} - 1$ .

sería de alguna forma más aleatorio. Sin embargo, cuando utilizamos esta ecuación, vemos que

$$(48.271 \cdot 179.424.105 + 1) \bmod (2^{31} - 1) = 179.424.105$$

Por tanto, si la semilla es 179.424.105, el generador se ve atrapado en un ciclo de periodo 1, lo que ilustra lo frágiles que son estos generadores.

El lector podría sentirse tentado a suponer que todas las máquinas disponen de una generador de números aleatorios al menos tan bueno como el mostrado en la Figura 9.2. Desafortunadamente, no es así. Muchas librerías tienen generadores basados en la función

$$X_{i+1} = (AX_i + C) \bmod 2^B$$

donde  $B$  se elige de forma que se ajuste al número de bits que componen un valor entero en esa máquina, y  $C$  es impar. Estas librerías, al igual que la rutina `nextInt` de la Figura 9.2, también devuelven el nuevo valor calculado `state` directamente, en lugar de devolver, por ejemplo, un valor comprendido entre 0 y 1. Lamentablemente, estos generadores siempre producen valores de  $X$ , que alternan entre par e impar –lo que constituye obviamente una propiedad deseable. De hecho, los  $k$  bits de menor peso se repiten cíclicamente con un periodo de  $2^k$  (como mucho). Muchos otros generadores de números aleatorios tienen ciclos mucho más pequeños que el que nosotros hemos proporcionado. Estos generadores no son adecuados para ninguna aplicación que requiera secuencias largas de números aleatorios. La librería Java tiene un generador de esta forma. Sin embargo, emplea un generador de congruencia lineal de 48 bits y devuelve solo los 32 bits más altos, evitando así el problema de la repetición cíclica de los bits de menor peso. Las constantes son  $A = 25.214.903.917$ ,  $B = 48$  y  $C = 11$ .

Este generador forma también la base de la rutina `drand48` utilizada en las librerías de C y C++. Puesto que Java proporciona valores enteros de 64 bits de longitud, implementar un generador básico de números aleatorios de 48 bits en lenguaje Java estándar es algo que se puede ilustrar con una sola página de código. Es algo mas lento que el generador de números aleatorios de 31 bits, pero no mucho más, y proporciona un periodo de secuencia significativamente mayor. La Figura 9.3 muestra una implementación razonable de este generador de números aleatorios.

Las líneas 10-13 muestran las constantes básicas del generador de números aleatorios. Dado que  $M$  es una potencia de 2, podemos utilizar operadores de bit (véase el Apéndice C para conocer más

```

1 package weiss.util;
2
3 /**
4  * Clase de números aleatorios, utilizando un generador
5  * de congruencia lineal de 48 bits.
6  * @author Mark Allen Weiss
7  */
8 public class Random48
9 {
10    private static final long A = 25214903917L;
```

Continúa

Figura 9.3 Generador de números aleatorios de 48 bits.

```
11  private static final long B = 48;
12  private static final long C = 11;
13  private static final long M = (1L<<B);
14  private static final long MASK = M-1;
15
16  public Random48( )
17      { this( System.nanoTime( ) ); }
18
19  public Random48( long initialValue )
20      { state = initialValue & MASK; }
21
22  public int nextInt( )
23      { return next( 32 ); }
24
25  public int nextInt( int N )
26      { return (int) ( Math.abs( nextLong( ) ) % N ); }
27
28  public double nextDouble( )
29      { return (((long) (next( 26 )) << 27 ) + next( 27 )) / (double)( 1L<< 53); }
30
31  public long nextLong( )
32      { return ( (long) ( next( 32 ) ) << 32 ) + next( 32 ); }
33
34 /**
35  * Devolver el número especificado de bits aleatorios
36  * @param bits número de bits que hay que devolver
37  * @return los bits aleatorios especificados
38  * @throws IllegalArgumentException si bits es mayor que 32
39  */
40  private int next( int bits )
41  {
42      if( bits <= 0 || bits > 32 )
43          throw new IllegalArgumentException( );
44
45      state = ( A * state + C ) & MASK;
46
47      return (int) ( state >>> ( B - bits ) );
48  }
49
50  private long state;
51 }
```

Figura 9.3 (Continuación).

detalles acerca de estos operadores). Se puede calcular  $M = 2^B$  mediante un desplazamiento de bits, y en lugar de utilizar el operador módulo %, podemos emplear el operador and bit a bit. Esto se debe a que `MASK=M-1` está compuesto por los 48 bits de menor peso todos puestos a 1, y un operador and bit a bit con `MASK` tiene el efecto de proporcionar un resultado de 48 bits.

La rutina `next` devuelve un número especificado (como mucho 32) de bits aleatorios a partir del estado calculado, utilizando los bits de mayor peso que son más aleatorios que los de menor peso. La línea 45 es una aplicación directa de la fórmula de congruencia lineal previamente enunciada, mientras que la línea 47 es un desplazamiento de bits (rellenando con cero los bits de mayor peso para evitar los números negativos). La rutina de cero parámetros `nextInt` obtiene 32 bits; la rutina `nextLong` obtiene 64 bits en dos llamadas separadas; `nextDouble` obtiene 53 bits (que representan la mantisa; los otros 11 bits de un valor `double` representan el exponente) también mediante dos llamadas separadas; y la rutina `nextInt` de un parámetro utiliza un operador módulo para obtener un número pseudoaleatorio dentro del rango especificado. Los ejercicios sugieren algunas mejoras posibles para la rutina `nextInt` de un solo parámetro cuando el parámetro `N` es una potencia de 2.

El generador de números aleatorios de 48 bits (e incluso el generador de 31 bits) resulta bastante adecuado para muchas aplicaciones, es simple de implementar utilizando aritmética de 64 bits y consume poco espacio. Sin embargo, los generadores de congruencia lineal son inadecuados para algunas aplicaciones, como por ejemplo la criptografía, o en simulaciones que requieran grandes cantidades de números aleatorios extremadamente independientes e incorrelados.

### 9.3 Números aleatorios no uniformes

No todas las aplicaciones requieren números aleatorios distribuidos uniformemente. Por ejemplo, las notas de los estudiantes de un curso con un gran número de alumnos no están, por regla general, distribuidas uniformemente. En lugar de ello, satisfacen la clásica distribución con una curva de campana, que se conoce de manera más formal con el nombre de *distribución normal* o *distribución gaussiana*. Puede utilizarse un generador de números aleatorios uniforme para generar números aleatorios que satisfagan otras distribuciones.

La distribución de Poisson modela el número de veces que se produce un suceso ocasional; este tipo de distribución se emplea en simulaciones.

Una distribución no uniforme importante que se presenta en las simulaciones es la *distribución de Poisson*, que modela el número de apariciones de un suceso que se produce raramente. Los sucesos que tienen lugar en las siguientes circunstancias satisfacen la distribución de Poisson:

1. La probabilidad de que se produzca un suceso dentro de una región de pequeño tamaño es proporcional al tamaño de la región.
2. La probabilidad de que se produzcan dos sucesos en una región de pequeño tamaño es proporcional al cuadrado del tamaño de la región y es, usualmente, lo suficientemente pequeña como para despreciarla.
3. El suceso consistente en obtener  $k$  sucesos en una región y el suceso consistente en obtener  $j$  sucesos en otra región disjunta con la primera son independientes. (Técnicamente, esta afirmación quiere decir que podemos obtener la probabilidad de que ambos sucesos se produzcan simultáneamente multiplicando la probabilidad de los sucesos individuales.)
4. El número de sucesos en una región de cierto tamaño es conocido.

Billetes premiados	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 primitiva si el número esperado de ganadores es 2.

Si el número medio de veces que se produce un suceso es  $a$ , la probabilidad de que se produzca exactamente  $k$  veces es  $a^k e^{-a} / k!$

La distribución de Poisson se aplica generalmente a sucesos que tienen una baja probabilidad individual. Por ejemplo, considere el suceso de adquirir un boleto de lotería primitiva ganador, siendo la probabilidad de ganar el premio igual a 1 entre 14.000.000. Presumiblemente, los números elegidos son más o menos aleatorios e independientes. Si una persona compra 100 boletos, la probabilidad de ganar será ahora de 1 entre 140.000 (la probabilidad se multiplica por un factor de 100), por lo que se cumple la condición 1. La probabilidad de que esa persona tenga dos boletos premiados es despreciable, por lo que se cumple la condición 2. Si alguna otra persona compra 10 boletos, la probabilidad de que esa persona gane es de 1 entre 1.400.000, y esa probabilidad es independiente de la probabilidad de que gane la primera persona, por lo que la condición 3 también se cumple. Suponga que se venden 28.000.000 de boletos. El número medio de boletos ganadores en esta situación será 2 (el número que necesitamos para la condición 4). El número real de boletos ganadores será una variable aleatoria con un valor esperado igual a 2 y que satisface la distribución de Poisson. Por tanto, la probabilidad de que se hayan vendido exactamente  $k$  boletos ganadores es  $2^k e^{-2} / k!$ , lo que nos da la distribución mostrada en la Figura 9.4. Si el número esperado de ganadores es la constante  $a$ , la probabilidad de que haya  $k$  boletos ganadores es  $a^k e^{-a} / k!$

Para generar un entero aleatorio sin signo de acuerdo con una distribución de Poisson que tenga un valor esperado igual a  $a$ , podemos adoptar la siguiente estrategia (cuya justificación matemática queda fuera del alcance de este libro): generamos repetidamente números aleatorios uniformemente distribuidos en el intervalo (0, 1) hasta que su producto sea menor o igual que  $e^{-a}$ . El código mostrado en la Figura 9.5 hace simplemente eso, utilizando una técnica matemáticamente

```

1  /**
2   * Devolver un int usando una distribución de Poisson y
3   * cambiar el estado interno.
4   * @param expectedValue la media de la distribución.
5   * @return el valor int psudoaleatorio.
6   */
7  public int nextPoisson( double expectedValue )
8  {
9      double limit = -expectedValue;
10     double product = Math.log( nextDouble( ) );
11     int count;
12
13     for( count = 0; product > limit; count++ )
14         product += Math.log( nextDouble( ) );
15
16     return count;
17 }

```

Figura 9.5 Generación de un número aleatorio de acuerdo con la distribución de Poisson.

La distribución exponencial negativa tiene la misma media y varianza. Se utiliza para modelar el tiempo transcurrido entre los instantes en que se producen sucesos aleatorios.

equivalente que es menos susceptible de desbordamiento. El código suma el logaritmo de los números aleatorios uniformemente distribuidos hasta que su suma es menor o igual que  $-a$ .

Otra distribución importante no uniforme es la *distribución exponencial negativa*, mostrada en la Figura 9.6, que tiene la misma media y varianza, y que se utiliza para modelar el tiempo transcurrido entre los instantes en que se producen sucesos aleatorios. Emplearemos este modelo en la aplicación de simulación de la Sección 13.2.

Hay otras muchas distribuciones que se utilizan de manera común. Nuestro principal objetivo aquí es mostrar que la mayoría se pueden generar a partir de la distribución uniforme. Consulte cualquier libro sobre probabilidad y estadística para obtener más información acerca de estas funciones.

## 9.4 Generación de una permutación aleatoria

Considere el problema de simular un juego de cartas. La baraja francesa consta de 52 cartas diferentes, y al repartir cartas para jugar una mano debemos extraerlas del mazo sin duplicados. En la práctica, necesitamos barajar las cartas y luego iterar a través del mazo. Queremos además que la operación de barajado sea equitativa; es decir, que cada una de las 52! posibles ordenaciones del mazo sea igualmente probable como resultado de la operación de barajado.

Una permutación aleatoria puede generarse en un tiempo lineal utilizando un número aleatorio por cada elemento.

Este tipo de problema implica el uso de una *permutación aleatoria*. En general, el problema consiste en generar una permutación aleatoria de 1, 2, ...,  $N$ , debiendo ser todas las permutaciones igualmente probables. La aleatoriedad de la permutación aleatoria está, por supuesto, limitada por la aleatoriedad del generador de números pseudoaleatorios. Por tanto, el que todas las permutaciones sean equiprobables depende de que todos los números aleatorios estén uniformemente distribuidos y sean independientes.

Vamos a demostrar que las permutaciones aleatorias pueden generarse en un tiempo lineal, utilizando un número aleatorio por cada elemento.

En la Figura 9.7 se muestra una rutina, *permute*, para generar una permutación aleatoria. El bucle realiza un barajado aleatorio. En cada iteración del bucle, intercambiamos  $a[j]$  con algún elemento de la matriz en las posiciones 0 a  $j$  (es posible no realizar ninguna permutación).

```

1  /**
2   * Devolver un valor double usando una distribución exponencial
3   * negativa y cambiar el estado interno.
4   * @param expectedValue la media de la distribución.
5   * @return el valor double pseudoaleatorio.
6   */
7  public double nextNegExp( double expectedValue )
8  {
9      return -expectedValue * Math.log( nextDouble( ) );
10 }
```

Figura 9.6 Generación de un número aleatorio de acuerdo con la distribución exponencial negativa.

```

1  /**
2   * Reorganizar aleatoriamente una matriz.
3   * Los números aleatorios usados dependen de la fecha y la hora.
4   * @param a la matriz.
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         swapReferences( a, j, r.nextInt( 0, j ) );
12 }

```

Figura 9.7 Una rutina de permutación.

Claramente, `permute` genera permutaciones a partir de una ordenación original. ¿Pero son todas las permutaciones igualmente probables? La respuesta es a la vez sí y no. La respuesta basada en el algoritmo es sí. Hay  $M$  posibles permutaciones y el número de diferentes resultados de las  $N - 1$  llamadas a `nextInt` en la línea 11 es también  $M$ . La razón es que la primera llamada produce 0 o 1, así que tiene dos posibles resultados. La segunda llamada produce 0, 1 o 2, por lo que tiene tres resultados. La última llamada tiene  $N$  resultados. El número total de resultados es el producto de todas estas posibilidades, porque cada número aleatorio es independiente de los números aleatorios anteriores. Lo único que nos queda por demostrar es que cada secuencia de números aleatorios se corresponde con una única permutación. Podemos hacerlo razonando hacia atrás (véase el Ejercicio 9.5).

La corrección de `permute` es algo bastante sutil.

Sin embargo, la respuesta en realidad es no –no todas las permutaciones son igualmente probables. Hay solo  $2^{31} - 2$  estados iniciales para el generador de números aleatorios, por lo que solo puede haber  $2^{31} - 2$  diferentes permutaciones. Esta condición podría ser un problema en algunas situaciones. Por ejemplo, un programa que genere 1.000.000 de permutaciones (quizá dividiendo el trabajo entre varias computadoras) para medir el rendimiento de un algoritmo de ordenación, generará casi con total seguridad algunas permutaciones dos veces –lamentablemente. Hacen falta mejores generadores de números aleatorios para que la práctica coincida con la teoría.

Observe que escribir de nuevo la llamada a `swap` usando la llamada a `r.nextInt(0, n-1)` no funciona, ni siquiera para el caso de tres elementos. Existen  $3! = 6$  posibles permutaciones y el número de secuencias diferentes que podrían calcularse mediante las tres llamadas a `nextInt` es  $3^3 = 27$ . Puesto que 6 no es divisor de 27, probablemente algunas permutaciones serán más probables que otras.

## 9.5 Algoritmos aleatorizados

Suponga que es usted un profesor que está asignando las tareas semanales a sus alumnos. Quiere cerciorarse de que sus alumnos están desarrollando sus propios programas o de que, como mínimo, comprenden el código que están enviando. Una solución es hacer un breve examen el mismo día que

se entrega cada programa. Sin embargo, estos exámenes quitan tiempo a las clases y, además, solo sirven realmente para algo en aproximadamente la mitad de las tareas de programación asignadas. Su tarea como profesor consiste en decidir cuándo realizar esos exámenes.

Por supuesto, si anuncia los exámenes de antemano eso podría interpretarse como una licencia implícita para hacer trampas en ese 50 por ciento de las tareas de programación para las que no va a haber examen. También podría adoptar la estrategia no anunciada de realizar los exámenes para tareas de programación alternativas, realizando un examen para una, pero no para la siguiente, etc. Pero los estudiantes rápidamente se darían cuenta de cuál es la estrategia. Otra posibilidad consiste en hacer los exámenes solo para lo que parecen ser tareas de programación importantes, pero eso conduciría a que los patrones de realización de exámenes fueran iguales año tras año. Teniendo en cuenta que los estudiantes se relacionan con los estudiantes de cursos anteriores, esta estrategia probablemente sería inútil después de un semestre.

Un método que parece eliminar todos estos problemas consiste en tirar una moneda al aire. Preparamos los exámenes para todos los programas (preparar los exámenes no consume tanto tiempo como realizarlos y calificarlos) y al principio de la clase, arrojamos una moneda al aire para decidir si hacemos el examen o no. De este modo, ni nosotros ni nuestros estudiantes podremos saber antes de la clase si se va a llevar a cabo un examen o no. Asimismo, los patrones no se repiten de un curso al siguiente. Los estudiantes pueden esperar que se realice un examen con un 50 por ciento de probabilidades, independientemente de los patrones de realización de exámenes anteriores. La desventaja de esta estrategia es que podríamos terminar no realizando ningún examen a lo largo de todo un semestre. Sin embargo, suponiendo que haya un gran número de tareas de programación que asignar a los estudiantes, no es probable que esto suceda, a menos que la moneda esté trucada. Cada semestre, el número esperado de exámenes será igual a la mitad del número de tareas de programación, y con una alta probabilidad, el número de exámenes que se termine realizando no se desviará demasiado con respecto a esa media.

Este ejemplo ilustra la idea de *algoritmo aleatorizado*, que emplea números aleatorios en lugar de decisiones deterministas para controlar el flujo de procesamiento. El tiempo de ejecución del algoritmo no depende solo de la entrada concreta, sino también de los números aleatorios que se presenten.

El tiempo de ejecución de caso peor de un algoritmo aleatorizado es siempre el mismo que el tiempo de ejecución de caso peor de un algoritmo no aleatorizado. La importante diferencia es que un buen algoritmo aleatorizado no tiene ninguna entrada que se considere mala –solo habrá malos números aleatorios (en relación con una entrada concreta). Esta diferencia puede parecer una cuestión filosófica, pero en realidad es muy importante como veremos en el siguiente ejemplo.

Imagine que el jefe nos pide que escribamos un programa para determinar la mediana de un grupo de 1.000.000 de números. Tenemos que enviar el programa y luego ejecutarlo con unos datos de entrada que el jefe va a elegir. Si la respuesta correcta se proporciona en unos pocos segundos de tiempo de computación (que es lo que cabría esperar para un algoritmo lineal), nuestro jefe estará muy satisfecho y obtendremos una gratificación. Pero si el programa no funciona o tarda demasiado tiempo, el jefe nos despedirá por incompetentes. Nuestro jefe cree ya en realidad que nos paga demasiado y espera tener que tomar la segunda de las posibles decisiones. ¿Qué deberíamos hacer?

Un algoritmo aleatorizado utiliza números aleatorios en lugar de decisiones deterministas para controlar el flujo de procesamiento.

El tiempo de ejecución de un algoritmo aleatorizado depende de los números aleatorios que se presenten, así como de la entrada concreta.

El algoritmo de selección rápida descrito en la Sección 8.7 puede parecer la forma conveniente de proceder. Aunque el algoritmo (véase la Figura 8.23) es muy rápido en promedio, recuerde que tiene un tiempo cuadrático de caso peor si el pivote es inadecuado una vez tras otra. Utilizando el particionamiento de la mediana de tres, hemos garantizado que este caso peor no se presente para una serie de entradas comunes, como aquellas que ya hayan sido ordenadas o aquellas que contengan muchos duplicados. Sin embargo, sigue existiendo un caso peor con un tiempo cuadrático y, como se muestra en el Ejercicio 8.9, el jefe va a leer nuestro programa, va a ver cómo estamos seleccionando el pivote y podrá construir esa entrada de caso peor. En consecuencia, seremos despedidos.

Utilizando números aleatorios, podemos garantizar estadísticamente que nuestro trabajo esté seguro. Iniciamos el algoritmo de selección rápida barajando aleatoriamente la entrada mediante las líneas 10 y 11 de la Figura 9.7.<sup>1</sup> Como resultado, el jefe pierde prácticamente la posibilidad de especificar la secuencia de entrada. Cuando ejecutamos el algoritmo de selección rápida, ahora estará trabajando con una entrada aleatoria, por lo que podemos esperar que se ejecute en un tiempo lineal. ¿Podría seguir necesitando un tiempo cuadrático? La respuesta es que sí. Para cualquier entrada original, la permutación podría llevarnos al caso peor del algoritmo de selección rápida, con lo que el resultado sería una ordenación en un tiempo cuadrático. Si tenemos la mala suerte de que eso suceda, perderemos nuestro trabajo. Sin embargo, este suceso es estadísticamente imposible. Para 1.000.000 de elementos, la posibilidad de emplear ni siquiera el doble de tiempo de lo que la media sugeriría es tan pequeña que podemos prácticamente ignorar dicha posibilidad. Es mucho más probable que la computadora se rompa. Nuestro trabajo no corre ningún peligro.

El algoritmo de selección rápida aleatorizada tiene estadísticamente garantizada la ejecución en un tiempo lineal.

En lugar de utilizar una técnica de barajado, podemos conseguir el mismo resultado seleccionando el pivote aleatoriamente en lugar de elegirlo de manera determinista. Tomamos un elemento aleatorio de la matriz y lo intercambiamos con el elemento situado en la posición `low`. Después, tomamos otro elemento aleatorio y lo intercambiamos con el elemento situado en la posición `high`. Tomamos un tercer elemento aleatorio y lo intercambiamos con el elemento situado en la posición central. Después, continuamos de la forma usual. Como antes, siempre será posible obtener particiones degeneradas, pero ahora esas particiones se producirán como resultado de la aparición de números aleatorios inadecuados, no de entradas inadecuadas.

Examinemos las diferencias entre los algoritmos aleatorizados y los no aleatorizados. Hasta ahora, nos hemos concentrado en los algoritmos no aleatorizados. A la hora de calcular sus tiempos de ejecución promedio, asumimos que todas las entradas son equiprobables. Esta suposición, sin embargo, no se cumple, porque las entradas casi ordenadas se presentan, por ejemplo, mucho más frecuentemente de lo que cabría estadísticamente esperar. Esta situación puede causar problemas a algunos algoritmos, como por ejemplo al de ordenación rápida. Pero cuando utilizamos un algoritmo aleatorizado, la entrada concreta deja de ser importante. Lo que son importantes son los números aleatorios y obtendremos un tiempo de ejecución *esperado*, con el que promediaremos todos los posibles números aleatorios para cualquier entrada concreta. La utilización del algoritmo de selección aleatoria con pivotes aleatorios (o con un paso de preprocessamiento que realice la permutación de la entrada) nos da un algoritmo con un tiempo esperado  $O(N)$ . Es decir, para

<sup>1</sup> Tenemos que asegurarnos de que el generador de números aleatorios sea suficientemente aleatorio y de que su salida no pueda ser predicha por el jefe.

cualquier entrada, incluyendo las entradas previamente ordenadas, se espera que el tiempo de ejecución sea  $O(N)$ , basándose en las propiedades estadísticas de los números aleatorios. Por un lado, una cota de tiempo esperado es algo más fuerte que una cota de caso promedio, porque las suposiciones utilizadas para generarla son menos restrictivas (números aleatorios en lugar de entradas aleatorias), pero es más débil que la cota de caso peor correspondiente. Por otro lado, en muchos casos las soluciones que tienen una buena cota de caso peor suelen requerir que el algoritmo incurra en más costes de procesamiento adicionales, para cerciorarse de que el caso peor no llegue a presentarse. El algoritmo de caso peor  $O(N)$  para la operación de selección por ejemplo, es un maravilloso resultado teórico, pero no es práctico.

Algunos algoritmos aleatorizados funcionan en un tiempo fijo pero cometen errores de forma aleatoria (presumiblemente con una baja probabilidad). Estos errores son falsos positivos o falsos negativos.

Los algoritmos aleatorizados se presentan en dos formas básicas. La primera, como ya hemos mostrado, siempre nos da una respuesta correcta, pero podría requerir una gran cantidad de tiempo, dependiendo de la suerte que tengamos con los números aleatorios. El segundo tipo es lo que vamos a ver en el resto de este capítulo. Algunos algoritmos aleatorizados funcionan en una cantidad fija de tiempo, pero cometen errores de forma aleatoria (presumiblemente con una baja probabilidad), denominándose a esos errores *falsos positivos* o *falsos negativos*. Esta técnica se acepta de modo común en el campo de la medicina. Los falsos positivos y los falsos negativos en

la mayoría de las pruebas médicas son, en realidad, bastante comunes, y algunas pruebas tienen tasas de error sorprendentemente altas. Además, para algunas pruebas los errores dependen del individuo, no de números aleatorios, por lo que la repetición de la prueba proporcionará casi con total seguridad otro falso resultado. En los algoritmos aleatorizados podemos volver a ejecutar la prueba con la misma entrada, utilizando números aleatorios diferentes. Si ejecutamos un algoritmo aleatorizado 10 veces y obtenemos 10 positivos –y asumiendo que un falso positivo sea un suceso improbable (por ejemplo, con una probabilidad de 1 entre 100)– la probabilidad de 10 falsos positivos consecutivos (1 entre  $100^{10}$ , o 1 entre cien trillones) es prácticamente cero.

## 9.6 Prueba aleatorizada de primalidad

Recuerde que en la Sección 7.4 hemos descrito algunos algoritmos numéricos y hemos demostrado cómo se pueden utilizar para implementar el esquema de cifrado RSA. Un paso importante en el algoritmo RSA consiste en generar dos números primos  $p$  y  $q$ . Podemos encontrar un número primo probando repetidamente con números impares sucesivos, hasta encontrar uno que sea primo. Por tanto, el problema se reduce a poder determinar si un cierto número especificado es primo o no.

El algoritmo de prueba de primalidad más simple es el de *división*. En este algoritmo, un número impar mayor que 3 será primo si no es divisible por ningún otro número impar menor o igual que  $\sqrt{N}$ . En la Figura 9.8 se muestra una implementación directa de esta estrategia.

El algoritmo de división es razonablemente rápido para números pequeños de 32 bits, pero no puede utilizarse para números de mayor tamaño, porque requeriría probar aproximadamente  $\sqrt{N}/2$  divisores, utilizando por tanto un tiempo  $O(\sqrt{N})$ .<sup>2</sup> Lo que necesitamos es una prueba cuyo tiempo

<sup>2</sup> Aunque  $\sqrt{N}$  puede parecer pequeño, si  $N$  es un número de 100 dígitos, entonces  $\sqrt{N}$  sigue siendo un número de 50 dígitos; las pruebas que tarden un tiempo  $O(\sqrt{N})$  son, por tanto, imposibles de aceptar para el tipo `BigInteger`.

```

1  /**
2   * Devuelve true si el entero impar n es primo.
3   */
4  public static boolean isPrime( long n )
5  {
6      for( int i = 3; i * i <= n; i += 2 )
7          if( n % i == 0 )
8              return false; // no primo
9
10     return true;      // primo
11 }

```

Figura 9.8 Prueba de primalidad mediante división.

de ejecución sea del mismo orden de magnitud que el de la rutina `power` de la Sección 7.4.2. Un teorema muy conocido, denominado *Pequeño teorema de Fermat*, parece bastante prometedor a este respecto. Vamos a enunciar el teorema y a proporcionar una demostración del mismo en el Teorema 9.1, en aras de la exhaustividad, aunque la demostración no es necesaria para comprender el algoritmo de comprobación de la primalidad.

El algoritmo de división es el algoritmo más simple para comprobar la primalidad. Es rápido para números pequeños (32 bits), pero no puede utilizarse para números de mayor tamaño.

**Teorema 9.1**

**Pequeño teorema de Fermat:** Si  $P$  es primo y  $0 < A < P$ , entonces  $A^{P-1} \equiv 1 \pmod{P}$ .

**Demostración**

Considere cualquier valor  $1 \leq k < P$ . Claramente,  $Ak \equiv 0 \pmod{P}$  es imposible porque  $P$  es primo y es mayor que  $A$  y  $k$ . Considere ahora cualquier  $1 \leq i < j < P$ .  $Ai \equiv Aj \pmod{P}$ , que es imposible por el argumento anterior porque  $1 \leq j - i < P$ . Por tanto, la secuencia  $A, 2A, \dots, (P-1)A$ , cuando se la considera  $\pmod{P}$ , es una permutación de  $1, 2, \dots, P-1$ . El producto de ambas secuencias  $\pmod{P}$  debe ser equivalente (y distinto de cero), lo que nos da la equivalencia  $A^{P-1} (P-1)! \equiv (P-1)!$ , de la que se deduce el teorema.

Si fuera cierta la implicación inversa a la que se presenta en el Pequeño teorema de Fermat, entonces tendríamos un algoritmo de comprobación de la primalidad que sería computacionalmente equivalente a la exponentiación modular (es decir,  $O(\log N)$ ). Lamentablemente, la implicación inversa no es cierta. Por ejemplo,  $2^{340} \equiv 1 \pmod{341}$ , pero 341 es número compuesto ( $11 \times 31$ ).

El Pequeño teorema de Fermat es una condición necesaria, pero no suficiente para establecer la primalidad.

Para llevar a cabo la comprobación de primalidad, necesitamos un teorema adicional, que es el Teorema 9.2.

**Teorema 9.2**

**Si  $P$  es primo y  $X^2 \equiv 1 \pmod{P}$ , entonces  $X \equiv \pm 1 \pmod{P}$ .**

**Demostración**

Puesto que  $X^2 - 1 \equiv 0 \pmod{P}$  implica  $(X-1)(X+1) \equiv 0 \pmod{P}$  y  $P$  es primo, entonces  $(X-1) \circ (X+1) \equiv 0 \pmod{P}$ .

Resulta entonces útil combinar los Teoremas 9.1 y 9.2. Sea  $A$  cualquier entero comprendido entre 2 y  $N - 2$ . Si calculamos  $A^{N-1} \pmod{N}$  y el resultado no es 1, sabemos que  $N$  no puede ser primo; en caso contrario, no se cumpliría el Pequeño teorema de Fermat. Como resultado,  $A$  es un valor que demuestra que  $N$  no es primo. Decimos entonces que  $A$  es un *testigo* de que  $N$  es compuesto. Todo número compuesto  $N$  tiene algunos testigos  $A$ , pero para algunos números, denominados *números de Carmichael*, estos testigos son difíciles de encontrar. Necesitamos asegurarnos de que haya una alta probabilidad de encontrar un testigo, independientemente de cuál sea la elección de  $N$ . Para mejorar nuestras probabilidades, utilizamos el Teorema 9.2.

En el curso de calcular  $A^t$ , calculamos  $(A^{\lfloor t/2 \rfloor})^2$ . Así que hacemos  $X = A^{\lfloor t/2 \rfloor}$  e  $Y = X^2$ . Observe que  $X$  e  $Y$  se calculan automáticamente como parte de la rutina `power`. Si  $Y$  es 1 y si  $X$  no es  $\pm 1 \pmod{N}$ , entonces por el Teorema 9.1,  $N$  no puede ser primo. Podemos devolver 0 para el valor de  $A^t$  cuando se detecte dicha condición, y  $N$  parecerá haber fallado la comprobación de primalidad implicada por el Pequeño teorema de Fermat.

La rutina `witness` (testigo) mostrada en la Figura 9.9, calcula  $A^t \pmod{P}$ , ampliada para devolver 0 si se detecta una violación del Teorema 9.1. Si `witness` no devuelve 1, entonces  $A$  es un testigo del hecho de que  $N$  no puede ser primo. La líneas 12 a 14 hacen una llamada recursiva y generan  $X$ . Entonces calculamos  $X^2$ , como es normal dentro del algoritmo `power`. Comprobamos si el Teorema 9.1 es violado, devolviendo 0 si lo es. En caso contrario, completamos los cálculos de `power`.

Si el algoritmo declara que un número no es primo, no lo será con un 100 por ciento de certidumbre. Cada intento aleatorio tiene como máximo una tasa de falsos positivos del 25 por ciento.

Algunos números compuestos pasarán la prueba y serán considerados primos, pero un número compuesto tiene una probabilidad muy baja de pasar 20 pruebas consecutivas con valores aleatorios independientes.

La única cuestión que queda por resolver es la de la corrección. Si nuestro algoritmo declara que  $N$  es compuesto, entonces  $N$  debe ser compuesto. Si  $N$  es compuesto, ¿son todos los valores  $2 \leq A \leq N - 2$  testigos de ello? La respuesta, lamentablemente, es que no. Es decir, algunas elecciones de  $A$  engañarán a nuestro algoritmo, induciéndole a declarar que  $N$  es primo. De hecho, si elegimos  $A$  aleatoriamente, tenemos como máximo una probabilidad de  $1/4$  de no detectar un número compuesto y de cometer por tanto un error. Observe que este resultado es cierto para cualquier  $N$ . Si solo se obtuviera promediando para todo valor de  $N$ , no tendríamos una rutina suficientemente buena. De forma análoga a lo que sucede con las pruebas médicas, nuestro algoritmo genera falsos positivos como máximo un 25 por ciento de las veces para cualquier  $N$ .

Estas probabilidades no parecen excesivamente buenas, porque una tasa de error del 25 por ciento se considera generalmente muy alta. Sin embargo, si utilizamos 20 valores independientes de  $A$ , la probabilidad de que ninguno de ellos sea testigo de que un número es compuesto es de  $1/4^{20}$ , que es aproximadamente de uno entre un billón. Estas probabilidades son mucho más razonables y pueden mejorarse todavía más utilizando más intentos. La rutina `isPrime`, que también se muestra en la Figura 9.9, utiliza cinco intentos.<sup>3</sup>

<sup>3</sup> Estas cotas son normalmente pesimistas y los análisis implican utilizar resultados de la teoría de números que son demasiado complicados para este libro de texto.

```
1  /**
2   * Método privado que implementa la comprobación básica de primalidad.
3   * Si witness no devuelve 1, n es definitivamente compuesto.
4   * Hacemos esto calculando  $a^i \pmod{n}$  y buscando de camino
5   * raíces cuadradas no triviales de 1.
6   */
7  private static long witness( long a, long i, long n )
8  {
9      if( i == 0 )
10         return 1;
11
12     long x = witness( a, i / 2, n );
13     if( x == 0 ) // Si n es recursivamente compuesto, parar
14         return 0;
15
16     // n no es primo si encontramos una raíz cuadrada de 1 no trivial
17     long y = ( x * x ) % n;
18     if( y == 1 && x != 1 && x != n - 1 )
19         return 0;
20
21     if( i % 2 != 0 )
22         y = ( a * y ) % n;
23
24     return y;
25 }
26
27 /**
28 * El número de testigos consultados en la prueba aleatorizada de primalidad.
29 */
30 public static final int TRIALS = 5;
31
32 /**
33 * Comprobación aleatorizada de primalidad.
34 * Ajuste TRIALS para incrementar el nivel de confianza.
35 * @param n el número que hay que comprobar.
36 * @return si es false, n es definitivamente no primo.
37 * Si es true, n es probablemente primo.
38 */
39 public static boolean isPrime( long n )
40 {
41     Random r = new Random();
42
43     for( int counter = 0; counter < TRIALS; counter++ )
44         if( witness( r.nextInt( (int) n - 3 ) + 2, n - 1, n ) != 1 )
45             return false;
46
47     return true;
48 }
```

Figura 9.9 Una comprobación aleatorizada de primalidad.

## Resumen

---

En este capítulo hemos descrito cómo se generan y se utilizan los números aleatorios. El generador de congruencia lineal es una buena opción para aplicaciones simples, siempre y cuando se tenga cuidado a la hora de elegir los parámetros  $A$  y  $M$ . Utilizando un generador de números aleatorios con distribución uniforme, podemos obtener números aleatorios con otras distribuciones, como la distribución de Poisson y la distribución exponencial negativa.

Los números aleatorios tienen muchas aplicaciones, incluyendo el estudio empírico de algoritmos, la simulación de sistemas de la vida real y el diseño de algoritmos que eviten de manera probabilística que se presente el caso peor. Utilizaremos números aleatorios en otras partes de este texto, especialmente en la Sección 13.2 y en el Ejercicio 21.22.

Con esto concluye la Parte Dos del libro. En la Parte Tres examinaremos algunas aplicaciones simples, comenzando con una exposición acerca de los juegos en el Capítulo 10, en la que se ilustrarán tres importantes técnicas de resolución de problemas.



### Conceptos clave

**algoritmo aleatorizado** Un algoritmo que emplea números aleatorios en lugar de decisiones deterministas para controlar el flujo de procesamiento. (396)

**algoritmo de división** El algoritmo más simple para comprobación de la primalidad. Es rápido para números de pequeño tamaño (32 bits), pero no es utilizable para números de mayor tamaño. (399)

**distribución de Poisson** Una distribución que modela el número de veces que se produce un suceso de baja probabilidad. (392)

**distribución exponencial negativa** Una forma de distribución utilizada para modelar el tiempo transcurrido entre sucesos aleatorios. Su media es igual a su varianza. (394)

**distribución uniforme** Una distribución en la que todos los números del rango especificado tienen la misma probabilidad de aparecer. (385)

**falsos positivos/falsos negativos** Errores cometidos aleatoriamente (presumiblemente con una baja probabilidad) por algunos algoritmos aleatorizados que funcionan en una cantidad fija de tiempo. (398)

**generador de congruencia lineal** Un buen algoritmo para generar distribuciones uniformes de números aleatorios. (385)

**generador de congruencia lineal de periodo completo** Un generador de números aleatorios que tiene un periodo  $M - 1$ . (387)

**números pseudoaleatorios** Números que tienen muchas propiedades de los números aleatorios. Los buenos generadores de números pseudoaleatorios son difíciles de encontrar. (384)

**Pequeño teorema de Fermat** Afirma que si  $P$  es primo y  $0 < A < P$ , entonces  $A^{P-1} \equiv 1 \pmod{P}$ . Es una condición necesaria pero no suficiente para establecer la primalidad. (399)

**periodo** La longitud de la secuencia hasta que se repite un número. Un generador de números aleatorios con periodo  $P$  genera la misma secuencia de números después de  $P$  iteraciones. (387)

**permutación** Una permutación de  $1, 2, \dots, N$  es una secuencia de  $N$  enteros que incluye a cada uno de los valores  $1, 2, \dots, N$  exactamente una vez. (384)

**permutación aleatoria** Una ordenación aleatoria de  $N$  elementos. Puede generarse en un tiempo lineal utilizando un número aleatorio por cada elemento. (394)

**semilla** El valor inicial de un generador de números aleatorios. (387)

**testigo de composición** Un valor de  $A$  que demuestra que un número no es primo, utilizando el Pequeño teorema de Fermat. (400)



## Errores comunes

1. El uso de una semilla inicial igual a cero proporciona números aleatorios inadecuados.
2. Los usuarios inexpertos en ocasiones reinicializan la semilla antes de generar una permutación aleatoria. Esta acción garantiza que se produzca repetidamente la misma permutación, la cual probablemente no es nuestra intención.
3. Muchos generadores de números aleatorios son notoriamente malos. Para aplicaciones serias en las que se necesitan largas secuencias de números aleatorios, el generador de congruencia lineal tampoco es satisfactorio.
4. Los bits de menor peso de los generadores de congruencia lineal se sabe que son en cierta medida no aleatorios, así que evite utilizarlos. Por ejemplo, `nextInt()%2` a menudo es una mala forma de simular el lanzamiento de una moneda.
5. Cuando se están generando números aleatorios en un cierto intervalo, un error común consiste en no respetar las fronteras y permitir que se genere algún número fuera del intervalo o no permitir que el número más pequeño se genere con una probabilidad equitativa.
6. Muchos generadores de permutaciones aleatorias no generan todas las permutaciones con igual probabilidad. Como se ha explicado en el texto, nuestro algoritmo está limitado por el generador de números aleatorios.
7. Manipular un generador de números aleatorios suele conducir a que se debiliten sus propiedades estadísticas.



## Internet

La mayor parte del código de este capítulo está disponible.

**Random.java**

Contiene ambas implementaciones de la clase `Random`.

**Numerical.java**

Contiene la rutina de comprobación de la primalidad mostrada en la Figura 9.9 y las rutinas matemáticas presentadas en la Sección 7.4.



## Ejercicios

### EN RESUMEN

- 9.1** Si se venden 42.000.000 de boletos de la lotería primitiva (con una probabilidad de 1 entre 14.000.000 de que un boleto resulte ganador), ¿cuál es el número esperado de ganadores? ¿Cuál es la probabilidad de que no haya ningún ganador? ¿Y cuál es la probabilidad de que haya un ganador?
- 9.2** ¿Por qué no se puede utilizar cero como semilla para el generador de congruencia lineal?
- 9.3** Demuestre cuál es el resultado de ejecutar el algoritmo de comprobación de la primalidad para  $N = 561$  con valores de  $A$  que van de 3 a 7.
- 9.4** Para el generador de números aleatorios descrito en el texto, determine los 10 primeros valores de `state`, suponiendo que se inicializa con un valor de 1.

### EN TEORÍA

- 9.5** Complete la demostración de que cada permutación obtenida en la Figura 9.7 es equiprobable.
- 9.6** Suponga que tenemos una moneda trucada que nos da cara con una probabilidad  $p$  y cruz con una probabilidad  $1 - p$ . Muestre cómo diseñar un algoritmo que utilice la moneda para generar un 0 o un 1 con igual probabilidad.

### EN LA PRÁCTICA

- 9.7** Demuestre que la Ecuación 9.4 es equivalente a la Ecuación 9.1 y que el programa resultante de la Figura 9.2 es correcto.
- 9.8** Ejecute 1.000.000 de veces el generador de Poisson mostrado en la Figura 9.5, utilizando un valor esperado de 2. ¿Concuerda la distribución con la de la Figura 9.4?
- 9.9** En el constructor `Random` de un solo parámetro de la Figura 9.2, ¿por qué no escribimos simplemente `initialValue=(M+1);`?
- 9.10** Escriba un programa que llame 100.000 veces a `nextInt` (que devuelve un valor `int` en el intervalo especificado) para generar números comprendidos entre 1 y 1.000. ¿Satisface las comprobaciones estadísticas más restrictivas proporcionadas en la Sección 9.2?
- 9.11** En la clase `Random48`, demuestre que `nextLong` no devuelve todos los posibles valores de tipo `long`.
- 9.12** Considere una elección entre dos candidatos en la que el ganador ha recibido una fracción  $p$  de los votos. Si los votos se cuentan secuencialmente, ¿cuál es la probabilidad de que el ganador estuviera por delante (o empatado) en cada etapa de la elección? Este problema es el denominado *problema del recuento*. Escriba un programa que verifique la respuesta,  $2 - \frac{1}{p}$ , suponiendo que  $p > \frac{1}{2}$  y que ha habido

un gran número de votos. (*Pista:* simule una elección con 10.000 votantes. Genere matrices aleatorias de  $10.000p$  unos y  $10.000(1 - p)$  ceros. Después verifique con una exploración secuencial que la diferencia entre la cantidad de unos y la de ceros nunca es negativa.)

## PROYECTOS DE PROGRAMACIÓN

- 9.13** Suponga que queremos generar una permutación aleatoria de  $N$  elementos distintos extraídos del rango 1, 2, ...,  $M$ . (El caso  $M = N$ , por supuesto, ya ha sido comentado.) El algoritmo de Floyd hace lo siguiente: en primer lugar, genera recursivamente una permutación de  $N - 1$  elementos distintos extraídos del rango  $M - 1$ . Después genera un entero aleatorio en el rango 1 a  $M$ . Si el entero aleatorio no se encuentra ya dentro de la permutación, lo añadimos; en caso contrario, añadimos  $M$ .
- Demuestre que este algoritmo no añade duplicados.
  - Demuestre que cada permutación es igualmente probable.
  - Proporcione una implementación recursiva del algoritmo.
  - Proporcione una implementación iterativa del algoritmo.
- 9.14** Un *paseo aleatorio* en dos dimensiones es un juego que se practica sobre el sistema de coordenadas *x-y*. Partiendo del origen (0, 0), cada iteración está compuesta de un paso aleatorio de longitud 1 unidad y que se puede ser hacia la izquierda, hacia arriba, hacia abajo o hacia la derecha. El paseo termina cuando el paseante vuelve al origen. (La probabilidad de que esto suceda es 1 en dos dimensiones, pero menor que 1 en tres dimensiones.) Escriba un programa que realice 100 paseos aleatorios independientes y calcule el número medio de pasos dados en cada dirección.
- 9.15** Un algoritmo de permutación alternativo consiste en llenar la matriz *a* desde *a[0]* a *a[n-1]*, de la forma siguiente. Para llenar *a[1]*, generamos números aleatorios hasta obtener uno que no haya sido utilizado previamente. Usamos una matriz de valores booleanos para realizar dicha comprobación. Proporcione un análisis del tiempo de ejecución esperado (le prevenimos que esto es complicado) y luego escriba un programa que compare este tiempo de ejecución tanto con su análisis como con el de la rutina mostrada en la Figura 9.7.
- 9.16** En la clase *Random48*, suponga que la rutina de un parámetro *nextInt* invoca a la rutina de cero parámetros *nextInt* en lugar de *nextLong* (y resuelve el caso extremo relativo a *Math.abs(Integer.MIN\_VALUE)*).
- Demuestre que, en este caso, si  $N$  es muy grande, entonces algunos restos son **significativamente** más probables que otros. Considere, por ejemplo,  $N = 2^{30} + 1$ .
  - Sugiera un remedio para el problema anterior.
  - ¿Se produce la misma situación cuando invocamos *nextLong*?
- 9.17** Una comprobación estadística simple y efectiva es la *prueba de ji-cuadrado*. Suponga que generamos  $N$  números positivos que pueden asumir uno de  $M$  posibles valores (por ejemplo, podríamos generar números comprendidos entre 1 y  $M$ ,

ambos inclusive). El número de veces que se presente cada número es una variable aleatoria cuya media es  $\mu = N / M$ . Para que la prueba funcione, deberíamos tener  $\mu > 10$ . Sea  $f_i$  el número de veces que se genera  $i$ . A continuación calcule el valor ji-cuadrado  $V = \sum (f_i - \mu)^2 / \mu$ . El resultado debería ser próximo a  $M$ . Si el resultado difiere sistemáticamente en más de  $2\sqrt{M}$  con respecto a  $M$  (es decir, más de una vez en 10 intentos), entonces el generador no ha pasado la comprobación. Implemente la prueba ji-cuadrado y ejecútela con su implementación del método `nextInt` (con `low = 1` y `high = 100`).

- 9.18** En la clase `Random48`, los  $b$  bits de menor peso de `state` se repiten con un periodo  $2^b$ .
- ¿Cuál es el periodo de la rutina `nextInt` de cero parámetros?
  - Demuestre que la rutina `nextInt` de un solo parámetro tendrá entonces un periodo de longitud  $2^{16}N$  si se la invoca con un valor  $N$  que sea una potencia de 2.
  - Modifique `nextInt` para detectar si  $N$  es una potencia de 2 y en caso afirmativo, utilice los bits de mayor peso de `state`, en lugar de los bits de menor peso.



## Referencias

Una buena explicación acerca de los generadores elementales de números aleatorios es la que se proporciona en [3]. El algoritmo de permutación se debe a R. Floyd y se presenta en [1]. El algoritmo aleatorizado de comprobación de la primalidad está tomado de [2] y [4]. Puede encontrar más información sobre números aleatorios en cualquier buen libro dedicado a estadística o probabilidad.

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. (Véase también *Technical Correspondence* en 36 (1993) 105–110, que proporciona el valor de  $A$  utilizado en la Figura 9.2.)
4. M. O. Rabin, "Probabilistic Algorithms for Testing Primality", *Journal of Number Theory* 12 (1980), 128–138.

parte  
tres

# Aplicaciones



Capítulo 10 Entretenimiento y juegos

Capítulo 11 Pilas y compiladores

Capítulo 12 Utilidades

Capítulo 13 Simulación

Capítulo 14 Grafos y caminos



## Entretenimiento y juegos

En este capítulo vamos a presentar tres importantes técnicas algorítmicas y a mostrar cómo utilizarlas implementando programas para resolver los problemas relacionados con el entretenimiento. El primer problema es el de las *sopas de letras*, que implica localizar palabras dentro de una cuadrícula bidimensional de caracteres. El segundo es el problema del juego óptimo en una partida de tres en raya.

En este capítulo, veremos

- Cómo utilizar el algoritmo de búsqueda binaria para incorporar información de búsquedas que no han tenido éxito y para resolver instancias de gran tamaño de sopas de letras en menos de un segundo.
- Cómo utilizar el algoritmo de *poda alfa-beta* para acelerar el algoritmo recursivo presentado en la Sección 7.7.
- Cómo utilizar mapas para aumentar la velocidad del algoritmo de tres en raya.

### 10.1 Sopa de letras

La entrada del problema de la *sopa de letras* es una matriz bidimensional de caracteres y una lista de palabras y el objetivo del juego es encontrar las palabras dentro de la cuadrícula. Dichas palabras pueden estar dispuestas en horizontal, en vertical o en diagonal en cualquier dirección (lo que nos da un total de ocho direcciones). Por ejemplo, la cuadrícula mostrada en la Figura 10.1 contiene las palabras *this*, *two*, *fat* y *that*. La palabra *this* comienza en la fila 0, columna 0 –el punto (0, 0)– y va hasta (0, 3); *two* ocupa desde (0, 0) a (2, 0); *fat* va desde (3, 0) a (1, 2); y *that* va desde (3, 3) a (0, 0). (Otras palabras, casi todas más cortas, no se enumeran aquí.)

La sopa de letras requiere buscar palabras dentro de una cuadrícula bidimensional de letras. Las palabras están orientadas en una de ocho posibles direcciones.

#### 10.1.1 Teoría

Podemos utilizar cualquiera de varios algoritmos simplistas para resolver el problema de la sopa de letras. La técnica más directa, basada en la fuerza bruta, sería la siguiente:

El algoritmo de fuerza bruta busca cada palabra dentro de la lista de palabras.

	0	1	2	3
0	t	h	i	s
1	w	a	t	s
2	o	a	h	g
3	f	g	d	t

Figura 10.1 Un ejemplo de cuadrícula para sopa de letras.

```

for each palabra W en la lista de palabras
  for each fila R
    for each columna C
      for each dirección D
        comprobar si W existe en la fila R, columna C en la dirección D
  
```

Puesto que hay ocho direcciones, este algoritmo requiere ocho comprobaciones de palabra/fila/columna ( $8WRC$ ). Los pasatiempos de sopa de letras que se publican en las revistas incluyen unas 40 palabras en aproximadamente una cuadrícula de  $16 \times 16$ , lo que implica unas 80.000 comprobaciones. Esta cantidad resulta ciertamente fácil de calcular con cualquier máquina moderna.

Un algoritmo alternativo busca a partir de cada punto de la cuadrícula en cada dirección y para cada longitud de palabra, y comprueba si la palabra está contenida en la lista de palabras.

Suponga, sin embargo, que consideramos una variante en la que solo nos dan la sopa de letras y la lista de palabras es esencialmente todo el diccionario de palabras en inglés. En este caso, el número de palabras podría ser 40.000 en lugar de 40, lo que da como resultado 80.000.000 de comprobaciones. Si doblamos el tamaño de la cuadrícula, necesitariamos 320.000.000 de comprobaciones, lo que ya deja de ser un cálculo trivial. Queremos un algoritmo que sea capaz de resolver una sopa de letras de este tamaño en una fracción de segundo (sin tener en cuenta el tiempo de E/S de disco), así que tenemos que tomar en consideración un algoritmo alternativo:

```

for each fila R
  for each columna C
    for each dirección D
      for each longitud de palabra L
        comprobar si los L caracteres que comienzan en la fila R columna C
          en la dirección D forman una palabra
  
```

Este algoritmo reordena el bucle para evitar tener que buscar cada palabra de la lista de palabras. Si suponemos que las palabras están limitadas a 20 caracteres, el número de comprobaciones utilizadas por el algoritmo será de  $160RC$ . Para una sopa de letras de  $32 \times 32$ , este número equivale aproximadamente a 160.000 comprobaciones. El problema, por supuesto, es que ahora tenemos que decidir si una palabra se encuentra dentro de la lista de palabras. Si utilizamos una búsqueda lineal estamos perdidos. Si empleamos una buena estructura de datos, cabe esperar que podamos

realizar una búsqueda eficiente. Si la lista de palabras está ordenada, que es algo que cabría esperar en un diccionario en línea, podríamos usar una búsqueda binaria (mostrada en la Figura 5.12) y realizar cada comprobación en aproximadamente  $\log W$  comparaciones de cadenas. Para 40.000 palabras,

Las consultas se pueden realizar mediante una búsqueda binaria.

hacer esto implica quizás 16 comparaciones por cada comprobación, lo que da un total inferior a 3.000.000 de comparaciones de cadenas. Esta cantidad de comparaciones se puede llevar a cabo ciertamente en unos cuantos segundos y es 100 veces mejor que el algoritmo anterior.

Podemos mejorar aun más el algoritmo basándonos en la siguiente observación. Suponga que estamos buscando en alguna dirección y vemos la secuencia de caracteres qx. Un diccionario de inglés no contiene ninguna palabra que comience por qx, así que ¿merece la pena continuar con el bucle más interno para todas las longitudes de palabra? Obviamente, la respuesta es no: si detectamos una secuencia de caracteres que no es un prefijo de ninguna palabra contenida en el diccionario, podemos buscar inmediatamente en otra dirección. Este algoritmo está dado por el siguiente pseudocódigo:

```

for each fila R
    for each columna C
        for each dirección D
            for each longitud de palabra L
                comprobar si los L caracteres que comienzan en la fila R columna C
                    en la dirección D forman una palabra
                if no forman un prefijo,
                    break; // el bucle más interno

```

El único detalle algorítmico que nos queda por concretar es la implementación de la comprobación de prefijos: suponiendo que la secuencia de caracteres actual no está en la lista de palabras, ¿cómo podemos decidir qué es un prefijo de alguna palabra contenida en la lista de palabras? La respuesta es bastante simple. Recuerde de la Sección 6.4.3 que el método `binarySearch` de la API de Colecciones devuelve el índice de una correspondencia o la posición del elemento más pequeño que sea al menos tan grande como el elemento buscado (como un número negativo). El llamante puede comprobar fácilmente si se ha encontrado una correspondencia. Si no se ha encontrado, verificar que la secuencia de caracteres es un prefijo de alguna palabra de la lista también es sencillo, porque si lo es, deberá ser un prefijo de la palabra situada en la posición señalada por el valor de retorno (en el Ejercicio 10.3 le pediremos que demuestre este resultado).

Si una secuencia de caracteres no es un prefijo de ninguna palabra del diccionario, podemos dar por terminada la búsqueda en esa dirección.

La comprobación de prefijos también se puede realizar mediante búsqueda binaria.

### 10.1.2 Implementación Java

Nuestra implementación Java se ajusta a la descripción del algoritmo de forma prácticamente textual. Diseñamos una clase `WordSearch` para almacenar la cuadrícula y la lista de palabras, así como los correspondientes flujos de entrada. El esqueleto de la clase se muestra en la Figura 10.2. La parte pública de la clase está compuesta por un constructor y un único método, `solvePuzzle`. La parte privada incluye los miembros de datos y las rutinas de soporte.

Nuestra implementación se ajusta a la descripción del algoritmo.

La Figura 10.3 proporciona el código del constructor. Se limita a abrir y leer los dos archivos correspondientes a la cuadrícula y la lista de palabras. La rutina de soporte `openFile`, mostrada en la Figura 10.4, pide repetidamente

El constructor abre y lee los archivos de datos. Hemos simplificado las comprobaciones de error en aras de la brevedad.

```
1 import java.io.BufferedReader;
2 import java.io.FileReader;
3 import java.io.InputStreamReader;
4 import java.io.IOException;
5
6 import java.util.Arrays;
7 import java.util.ArrayList;
8 import java.util.Iterator;
9 import java.util.List;
10
11
12 // Interfaz de la clase WordSearch: resolver la sopa de letras
13 //
14 // CONSTRUCCIÓN: sin ningún inicializador
15 // *****OPERACIONES PÚBLICAS*****
16 // int solvePuzzle( ) --> Imprimir todas las palabras encontradas en
17 // la sopa de letras; devolver el número de correspondencias
18
19 public class WordSearch
20 {
21     public WordSearch( ) throws IOException
22     { /* Figura 10.3 */ }
23     public int solvePuzzle( )
24     { /* Figura 10.7 */ }
25
26     private int rows;
27     private int columns;
28     private char theBoard[ ][ ];
29     private String [ ] theWords;
30     private BufferedReader puzzleStream;
31     private BufferedReader wordStream;
32     private BufferedReader in = new
33             BufferedReader( new InputStreamReader( System.in ) );
34
35     private static int prefixSearch( String [ ] a, String x )
36     { /* Figura 10.8 */ }
37     private BufferedReader openFile( String message )
38     { /* Figura 10.4 */ }
39     private void readWords( ) throws IOException
40     { /* Figura 10.5 */ }
41     private void readPuzzle( ) throws IOException
42     { /* Figura 10.6 */ }
43     private int solveDirection( int baseRow, int baseCol,
44                               int rowDelta, int colDelta )
45     { /* Figura 10.8 */ }
46 }
```

---

Figura 10.2 Esqueleto de la clase WordSearch.

```
1 /**
2  * Constructor para la clase WordSearch.
3  * Pide y lee los archivos de sopa de letras y del diccionario.
4  */
5 public WordSearch( ) throws IOException
6 {
7     puzzleStream = openFile( "Enter puzzle file" );
8     wordStream = openFile( "Enter dictionary name" );
9     System.out.println( "Reading files..." );
10    readPuzzle( );
11    readWords( );
12 }
```

Figura 10.3 Constructor de la clase WordSearch.

```
1 /**
2  * Muestra una petición y abre un archivo.
3  * Reintentar hasta que la apertura tenga éxito.
4  * El programa sale si se alcanza el final del archivo.
5  */
6 private BufferedReader openFile( String message )
7 {
8     String fileName = "";
9     FileReader theFile;
10    BufferedReader fileIn = null;
11
12    do
13    {
14        System.out.println( message + ":" );
15
16        try
17        {
18            fileName = in.readLine( );
19            if( fileName == null )
20                System.exit( 0 );
21            theFile = new FileReader( fileName );
22            fileIn = new BufferedReader( theFile );
23        }
24        catch( IOException e )
25        {
26            System.err.println( "Cannot open " + fileName );
27        }
28    } while( fileIn == null );
29
30    System.out.println( "Opened " + fileName );
31    return fileIn;
32 }
```

Figura 10.4 La rutina openFile para abrir el archivo de cuadricula o el de la lista de palabras.

un archivo hasta que se produzca una apertura de archivo con éxito. La rutina `readWords`, mostrada en la Figura 10.5, lee la lista de palabras. El código incluye una comprobación de error para cerciorarse de que la lista de palabras ha sido ordenada. De forma similar, `readPuzzle`, mostrada en la Figura 10.6, lee la cuadrícula y también se preocupa del tratamiento de errores. Tenemos que asegurarnos de que podemos tratar los casos en los que no haya sopa de letras y también queremos advertir al usuario en el caso de que la cuadrícula no sea rectangular.

Utilizamos dos bucles para iterar a través de las ocho direcciones.

La rutina `solvePuzzle`, mostrada en la Figura 10.7, anida los bucles de fila, de columna y de dirección, y luego invoca a la rutina privada `solveDirection` para cada posibilidad. El valor de retorno es el número de correspondencias encontradas. Proporcionamos una dirección indicando una dirección de columna y luego una dirección de fila. Por ejemplo, la dirección sur está indicada por `cd=0` y `rd=1` y la dirección noreste por `cd=1` y `rd=-1`; `cd` puede variar de -1 a 1 y `rd` de -1 a 1, con la única restricción de que ambos no pueden ser 0 simultáneamente. Todo lo que resta por hacer es proporcionar `solveDirection`, que está codificada en la Figura 10.8. La rutina `solveDirection` construye una cadena comenzando por la fila y columna base, extendiéndose en la dirección apropiada.

```

1  /**
2   * Rutina para leer el diccionario.
3   * Se imprime un mensaje de error si el diccionario no está ordenado.
4   */
5  private void readWords() throws IOException
6  {
7      List<String> words = new ArrayList<String>();
8
9      String lastWord = null;
10     String thisWord;
11
12     while( ( thisWord = wordStream.readLine() ) != null )
13     {
14         if( lastWord != null && thisWord.compareTo( lastWord ) < 0 )
15         {
16             System.err.println( "Dictionary is not sorted... skipping" );
17             continue;
18         }
19         words.add( thisWord );
20         lastWord = thisWord;
21     }
22
23     theWords = new String[ words.size() ];
24     theWords = words.toArray( theWords );
25 }
```

Figura 10.5 La rutina `readWords` para leer la lista de palabras.

```
1  /**
2  * Rutina para leer la cuadrícula.
3  * Comprueba que la cuadrícula sea rectangular.
4  * Se omiten las comprobaciones de que la capacidad sea excedida.
5  */
6  private void readPuzzle( ) throws IOException
7  {
8      String oneLine;
9      List<String> puzzleLines = new ArrayList<String>( );
10
11     if( ( oneLine = puzzleStream.readLine( ) ) == null )
12         throw new IOException( "No lines in puzzle file" );
13
14     columns = oneLine.length( );
15     puzzleLines.add( oneLine );
16
17     while( ( oneLine = puzzleStream.readLine( ) ) != null )
18     {
19         if( oneLine.length( ) != columns )
20             System.err.println( "Puzzle is not rectangular; skipping row" );
21         else
22             puzzleLines.add( oneLine );
23     }
24
25     rows = puzzleLines.size( );
26     theBoard = new char[ rows ][ columns ];
27
28     int r = 0;
29     for( String theLine : puzzleLines )
30         theBoard[ r++ ] = theLine.toCharArray( );
31 }
```

Figura 10.6 La rutina `readPuzzle` para leer la cuadrícula.

También asumimos que no están permitidas las correspondencias de una sola letra (porque cualquier correspondencia de una única letra sería impresa ocho veces). En las líneas 14 a 16, iteramos y ampliamos la cadena, al mismo tiempo que nos cercioramos de no ir más allá de los límites de la cuadrícula. En la línea 18 añadimos el siguiente carácter, utilizando `+=`, y hacemos una búsqueda binaria en la línea 19. Si no tenemos un prefijo, podemos dejar de buscar y volver. En caso contrario, sabemos que tenemos que continuar, después de comprobar en la línea 26 si se ha producido una correspondencia exacta. La línea 35 devuelve el número de correspondencias encontradas, una vez que la llamada a `solveDirection` ya no puede encontrar más palabras. En la Figura 10.9 se muestra un programa `main` simple.

```

1  /**
2   * Rutina para resolver la sopa de letras.
3   * Realiza comprobaciones en las ocho direcciones.
4   * @return número de correspondencias
5   */
6  public int solvePuzzle( )
7  {
8      int matches = 0;
9
10     for( int r = 0; r < rows; r++ )
11         for( int c = 0; c < columns; c++ )
12             for( int rd = -1; rd <= 1; rd++ )
13                 for( int cd = -1; cd <= 1; cd++ )
14                     if( rd != 0 || cd != 0 )
15                         matches += solveDirection( r, c, rd, cd );
16
17     return matches;
18 }

```

**Figura 10.7** La rutina `solvePuzzle` para buscar en todas las direcciones a partir de todos los puntos de partida.

```

1 /**
2  * Buscar en la cuadrícula a partir de un punto inicial y una dirección.
3  * @return número de correspondencias
4  */
5 private int solveDirection( int baseRow, int baseCol,
6 int rowDelta, int colDelta )
7 {
8     String charSequence = "";
9     int numMatches = 0;
10    int searchResult;
11
12    charSequence += theBoard[ baseRow ][ baseCol ];
13
14    for( int i = baseRow + rowDelta, j = baseCol + colDelta;
15        i >= 0 && j >= 0 && i < rows && j < columns;
16        i += rowDelta, j += colDelta )
17    {
18        charSequence += theBoard[ i ][ j ];
19        searchResult = prefixSearch( theWords, charSequence );
20
21        if( searchResult == theWords.length )
22            break;
23        if( !theWords[ searchResult ].startsWith( charSequence ) )
24            break;
25
26        if( theWords[ searchResult ].equals( charSequence ) )

```

Continúa

**Figura 10.8** Implementación de una única búsqueda.

```

27     {
28         numMatches++;
29         System.out.println( "Found " + charSequence + " at " +
30                             baseRow + " " + baseCol + " to " +
31                             i + " " + j );
32     }
33 }
34
35     return numMatches;
36 }
37
38 /**
39 * Realiza la búsqueda binaria de palabras.
40 * Devuelve la última posición examinada. O bien esta posición
41 * se corresponde con x, o x es un prefijo or no hay ninguna
42 * palabra para la que x sea un prefijo.
43 */
44 private static int prefixSearch( String [ ] a, String x )
45 {
46     int idx = Arrays.binarySearch( a, x );
47
48     if( idx < 0 )
49         return -idx - 1;
50     else
51         return idx;
52 }

```

Figura 10.8 (Continuación).

```

1 // Rutina main simple
2 public static void main( String [ ] args )
3 {
4     WordSearch p = null;
5
6     try
7     {
8         p = new WordSearch( );
9     }
10    catch( IOException e )
11    {
12        System.out.println( "IO Error: " );
13        e.printStackTrace( );
14        return;
15    }
16
17    System.out.println( "Solving..." );
18    p.solvePuzzle( );
19 }

```

Figura 10.9 Una simple rutina main para el problema de la sopa de letras.

## 10.2 El juego de las tres en raya

La estrategia *minimax* examina muchas posiciones. Podemos hacerlo con menos sin perder ninguna información.

Recuerde de la Sección 7.7 que un algoritmo simple, conocido con el nombre de *estrategia minimax*, permite a la computadora seleccionar un movimiento óptimo en una partida del juego de las tres en raya. Esta estrategia recursiva implica las siguientes decisiones.

1. Una *posición terminal* se puede evaluar de forma inmediata, por lo que si la posición es terminal, hay que devolver su valor.
2. En caso contrario, si es el turno para que mueva la computadora, devolvemos el valor máximo de todas las posiciones alcanzables al realizar un movimiento. Los valores alcanzables se calculan de forma recursiva.
3. En caso contrario, le toca mover al jugador humano. Devolvemos el valor mínimo de todas las posiciones alcanzables realizando un único movimiento. Los valores alcanzables se calculan de forma recursiva.

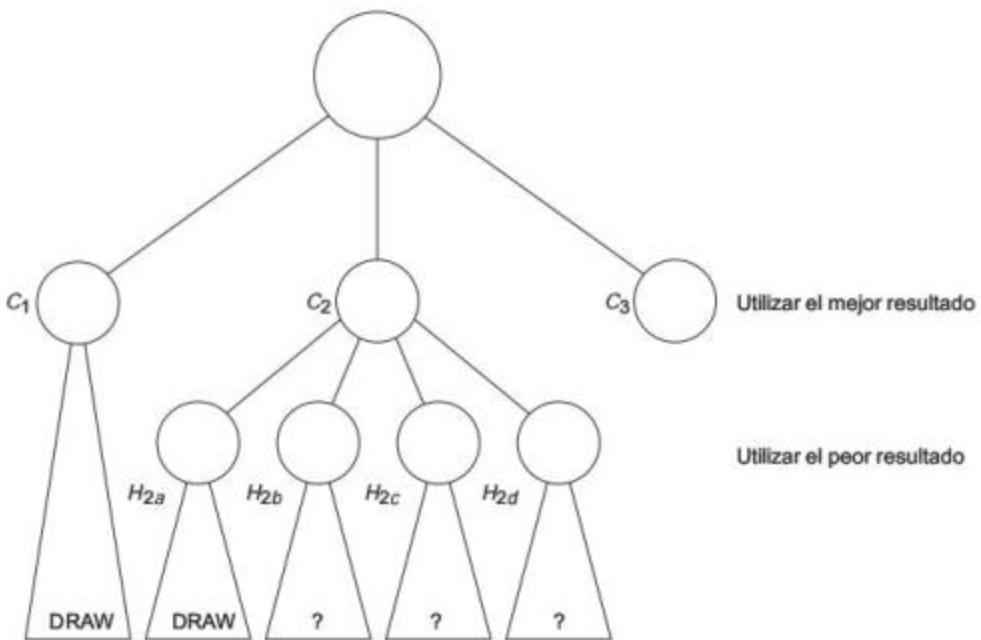
### 10.2.1 Poda alfa-beta

Una *refutación* es un movimiento de respuesta que demuestra que un movimiento propuesto no constituye una mejora con respecto a los movimientos anteriormente considerados. Si encontramos una refutación, no tenemos por qué examinar más movimientos y la llamada recursiva puede volver.

Aunque la estrategia minimax nos da un movimiento óptimo para juego de las tres en raya, realiza un montón de búsquedas. Específicamente, para seleccionar el primer movimiento hace aproximadamente unas 500.000 llamadas recursivas. Una razón para que se produzca este gran número de llamadas es que el algoritmo realiza más búsquedas de las necesarias. Suponga que la computadora está considerando cinco movimientos:  $C_1$ ,  $C_2$ ,  $C_3$ ,  $C_4$  y  $C_5$ . Suponga también que la evaluación recursiva de  $C_1$  revela que  $C_1$  fuerza un empate. Ahora evaluamos  $C_2$ . En esta etapa, tenemos una posición a partir de la cual le correspondería mover a un jugador humano. Suponga que en respuesta a  $C_2$ , el jugador humano puede considerar  $H_{2a}$ ,  $H_{2b}$ ,  $H_{2c}$  y  $H_{2d}$ . Además, suponga que una evaluación de  $H_{2a}$  muestra un empate forzado. Automáticamente,  $C_2$  será, en el caso mejor, un empate y

posiblemente incluso una derrota de la computadora (porque se supone que el jugador humano va a jugar de forma óptima). Puesto que necesitamos mejorar con respecto a  $C_1$ , no tenemos que evaluar ninguna de las opciones  $H_{2b}$ ,  $H_{2c}$  y  $H_{2d}$ . Decimos entonces que  $H_{2a}$  es una *refutación*, lo que significa que demuestra que  $C_2$  no es un mejor movimiento que los que ya hemos visto anteriormente. Por tanto, proporcionamos como valor de retorno que  $C_2$  es un empate y mantenemos  $C_1$  como el mejor movimiento visto hasta el momento, como se muestra en la Figura 10.10. Por tanto, en general, una *refutación* es un movimiento de respuesta que demuestra que un movimiento propuesto no constituye un mejora con respecto a los movimientos previamente considerados.

No necesitamos evaluar cada nodo completamente; para algunos nodos, una refutación basta y algunos bucles puede terminar anticipadamente. Específicamente, cuando el jugador humano evalúa una posición, como  $C_2$ , en caso de que se encuentre una refutación, será tan buena como el movimiento mejor en términos absolutos. La misma lógica se aplica a la computadora. En cualquier punto de la búsqueda, *alpha* es el valor que el jugador humano tiene que refutar y *beta* es el valor que la computadora tiene que refutar. Cuando se hace una búsqueda en el lado del jugador humano,



**Figura 10.10** Poda alfa-beta: después de evaluar  $H_{2a}$ ,  $C_2$ , que es el mínimo de los  $H_2$ , es como mucho un empate. En consecuencia, no puede constituir una mejora con respecto a  $C_1$ . Por tanto, no necesitamos evaluar  $H_{2b}$ ,  $H_{2c}$  y  $H_{2d}$  y podemos continuar directamente con  $C_3$ .

cualquier movimiento menor que `alpha` es equivalente a `alpha`; cuando se hace una búsqueda en el lado de la computadora, cualquier movimiento mayor que `beta` es equivalente a `beta`. Esta estrategia de reducir el número de posiciones evaluadas en un búsqueda minimax se suele denominar *poda alfa-beta*.

Como se muestra en la Figura 10.11, la poda alfa-beta solo requiere unos cuantos cambios en `chooseMove`. Tanto `alpha` como `beta` se pasan como parámetros adicionales. Inicialmente, `chooseMove` se inicia con `alpha` y `beta` representando a `HUMAN_WIN` y `COMPUTER_WIN`, respectivamente, que son los valores que indican que gana el jugador humano o que gana la computadora. Las líneas 17 a 21 reflejan un cambio en la inicialización de `value`. La evaluación de movimientos es solo ligeramente más compleja que la originalmente mostrada en la Figura 7.29. La llamada recursiva en la línea 30 incluye los parámetros `alpha` y `beta`, que se ajustan en las líneas 37 o 39 en caso necesario. El único cambio adicional es la línea 42, que fuerza un retorno inmediato cuando se encuentra una refutación.

Para aprovechar al máximo la poda alfa-beta, los programas de juego suelen intentar aplicar heurísticos para situar los mejores movimientos en las primeras etapas de la búsqueda. Este enfoque hace que se produzca una poda aún mayor que la que cabría esperar a partir de una búsqueda aleatoria de posiciones. En la práctica, la poda alfa-beta limita las búsquedas a nodos  $O(\sqrt{N})$ , donde  $N$  es el número de nodos que se examinarían sin la poda alfa-beta, lo que da como resultado un enorme ahorro. El ejemplo del juego de las tres en raya no es el ideal, ya que hay

La poda alfa-beta se utiliza para reducir el número de posiciones evaluadas dentro de una búsqueda minimax. Alfa es el valor que tiene que refutar el jugador humano, mientras que beta es el valor que tiene que refutar la computadora.

La poda alfa-beta funciona mejor cuando encuentra las refutaciones en una etapa más temprana.

```

1 // Encontrar movimiento óptimo
2 private Best chooseMove( int side, int alpha, int beta, int depth )
3 {
4     int opp;           // El otro lado
5     Best reply;       // Mejor respuesta del oponente
6     int dc;           // Variable de almacenamiento
7     int simpleEval;  // Resultado de una evaluación inmediata
8     int bestRow = 0;
9     int bestColumn = 0;
10    int value;
11
12    if( simpleEval = positionValue( ) ) != UNCLEAR )
13        return new Best( simpleEval );
14
15    if( side == COMPUTER )
16    {
17        opp = HUMAN; value = alpha;
18    }
19    else
20    {
21        opp = COMPUTER; value = beta;
22    }
23
24 Outer:
25    for( int row = 0; row < 3; row++ )
26        for( int column = 0; column < 3; column++ )
27            if( squareIsEmpty( row, column ) )
28            {
29                place( row, column, side );
30                reply = chooseMove( opp, alpha, beta, depth + 1 );
31                place( row, column, EMPTY );
32
33                if( side == COMPUTER && reply.val > value ||
34                    side == HUMAN && reply.val < value )
35                [
36                    if( side == COMPUTER )
37                        alpha = value = reply.val;
38                    else
39                        beta = value = reply.val;
40
41                    bestRow = row; bestColumn = column;
42                    if( alpha >= beta )
43                        break Outer; // Refutación
44                ]
45            }
46
47    return new Best( value, bestRow, bestColumn );
48 }

```

Figura 10.11 La rutina chooseMove para calcular un movimiento óptimo del juego de las tres en raya, utilizando la poda alfa-beta.

demasiados valores idénticos. Aun así, la búsqueda inicial se reduce a aproximadamente 18.000 posiciones.

### 10.2.2 Tablas de transposición

Otra práctica comúnmente empleada es la de utilizar una tabla para llevar la cuenta de todas las posiciones que se han evaluado. Por ejemplo, en el curso de la búsqueda del primer movimiento, el programa examinará las posiciones mostradas en la Figura 10.12. Si guardamos los valores de las posiciones, la segunda aparición de una posición no necesita ser recalculada; esencialmente, se transforma en una posición terminal. La estructura de datos que registra y almacena las posiciones previamente evaluadas se denomina *tabla de transposición*; se implementa como un mapa que asigna valores a posiciones.<sup>1</sup>

No necesitamos un mapa ordenado, por lo que para implementar la tabla de transposición se emplea como implementación subyacente el `HashMap`, un mapa desordenado con una estructura de datos denominada *tabla hash*. Hablaremos de las tablas hash en el Capítulo 20.

Para implementar la tabla de transposición definimos primero una clase `Position`, como se muestra en la Figura 10.13, que utilizaremos para almacenar cada posición. Los valores del tablero serán `HUMAN`, `COMPUTER` o `EMPTY` (que definiremos enseguida en la clase `TicTacToe`, como se muestra en la Figura 10.14). El `HashMap` requiere que definamos `equals` y `hashCode`. Recuerde que si `equals` declara dos objetos `Position` como iguales, `hashCode` debe proporcionar valores idénticos para esos objetos. También proporcionamos un constructor que se puede inicializar con una matriz que representa el tablero.

Un problema importante es el relativo a si merece la pena incluir todas las posiciones en la tabla de transposición. El coste adicional de mantener la tabla sugiere que las posiciones cercanas al final de la recursión no deberían guardarse porque:

Una tabla de transposición almacena las posiciones previamente evaluadas.

Se utiliza un mapa para implementar la tabla de transposición. A menudo, la implementación subyacente es una tabla hash.

No almacenamos en la tabla de transposición las posiciones que se encuentran al final de la recursión.

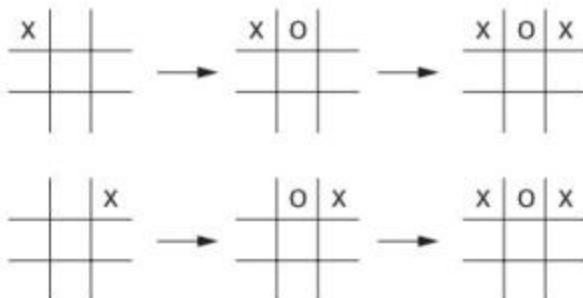


Figura 10.12 Dos búsquedas que llegan a posiciones idénticas.

<sup>1</sup> Aunque en un contexto diferente, en la Sección 7.6 hemos hablado de esta técnica genérica, que almacena valores en una tabla para evitar las llamadas recursivas repetidas. Esta técnica se conoce también con el nombre *memorización*. El término *tabla de transposición* induce ligeramente a error, porque algunas implementaciones más inteligentes de esta técnica reconocen y evitan buscar no solo las posiciones exactamente idénticas, sino también las que son simétricamente idénticas.

- Hay demasiadas.
- El objetivo de la poda alfa-beta y de las tablas de transposición consiste en reducir los tiempos de búsqueda, evitando las llamadas recursivas en las primeras etapas del juego; ahorrarse una llamada recursiva cuando estamos a gran profundidad dentro de la búsqueda no reduce enormemente el número de posiciones examinadas, porque de todos modos esa llamada recursiva solo iba a examinar unas pocas posiciones.

```
1 final class Position
2 {
3     private int [][] board;
4
5     public Position( int [][] theBoard )
6     {
7         board = new int[ 3 ][ 3 ];
8         for( int i = 0; i < 3; i++ )
9             for( int j = 0; j < 3; j++ )
10                board[ i ][ j ] = theBoard[ i ][ j ];
11    }
12
13    public boolean equals( Object rhs )
14    {
15        if( !(rhs instanceof Position) )
16            return false;
17
18        Position other = (Position) rhs;
19
20        for( int i = 0; i < 3; i++ )
21            for( int j = 0; j < 3; j++ )
22                if( board[ i ][ j ] != (Position) rhs ).board[ i ][ j ] )
23                    return false;
24        return true;
25    }
26
27    public int hashCode( )
28    {
29        int hashVal = 0;
30
31        for( int i = 0; i < 3; i++ )
32            for( int j = 0; j < 3; j++ )
33                hashVal = hashVal * 4 + board[ i ][ j ];
34
35        return hashVal;
36    }
37 }
```

---

Figura 10.13 La clase Position.

Vamos a mostrar cómo se aplica esta técnica al juego de las tres en raya cuando implementamos la tabla de transposición. Los cambios necesarios en la clase `TicTacToe` se muestran en la Figura 10.14. Las adiciones son el nuevo miembro de datos en la línea 7 y la nueva declaración para `chooseMove` en la línea 14. Ahora pasamos `alpha` y `beta` (como en la poda alfa-beta) y también la profundidad `depth` de la recursión, que es cero de manera predeterminada. La llamada inicial a `chooseMove` se muestra en la línea 11.

El método `chooseMove` tiene parámetros adicionales, todos los cuales tienen valores predeterminados.

La Figura 10.15 muestra la nueva rutina `chooseMove`. En la línea 8 declaramos un objeto `Position`, `thisPosition`. Cuando llegue el momento, se colocará en la tabla de transposición. `tableDepth` nos dice qué profundidad dentro de la búsqueda será la que permita colocar posiciones en la tabla de transposición. Experimentando, nosotros hemos determinado que la profundidad 5 era la óptima. Permitir que se guarden las posiciones de profundidad 6 resulta contraproducente, porque el coste adicional de mantener una tabla de transposición más grande no se ve compensado por la escasa reducción en el número de posiciones examinadas.

Las líneas 17 a 24 son nuevas. Si estamos en la primera llamada a `chooseMove`, inicializamos la tabla de transposición. En caso contrario, si estamos a una profundidad apropiada, determinamos si la posición actual ha sido evaluada; si lo ha sido, devolvemos su valor. El código tiene dos trucos. En primer lugar, podemos transponer únicamente a profundidad 3 o superior, como sugiere la Figura 10.12. La única otra diferencia es la adición de las líneas 57 y 58. Inmediatamente antes de volver, almacenamos el valor de la posición dentro de la tabla de transposición.

El código tiene unos cuantos trucos, pero ninguno especialmente importante.

El uso de la tabla de transposición en este algoritmo de las tres en raya nos evita tener que considerar aproximadamente la mitad de las posiciones, con solo un ligero coste adicional debido a las operaciones con la tabla de transposición. La velocidad del programa casi se dobla.

```

1 // Directivas de importación originales más:
2 import java.util.Map;
3 import java.util.HashMap;
4
5 class TicTacToe
6 {
7     private Map<Position, Integer> transpositions
8         = new HashMap<Position, Integer>();
9
10    public Best chooseMove( int side )
11        { return chooseMove( side, HUMAN_WIN, COMPUTER_WIN, 0 ); }
12
13    // Encontrar el movimiento óptimo
14    private Best chooseMove( int side, int alpha, int beta, int depth )
15        { /* Figuras 10.15 y 10.16 */ }
16
17    ...
18 }
```

Figura 10.14 Cambios en la clase `TicTacToe` para incorporar la tabla de transposición y la poda alfa-beta.

```

1      // Encontrar el movimiento óptimo.
2  private Best chooseMove( int side, int alpha, int beta, int depth )
3  {
4      int opp;           // El otro lado
5      Best reply;       // Mejor respuesta del oponente
6      int dc;           // Variable de almacenamiento
7      int simpleEval;   // Resultado de una evaluación inmediata
8      Position thisPosition = new Position( board );
9      int tableDepth = 5; // Máx. profund. colocada en la tabla de transp.
10     int bestRow = 0;
11     int bestColumn = 0;
12     int value;
13
14     if( simpleEval == positionValue( ) ) != UNCLEAR )
15         return new Best( simpleEval );
16
17     if( depth == 0 )
18         transpositions.clear( );
19     else if( depth >= 3 && depth <= tableDepth )
20     {
21         Integer lookupVal = transpositions.get( thisPosition );
22         if( lookupVal != null )
23             return new Best( lookupVal );
24     }
25
26     if( side == COMPUTER )
27     {
28         opp = HUMAN; value = alpha;
29     }
30     else
31     {
32         opp = COMPUTER; value = beta;
33     }
34     Outer:
35     for( int row = 0; row < 3; row++ )
36         for( int column = 0; column < 3; column++ )
37             if( squareIsEmpty( row, column ) )
38             {
39                 place( row, column, side );
40                 reply = chooseMove( opp, alpha, beta, depth + 1 );
41                 place( row, column, EMPTY );

```

*Continúa*

---

Figura 10.15 El algoritmo de tres en raya con poda alfa-beta y tabla de transposición.

```
42
43     if( side == COMPUTER && reply.val > value || 
44         side == HUMAN && reply.val < value )
45     {
46         if( side == COMPUTER )
47             alpha = value = reply.val;
48         else
49             beta = value = reply.val;
50
51         bestRow = row; bestColumn = column;
52         if( alpha >= beta )
53             break Outer; // Refutación
54     }
55 }
56
57 if( depth <= tableDepth )
58     transpositions.put( thisPosition, value );
59
60 return new Best( value, bestRow, bestColumn );
61 }
```

Figura 10.15 (Continuación).

### 10.2.3 Ajedrez por computadora

En un juego complejo como el ajedrez o el Go, no es factible prolongar la búsqueda hasta los nodos terminales: algunas estimaciones indican que hay aproximadamente  $10^{100}$  posiciones legales en el ajedrez, y ni todos los trucos del mundo permitirían reducir ese número a un nivel manejable. En este caso, tenemos que detener la búsqueda después de alcanzar una cierta profundidad de recursión. Los nodos en los que la recursión se detiene se convierten en nodos terminales. Estos nodos terminales se evalúan con una función que estima el valor de la posición. Por ejemplo, en un programa de ajedrez, la función de evaluación mide variables tales como la cantidad relativa y la importancia de las piezas, así como otros factores posicionales.

Las posiciones terminales no se pueden explorar en el ajedrez por computadora. En los mejores programas, la función de evaluación incorpora una considerable cantidad de conocimientos.

Las computadoras son especialmente proclives a realizar jugadas que implican combinaciones complejas que tienen como resultado intercambios de material. La razón es que la fortaleza relativa de las piezas se puede evaluar fácilmente. Sin embargo, ampliar la profundidad de búsqueda simplemente un nivel requiere multiplicar la velocidad de procesamiento aproximadamente por seis (porque el número de posiciones se multiplica aproximadamente por 36). Cada nivel adicional de búsqueda mejora enormemente las capacidades del programa, hasta un cierto límite (que parece que ha sido alcanzado ya por los mejores programas). Por otro lado, generalmente, las computadoras no son demasiado buenas a la hora de jugar tranquilas partidas posicionales en las

Los mejores programas de ajedrez por computadora juegan en el nivel de gran maestro.

que se requieren evaluaciones más sutiles y un conocimiento más profundo del juego. Sin embargo, esta carencia solo es aparente cuando la computadora se enfrenta a un oponente muy fuerte. Los programas para jugar al ajedrez por computadora que se comercializan de forma masiva son mejores que casi todos los jugadores humanos actuales, salvo unos pocos de ellos.

En 1997, el programa informático *Deep Blue*, utilizando una enorme potencia de procesamiento (que permitió evaluar hasta 200 millones de movimientos por segundo), fue capaz de derrotar al entonces campeón mundial de ajedrez en una competición a seis partidas. Su función de evaluación, aunque es alto secreto, se sabe que contiene un gran número de factores, que fue elaborada con la ayuda de varios grandes maestros del ajedrez y que se obtuvo como resultado de años de experimentación. Escribir el mejor programa de ajedrez por computadora no es, ciertamente, una tarea trivial.

## Resumen

---

En este capítulo hemos presentado una aplicación de la búsqueda binaria y algunas técnicas algorítmicas que se utilizan comúnmente para resolver sopas de letras y a la hora de practicar juegos como el ajedrez, las damas y Otelo. Los mejores programas para estos juegos son todos ellos de primera fila. Sin embargo, el juego de Go parece ser demasiado complejo para las técnicas existentes de búsqueda por computadora.



### Conceptos clave

**estrategia minimax** Una estrategia recursiva que permite a la computadora seleccionar un movimiento óptimo en una partida de tres en raya. (418)

**poda alfa-beta** Una técnica utilizada para reducir el número de posiciones que se evalúan en una búsqueda minimax. Alfa es el valor que el jugador humano tiene que refutar, mientras que beta es el valor que la computadora tiene que refutar. (419)

**posición terminal** Una posición en un juego que se puede evaluar de forma inmediata. (418)

**refutación** Una jugada de respuesta que demuestra que un movimiento propuesto no constituye una mejora con respecto a los movimientos anteriormente considerados. Si encontramos una refutación, no tenemos que examinar más movimientos y la llamada recursiva puede volver. (418)

**sopa de letras** Un programa que requiere buscar palabras en una cuadrícula bidimensional de letras. Las palabras pueden estar orientadas en una de ocho direcciones posibles. (409)

**tabla de transposición** Un mapa que almacena las posiciones previamente evaluadas. (421)



## Errores comunes

1. A la hora de utilizar una tabla de transposición, hay que limitar el número de posiciones almacenadas, para evitar quedarse sin memoria.
2. Verificar las suposiciones es importante. Por ejemplo, en el problema de la sopa de letras, asegúrese de que el diccionario esté ordenado. Un error común consiste en olvidarse comprobar las suposiciones.



## Internet

Tanto el juego de la sopa de letras como el de las tres en raya están completamente codificados, aunque la interfaz de este último es bastante mejorable.

### **WordSearch.java**

Contiene el algoritmo de la sopa de letras.

### **TicTacToe.java**

Contiene la clase TicTacToe; se proporciona una rutina main por separado en **TicTacMain.java**.



## Ejercicios

### EN RESUMEN

- 10.1** Para la situación de la Figura 10.16
- a. ¿Cuál de las respuestas al movimiento  $C_2$  es una refutación?
  - b. ¿Cuál es el valor de la posición?
- 10.2** ¿Qué comprobaciones de error faltan en la Figura 10.6?

### EN TEORÍA

- 10.3** Explique cómo cambia el tiempo de ejecución del algoritmo de la sopa de letras cuando
- a. El número de palabras se duplica.
  - b. Se duplica el número de filas y columnas (simultáneamente).
- 10.4** Verifique que, si  $x$  es un prefijo de alguna palabra en la matriz ordenada  $a$ , entonces  $x$  es un prefijo de la palabra situada en el índice que `prefixSearch` devuelve.

### EN LA PRÁCTICA

- 10.5** Compare el rendimiento del algoritmo de la sopa de letras con y sin la búsqueda de prefijos.
- 10.6** Sustituya el `HashMap` por un `TreeMap` en el programa de las tres en raya y compare el rendimiento de ambas versiones.

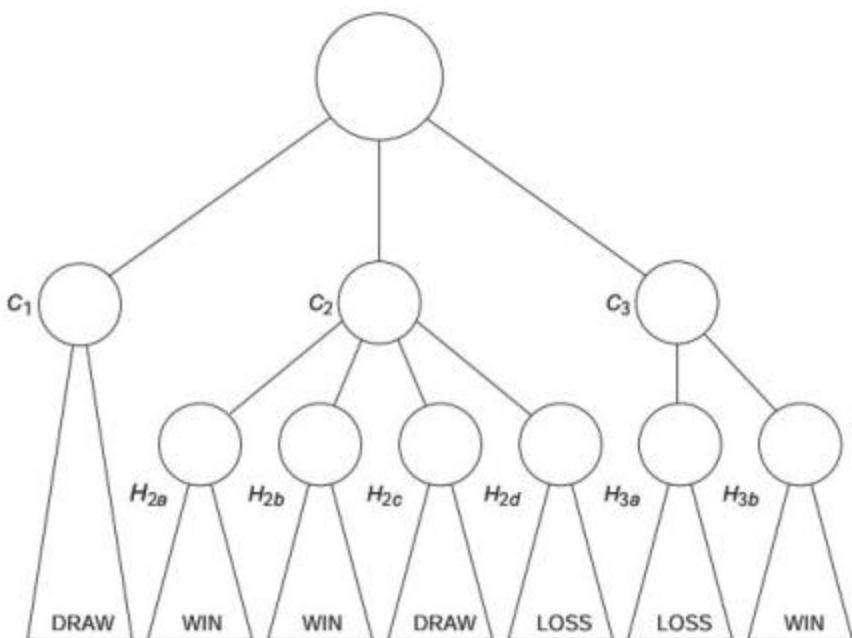


Figura 10.16 Ejemplo de poda alfa-beta para el Ejercicio 10.1.

- 10.7** Para el problema de la sopa de letras, sustituya la búsqueda binaria por una búsqueda secuencial. ¿Cómo afecta este cambio al rendimiento?

### PROYECTOS DE PROGRAMACIÓN

- 10.8** El juego de Boggle está formado por una cuadrícula de letras y una lista de palabras. El objetivo es encontrar palabras en la cuadrícula, pero con la condición de que dos letras adyacentes deben ser adyacentes en la cuadrícula (es decir, al norte, al sur, al este o al oeste una de otra) y cada elemento de la cuadrícula puede utilizarse como mucho una vez por palabra. Escriba un programa para jugar a Boggle.
- 10.9** Escriba un programa para jugar a las tres en raya en un tablero de  $5 \times 5$ , donde cuatro en fila ganan. ¿Puede realizar la búsqueda para nodos terminales?
- 10.10** Escriba un programa para jugar a MAXIT. El tablero está representado por una cuadrícula  $N \times N$  de números aleatoriamente situados al principio del juego. Se designa una posición como posición inicial actual. Los dos jugadores juegan alternativamente. En cada turno, un jugador debe seleccionar un elemento de la cuadrícula en la fila o columna actual. El valor de la posición seleccionada se suma a la puntuación del jugador y esa posición se convierte en la posición actual y no puede volver a ser seleccionada. Los jugadores van jugando alternativamente hasta que todos los elementos de la cuadrícula en la fila y la columna actuales han sido seleccionados, en cuyo momento el juego termina y gana el jugador con la puntuación más alta.



## Referencias

Si está interesado en los juegos por computadora un buen punto de partida es el artículo citado en [1]. En este número especial de la revista, dedicado exclusivamente a este tema, también encontrará una gran cantidad de información y referencias a otros trabajos dedicados al ajedrez, las damas y otros juegos por computadora.

1. K. Lee y S. Mahajan, "The Development of a World Class Othello Program", *Artificial Intelligence* 43 (1990), 21–36.



## Pilas y compiladores

Las pilas se utilizan de manera intensiva en los compiladores. En este capítulo vamos a presentar dos componentes simples de un compilador: un comprobador de equilibrado de los símbolos y una calculadora simple. Vamos a hacer esto para proporcionar algunos ejemplos de algoritmos sencillos que usan pilas y para mostrar también cómo se emplean las clases de la API de Colecciones descritas en el Capítulo 6.

En este capítulo, veremos

- Cómo utilizar una pila para comprobar el equilibrado de los símbolos.
- Cómo utilizar una *máquina de estados* para analizar los símbolos en un programa de análisis de equilibrado de los símbolos.
- Cómo utilizar un *análisis de precedencia de operadores* para evaluar expresiones infijas en un programa sencillo de calculadora.

### 11.1 Comprobador de equilibrado de los símbolos

Como se ha explicado en la Sección 6.6, los compiladores comprueban los programas para ver que no existan errores sintácticos. Frecuentemente, sin embargo, la falta de un símbolo (como por ejemplo la falta de un \*/ que marca el final de un comentario o de una llave de cierre, }) puede hacer que el compilador genere numerosas líneas de mensajes de diagnóstico sin llegar a identificar el error real. Una herramienta útil para ayudar a depurar los mensajes de error de los compiladores es un programa que comprueba si los símbolos están equilibrados. En otras palabras, cada { debe corresponderse con una }, cada [ debe corresponder con un ], etc. Sin embargo, limitarse a contar el número de veces que aparece cada símbolo no es suficiente. Por ejemplo, la secuencia [0] es legal, pero la secuencia [(]) es errónea.

#### 11.1.1 Algoritmo básico

En el contexto de este problema, un pila es útil, porque sabemos que cuando nos encontramos con un símbolo de cierre como ), este deberá corresponderse con el símbolo (, que nos hayamos encontrado más recientemente. Por tanto, colocando un símbolo de apertura en una pila, podemos

Se puede utilizar una pila para detectar símbolos para los que no existe el otro símbolo correspondiente.

determinar fácilmente si un símbolo de cierre tiene sentido. Específicamente, tenemos el siguiente algoritmo:

1. Crear una pila vacía.
2. Leer los símbolos hasta el final del archivo.
  - a. Si el símbolo es un símbolo de apertura, introducirlo en la pila.
  - b. Si es un símbolo de cierre, hacer lo siguiente.
    - i. Si la pila está vacía, informar de un error.
    - ii. En caso contrario extraer un símbolo de la pila. Si el símbolo extraído no es el correspondiente símbolo de apertura, informar de un error.
3. Al final del archivo, si la pila no está vacía, informar de un error.

En este algoritmo, ilustrado en la Figura 11.1, los símbolos cuarto, quinto y sexto generan errores. El símbolo } es un error porque el símbolo extraído de la parte superior de la pila es un (, por lo que se detecta que no existe una correspondencia. El símbolo ) es un error porque la pila está vacía, así que no hay ningún símbolo ( correspondiente. El símbolo [ es un error que se detecta cuando nos encontramos el final del archivo y la pila no está vacía.

Los símbolos incluidos en los comentarios, constantes de cadenas de caracteres y constantes de caracteres no necesitan estar equilibrados.

Para hacer que esta técnica funcione para los programas Java, necesitamos considerar todos los contextos en los que los paréntesis, llaves y corchetes no tienen por qué tener una correspondencia. Por ejemplo, no debemos considerar un paréntesis como un símbolo si este aparece dentro de un comentario, dentro de una constante de cadena de caracteres o dentro de una cadena de caracteres. Necesitamos, por tanto, rutinas que nos permitan saltarnos los comentarios, la constantes de cadenas de caracteres y las constantes de caracteres. Una constante de carácter en Java puede ser difícil de reconocer debido a las muchas secuencias de escape posibles, así que tenemos que simplificar las cosas. Queremos diseñar un programa que funcione para la mayoría de los archivos de entrada que probablemente nos vayamos a encontrar.

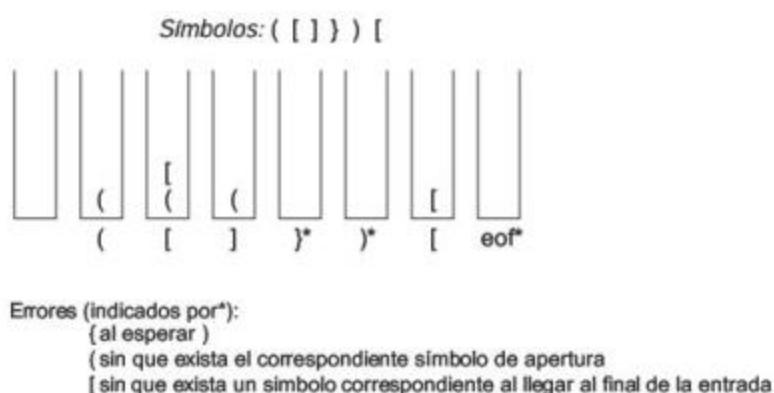


Figura 11.1 Operaciones con la pila en un algoritmo de comprobación de equilibrado de los símbolos.

Para que el programa sea útil, no solo debemos informar de las faltas de correspondencia, sino también tratar de identificar dónde se han producido. En consecuencia, llevaremos la cuenta de los números de las líneas en las que los símbolos aparecen. Cuando nos encontramos un error, siempre resulta difícil obtener un mensaje preciso. Si existe un símbolo } extra, ¿quiere eso decir que el símbolo } sobra? ¿O lo que falta es un símbolo { anterior? Vamos a hacer el tratamiento de errores lo más simple posible, pero una vez que se haya informado de un error, el programa puede verse confundido y comenzar a indicar la existencia de muchos otros errores. Por tanto, solo el primer error puede ser considerado significativo. Aun así, el programa aquí desarrollado es muy útil.

Los números de línea son necesarios para poder generar mensajes de error significativos.

### 11.1.2 Implementación

El programa tiene dos componentes básicos. Una parte, denominada *tokenización* o *simbolización*, que es el proceso de analizar un flujo de entrada en busca de los símbolos de apertura y de cierre (*tokens* o símbolos sintácticos) y generar la secuencia de símbolos sintácticos que es necesario reconocer. La segunda parte consiste en ejecutar el algoritmo de comprobación de equilibrado de los símbolos, basándose en los símbolos sintácticos. Los dos componentes básicos se representan como clases separadas.

La simbolización es el proceso de generar la secuencia de símbolos sintácticos que es necesario reconocer.

La Figura 11.2 muestra el esqueleto de la clase `Tokenizer` y la Figura 11.3 muestra el esqueleto de la clase `Balance`. La clase `Tokenizer` proporciona un constructor que requiere un `Reader` y luego proporciona una serie de métodos accesores que se pueden utilizar para obtener:

- El siguiente símbolo sintáctico (o bien un símbolo de apertura/cierre para el código de este capítulo o un identificador en el caso del código del Capítulo 12).
- El número de línea actual.
- El número de errores (comillas y comentarios para los que no existe correspondencia).

La clase `Tokenizer` mantiene la mayor parte de esta información en miembros de datos privados. La clase `Balance` también proporciona un constructor similar, pero su única rutina públicamente visible es `checkBalance`, mostrada en la línea 24. Todo lo demás son rutinas de soporte o miembros de datos de la clase.

Comenzaremos describiendo la clase `Tokenizer`. Es una referencia a un objeto `PushbackReader` y se inicializa durante la construcción. Debido a la jerarquía de E/S (véase la Sección 4.5.3), puede construirse con cualquier objeto `Reader`. El carácter que está siendo actualmente explorado se almacena en `ch` y el número de línea actual se almacena en `currentLine`. Por último, en la línea 53 se declara un entero que cuenta el número de errores. El constructor, mostrado en las líneas 19 a 21, inicializa la cuenta de errores a 0 y el número de línea actual a 1 y establece la referencia `PushbackReader`.

Ahora podemos implementar los métodos de la clase, los cuales, como ya hemos comentado, se ocupan de llevar la cuenta de la línea actual y de tratar de diferenciar los caracteres que representan los símbolos de apertura y cierre, de aquellos contenidos en comentarios, constantes de caracteres y constantes de cadenas de caracteres. Este proceso general de reconocimiento de

```

1 import java.io.Reader;
2 import java.io.PushbackReader;
3 import java.io.IOException;
4
5 // Clase Tokenizer.
6 //
7 // CONSTRUCCIÓN: con un objeto Reader
8 // *****OPERACIONES PÚBLICAS*****
9 // char getNextOpenClose( ) --> Obtener siguiente símbolo apert./cierre
10 // int getLineNumber( ) --> Devolver nº de línea actual
11 // int getErrorCount( ) --> Devolver nº de errores de análisis sintáctico
12 // String getNextID( ) --> Obtener siguiente identificador Java
13 // (véase la Sección 12.2)
14 // *****ERRORES*****
15 // Se realiza la comprobación de errores en comentarios y comillas
16
17 public class Tokenizer
18 {
19     public Tokenizer( Reader inStream )
20         { errors = 0; ch = '\0'; currentLine = 1;
21           in = new PushbackReader( inStream ); }
22
23     public static final int SLASH_SLASH = 0;
24     public static final int SLASH_STAR = 1;
25
26     public int getLineNumber( )
27         { return currentLine; }
28     public int getErrorCount( )
29         { return errors; }
30     public char getNextOpenClose( )
31         { /* Figura 11.7 */ }
32     public char getNextID( )
33         { /* Figura 12.29 */ }
34
35     private boolean nextChar( )
36         { /* Figura 11.4 */ }
37     private void putBackChar( )
38         { /* Figura 11.4 */ }
39     private void skipComment( int start )
40         { /* Figura 11.5 */ }
41     private void skipQuote( char quoteType )
42         { /* Figura 11.6 */ }

```

*Continúa***Figura 11.2** El esqueleto de la clase `Tokenizer`, utilizado para extraer símbolos sintácticos de un flujo de datos de entrada.

```
43  private void processSlash( )
44  { /* Figura 11.7 */ }
45  private static final boolean isIdChar( char ch )
46  { /* Figura 12.27 */ }
47  private String getRemainingString( )
48  { /* Figura 12.28 */ }

49
50  private PushbackReader in; // El flujo de entrada
51  private char ch;           // Carácter actual
52  private int currentLine;   // Línea actual
53  private int errors;        // Número de errores encontrados
54 }
```

Figura 11.2 (Continuación).

los símbolos sintácticos incluidos dentro de un flujo de símbolos se denomina *análisis léxico*. La Figura 11.4 muestra una par de rutinas, `nextChar` y `putBackChar`. El método `nextChar` lee el siguiente carácter de `in`, lo asigna a `ch` y actualiza `currentLine` si se encuentra una nueva línea. Devuelve `false` solo si se ha alcanzado el final del archivo. El procedimiento complementario, `putBackChar`, pone de nuevo el carácter actual, `ch`, en el flujo de entrada, y decremente `currentLine` si el carácter es una nueva línea. Claramente, `putBackChar` debe ser invocado como máximo una vez entre llamadas sucesivas a `nextChar`; como se trata de una rutina privada, no nos preocupa el que alguien pueda abusar de este mecanismo desde la clase escrita por el usuario. Volver a poner los caracteres en el flujo de datos de entrada es una técnica comúnmente utilizada en el análisis sintáctico. En muchos casos, estamos obligados a leer un carácter de más, por lo que resulta útil poder deshacer la operación de lectura. En nuestro caso, esto sucede después de procesar un símbolo `/`. Debemos determinar si el siguiente carácter marca el símbolo de comienzo de un comentario, si no lo hace, no podemos limitarnos a descartarlo, porque podría tratarse de un símbolo de apertura o de cierre o de una comilla. Por tanto, lo que hacemos es pretender que ese carácter nunca ha sido leído.

A continuación está la rutina `skipComment`, mostrada en la Figura 11.5. Su propósito es saltarse los caracteres del comentario y posicionarse dentro del flujo de datos de entrada, de tal manera que el siguiente carácter leido sea el primer carácter después de que termine el comentario. Esta técnica se ve complicada por el hecho de que los comentarios pueden comenzar con los símbolos `//`, en cuyo caso el comentario termina en el mismo punto en que lo hace la línea, o por `/*`, en cuyo caso el comentario termina con `*/`.<sup>1</sup> En el caso de `//`, seguimos obteniendo el siguiente carácter hasta alcanzar el final del archivo (en cuyo caso, la primera mitad del operador `&&` fallará) o hasta obtener una nueva línea. En ese punto, volvemos. Observe que el número de línea es actualizado automáticamente por `nextChar`. En el otro caso, el de `/*`, el procesamiento comienza en la línea 17.

El análisis léxico se utiliza para ignorar los comentarios y reconocer los símbolos.

<sup>1</sup> No vamos a considerar aquí los casos especiales en los que interviene \, en lugar de `/**/`.

```
1 import java.io.Reader;
2 import java.io.FileReader;
3 import java.io.IOException;
4 import java.io.InputStreamReader;
5
6 import java.util.Stack;
7
8
9 // Clase Balance: comprobar equilibrado de los símbolos
10 //
11 // CONSTRUCCIÓN: con un objeto Reader
12 // *****OPERACIONES PÚBLICAS*****
13 // int checkBalance( ) --> Imprimir faltas de correspondencias
14 // devolver número de errores
15 // *****ERRORES*****
16 // Se realiza la comprobación de errores en comentarios y comillas
17 // main comprueba el equilibrado de los símbolos.
18
19 public class Balance
20 {
21     public Balance( Reader inStream )
22     { errors = 0; tok = new Tokenizer( inStream ); }
23
24     public int checkBalance( )
25     { /* Figura 11.8 */ }
26
27     private Tokenizer tok;
28     private int errors;
29
30     /**
31      * Clase anidada Symbol;
32      * representa lo que se colocará en la pila.
33      */
34     private static class Symbol
35     {
36         public char token;
37         public int theLine;
38
39         public Symbol( char tok, int line )
40         {
41             token = tok;
42             theLine = line;
43         }
44     }
45
46     private void checkMatch( Symbol opSym, Symbol clSym )
47     { /* Figura 11.9 */ }
48 }
```

Figura 11.3 Esqueleto de clase para un programa de equilibrado de los símbolos.

```

1  /**
2   * nextChar configura ch basándose en el sig. car. del flujo de entrada.
3   * putBackChar pone el carácter de nuevo en el flujo de entrada.
4   * Sólo debe utilizarse una vez después de una llamada a nextChar.
5   * Ambas rutinas ajustan currentLine en caso necesario.
6   */
7  private boolean nextChar( )
8  {
9      try
10     {
11         int readVal = in.read( );
12         if( readVal == -1 )
13             return false;
14         ch = (char) readVal;
15         if( ch == '\n' )
16             currentLine++;
17         return true;
18     }
19     catch( IOException e )
20     {
21         return false;
22     }
23     private void putBackChar( )
24     {
25         if( ch == '\n' )
26             currentLine--;
27         try
28         {
29             in.unread( (int) ch );
30         }
31     }

```

**Figura 11.4** La rutina nextChar para leer el siguiente carácter, actualizar currentLine si fuera necesario y devolver true si no estamos al final del archivo; y la rutina putBackChar para colocar de nuevo ch en el flujo de entrada y actualizar currentLine en caso necesario.

La rutina skipComment utiliza una máquina de estados simplificada. La *máquina de estados* es una técnica común utilizada para analizar sintácticamente símbolos; en cualquier punto, esa máquina se encuentra en un cierto estado, y cada carácter de entrada la lleva a otro estado nuevo. Al final, termina por alcanzar un estado que indica que se ha reconocido un símbolo.

En skipComment, en cualquier momento, la máquina de estados habrá logrado encontrar la correspondencia con 0, 1 o 2 caracteres del terminador /\*, lo que se corresponde con los estados 0, 1 y 2. Si logra encontrar la correspondencia con dos caracteres, la rutina puede volver. Por tanto, dentro del bucle, solo puede estar en el estado 0 o 1, porque si se encuentra en el

La máquina de estados es una técnica común utilizada para analizar sintácticamente símbolos; en cualquier punto, esa máquina se encuentra en un cierto estado, y cada carácter de entrada la lleva a otro estado nuevo. Al final, termina por alcanzar un estado que indica que se ha reconocido un símbolo.

```

1  /**
2   * Precondición: estamos a punto de procesar un comentario;
3   * ya hemos visto el símbolo de inicio de comentario.
4   * Postcondición: la posición dentro del flujo estará fijada
5   * inmediatamente después del símbolo sintáctico de final de comentario.
6   */
7  private void skipComment( int start )
8  {
9      if( start == SLASH_SLASH )
10     {
11         while( nextChar( ) && ( ch != '\n' ) )
12             ;
13         return;
14     }
15
16     // Buscar una secuencia */
17     boolean state = false;      // True si hemos visto *
18
19     while( nextChar( ) )
20     {
21         if( state && ch == '*' )
22             return;
23         state = ( ch == '*' );
24     }
25     errors++;
26     System.out.println( "Unterminated comment!" );
27 }

```

Figura 11.5 La rutina skipComment para posicionarnos inmediatamente después de un comentario ya comenzado.

estado 1 y ve un carácter /, vuelve inmediatamente. Por tanto, el estado se puede representar mediante una variable booleana que será true si la máquina de estados se encuentra en el estado 1. Si no vuelve, o bien pasa al estado 1 si encuentra un carácter \* o pasa al estado 0 si no lo encuentra. Este procedimiento se enumera sucintamente en la línea 23.

Si nunca encontramos el símbolo de final de comentario, nextChar terminará por devolver false y el bucle while terminará, provocando un mensaje de error. El método skipQuote, mostrado en la Figura 11.6, es similar. Aquí, el parámetro es el carácter de comilla de apertura, que puede ser " o '. En cualquiera de los casos, necesitamos ver dicho carácter como comilla de cierre. Sin embargo, debemos estar preparados para tratar el carácter \; en caso contrario, nuestro programa informará de errores incluso cuando se ejecute con su propio código fuente. De ese modo, iremos procesando repetidamente caracteres. Si el carácter actual es una comilla de cierre, habremos terminado. Si se trata de una nueva línea, entonces tendremos una constante de carácter o una constante de cadena de caracteres sin terminar. Y si se trata de una barra inclinada a la izquierda, tendremos que procesar un carácter extra sin examinarlo.

```
1  /**
2   * Precondición: estamos a punto de procesar unas comillas;
3   * ya hemos visto el símbolo de comillas de apertura.
4   * Postcondición: el flujo de datos se encontrará inmediatamente
5   * después de las comillas correspondientes
6   */
7  private void skipQuote( char quoteType )
8  {
9      while( nextChar( ) )
10     [
11         if( ch == quoteType )
12             return;
13         if( ch == '\n' )
14         [
15             errors++;
16             System.out.println( "Missing closed quote at line " +
17                                 currentLine );
18             return;
19         ]
20         else if( ch == '\\' )
21             nextChar( );
22     ]
23 }
```

Figura 11.6 La rutina `skipQuote` para desplazarnos más allá de una constante de caracteres o de una cadena que ya haya comenzado.

Una vez que hemos escrito la rutina para saltar las partes del programa necesarias, escribir `getNextOpenClose` es más fácil. El cuerpo principal de la lógica de la rutina se deja para `processSlash`. Si el carácter actual es un /, leemos un segundo carácter para ver si tenemos un comentario. En caso afirmativo, invocamos a `skipComment`; si no, deshacemos la segunda lectura. Si tenemos una comilla, llamamos a `skipQuote`. Si tenemos un símbolo de apertura o de cierre, podemos volver. En caso contrario, continuamos leyendo hasta terminar consumiendo todos los caracteres de entrada o hasta encontrar un símbolo de apertura o cierre. Tanto `getNextOpenClose` como `processSlash` se muestran en la Figura 11.7.

Los métodos `getLineNumber` y `getErrorCount` son métodos de una sola línea que devuelven los valores de los correspondientes miembros de datos y se muestran en la Figura 11.2. Hablaremos de la rutina `getNextID` en la Sección 12.2.2, cuando sea necesaria.

En la clase `Balance`, el algoritmo de equilibrado de símbolos requiere que coloquemos los símbolos en una pila. Para imprimir mensajes de diagnóstico, almacenamos un número de línea con cada símbolo, como se ha indicado anteriormente en la clase anidada `Symbol` en las líneas 34 a 44 de la Figura 11.3.

La rutina `checkBalance` se implementa como se muestra en la Figura 11.8. Esta rutina se adapta a la descripción del algoritmo que hemos proporcionado de forma casi textual. En la línea 9 se declara

```

1  /**
2   * Obtener el siguiente simbolo de apertura o cierre.
3   * Devolver false si estamos al final del archivo.
4   * Saltarse los comentarios y las constantes de caracteres y de cadena.
5   */
6  public char getNextOpenClose( )
7  {
8      while( nextChar( ) )
9      {
10         if( ch == '/' )
11             processSlash( );
12         else if( ch == '\'' || ch == '"' )
13             skipQuote( ch );
14         else if( ch == '(' || ch == '[' || ch == '{' ||
15             ch == ')' || ch == ']' || ch == '}' )
16             return ch;
17     }
18     return '\0'; // Fin del archivo
19 }
20
21 /**
22  * Después de ver la barra de apertura, procesar el siguiente carácter.
23  * Si se trata de un principio de comentario, procesarlo; en otro caso,
24  * volver a poner en el flujo de datos el carácter si no es una nueva línea
25  */
26 private void processSlash( )
27 {
28     if( nextChar( ) )
29     {
30         if( ch == '*' )
31         {
32             // Comentario Javadoc
33             if( nextChar( ) && ch != '*' )
34                 putBackChar( );
35             skipComment( SLASH_STAR );
36         }
37         else if( ch == '/' )
38             skipComment( SLASH_SLASH );
39         else if( ch != '\n' )
40             putBackChar( );
41     }
42 }

```

Figura 11.7 La rutina `getNextOpenClose` para saltarse los comentarios y las comillas y devolver el siguiente carácter de apertura o cierre, junto con la rutina `processSlash`.

```
1  /**
2  * Imprimir un mensaje de error para los símbolos no equilibrados.
3  * @return número de errores detectados.
4  */
5  public int checkBalance( )
6  {
7      char ch;
8      Symbol match = null;
9      Stack<Symbol> pendingTokens = new Stack<Symbol>();
10
11     while( ( ch = tok.getNextOpenClose( ) ) != '\0' )
12     {
13         Symbol lastSymbol = new Symbol( ch, tok.getLineNumber( ) );
14
15         switch( ch )
16         {
17             case '(': case '[': case '{':
18                 pendingTokens.push( lastSymbol );
19                 break;
20
21             case ')': case ']': case '}':
22                 if( pendingTokens.isEmpty( ) )
23                 {
24                     errors++;
25                     System.out.println( "Extraneous " + ch +
26                                         " at line " + tok.getLineNumber( ) );
27                 }
28                 else
29                 {
30                     match = pendingTokens.pop( );
31                     checkMatch( match, lastSymbol );
32                 }
33                 break;
34
35             default: // No puede producirse
36                 break;
37         }
38     }
39
40     while( !pendingTokens.isEmpty( ) )
41     {
42         match = pendingTokens.pop( );
43         System.out.println( "Unmatched " + match.token +
44 " at line " + match.theLine );
45         errors++;
46     }
47     return errors + tok.getErrorCount( );
48 }
```

Figura 11.8 El algoritmo checkBalance.

una pila que almacena los símbolos de apertura pendientes. Los símbolos de apertura se insertan en la pila junto con el número de línea actual. Cuando se encuentra un símbolo de cierre y la pila está vacía, eso quiere decir que el símbolo de cierre sobra; en caso contrario, extraemos el elemento superior de la pila y verificamos que el símbolo de apertura que estaba almacenado en la pila se corresponde con el símbolo que acabamos de leer. Para hacer esto, utilizamos la rutina `checkMatch`, que se muestra en la Figura 11.9. Una vez que nos encontramos con el final de la entrada, todos los símbolos que permanezcan en la pila no tendrán sus símbolos de cierre correspondientes; por ello, los imprimimos repetidamente en el bucle `while` que comienza en la línea 40. Después, devolvemos el número total de errores detectados.

La rutina `checkBalance` realiza todo el trabajo algorítmico.

La implementación actual permite realizar varias llamadas a `checkBalance`. Sin embargo, si no se reinicializa externamente el flujo de datos de entrada, todo lo que sucederá es que se detecta inmediatamente el final del archivo y se vuelve de forma inmediata. Podemos añadir funcionalidad a la clase `Tokenizer`, permitiéndola cambiar el origen del flujo de datos y luego añadir funcionalidad a la clase `Balance` para cambiar el flujo de datos de entrada (pasándola el cambio de la clase `Tokenizer`). Dejamos esta tarea para el lector, para que la realice en el Ejercicio 11.8.

La Figura 11.10 muestra que esperamos que se cree un objeto `Balance` y que luego se invoque `checkBalance`. En nuestro ejemplo, si no hay ningún argumento de la línea de comandos, se conecta el `Reader` asociado a `System.in` (a través del puente `InputStreamReader`); en caso contrario, utilizamos repetidamente objetos `Reader` asociados con los archivos proporcionados en la lista de argumentos de la línea de comandos.

```

1  /**
2   * Imprimir un mensaje de error si clSym no se corresponde con opSym.
3   * Actualizar errores.
4   */
5  private void checkMatch( Symbol opSym, Symbol clSym )
6  {
7      if( opSym.token == '*' && clSym.token != '*' || 
8          opSym.token == '[' && clSym.token != ']' || 
9          opSym.token == '(' && clSym.token != ')' )
10     {
11         System.out.println( "Found " + clSym.token + " on line " +
12             tok.getLineNumber( ) + "; does not match " + opSym.token
13             + " at line " + opSym.theLine );
14         errors++;
15     }
16 }
```

Figura 11.9 La rutina `checkMatch` para comprobar que el símbolo de cierre se corresponde con el símbolo de apertura.

```

1 // Rutina main para el comprobador de equilibrado de los símbolos.
2 // Si no hay parámetros de linea de comandos, se usa la entrada estándar.
3 // En caso contrario, se usan los archivos de la linea comandos.
4 public static void main( String [ ] args )
5 {
6     Balance p;
7
8     if( args.length == 0 )
9     {
10
11         p = new Balance( new InputStreamReader( System.in ) );
12         if( p.checkBalance( ) == 0 )
13             System.out.println( "No errors!" );
14         return;
15     }
16
17     for( int i = 0; i < args.length; i++ )
18     {
19         FileReader f = null;
20         try
21         {
22             f = new FileReader( args[ i ] );
23
24             System.out.println( args[ i ] + ": " );
25             p = new Balance( f );
26             if( p.checkBalance( ) == 0 )
27                 System.out.println( "...no errors!" );
28         }
29         catch( IOException e )
30             { System.err.println( e + args[ i ] ); }
31         finally
32         {
33             try
34             { if( f != null ) f.close( ); }
35             catch( IOException e )
36             { }
37         }
38     }
39 }
```

Figura 11.10 La rutina main con argumentos de la línea de comandos.

## 11.2 Una calculadora simple

Algunas de las técnicas utilizadas para implementar compiladores pueden emplearse, a menor escala, en la implementación de una calculadora de bolsillo típica. Normalmente, las calculadoras evalúan *expresiones infijas*, como  $1+2$ , las cuales constan de un operador binario con argumentos a su izquierda y a su derecha. Este formato, aunque suele ser muy fácil de evaluar, puede ser más complejo. Considere la siguiente expresión

$1 + 2 * 3$

En una expresión infija, un operador binario tiene argumentos a su izquierda y a su derecha.

Matemáticamente, esta expresión se evalúa como 7, porque el operador de multiplicación tiene mayor precedencia que el de la suma. Algunas calculadoras darían como resultado el valor 9, lo que ilustra que una evaluación simple de izquierda a derecha no es suficiente; no podemos comenzar evaluando  $1+2$ . Ahora considere las expresiones

$10 - 4 - 3$   
 $2 ^ 3 ^ 3$

Cuando hay varios operadores, la precedencia y la asociatividad determinan cómo se procesan esos operadores.

en donde  $^$  es el operador de exponenciación. ¿Qué resta y qué exponenciación hay que evaluar primero? Por un lado, las restas se procesan de izquierda a derecha, lo que nos da el resultado 3. Por otro lado, la exponenciación generalmente se procesa de derecha a izquierda, reflejando así la operación matemática  $2^{3^3}$  en lugar de  $(2^3)^3$ . Por tanto, la resta es asociativa de izquierda a derecha, mientras que la exponenciación es asociativa de derecha a izquierda. Todas estas posibilidades sugieren que evaluar una expresión como

$1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2$

sería muy complicado.

Si los cálculos se realizan con matemática de números enteros (es decir, redondeando hacia abajo en la división), la respuesta es -8. Para mostrar este resultado, vamos a insertar paréntesis con el fin de clarificar el orden de los cálculos:

$(1 - 2) - (((4 ^ 5) * 3) * 6) / (7 ^ (2 ^ 2))$

Aunque los paréntesis hacen que el orden de las evaluaciones no sea ambiguo, no necesariamente contribuyen a clarificar el mecanismo de evaluación. Otra forma distinta de expresión, denominada *expresión postfija*, que puede ser evaluada mediante una máquina postfija sin ninguna regla de precedencia, proporciona un mecanismo directo para la evaluación de expresiones. En las siguientes secciones vamos a explicar cómo funciona. En primer lugar, examinaremos la forma de expresión postfija y mostraremos cómo pueden evaluarse las expresiones mediante una simple exploración de izquierda a derecha. A continuación, mostraremos algorítmicamente cómo las anteriores expresiones, que se presentan como expresiones infijas, pueden convertirse a postfijas. Finalmente, proporcionaremos un programa Java que evalúa expresiones infijas que contengan operadores aditivos, multiplicativos y de exponenciación, así como paréntesis para indicar la precedencia. Utilizaremos un algoritmo denominado *análisis de precedencia de operadores* para convertir una expresión infija en una expresión postfija, con el fin de evaluar la expresión infija.

### 11.2.1 Máquinas postfijas

Una expresión postfija es una serie de operadores y operandos. Una máquina postfija se utiliza para evaluar una expresión postfija de la forma siguiente. Cuando se ve un operando, ese operando se inserta en una pila. Cuando se ve un operador, se extrae de la pila el número apropiado de operandos, se evalúa el operador y el resultado vuelve a ponerse en la pila. Para los operadores binarios, que son los más comunes, se extraen dos operandos. Una vez que se ha evaluado la expresión postfija completa, el resultado debería ser un único elemento situado en la pila, que representa la respuesta. La forma postfija representa una forma natural de evaluar expresiones, porque no se necesitan reglas de precedencia.

Una expresión postfija se puede evaluar de la forma siguiente: los operandos se insertan en una única pila. Un operador extrae sus operandos y luego inserta el resultado. Al final de la evaluación, la pila deberá contener un único elemento, que representa el resultado.

Un ejemplo simple sería la expresión postfija

1 2 3 \* +

La evaluación se realiza de la forma siguiente: 1, luego 2 y a continuación 3 se insertan en la pila. Para procesar \*, extraemos los dos elementos de la parte superior de la pila: 3 y luego 2. Observe que el primer elemento extraído se convierte en el parámetro del lado derecho, rhs, del operador binario y que el segundo elemento extraído es el parámetro del lado izquierdo, lhs; por tanto, los parámetros se extraen en orden inverso. Para la multiplicación, el orden no importa, pero para la resta y la división sí. El resultado de la multiplicación es 6, y ese resultado se introduce de nuevo en la pila. En este punto, el elemento superior de la pila será 6; por debajo de él estará 1. Para procesar el operador +, se extraen el 6 y el 1, y se inserta en la pila el resultado de su suma, 7. En este punto, la expresión completa habrá sido leída y la pila contendrá un único elemento. Por tanto, la respuesta final es 7.

Toda expresión infija válida puede convertirse a forma postfija. Por ejemplo, la larga expresión infija que antes hemos comentado se puede escribir en notación postfija de la forma siguiente:

1 2 - 4 5 ^ 3 \* 6 \* 7 2 2 ^ ^ / -

La Figura 11.11 muestra los pasos utilizados por la máquina postfija para evaluar esta expresión. Cada paso implica una única introducción en la pila. En consecuencia, como hay 9 operandos y 8 operadores, habrá 17 pasos y 17 inserciones en la pila. Claramente, el tiempo requerido para evaluar una expresión postfija es lineal.

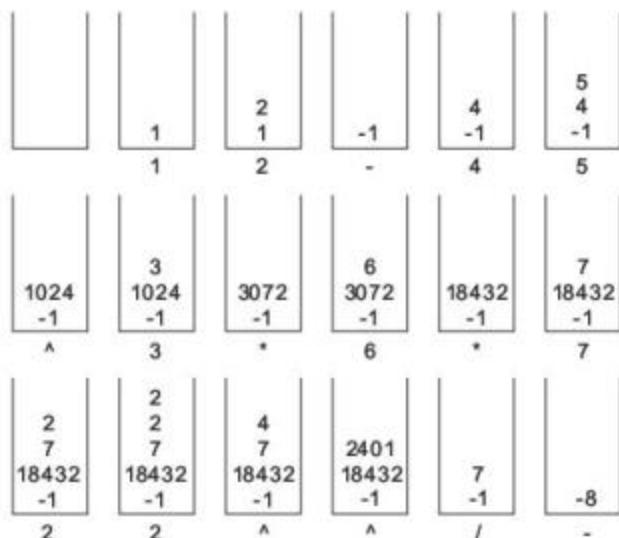
La evaluación de una expresión postfija requiere un tiempo lineal.

La tarea que nos resta es escribir un algoritmo para convertir de notación infija a notación postfija. Una vez que dispongamos de él, también tendremos un algoritmo que permita evaluar una expresión infija.

### 11.2.2 Conversión de notación infija a notación postfija

El principio básico implicado en el algoritmo de análisis de la precedencia de operadores, que convierte una expresión infija en una expresión postfija es el siguiente: cuando nos encontramos un operando, podemos proporcionarlo inmediatamente como salida. Sin embargo, cuando encontramos un operador, nunca podemos proporcionarlo como salida, porque tenemos que esperar a encontrar el segundo operando, por tanto, deberemos guardarlo. En una expresión tal como

Expresión postfija: 12 - 4 5 ^ 3 \* 6 \* 7 2 2 ^ ^ / -



**Figura 11.11** Pasos de la evaluación de una expresión posicional.

El algoritmo de análisis de precedencia de operadores convierte una expresión infix en una expresión postfix, para poder así evaluar la expresión infix.

Se emplea una pila de operadores para almacenar los operadores que nos hemos encontrado pero que no han sido todavía proporcionados como salida.

Cuando encontramos un operador a la entrada, se extraen de la pila los operadores de mayor prioridad (o los operadores de igual prioridad con asociatividad izquierda), lo que significa que hay que aplicar esos operadores. A continuación, se inserta en la pila el operador de entrada.

$1 + 2 * 3 ^ 4$

que en forma postfija es

1 2 3 4 ⋯ \*

Una expresión postfija tiene en algunos casos los operadores en el orden inverso de aquel en el que aparecen en la correspondiente expresión infija. Por supuesto, este orden solo se puede presentar si la precedencia de los operadores implicados se incrementa a medida que vamos de izquierda a derecha. Aun así, esta condición sugiere que una pila es apropiada para almacenar los operadores. Entonces, aplicando esta lógica, cuando leamos un operador, deberemos incluirlo en una pila de alguna forma. En consecuencia, en algún punto, será necesario extraer el operador de la pila. El resto del algoritmo implica decidir en qué momento entran y salen de la pila los operadores.

En otra expresión infinita simple como

$$2 \wedge 5 = 1$$

cuando alcanzamos el operador  $-$ ,  $2$  y  $5$  ya habrán sido proporcionados como salida y  $^$  se encontrará en la pila. Puesto que el operador  $-$  tiene menor precedencia que  $^$ , tendremos que aplicar  $^$  a  $2$  y a  $5$ . Por tanto, tendremos que extraer de la pila  $^$  y cualquier otro operando de mayor precedencia que  $-$ . Despu s de hacer esto, insertamos el operador  $-$ . La expresi n postfija resultante es

25^1-

En general, cuando estamos procesando un operador de la entrada, proporcionaremos como salida aquellos operadores de la pila que debamos procesar en aplicación de las reglas de precedencia (y de asociatividad).

Un segundo ejemplo sería la expresión infija

$3 * 2 ^ 5 - 1$

Cuando alcanzamos el operador  $^$ , 3 y 2 ya habrán sido proporcionados como salida y  $*$  se encontrará en la pila. Puesto que  $^$  tiene una mayor precedencia que  $*$ , no se extrae nada y  $^$  pasa a la pila. El 5 se proporciona como salida inmediatamente. Después nos encontramos con un  $-$ . Las reglas de precedencia nos dicen que hay que extraer  $^$ , seguido del operador  $*$ . En este punto, ya no queda nada que extraer de la pila, por lo que habremos terminado con la labor de extracción, y el  $-$  pasará a la pila. Después proporcionaremos como salida 1. Cuando alcanzamos el final de la expresión infija, podemos extraer de la pila los operadores restantes. La expresión postfija resultante es

$3 2 5 ^ * 1 -$

Antes de presentar el algoritmo que resume todo esto, es necesario responder a algunas cuestiones. En primer lugar, si el símbolo actual es un  $+$  y en la parte superior de la pila hay un  $+$ , ¿debemos extraer de la pila el  $+$  que allí se encuentra, o ese símbolo debe permanecer en la pila? La respuesta está determinada por la decisión de si el símbolo  $+$  de entrada implica que se ha completado el  $+$  de la pila. Puesto que el operador  $+$  es asociativo de izquierda a derecha, la respuesta es sí. Sin embargo, si estamos hablando del operador  $^$ , que es asociativo de derecha a izquierda, la respuesta es no. Por tanto, al examinar dos operadores de igual precedencia, tenemos que recurrir a las reglas de asociatividad para decidir, tal y como se muestra en la Figura 11.12.

¿Qué sucede con los paréntesis? Un paréntesis izquierdo puede considerarse un operador de alta precedencia cuando es un símbolo de entrada, mientras que si se encuentra en la pila hay que tratarlo como un operador de baja precedencia. En consecuencia, el paréntesis izquierdo de entrada se inserta simplemente en la pila. Cuando aparece un paréntesis derecho en la entrada, extraemos la pila de operadores hasta encontrar un paréntesis izquierdo. Los operadores se escriben como salida, pero los paréntesis no.

A continuación se presenta un resumen de los distintos casos con los que nos podemos encontrar en el algoritmo de análisis de precedencia de los operadores. Con la excepción de los paréntesis, todo lo que extrae de la pila se proporciona como salida.

Un paréntesis izquierdo se trata como un operador de alta precedencia cuando es un símbolo de entrada, pero como un operador de baja precedencia cuando se encuentra en la pila. Un paréntesis izquierdo solo puede ser eliminado por un paréntesis derecho.

Expresión infija	Expresión postfija	Asociatividad
$2 + 3 + 4$	$2 3 + 4 +$	Asociatividad izquierda; la entrada $+$ es de menor prioridad que el $+$ de la pila.
$2 ^ 3 ^ 4$	$2 3 4 ^ ^$	Asociatividad derecha; la entrada $^$ es de mayor prioridad que el $^$ de la pila.

Figura 11.12 Ejemplos de utilización de la asociatividad para romper los empates relativos a la precedencia.

- *Operandos*. Se proporcionan como salida inmediatamente.
- *Paréntesis de cierre*. Se extraen símbolos de la pila hasta que aparece un paréntesis de apertura.
- *Operadores*. Se extraen todos los símbolos de la pila hasta que aparece un símbolo de menor precedencia o un símbolo de igual precedencia y con asociatividad derecha. Después se inserta el operador.
- *Final de la entrada*. Se extraen todos los símbolos restantes de la pila.

Como ejemplo, la Figura 11.13 muestra cómo procesa el algoritmo la expresión

$1 - 2 ^ 3 ^ 3 - ( 4 + 5 * 6 ) * 7$

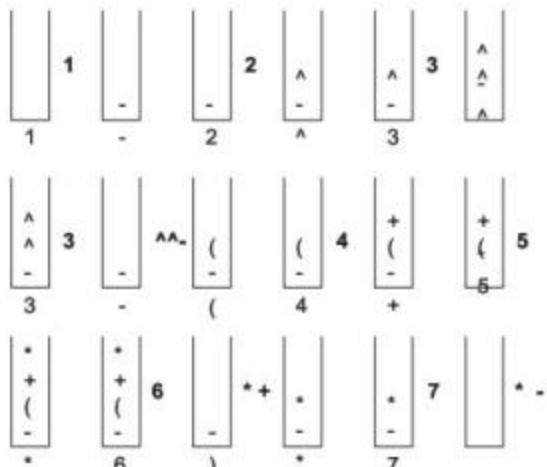
Debajo de cada pila se indica el símbolo leído. A la derecha de cada pila, en negrita, se muestra la salida generada.

### 11.2.3 Implementación

La clase `Evaluator`  
analiza y evalúa  
expresiones infixas.

Con esto disponemos ya de los conocimientos teóricos requeridos para implementar una calculadora simple. Nuestra calculadora soporta la suma, la resta, la multiplicación, la división y la exponentiación. Escribimos una clase `Evaluator` que funciona con enteros de tipo `long`. Hacemos una suposición para simplificar las cosas: no se permiten los números negativos. Distinguir entre el operador menos binario y el menos unario requiere un trabajo adicional en la rutina de exploración de la entrada y también complica las cosas, por el hecho de introducir un operador no binario. La incorporación de los operadores unarios no es difícil, pero el código extra no ilustra ningún concepto interesante por sí mismo, así que le dejamos dicha tarea al lector para que la acometa como ejercicio.

Infixa:  $1 - 2 ^ 3 ^ 3 - ( 4 + 5 * 6 ) * 7$



Postfixa:  $1\ 2\ 3\ 3\ ^\ ^\ -\ 4\ 5\ 6\ *\ +\ 7\ *\ -$

Figura 11.13 Conversión de notación infixa a postfix.

La Figura 11.14 muestra el esqueleto de la clase `Evaluator`, que se utiliza para procesar una única cadena de caracteres de entrada. El algoritmo básico de evaluación requiere utilizar dos pilas. La primera pila se utiliza para evaluar la expresión infija y generar la expresión postfija. Se trata de la línea de operadores declarada en la línea 34. Un valor `int` representa diferentes tipos de símbolos sintácticos como `PLUS`, `MINUS`, etc. Estas constantes se muestran posteriormente. En lugar de proporcionar explícitamente como salida la expresión postfija, enviamos cada símbolo postfijo a la máquina postfija a medida que se van generando. Por tanto, necesitamos también una pila que almacene los operandos; la pila de la máquina postfija se declara en la línea 35. El miembro de datos restante es un objeto  `StringTokenizer` utilizado para analizar la línea de entrada.

Como sucedía con el comprobador de equilibrado de los símbolos, podemos escribir una clase `Tokenizer` que puede emplearse para obtener la secuencia de símbolos sintácticos. Aunque podríamos reutilizar el código, existen de hecho pocos aspectos comunes, así que hemos preferido escribir una clase `Tokenizer` exclusiva de esta aplicación. Aquí, sin embargo, los símbolos sintácticos son algo más complejos porque, si leemos un operando, el tipo de símbolo sintáctico es `VALUE`, pero también debemos saber cuál es el valor que ha sido leído. Para evitar confusiones, denominamos a la clase `EvalTokenizer` y hacemos que sea una clase anidada. Su colocación se muestra en la línea 22; su implementación, junto con la clase `Token` anidada, se muestra en la Figura 11.15. Un `Token` almacena tanto un tipo de símbolo sintáctico, como su valor numérico (en caso de que el símbolo sintáctico sea un `VALUE`). Se pueden utilizar métodos accesores para obtener información acerca de un símbolo sintáctico. (El método `getValue` podría hacerse más robusto haciendo que informara de un error si `type` no es `VALUE`.) La clase `EvalTokenizer` tiene un método.

La Figura 11.16 muestra la rutina `getToken`. La línea 10 comprueba si estamos al final de la línea entrada. Cuando `getToken` pasa más allá de la línea 11, sabemos que hay más símbolos sintácticos disponibles. Si no hemos alcanzado el final de la línea, comprobamos si hay una correspondencia con cualquiera de los operadores de un solo carácter. Si es así, devolvemos el símbolo sintáctico apropiado. En caso contrario, esperamos que lo que resta sea un operando, así que utilizamos `Long.parseLong` para obtener el valor y luego devolvemos un objeto `Token`, construyendo explícitamente un objeto `Token` basado en el valor leído.

Ahora podemos examinar los métodos de la clase `Evaluator`. El único método públicamente visible es `getValue`. Mostrado en la Figura 11.17, `getValue` lee repetidamente un símbolo sintáctico y lo procesa, hasta detectar el final de línea. En ese punto, el elemento situado en la parte superior de la pila será la respuesta.

Las Figuras 11.18 y 11.19 muestran las rutinas utilizadas para implementar la máquina postfija. La rutina de la Figura 11.18 se emplea para extraer un elemento de la pila postfija e imprimir un mensaje de error en caso necesario. La rutina `binaryOp` de la Figura 11.19 aplica `topOp` (que se espera que sea el elemento situado en la parte superior de la pila de operadores) a los dos elementos situados en la parte superior de la pila postfija, y sustituye estos por el resultado. También extrae elementos de la pila de operadores (en la línea 33), señalando que el procesamiento de `topOp` se ha completado.

La Figura 11.20 declara una *tabla de precedencia*, que almacena las precedencias entre operadores y se utiliza para decidir qué hay que extraer de la pila de operadores. Los operadores se enumeran en el mismo orden que en las constantes de los símbolos sintácticos.

Necesitamos dos pilas:  
una pila de operadores  
y una pila para la máquina  
postfija.

Se utiliza una tabla de  
precedencia para decidir  
qué hay que extraer de  
la pila de operadores.  
Los operadores con  
asociatividad izquierda  
tienen una precedencia  
en la pila de operadores  
configurada con 1 unidad  
más que la precedencia  
del símbolo de entrada.  
En los operadores con  
asociatividad derecha es  
al revés.

```
1 import java.util.Stack;
2 import java.util.StringTokenizer;
3 import java.io.IOException;
4 import java.io.BufferedReader;
5 import java.io.InputStreamReader;
6
7 // Interfaz de la clase Evaluator: evalúa expresiones infijas.
8 //
9 // CONSTRUCCIÓN: con un String
10 //
11 // *****OPERACIONES PÚBLICAS*****
12 // long getValue( ) --> Devolver valor de expresión infija
13 // *****ERRORES*****
14 // Se realizan algunas comprobaciones de errores
15
16 public class Evaluator
17 {
18     private static class Precendence
19         { /* Figura 11.20 */ }
20     private static class Token
21         { /* Figura 11.15 */ }
22     private static class EvalTokenizer
23         { /* Figura 11.15 */ }
24
25     public Evaluator( String s )
26     {
27         opStack = new Stack<Integer>(); opStack.push( EOL );
28         postfixStack = new Stack<Long>();
29         str = new StringTokenizer( s, "+*-^() ", true );
30     }
31     public long getValue( )
32     { /* Figura 11.17 */ }
33
34     private Stack<Integer> opStack; // Pila de operadores para la conversión
35     private Stack<Long> postfixStack; // Pila para la máquina postfija
36     private StringTokenizer str; // Flujo StringTokenizer
37
38     private void processToken( Token lastToken )
39     { /* Figura 11.21 */ }
40     private long getTop( )
41     { /* Figura 11.18 */ }
42     private void binaryOp( int topOp )
43     { /* Figura 11.19 */ }
44 }
```

Figura 11.14 El esqueleto de la clase Evaluator.

```

1  private static class Token
2  {
3      public Token( )
4          { this( EOL ); }
5      public Token( int t )
6          { this( t, 0 ); }
7      public Token( int t, long v )
8          { type = t; value = v; }
9
10     public int getType( )
11         { return type; }
12     public long getValue( )
13         { return value; }
14
15     private int type = EOL;
16     private long value = 0;
17 }
18
19 private static class EvalTokenizer
20 {
21     public EvalTokenizer( StringTokenizer is )
22         { str = is; }
23
24     /**
25      * Encontrar el siguiente símbolo sintáctico, saltando los blancos
26      * y devolverlo. Para el símbolo sintáctico VALUE, colocar el valor
27      * procesado en currentValue.
28      * Imprimir mensaje de error si no se reconoce la entrada.
29      */
30     public Token getToken( )
31         { /* Figura 11.16 */ }
32
33     private StringTokenizer str;
34 }

```

Figura 11.15 Las clases anidadas Token y EvalTokenizer.

Deseamos asignar un número a cada nivel de precedencia. Cuanto mayor sea el número, mayor será la precedencia. Podríamos asignar a los operadores aditivos el nivel de precedencia 1, a los operadores multiplicativos el nivel de precedencia 3, a la exponentiación una precedencia 5 y a los paréntesis el nivel de precedencia 99. Sin embargo, también tenemos que tener en cuenta la asociatividad. Para ello, asignamos a cada operador un número que representa su precedencia cuando los encontramos como símbolo de entrada y un segundo número que representa su precedencia

```

1  /**
2   * Encontrar el siguiente símbolo sintáct., saltando blancos y devolverlo.
3   * Para símb. sintáctico VALUE, colocar el valor procesado en currentValue.
4   * Imprimir mensaje de error si no se reconoce la entrada.
5   */
6  public Token getToken( )
7  {
8      long theValue;
9
10     if( !str.hasMoreTokens( ) )
11         return new Token( );
12
13     String s = str.nextToken( );
14     if( s.equals( " " ) ) return getToken( );
15     if( s.equals( "^" ) ) return new Token( EXP );
16     if( s.equals( "/" ) ) return new Token( DIV );
17     if( s.equals( "*" ) ) return new Token( MULT );
18     if( s.equals( "(" ) ) return new Token( OPAREN );
19     if( s.equals( ")" ) ) return new Token( CPAREN );
20     if( s.equals( "+" ) ) return new Token( PLUS );
21     if( s.equals( "-" ) ) return new Token( MINUS );
22
23     try
24     {
25         theValue = Long.parseLong( s );
26     }
27     catch( NumberFormatException e )
28     {
29         System.err.println( "Parse error" );
30         return new Token( );
31     }
32 }
33
34     return new Token( VALUE, theValue );
35 }
```

**Figura 11.16** La rutina getToken para devolver el siguiente símbolo sintáctico del flujo de datos de entrada.

cuando se encuentra en la pila de operadores. Un operador con asociatividad izquierda tendrá su precedencia en la pila de operadores configurada con 1 unidad más alta que la precedencia como símbolo de entrada, mientras que en un operador con asociatividad derecha es al revés. Por tanto, la precedencia del operador + en la pila es 2.

Una consecuencia de esta regla es que cualesquiera dos operadores que tengan distintas precedencias seguirán estando ordenados correctamente. Sin embargo, si hay un + en la pila de operadores y el símbolo de entrada es también el mismo, el operador en la parte superior de la pila parecerá tener una mayor precedencia y será por tanto extraído de la pila. Esto es lo que queremos que ocurra para los operadores con asociatividad izquierda.

```

1  /**
2  * Rutina pública que realiza la evaluación.
3  * Examinar la máquina postfija para ver si queda un único resultado y,
4  * de ser así, devolverlo; en caso contrario, imprimir error.
5  * @return el resultado.
6  */
7  public long getValue( )
8  {
9      EvalTokenizer tok = new EvalTokenizer( str );
10     Token lastToken;
11
12     do
13     {
14         lastToken = tok.getToken( );
15         processToken( lastToken );
16         } while( lastToken.getType( ) != EOL );
17
18         if( postfixStack.isEmpty( ) )
19         {
20             System.err.println( "Missing operand!" );
21             return 0;
22         }
23
24         long theResult = postfixStack.pop( );
25         if( !postfixStack.isEmpty( ) )
26             System.err.println( "Warning: missing operators!" );
27
28         return theResult;
29     }

```

Figura 11.17 La rutina `getValue` para leer y procesar símbolos sintácticos y luego devolver el elemento situado en la parte superior de la pila.

```

1  /*
2  * Extraer elemento de la pila de la máq. postfija; devolver el resultado.
3  * Si la pila está vacía, imprimir un mensaje de error.
4  */
5  private long postfixPop( )
6  {
7      if( postfixStack.isEmpty( ) )
8      {
9          System.err.println( "Missing operand" );
10         return 0;
11     }
12     return postfixStack.pop( );
13 }

```

Figura 11.18 Las rutinas para extraer el elemento situado en la parte superior de la pila postfija.

```

1  /**
2  * Procesar un operador extrayendo dos elementos de la pila postfija,
3  * aplicando el operador e introduciendo en la pila el resultado.
4  * Imprimir error si falta paréntesis de cierre o hay división por 0.
5  */
6  private void binaryOp( int topOp )
7  {
8      if( topOp == OPAREN )
9      {
10          System.err.println( "Unbalanced parentheses" );
11          opStack.pop( );
12          return;
13      }
14      long rhs = postfixPop( );
15      long lhs = postfixPop( );
16
17      if( topOp == EXP )
18          postfixStack.push( pow( lhs, rhs ) );
19      else if( topOp == PLUS )
20          postfixStack.push( lhs + rhs );
21      else if( topOp == MINUS )
22          postfixStack.push( lhs - rhs );
23      else if( topOp == MULT )
24          postfixStack.push( lhs * rhs );
25      else if( topOp == DIV )
26          if( rhs != 0 )
27              postfixStack.push( lhs / rhs );
28          else
29          {
30              System.err.println( "Division by zero" );
31              postfixStack.push( lhs );
32          }
33      opStack.pop( );
34  }

```

**Figura 11.19** La rutina `BinaryOp` para aplicar `topOp` a la pila postfija.

De forma similar, si hay `^` en la pila de operadores y ese mismo símbolo es el símbolo de entrada, el operador de la parte superior de pila parecerá tener una menor precedencia, por lo que no será extraído de la misma. Esto es lo que queremos que suceda para los operadores con asociatividad derecha. El símbolo sintáctico `VALUE` nunca llega a ser insertado en la pila, por lo que su precedencia es irrelevante. El símbolo sintáctico de fin de línea es el que recibe la precedencia más baja, porque se inserta en la pila para actuar como centinela (de lo cual se encarga el constructor). Si lo tratamos como un operador con asociatividad derecha, queda cubierto por el caso relativo a los operadores.

```

1  private static final int EOL = 0;
2  private static final int VALUE = 1;
3  private static final int OPAREN = 2;
4  private static final int CPAREN = 3;
5  private static final int EXP = 4;
6  private static final int MULT = 5;
7  private static final int DIV = 6;
8  private static final int PLUS = 7;
9  private static final int MINUS = 8;
10
11 private static class Precedence
12 {
13     public int inputSymbol;
14     public int topOfStack;
15
16     public Precedence( int inSymbol, int topSymbol )
17     {
18         inputSymbol = inSymbol;
19         topOfStack = topSymbol;
20     }
21 }
22
23 // precTable se ajusta al orden de enumeración de símbolos sintáct.
24 private static Precedence [ ] precTable =
25 {
26     new Precedence( 0, -1 ),           // EOL
27     new Precedence( 0, 0 ),            // VALUE
28     new Precedence( 100, 0 ),          // OPAREN
29     new Precedence( 0, 99 ),           // CPAREN
30     new Precedence( 6, 5 ),            // EXP
31     new Precedence( 3, 4 ),            // MULT
32     new Precedence( 3, 4 ),            // DIV
33     new Precedence( 1, 2 ),             // PLUS
34     new Precedence( 1, 2 )             // MINUS
35 }

```

Figura 11.20 Tabla de precedencias utilizada para evaluar un expresión infija.

El método que queda es `processToken`, que se muestra en la Figura 11.21. Cuando nos encontramos con un operando, lo insertamos en la pila postfija. Cuando nos encontramos con un paréntesis de cierre, extraemos y procesamos repetidamente el operador situado en la parte superior de la pila de operadores, hasta que aparezca el paréntesis de apertura (líneas 18-19). Después, el paréntesis de apertura se extrae en la línea 21. (La comprobación de la línea 20 se utiliza para

evitar extraer el centinela, en el caso de que falte un paréntesis de apertura.) En caso contrario, nos encontraremos en el caso genérico de los operadores, que se describe suavemente mediante el código de las líneas 27 a 31.

En la Figura 11.22 se proporciona una rutina `main` simple. La rutina lee repetidamente una línea de entrada, instancia un objeto `Evaluator` y calcula su valor.

```

1  /**
2   * Después de leer un símbolo sintáctico, usar el algoritmo de análisis
3   * de precedencia de operadores para procesarlo; aquí se detectan los
4   * paréntesis de apertura que faltan.
5   */
6  private void processToken( Token lastToken )
7  {
8      int topOp;
9      int lastType = lastToken.getType( );
10
11     switch( lastType )
12     {
13         case VALUE:
14             postfixStack.push( lastToken.getValue( ) );
15             return;
16
17         case CPAREN:
18             while( ( topOp = opStack.peek( ) ) != OPAREN && topOp != EOL )
19                 binaryOp( topOp );
20             if( topOp == OPAREN )
21                 opStack.pop( ); // Desembarazarse de paréntesis de apertura
22             else
23                 System.err.println( "Missing open parenthesis" );
24             break;
25
26         default: // Caso general de operadores
27             while( precTable[ lastType ].inputSymbol <=
28                   precTable[ topOp = opStack.peek( ) ].topOfStack )
29                 binaryOp( topOp );
30             if( lastType != EOL )
31                 opStack.push( lastType );
32             break;
33     }
34 }
```

Figura 11.21 La rutina `processToken` para procesar `lastToken`, utilizando el algoritmo de análisis de precedencia de operadores.

```

1  /**
2   * Rutina main simple para trabajar con la clase Evaluator.
3   */
4  public static void main( String [ ] args )
5  {
6      String str;
7      Scanner in = new Scanner( System.in );
8
9      System.out.println( "Enter expressions, one per line:" );
10     while( in.hasNextLine( ) )
11     {
12         str = in.nextLine( );
13         System.out.println( "Read: " + str );
14         Evaluator ev = new Evaluator( str );
15         System.out.println( ev.getValue( ) );
16         System.out.println( "Enter next expression:" );
17     }
18 }
```

Figura 11.22 Una rutina main simple para evaluar las expresiones repetidamente.

### 11.2.4 Árboles de expresión

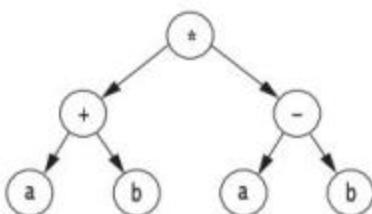
La Figura 11.23 muestra un ejemplo de un *árbol de expresión*, cuyas hojas son operandos (por ejemplo, constantes o nombres de variables), mientras que los otros nodos contienen operadores. Este árbol concreto es binario, porque todas las operaciones son binarias. Aunque se trata del caso más simple, lo cierto es que los nodos pueden tener más de dos hijos. Un nodo también puede tener un solo hijo, como sucede con el operador unario menos.

En un *árbol de expresión*, las hojas contienen operandos, mientras que los restantes nodos contienen operadores.

Evaluamos un árbol de expresión  $T$  aplicando el operador situado en la raíz a los valores obtenidos evaluando recursivamente los subárboles izquierdo y derecho. En este ejemplo, el subárbol izquierdo se evalúa como  $(a+b)$ , mientras que el subárbol derecho se evalúa como  $(a-b)$ . Por tanto, todo el árbol representará  $((a+b)*(a-b))$ . Podemos producir una expresión infija (con un montón de paréntesis innecesarios) generando recursivamente una expresión izquierda entre paréntesis, imprimiendo el operador situado en la raíz y generando recursivamente una expresión derecha entre paréntesis. Esta estrategia general (izquierda, nodo, derecha) se denomina *recorrido en orden*. Este tipo de recorrido es fácil de recordar, debido al tipo de expresión que produce.

Una segunda estrategia consiste en imprimir recursivamente el subárbol izquierdo, luego el subárbol derecho y después el operador (sin paréntesis). Al hacer esto obtenemos la expresión postfija, por lo que esta estrategia se denomina *recorrido en postorden del árbol*. Una tercera estrategia para evaluar un árbol da como resultado una expresión prefija. Hablaremos de todas estas estrategias en el Capítulo 18. El árbol de expresión (y sus generalizaciones)

Se puede utilizar la impresión recursiva del árbol de expresión para obtener una expresión infija, postfija o prefija.

Figura 11.23 Árbol de expresión para  $(a+b)*(a-b)$ .

constituye una útil estructura de datos en el diseño de compiladores, porque nos permite ver una expresión completa. Esta capacidad facilita la generación de código y, en algunos casos, incrementa enormemente los esfuerzos de optimización.

Los árboles de expresión se pueden construir a partir de una expresión postfixa de forma similar a como se realizaba la evaluación de una expresión postfixa.

Resulta de especial interés la construcción de un árbol de expresión a partir de una expresión infija. Como ya hemos visto, siempre podemos convertir una expresión infija a una expresión postfixa, por lo que lo único que necesitamos es mostrar cómo construir un árbol de expresión a partir de una expresión postfixa. No es sorprendente que este procedimiento sea muy simple. Lo que hacemos es mantener una pila de árboles. Cuando nos encontramos con un operando, creamos un árbol de un único nodo y lo insertamos en la pila. Cuando nos encontramos con un operador, extraemos y combinamos los dos árboles situados en la parte superior de la pila. En el nuevo árbol, el nodo será el operador, el hijo derecho será el primer árbol extraído de la pila, y el hijo izquierdo será el segundo árbol extraído. Después volvemos a insertar el resultado en la pila. Este algoritmo es prácticamente el mismo que se utilizaba para una evaluación postfixa, salvo porque se sustituye el cálculo del resultado de un operador binario por una operación de creación de un árbol.

## Resumen

En este capítulo hemos examinado dos aplicaciones de las pilas en los lenguajes de programación y en el diseño de compiladores. Hemos visto que, aunque la pila es una estructura simple, tiene una gran potencia. Las pilas se pueden utilizar para decidir si una secuencia de símbolos está bien equilibrada. El algoritmo resultante requiere un tiempo de ejecución y, lo que es igualmente importante, está compuesto por una única exploración secuencial de la entrada. El análisis de precedencia de operadores es una técnica que se puede emplear para analizar sintácticamente expresiones infijas. También requiere un tiempo de ejecución lineal y una única exploración secuencial. En el algoritmo de análisis de precedencia de operadores se utilizan dos pilas. Aunque las pilas almacenan diferentes tipos de objetos, el código genérico de tratamiento de pilas nos permite el uso de una única implementación de pila para ambos tipos de objeto.



## Conceptos clave

**análisis léxico** El proceso de reconocer símbolos sintácticos en un flujo de símbolos. (435)

**análisis sintáctico de precedencia de operadores** Un algoritmo que convierte una expresión infija en una expresión postfija para evaluar la expresión infija. (446)

**árbol de expresión** Un árbol en el que las hojas contienen operandos y los demás nodos contienen operadores. (457)

**expresión infija** Una expresión en la que un operador binario tiene argumentos a su izquierda y a su derecha. Cuando hay varios operadores, las reglas de precedencia y la asociatividad determinan cómo deben procesarse los operadores. (444)

**expresión postfija** Una expresión que puede ser evaluada por una máquina postfija, sin utilizar ninguna regla de precedencia. (444)

**máquina de estados** Una técnica común utilizada para el análisis sintáctico de símbolos; en cualquier punto, la máquina se encuentra en un cierto estado y cada carácter de entrada la lleva a un estado nuevo. Al final, la máquina de estados terminará por alcanzar un estado en el que se habrá reconocido un símbolo. (437)

**máquina postfija** Máquina utilizada para evaluar una expresión postfija. El algoritmo que se emplea es el siguiente: los operandos se introducen en una pila y cada operador extrae sus operandos y luego introduce el resultado en la pila. Al final de la evaluación, la pila debería contener exactamente un elemento, el cual representará el resultado. (445)

**simbolización** El proceso de generar la secuencia de símbolos sintácticos (*tokens*) a partir de un flujo de datos de entrada. (433)

**tabla de precedencia** Una tabla utilizada para decidir qué hay que extraer de la pila de operadores. Los operadores con asociatividad izquierda tienen una precedencia en la pila de operadores que es 1 unidad mayor que su precedencia como símbolo de entrada. En los operadores con asociatividad derecha sucede al revés. 0



## Errores comunes

1. En el código de producción, los errores de entrada tienen que ser tratados con el máximo cuidado. Si no se es suficientemente estricto a este respecto, se producirán errores de programación.
2. Para la rutina de equilibrado de símbolos, un error bastante común consiste en tratar las comillas de manera incorrecta.
3. En el algoritmo de conversión de notación infija a postfija, la tabla de precedencia debe reflejar la reglas de precedencia y asociatividad correctas.



## Internet

Están disponibles ambos programas de aplicación. Probablemente debería descargar el programa de equilibrado: le ayudará a depurar otros programas Java.

**Balance.java**

Contiene el programa de equilibrado de símbolos.

**Tokenizer.java**

Contiene la implementación de la clase `Tokenizer` para comprobar programas Java.

**Evaluator.java**

Contiene el evaluador de expresiones.



## Ejercicios

### EN RESUMEN

- 11.1** Para la expresión infija `a + b ^ c * d / ( e + f )`, haga lo siguiente:
- Muestre cómo el algoritmo de análisis de precedencia de operadores genera la correspondiente expresión postfija.
  - Muestre cómo una máquina postfija evalúa la expresión postfija resultante.
- 11.2** Muestre el resultado de ejecutar el programa de equilibrado de símbolos sobre
- `}`
  - `( )`
  - `[ [ [`
- 11.3** Proporcione la expresión postfija correspondiente a
- `1 + 2 - 3 ^ 4`
  - `1 + 2 + 4 / 3`
  - `1 + 2 + * 3 - 4 ^ 5 + 6`

### EN TEORÍA

- 11.4** En términos generales, explique cómo se incorporarían operadores unarios a los evaluadores de expresiones. Suponga que los operadores unarios preceden a sus operandos y tienen una alta precedencia. Incluya una descripción de cómo serían reconocidos por la máquina de estados.
- 11.5** Para el programa de equilibrado de símbolos, explique cómo imprimir un mensaje de error que tienda a reflejar la probable causa del error.

### EN LA PRÁCTICA

- 11.6** El evaluador infijo acepta expresiones ilegales en las que los operadores están colocados incorrectamente.
- ¿Cómo se evaluaría `1 2 3 + 5 1 3 - +?`
  - ¿Cómo podemos detectar estas ilegalidades?
- 11.7** El uso del operador `^` para la exponenciación puede llegar a confundir a los programadores Java (porque es el operador bit a bit para OR exclusiva). Reescriba la clase `Evaluator` utilizando `**` como operador de exponenciación.

## PROYECTOS DE PROGRAMACIÓN

- 11.8** Para el comprobador de equilibrado de los símbolos, modifique la clase `Tokenizer` añadiendo un método público que pueda cambiar el flujo de datos de entrada. Despues añada un método público a `Balance` que permita que `Balance` cambie el flujo de datos de entrada.
- 11.9** Implemente un evaluador de expresiones Java que incluya variables. Suponga que existen como máximo 26 variables (en concreto, de A a Z) y que se puede realizar una asignación a una variable mediante un operador – de baja precedencia.
- 11.10** Modifique el evaluador de expresiones para tratar el caso de números de entrada negativos.
- 11.11** Escriba un programa que lea una expresión infija y genere una expresión postfija.
- 11.12** Implemente un evaluador de expresiones Java completo. Trate todos los operadores Java que puedan aceptar constantes y que tengan sentido aritmético (por ejemplo, no implemente `[]`)



## Referencias

El algoritmo de conversión de notación infija a postfija (*análisis de precedencia de operadores*) se describió por primera vez en [3]. Dos buenos libros sobre construcción de operadores son [1] y [2].

1. A. V. Aho, M. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 2007.
2. C. N. Fischer y R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin Cummings, Redwood City, CA, 1991.
3. R. W. Floyd, "Syntactic Analysis and Operator Precedence", *Journal of the ACM* 10:3 (1963), 316–333.



## Utilidades

En este capítulo, vamos a analizar dos aplicaciones de utilidad para estructuras de datos: la compresión de datos y las referencias cruzadas. La compresión de datos es una importante técnica dentro del campo de las Ciencias de la computación. Se puede utilizar para reducir el tamaño de los archivos almacenados en disco (lo que en la práctica tiene el efecto de aumentar la capacidad del disco) y también para incrementar la tasa efectiva de transmisión a través de modems (transmitiendo menos datos). Casi todos los modems más recientes realizan algún tipo de compresión. Las referencias cruzadas son una técnica de exploración y de ordenación que se utiliza por ejemplo, para preparar el índice de un libro.

En este capítulo, veremos

- Una implementación de un algoritmo de compresión de archivos denominado *algoritmo de Huffman*.
- Una implementación de un programa de referencias cruzadas que enumera, de manera ordenada, todos los identificadores del programa y proporciona los números de línea en los que aparecen.

### 12.1 Compresión de archivos

El conjunto de caracteres ASCII está compuesto por aproximadamente 100 caracteres imprimibles. Para poder diferenciar estos caracteres, se requieren  $\lceil \log 100 \rceil = 7$  bits. Siete bits permiten representar 128 caracteres, así que el conjunto de caracteres ASCII añade algunos otros caracteres “no imprimibles”. Además, se añade un octavo bit para permitir la realización de comprobaciones de paridad. Lo importante es, sin embargo, que si el tamaño del conjunto de caracteres es  $C$ , entonces hacen falta  $\lceil \log C \rceil$  bits, si se utiliza una codificación estándar de longitud fija.

La codificación estándar de  
C caracteres utiliza  
 $\lceil \log C \rceil$  bits.

Suponga que tenemos un archivo que contiene solo los caracteres *a*, *e*, *i*, *s* y *t*, espacios en blanco (*sp*) y caracteres de nueva línea (*nl*). Suponga además que el archivo contiene 10 letras *a*, 15 letras *e*, 12 letras *i*, 3 letras *s*, 4 letras *t*, 13 espacios en blanco y 1 carácter de nueva línea. Como muestra la Figura 12.1, representar este archivo requiere 174 bits, porque hay 58 caracteres y cada carácter precisa 3 bits.

El proceso de reducir el número de bits necesarios para representar los datos se denomina *compresión*, que en la práctica está compuesta de dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión).

En muchas situaciones, es deseable reducir el tamaño de un archivo. Por ejemplo, el espacio en disco resulta escaso en casi todas las máquinas, así que reducir la cantidad de espacio requerido por los archivos incrementa la capacidad efectiva del disco. Cuando se transmiten datos a través de líneas telefónicas mediante un módem, la tasa efectiva de transmisión se incrementa si es posible reducir la cantidad de datos transmitidos. El proceso de reducir el número de bits necesarios para representar los datos se denomina *compresión*, que en la práctica está compuesta de dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión). Una estrategia simple de la que hablaremos en este capítulo, permite ahorrar un 25 por ciento de espacio en algunos archivos de gran tamaño y hasta un 50 o un 60 por ciento en algunos archivos de datos grandes. Diversas extensiones proporcionan una compresión todavía mejor.

En un código de longitud variable, los caracteres más frecuentes tienen la representación más corta.

En la vida real, los archivos pueden ser muy grandes. Muchos archivos de gran tamaño se generan como salida de algún programa informático y suele existir una gran disparidad entre los caracteres más y menos frecuentemente utilizados. Por ejemplo, muchos archivos de datos grandes tienen un número desproporcionadamente grande de dígitos, caracteres en blanco y de nueva línea, pero aparece pocas veces la letra *q* o la letra *x*.

En muchas situaciones, es deseable reducir el tamaño de un archivo. Por ejemplo, el espacio en disco resulta escaso en casi todas las máquinas, así que reducir la cantidad de espacio requerido por los archivos incrementa la capacidad efectiva del disco. Cuando se transmiten datos a través de líneas

telefónicas mediante un módem, la tasa efectiva de transmisión se incrementa si es posible reducir la cantidad de datos transmitidos. El proceso de reducir el número de bits necesarios para representar los datos se denomina *compresión*, que en la práctica está compuesta de dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión). Una estrategia simple de la que hablaremos en este capítulo, permite ahorrar un 25 por ciento de espacio en algunos archivos de gran tamaño y hasta un 50 o un 60 por ciento en algunos archivos de datos grandes. Diversas extensiones proporcionan una compresión todavía mejor.

La estrategia general consiste en conseguir que la longitud de código varíe de un carácter a otro y asegurarse de que los caracteres que aparecen más frecuentemente tengan asociados códigos de pequeña longitud. Si todos los caracteres se presentan con una frecuencia idéntica o muy similar no es posible conseguir ningún ahorro.

### 12.1.1 Códigos prefijo

En un *trie binario*, una rama a la izquierda representa un 0 y una rama a la derecha representa un 1. El camino seguido hasta un nodo indica su representación.

El código binario presentado en la Figura 12.1 se puede representar mediante el árbol binario mostrado en la Figura 12.2. En esta estructura de datos, denominada *trie binario*, los caracteres se almacenan solo en los nodos hoja; la representación de cada carácter se puede determinar comenzando por la raíz y anotando el camino recorrido, utilizando un 0 para indicar la rama izquierda y un 1 para indicar la rama derecha. Por ejemplo, a *sse* llega yendo

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
<b>Total</b>			<b>174</b>

Figura 12.1 Esquema estándar de codificación.

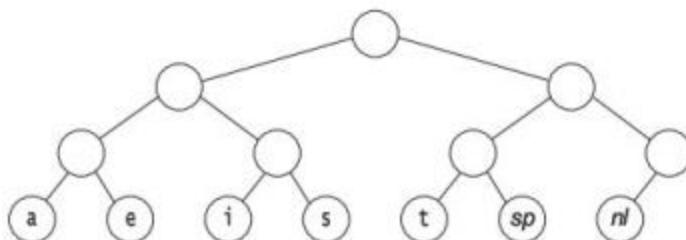


Figura 12.2 Representación del código original mediante un árbol.

hacia la izquierda, luego hacia la derecha y finalmente a la derecha. Esto se codificaría como 011. Si el carácter  $c$  está a una profundidad  $d$ , y aparece  $f$  veces, el coste del código es  $\sum d_f$ .

Podemos obtener un código mejor que el de la Figura 12.2 si nos damos cuenta de que  $nl$  es un hijo único. Situándolo más arriba (es decir, sustituyendo a su padre), obtenemos el nuevo árbol mostrado en la Figura 12.3. Este nuevo árbol tiene un coste de 173, pero sigue estando muy lejos de ser óptimo.

Observe que el árbol de la Figura 12.3 es un *árbol completo*, en el que todos los nodos son hojas o tienen dos hijos. Un código óptimo siempre tiene esta propiedad; en caso contrario, como ya hemos visto, los nodos con solo un hijo podrían desplazarse un nivel hacia arriba. Si los caracteres se colocan únicamente en las hojas, cualquier secuencia de bits se puede decodificar siempre de forma no ambigua.

Por ejemplo, suponga que la cadena codificada es 0100111100010110001000111. La Figura 12.3 muestra que 0 y 01 no son códigos de caracteres, pero que 010 representa la *i*, por lo que el primer carácter será *i*. Despues sigue 011, que es una *s*. Despues sigue 11, que es un carácter de nueva línea (*nl*). El resto del código es *a*, *sp*, *t*, *i*, *e* y *nl*.

Los códigos de los distintos caracteres pueden tener diferentes longitudes, siempre y cuando ningún código de carácter sea prefijo de otro código de carácter. Este tipo de codificación se denomina *código prefijo*. A la inversa, si un carácter está contenido en un nodo que no sea una hoja, deja de ser posible que la decodificación sea no ambigua.

Por tanto, nuestro problema básico consiste en encontrar el árbol binario de coste mínimo (tal como se ha definido previamente el coste), en el que todos los caracteres estén contenidos dentro las hojas. El árbol mostrado en la Figura 12.4 es óptimo para nuestro alfabeto de ejemplo. Como se muestra

En un *árbol completo*, todos los nodos son hojas o tienen dos hijos.

En un *código prefijo*, ningún código de carácter es prefijo de ningún otro código de carácter. Esta propiedad garantiza si los caracteres se encuentran únicamente en las hojas. Un código prefijo puede decodificarse de forma no ambigua.

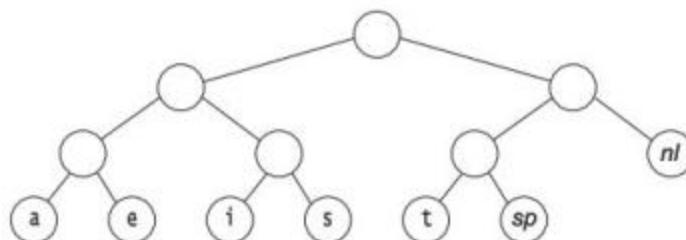
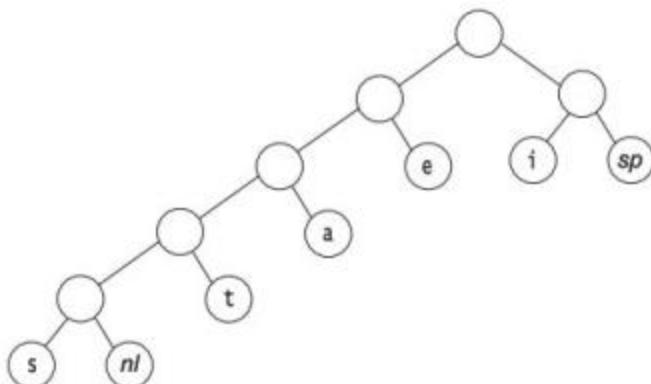


Figura 12.3 Un árbol ligeramente mejor.



**Figura 12.4** Un árbol de código prefijo óptimo.

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
<b>Total</b>			<b>146</b>

**Figura 12.5** Código prefijo óptimo.

en la Figura 12.5, este código solo requiere 146 bits. Existen muchos códigos óptimos, que se pueden obtener intercambiando hijos en el árbol de codificación.

### 12.1.2 Algoritmo de Huffman

*El algoritmo de Huffman* construye un código prefijo óptimo. Funciona combinando repetidamente los dos árboles de menor peso.

¿Cómo se construye el árbol de codificación? El algoritmo del sistema de codificación fue desarrollado por Huffman en 1952. Comúnmente denominado *algoritmo de Huffman*, construye un código prefijo óptimo combinando árboles repetidamente hasta obtener el árbol final.

A lo largo de esta sección, denominaremos  $C$  al número de caracteres. En el algoritmo de Huffman lo que hacemos es mantener un bosque de árboles. El *peso* de un árbol será la suma de las frecuencias de sus hojas. Se seleccionan  $C - 1$  veces dos árboles,  $T_1$  y  $T_2$ , de peso mínimo, rompiendo los empates arbitrariamente, y se forma un nuevo árbol con los subárboles  $T_1$  y  $T_2$ . Al principio del algoritmo, hay  $C$  árboles de un único nodo (uno por cada carácter). Al final del algoritmo, hay un solo árbol, que nos da el árbol óptimo de Huffman. En el Ejercicio 12.7 pediremos al lector que demuestre que el algoritmo de Huffman da como resultado un árbol óptimo.

Un ejemplo ayudará a clarificar el funcionamiento del algoritmo. La Figura 12.6 muestra el bosque inicial; el peso de cada árbol se muestra al lado de la raíz. Los dos árboles de menor peso se combinan creando el bosque mostrado en la Figura 12.7. La nueva raíz es  $T1$ . Designamos a  $s$  como hijo izquierdo de forma completamente arbitraria; puede utilizarse cualquier procedimiento que deseemos para romper los empates. El peso total del nuevo árbol será simplemente la suma de los pesos de los árboles usados para componerlo, así que puede calcularse fácilmente.

Los empates se rompen de forma arbitraria.

Ahora habrá seis árboles, y de nuevo volvemos a seleccionar los dos árboles de menor peso,  $T1$  y  $t$ . Esos árboles se combinan para formar un nuevo árbol con raíz  $T2$  y peso 8, como se muestra en la Figura 12.8. El tercer paso, combina  $T2$  y  $a$ , creando  $T3$ , con peso  $10 + 8 = 18$ . La Figura 12.9 muestra el resultado de esta operación.



Figura 12.6 Etapa inicial del algoritmo de Huffman



Figura 12.7 Algoritmo de Huffman después de la primera combinación.

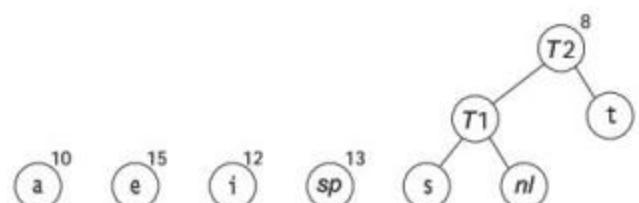


Figura 12.8 Algoritmo de Huffman después de la segunda combinación.

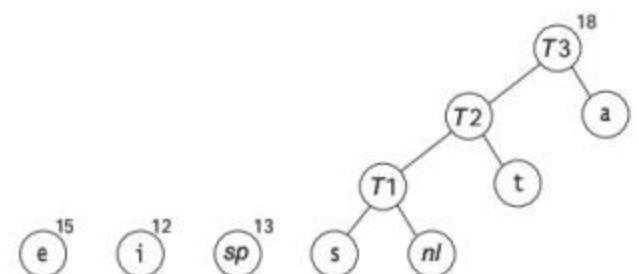


Figura 12.9 Algoritmo de Huffman después de la tercera combinación.

Después de completar la tercera combinación, los dos árboles de menor peso son los árboles de un único nodo que representan a *i* y *sp*. La Figura 12.10 muestra cómo se combinan estos árboles para formar un nuevo árbol con raíz *T*4. El quinto paso consiste en combinar los árboles con raíces *e* y *T*3, porque son los dos árboles con menor peso, dando el resultado mostrado en la Figura 12.11.

Finalmente, se obtiene un árbol óptimo, mostrado anteriormente en la Figura 12.4, combinando los dos árboles restantes. La Figura 12.12 muestra el árbol óptimo, con raíz *T*6.

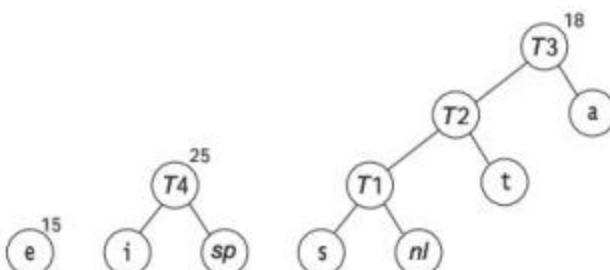


Figura 12.10 Algoritmo de Huffman después de la cuarta combinación.

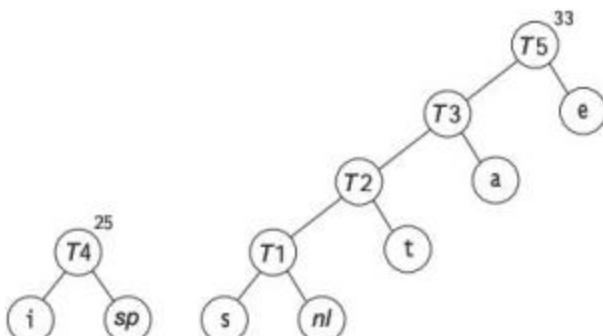


Figura 12.11 Algoritmo de Huffman después de la quinta combinación.

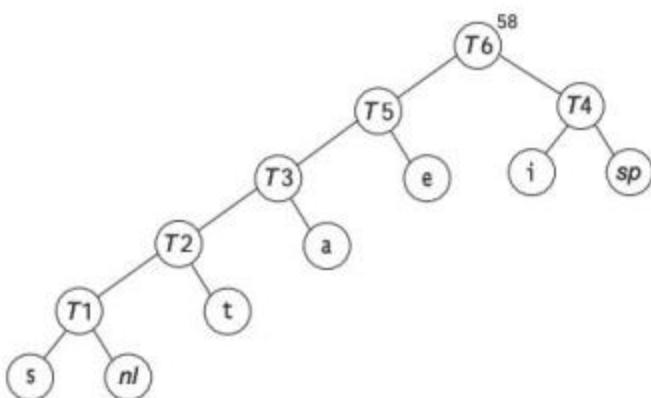


Figura 12.12 Algoritmo de Huffman después de la combinación final.

### 12.1.3 Implementación

Ahora vamos a proporcionar una implementación del algoritmo de codificación de Huffman, sin tratar de realizar ninguna optimización significativa; simplemente, deseamos obtener un programa funcional que ilustre los problemas algorítmicos básicos. Después de analizar la implementación, comentaremos las posibles mejoras. Aunque habría que añadir al programa algunos mecanismos significativos de comprobación de errores, aquí no lo hemos hecho, porque no queríamos oscurecer las ideas básicas.

La Figura 12.13 ilustra algunas de las clases de E/S y de las constantes que hay que utilizar. Mantenemos una cola con prioridad de nodos de árbol (recuerde que tenemos que seleccionar dos árboles con el menor peso).

Además de las clases de E/S estándar, nuestro programa está compuesto por varias clases adicionales. Puesto que necesitamos realizar E/S bit a bit, hemos escrito clases envoltorio que reflejan flujos de entrada y salida de bits. Hemos escrito otras clases para mantener el recuento de caracteres y para crear y devolver información acerca de un árbol de codificación de Huffman. Finalmente, hemos escrito envoltorios para compresión y descompresión de flujos de datos. Resumiendo, las clases que hemos escrito son:

<code>BitInputStream</code>	Envuelve un <code>InputStream</code> y permite la entrada bit a bit.
<code>BitOutputStream</code>	Envuelve un <code>OutputStream</code> y permite la salida bit a bit.
<code>CharCounter</code>	Mantiene el recuento de caracteres.
<code>HuffmanTree</code>	Manipula árboles de codificación de Huffman.
<code>HZIPInputStream</code>	Contiene un envoltorio de descompresión.
<code>HZIPOutputStream</code>	Contiene un envoltorio de compresión.

```
1 import java.io.IOException;
2 import java.io.InputStream;
3 import java.io.OutputStream;
4 import java.io.FileInputStream;
5 import java.io.FileOutputStream;
6 import java.io.DataInputStream;
7 import java.io.DataOutputStream;
8 import java.io.BufferedInputStream;
9 import java.io.BufferedOutputStream;
10 import java.util.PriorityQueue;
11
12 interface BitUtils
13 {
14     public static final int BITS_PER_BYTES = 8;
15     public static final int DIFF_BYTES = 256;
16     public static final int EOF = 256;
17 }
```

Figura 12.13 Las directivas `import` y algunas constantes utilizadas en los principales algoritmos del programa de compresión.

## Clases para entrada y salida de flujos de bits

Las clases `BitInputStream` y `BitOutputStream` son similares y se muestran en las Figuras 12.14 y 12.15, respectivamente. Ambas trabajan envolviendo un flujo de datos. Se almacena una referencia al flujo de datos en forma de miembro de datos privado. Cada octavo bit de lectura, `readBit`, del `BitInputStream` (o cada octavo bit de escritura, `writeBit`, de la clase `BitOutputStream`) hace que se lea (o escriba) un `byte` en el flujo de datos subyacente. El `byte` se almacena en un buffer, denominado apropiadamente `buffer`, y `bufferPos` proporciona una indicación de cuánta parte del buffer está libre.

Los métodos `getBit` y `setBit` se utilizan para acceder a un bit individual dentro de un `byte` de 8 bits; funcionan utilizando operaciones de bit (el Apéndice C describe los operadores de bit con más detalle). En `readBit`, comprobamos en la línea 19 si los bits del buffer han sido ya utilizados. En caso afirmativo, obtenemos 8 bits más en la línea 21 y reinicializamos el indicador de posición en la línea 24. Después, llamamos a `getBit` en la línea 27.

La clase `BitOutputStream` es similar a `BitInputStream`. Una diferencia es que proporcionamos un método `flush` de vaciado porque han podido quedar bits en el buffer al final de una secuencia de llamadas a `writeBit`. El método `flush` se invoca cuando una llamada a `writeBit` rellena el buffer y también es invocado por `close`.

Ninguna de las clases realiza comprobación de errores; en lugar de ello, propagan cualquier excepción `IOException` que se produzca. De ese modo, está disponible la posibilidad de realizar una comprobación completa de errores.

## La clase para el recuento de caracteres

La Figura 12.16 proporciona la clase `CharCounter`, que se utiliza para obtener los recuentos de caracteres en un flujo de datos de entrada (normalmente un archivo). Alternativamente, los recuentos de caracteres se pueden configurar manualmente y obtenerse con posterioridad. (Implicitamente, estamos tratando los bytes de ocho bits como caracteres ASCII para este programa).

## La clase para el árbol de Huffman

El árbol se mantiene mediante una colección de nodos. Cada nodo tiene enlaces a su hijo izquierdo, a su hijo derecho y a su padre (en el Capítulo 18 hablaremos detalladamente de la implementación de los árboles). La declaración de nodos se muestra en la Figura 12.17.

El esqueleto de la clase `HuffmanTree` se proporciona en la Figura 12.18. Podemos crear un objeto `HuffmanTree` proporcionando un objeto `CharCounter`, en cuyo caso el árbol se construye inmediatamente. Alternativamente, puede crearse sin un objeto `CharCounter`. En ese caso, los recuentos de caracteres se leen mediante una llamada posterior a `readEncodingTable`, después de lo cual se construye el árbol.

La clase `HuffmanTree` proporciona el método `writeEncodingTable` para escribir el árbol en un flujo de datos de salida (en una forma adecuada para una llamada a `readEncodingTable`). También proporciona métodos públicos para convertir de un carácter a un código, y viceversa.<sup>1</sup> Los códigos se representan mediante un `int[]` o un `String`, según sea apropiado, en el que cada elemento puede ser 0 o 1.

---

<sup>1</sup> Aviso técnico: se utiliza un `int` en lugar de un `byte` para que se puedan utilizar todos los caracteres y el símbolo EOF.

```
1 // Clase BitInputStream: clase envoltorio de flujos para entrada bit a bit.
2 //
3 // CONSTRUCCIÓN: con un InputStream abierto.
4 //
5 // *****OPERACIONES PÚBLICAS*****
6 // int readBit( ) --> Leer un bit como 0 o 1
7 // void close( ) --> Cerrar flujo de datos subyacente.
8
9 public class BitInputStream
10 {
11     public BitInputStream( InputStream is )
12     {
13         in = is;
14         bufferPos = BitUtils.BITS_PER_BYTES;
15     }
16
17     public int readBit( ) throws IOException
18     {
19         if( bufferPos == BitUtils.BITS_PER_BYTES )
20         {
21             buffer = in.read( );
22             if( buffer == -1 )
23                 return -1;
24             bufferPos = 0;
25         }
26
27         return getBit( buffer, bufferPos++ );
28     }
29
30     public void close( ) throws IOException
31     {
32         in.close( );
33     }
34
35     private static int getBit( int pack, int pos )
36     {
37         return ( pack & ( 1 << pos ) ) != 0 ? 1 : 0;
38     }
39
40     private InputStream in;
41     private int buffer;
42     private int bufferPos;
43 }
```

Figura 12.14 La clase BitInputStream.

```

1 // Clase BitOutputStream: clase envoltorio de flujos para salida bit a bit.
2 //
3 // CONSTRUCCIÓN: con un OutputStream abierto.
4 //
5 // *****OPERACIONES PÚBLICAS*****
6 // void writeBit( val )    --> Escribir un bit (0 o 1)
7 // void writeBits( vals ) --> Escribir matriz de bits
8 // void flush( )           --> Vaciar bits en buffer
9 // void close( )          --> Cerrar flujo de datos subyacente
10
11 public class BitOutputStream
12 {
13     public BitOutputStream( OutputStream os )
14     { bufferPos = 0; buffer = 0; out = os; }
15
16     public void writeBit( int val ) throws IOException
17     {
18         buffer = setBit( buffer, bufferPos++, val );
19         if( bufferPos == BitUtils.BITS_PER_BYTES )
20             flush( );
21     }
22
23     public void writeBits( int [ ] val ) throws IOException
24     {
25         for( int i = 0; i < val.length; i++ )
26             writeBit( val[ i ] );
27     }
28
29     public void flush( ) throws IOException
30     {
31         if( bufferPos == 0 )
32             return;
33         out.write( buffer );
34         bufferPos = 0;
35         buffer = 0;
36     }
37
38     public void close( ) throws IOException
39     { flush( ); out.close( ); }
40
41     private int setBit( int pack, int pos, int val )
42     {
43         if( val == 1 )

```

*Continúa*

Figura 12.15 La clase BitOutputStream.

```

44     pack |= ( val << pos );
45     return pack;
46 }
47
48 private OutputStream out;
49 private int buffer;
50 private int bufferPos;
51 }

```

**Figura 12.15 (Continuación).**

```

1 // Clase CharCounter: clase para el recuento de caracteres.
2 //
3 // CONSTRUCCIÓN: sin parámetros o con un InputStream abierto.
4 //
5 // *****OPERACIONES PÚBLICAS*****
6 // int getCount( ch )      --> Devuelve # ocurrencias de ch
7 // void setCount( ch, count ) --> Establece # ocurrencias de ch
8 // *****ERRORES*****
9 // No hay comprobación de errores.
10
11 class CharCounter
12 {
13     public CharCounter( )
14     {
15
16         public CharCounter( InputStream input ) throws IOException
17         {
18             int ch;
19             while( ( ch = input.read( ) ) != -1 )
20                 theCounts[ ch ]++;
21         }
22
23         public int getCount( int ch )
24             { return theCounts[ ch & 0xff ]; }
25
26         public void setCount( int ch, int count )
27             { theCounts[ ch & 0xff ] = count; }
28
29         private int [ ] theCounts = new int[ BitUtils.DIFF_BYTES ];
30     }

```

**Figura 12.16 La clase CharCounter.**

```

1 // Nodo básico en un árbol de codificación Huffman.
2 class HuffNode implements Comparable<HuffNode>
3 {
4     public int value;
5     public int weight;
6
7     public int compareTo( HuffNode rhs )
8     {
9         return weight - rhs.weight;
10    }
11
12    HuffNode left;
13    HuffNode right;
14    HuffNode parent;
15
16    HuffNode( int v, int w, HuffNode lt, HuffNode rt, HuffNode pt )
17        { value = v; weight = w; left = lt; right = rt; parent = pt; }
18 }

```

**Figura 12.17** Declaración de nodos para el árbol de codificación de Huffman.

Internamente, `root` es una referencia al nodo raíz del árbol y `theCounts` es un objeto `CharCounter` que se puede utilizar para inicializar los nodos del árbol. También mantenemos una matriz, `theNodes`, que asigna a cada carácter el nodo del árbol que lo contiene.

La Figura 12.19 muestra los constructores y la rutina pública utilizada para devolver el código correspondiente a un carácter determinado. Los constructores comienzan con árboles vacíos, y el constructor de un solo parámetro inicializa el objeto `CharCounter` e inmediatamente después llama a la rutina privada `createTree`. El objeto `CharCounter` se inicializa para que esté vacío en el constructor de cero parámetros.

Para `getCode` obtenemos, consultando `theNodes`, el nodo del árbol que almacena el carácter cuyo código estamos buscando. Si el carácter no está representado, señalizamos el error devolviendo una referencia `null`. En caso contrario, empleamos un sencillo bucle para recorrer el árbol hacia arriba, siguiendo los enlaces a los nodos padre, hasta alcanzar la raíz, que no tiene parent. Cada paso añade un 0 o un 1 al principio de una cadena, que se convierte a una matriz de `int` antes de volver (por supuesto, esto crea muchas cadenas de caracteres temporales, dejamos como ejercicio para el lector el optimizar este paso).

El método `getChar` mostrado en la Figura 12.20 es más simple: comenzamos en la raíz y vamos bifurcándonos a izquierda y derecha según indique el código. Alcanzar `null` prematuramente genera un error. Si eso no sucede, devolvemos el valor almacenado en el nodo (que para los nodos que no sean hojas resulta ser el símbolo `INCOMPLETE`, que indica que el código es incompleto).

En la Figura 12.21 tenemos rutinas para leer y escribir la tabla de codificación. El formato que empleamos es simple y no es necesariamente el más eficiente en términos de espacio. Para cada carácter que tenga un código, lo escribimos (utilizando un byte) y luego escribimos su recuento de caracteres correspondiente (utilizando cuatro bytes). Indicamos el final de la tabla escribiendo una

```
1 // Interfaz de la clase para árboles de Huffman:  
2 // manipulación de un árbol de codificación de Huffman.  
3 // CONSTRUCCIÓN: sin parámetros o con un objeto CharCounter.  
4 //  
5 // *****OPERACIONES PÚBLICAS*****  
6 // int [ ] getCode( ch )           --> Devuelve código de un carácter dado  
7 // int getChar( code )           --> Devuelve el carácter de un código dado  
8 // void writeEncodingTable( out ) --> Escribe en out la tabla de codificación  
9 // void readEncodingTable( in )   --> Lee de in la tabla de codificación  
10 // *****ERRORES*****  
11 // Comprobación del error debido a un código ilegal  
12  
13 class HuffmanTree  
14 {  
15     public HuffmanTree( )  
16         { /* Figura 12.19 */ }  
17     public HuffmanTree( CharCounter cc )  
18         { /* Figura 12.19 */ }  
19  
20     public static final int ERROR = -3;  
21     public static final int INCOMPLETE_CODE = -2;  
22     public static final int END = BitUtils.DIFF_BYTES;  
23  
24     public int [ ] getCode( int ch )  
25         { /* Figura 12.19 */ }  
26     public int getChar( String code )  
27         { /* Figura 12.20 */ }  
28  
29     // Escribir la tabla de codificación utilizando recuentos de caracteres  
30     public void writeEncodingTable( DataOutputStream out ) throws IOException  
31         { /* Figura 12.21 */ }  
32     public void readEncodingTable( DataInputStream in ) throws IOException  
33         { /* Figura 12.21 */ }  
34  
35     private CharCounter theCounts;  
36     private HuffNode [ ] theNodes = new HuffNode[ BitUtils.DIFF_BYTES + 1 ];  
37     private HuffNode root;  
38  
39     private void createTree( )  
40         { /* Figura 12.22 */ }  
41 }
```

Figura 12.18 Esqueleto de la clase HuffmanTree.

```
1  public HuffmanTree( )
2  {
3      theCounts = new CharCounter( );
4      root = null;
5  }
6
7  public HuffmanTree( CharCounter cc )
8  {
9      theCounts = cc;
10     root = null;
11     createTree( );
12 }
13
14 /**
15 * Devolver el código correspondiente al carácter ch.
16 * (El parámetro es un int para permitir EOF).
17 * Si no se encuentra el código, devolver una matriz de longitud 0.
18 */
19 public int [ ] getCode( int ch )
20 {
21     HuffNode current = theNodes[ ch ];
22     if( current == null )
23         return null;
24
25     String v = "";
26     HuffNode par = current.parent;
27
28     while ( par != null )
29     {
30         if( par.left == current )
31             v = "0" + v;
32         else
33             v = "1" + v;
34         current = current.parent;
35         par = current.parent;
36     }
37
38     int [ ] result = new int[ v.length( ) ];
39     for( int i = 0; i < result.length; i++ )
40         result[ i ] = v.charAt( i ) == '0' ? 0 : 1;
41
42     return result;
43 }
```

Figura 12.19 Algunos de los métodos del árbol de Huffman, incluyendo los constructores y la rutina para devolver el código correspondiente a un carácter determinado.

```
1  /**
2   * Obtener el carácter correspondiente a un código.
3   */
4  public int getChar( String code )
5  {
6      HuffNode p = root;
7      for( int i = 0; p != null && i < code.length( ); i++ )
8          if( code.charAt( i ) == '0' )
9              p = p.left;
10         else
11             p = p.right;
12
13         if( p == null )
14             return ERROR;
15
16         return p.value;
17     }
```

Figura 12.20 Una rutina de decodificación (para generar un carácter a partir de un código).

tabla adicional que contiene un carácter terminador nulo '\0' con un recuento de cero. El recuento igual a cero es la señal especial.

El método `readEncodingTable` inicializa todos los recuentos de caracteres a cero y luego lee la tabla, actualizando los recuentos a medida que los lee. Invoca a `calls createTree`, mostrado en la Figura 12.22, para construir el árbol de Huffman.

En dicha rutina, mantenemos una cola con prioridad de tres nodos. Para hacer esto, debemos proporcionar una función de comparación para nodos de árbol. Recuerde de la Figura 12.17 que `HuffNode` implementa `Comparable<HuffNode>`, ordenando los objetos `HuffNode` según el peso de cada nodo.

Después buscamos caracteres que hayan aparecido al menos una vez. Cuando la comprobación de la línea 9 tiene éxito, tendremos uno de tales caracteres. Creamos un nuevo nodo de árbol en las líneas 11 y 12, lo añadimos a `theNodes` en la línea 13 y luego lo añadimos a la cola con prioridad en la línea 14. En las líneas 17 y 18 añadimos el símbolo de final de archivo (*EOF*, *end-of-file*). El bucle que abarca de las líneas 20 a 28 es una traducción línea a línea del algoritmo de construcción del árbol. Mientras que tengamos dos o más árboles, extraemos dos árboles de la cola con prioridad, combinamos el resultado y volvemos a colocarlo en la cola con prioridad. Al final del bucle, solo quedará un árbol en la cola con prioridad, y podremos extraerlo y configurar `root`.

El árbol generado por `createTree` depende de cómo rompa los empates la cola con prioridad. Lamentablemente, esto quiere decir que si el programa se compila en dos máquinas diferentes, con dos implementaciones distintas de colas con prioridad, podría darse el caso de que se comprimiera un archivo en la primera máquina y luego fuéramos incapaces de obtener otra vez el archivo original al tratar de descomprimirlo en la segunda máquina. Evitar este problema requiere una cierta cantidad de trabajo adicional.

```
1  /**
2  * Escribe una tabla de codificación en un flujo de datos de salida.
3  * El formato es carácter, recuento (como bytes).
4  * Un recuento igual a cero marca el fin de la tabla de codificación.
5  */
6  public void writeEncodingTable( DataOutputStream out ) throws IOException
7  {
8      for( int i = 0; i < BitUtils.DIFF_BYTES; i++ )
9      {
10         if( theCounts.getCount( i ) > 0 )
11         {
12             out.writeByte( i );
13             out.writeInt( theCounts.getCount( i ) );
14         }
15     }
16     out.writeByte( 0 );
17     out.writeInt( 0 );
18 }
19
20 /**
21 * Leer la tabla de codificación de un flujo de datos de entrada en el
22 * formato dado y construir el árbol de Huffman. Luego se configurará
23 * la posición dentro del flujo de datos para leer los datos comprimidos.
24 */
25 public void readEncodingTable( DataInputStream in ) throws IOException
26 {
27     for( int i = 0; i < BitUtils.DIFF_BYTES; i++ )
28         theCounts.setCount( i, 0 );
29
30     int ch;
31     int num;
32
33     for( ; ; )
34     {
35         ch = in.readByte();
36         num = in.readInt();
37         if( num == 0 )
38             break;
39         theCounts.setCount( ch, num );
40     }
41
42     createTree();
43 }
```

---

Figura 12.21 Rutinas para leer y escribir tablas de codificación.

```

1  /**
2   * Construir el árbol de codificación de Huffman.
3   */
4  private void createTree( )
5  {
6      PriorityQueue<HuffNode> pq = new PriorityQueue<HuffNode>();
7
8      for( int i = 0; i < BitUtils.DIFF_BYT
9          if( theCounts.getCount( i ) > 0 )
10         {
11             HuffNode newNode = new HuffNode( i,
12                 theCounts.getCount( i ), null, null, null );
13             theNodes[ i ] = newNode;
14             pq.add( newNode );
15         }
16
17     theNodes[ END ] = new HuffNode( END, 1, null, null, null );
18     pq.add( theNodes[ END ] );
19
20     while( pq.size( ) > 1 )
21     {
22         HuffNode n1 = pq.remove( );
23         HuffNode n2 = pq.remove( );
24         HuffNode result = new HuffNode( INCOMPLETE_CODE,
25             n1.weight + n2.weight, n1, n2, null );
26         n1.parent = n2.parent = result;
27         pq.add( result );
28     }
29
30     root = pq.element( );
31 }

```

Figura 12.22 Una rutina para construir el árbol de codificación de Huffman.

### Clases para flujos de datos comprimidos

Todo lo que nos queda por hacer es escribir envoltorios de compresión y descompresión de flujos de datos y luego una rutina `main` que los invoque. Volvemos a recalcar que no hemos incluido las necesarias comprobaciones de errores, porque de lo único de lo que se trata es de ilustrar las ideas algorítmicas básicas.

La clase `HZIPOutputStream` se muestra en la Figura 12.23. El constructor inicia un `DataOutputStream`, en el que podemos escribir el flujo de datos comprimido. También mantenemos un `ByteArrayOutputStream`. Cada llamada a `write` hace que se añada información al `ByteArrayOutputStream`. Cuando se invoca `close`, se escribe el flujo de datos comprimidos real.

```
1 import java.io.IOException;
2 import java.io.OutputStream;
3 import java.io.DataOutputStream;
4 import java.io.ByteArrayInputStream;
5 import java.io.ByteArrayOutputStream;
6
7 /**
8 * Las escrituras a HZIPOutputStream están comprimidas y
9 * se envían al flujo de salida envuelto.
10 * No se lleva a cabo ninguna escritura hasta close.
11 */
12 public class HZIPOutputStream extends OutputStream
13 {
14     public HZIPOutputStream( OutputStream out ) throws IOException
15     {
16         dout = new DataOutputStream( out );
17     }
18
19     public void write( int ch ) throws IOException
20     {
21         byteOut.write( ch );
22     }
23
24     public void close( ) throws IOException
25     {
26         byte [ ] theInput = byteOut.toByteArray( );
27         ByteArrayInputStream byteIn = new ByteArrayInputStream( theInput );
28
29         CharCounter countObj = new CharCounter( byteIn );
30         byteIn.close( );
31
32         HuffmanTree codeTree = new HuffmanTree( countObj );
33         codeTree.writeEncodingTable( dout );
34
35         BitOutputStream bout = new BitOutputStream( dout );
36
37         for( int i = 0; i < theInput.length; i++ )
38             bout.writeBits( codeTree.getCode( theInput[ i ] & 0xff ) );
39             bout.writeBits( codeTree.getCode( BitUtils.EOF ) );
40
41         bout.close( );
42         byteOut.close( );
43     }
44
45     private ByteArrayOutputStream byteOut = new ByteArrayOutputStream( );
46     private DataOutputStream dout;
47 }
```

---

Figura 12.23 La clase HZIPOutputStream.

La rutina `close` extrae todos los bytes que han sido almacenados en el `ByteArrayOutputStream` para su lectura en la línea 26. Luego construye un objeto `CharCounter` en la línea 29 y un objeto `HuffmanTree` en la línea 32. Puesto que `CharCounter` necesita un `InputStream`, construimos un `ByteArrayInputStream` a partir de la matriz de bytes que acabamos de extraer. En la línea 33 escribimos la tabla de codificación.

Llegados a este punto, estamos listos para llevar a cabo la codificación principal. Creamos un objeto de flujo de bits de salida en la línea 35. El resto del algoritmo obtiene repetidamente un carácter y escribe su código (línea 38). Hay un fragmento de código interesante en la línea 38: el `int` pasado a `getCode` puede confundirse con `EOF` si simplemente utilizamos el byte, porque el bit alto puede interpretarse como un bit de signo. Por ello, empleamos una máscara de bits. Cuando salimos del bucle, habremos alcanzado el final del archivo, por tanto, escribimos el código de final del archivo en la línea 39. El método `close` del `BitOutputStream` vuelca cualquier bit restante en el archivo de salida, por lo que no hace falta ninguna llamada explícita a `flush`.

A continuación se muestra la clase `HZIPInputStream`, en la Figura 12.24. El constructor crea un `DataInputStream` y construye un objeto `HuffmanTree` leyendo la tabla de codificación (líneas 15 y 16) desde el flujo de datos comprimido. Después creamos un flujo de entrada de bits en la línea 18. El trabajo sucio se lleva cabo en el método `read`.

El objeto `bits`, declarado en la línea 23, representa el código (Huffman) que estamos examinando actualmente. Cada vez que leemos un bit en la línea 29, añadimos ese bit al final del código Huffman (en la línea 33). A continuación, buscamos el código Huffman en la línea 34. Si está incompleto, continuamos con el bucle (líneas 35 y 36). Si hay un código Huffman ilegal, generamos una `IOException` (líneas 37 a 38). Si llegamos al código de final de archivo, devolvemos `-1`, como es lo habitual para `read` (líneas 39 y 40); en caso contrario, habremos encontrado una correspondencia, así que devolvemos el carácter que se corresponde con el código Huffman (línea 42).

## La rutina `main`

La rutina `main` se proporciona en el código en línea. Si se invoca con el argumento `-c`, comprime; con el argumento `-u`, descomprime. La Figura 12.25 ilustra el envolvimiento de flujos de datos para compresión y descompresión. La compresión añade un `".huf"` al nombre de archivo; la descompresión añade `".uc"` al nombre de archivo, para evitar sobreescribir los archivos originales.

```
1 import java.io.IOException;
2 import java.io.InputStream;
3 import java.io.DataInputStream;
4
5 /**
6  * HZIPInputStream envuelve un flujo de datos de entrada. read devuelve un
7  * byte no comprimido a partir del flujo de datos de entrada envuelto.
8 */
```

Continúa

Figura 12.24 La clase `HZIPInputStream`.

```
9 public class HZIPInputStream extends InputStream
10 {
11     public HZIPInputStream( InputStream in ) throws IOException
12     {
13         DataInputStream din = new DataInputStream( in );
14
15         codeTree = new HuffmanTree( );
16         codeTree.readEncodingTable( din );
17
18         bin = new BitInputStream( in );
19     }
20
21     public int read( ) throws IOException
22     {
23         String bits = "";
24         int bit;
25         int decode;
26
27         while( true )
28         {
29             bit = bin.readBit( );
30             if( bit == -1 )
31                 throw new IOException( "Unexpected EOF" );
32
33             bits += bit;
34             decode = codeTree.getChar( bits );
35             if( decode == HuffmanTree.INCOMPLETE_CODE )
36                 continue;
37             else if( decode == HuffmanTree.ERROR )
38                 throw new IOException( "Decoding error" );
39             else if( decode == HuffmanTree.END )
40                 return -1;
41             else
42                 return decode;
43         }
44     }
45
46     public void close( ) throws IOException
47     {
48         bin.close( );
49     }
50     private BitInputStream bin;
51     private HuffmanTree codeTree;
52 }
```

---

Figura 12.24 (Continuación).

```
1 class Hzip
2 {
3     public static void compress( String inFile ) throws IOException
4     {
5         String compressedFile = inFile + ".huf";
6         InputStream in = new BufferedInputStream(
7             new FileInputStream( inFile ) );
8         OutputStream fout = new BufferedOutputStream(
9             new FileOutputStream( compressedFile ) );
10        HZIPOutputStream hzout = new HZIPOutputStream( fout );
11        int ch;
12        while( ( ch = in.read( ) ) != -1 )
13            hzout.write( ch );
14        in.close( );
15        hzout.close( );
16    }
17
18    public static void uncompress( String compressedFile ) throws IOException
19    {
20        String inFile;
21        String extension;
22
23        inFile = compressedFile.substring( 0, compressedFile.length( ) - 4 );
24        extension = compressedFile.substring( compressedFile.length( ) - 4 );
25
26        if( !extension.equals( ".huf" ) )
27        {
28            System.out.println( "Not a compressed file!" );
29            return;
30        }
31
32        inFile += ".uc"; // para depuración, para no sobreescribir el original
33        InputStream fin = new BufferedInputStream(
34            new FileInputStream( compressedFile ) );
35        DataInputStream in = new DataInputStream( fin );
36        HZIPInputStream hzin = new HZIPInputStream( in );
37
38        OutputStream fout = new BufferedOutputStream(
39            new FileOutputStream( inFile ) );
40        int ch;
41        while( ( ch = hzin.read( ) ) != -1 )
42            fout.write( ch );
43
44        hzin.close( );
45        fout.close( );
46    }
47 }
```

Figura 12.25 Una rutina `main` simple para compresión y descompresión de archivos.

## Mejora del programa

El programa, tal como está escrito, sirve a su propósito principal de ilustrar los fundamentos del algoritmo de codificación de Huffman. Consigue cierta compresión, incluso con tamaños de archivo moderados. Por ejemplo, obtiene una compresión de aproximadamente el 40 por ciento cuando se ejecuta sobre su propio archivo fuente, `Hzip.java`. Sin embargo, el programa podría mejorarse de múltiples maneras.

1. La comprobación de errores es limitada. Un programa de producción debería garantizar rigurosamente que el archivo que se está descomprimiendo es, en realidad, un archivo comprimido. (Una forma de que pueda hacerlo es escribir información adicional en la tabla de codificación.) Las rutinas internas deberían tener más comprobaciones.
2. Se ha hecho muy poco esfuerzo para minimizar el tamaño de la tabla de codificación. Para archivos grandes, esta carencia tiene pocas consecuencias, pero para archivos más pequeños una tabla de codificación grande puede resultar inaceptable, porque la propia tabla de codificación consume espacio.
3. Un programa robusto comprobaría el tamaño del archivo comprimido resultante y abortaría la ejecución si ese tamaño fuera mayor que el del original.
4. En muchos lugares no hemos hecho apenas esfuerzos para optimizar la velocidad. Se podría emplear la técnica de memorización para evitar buscar repetidamente en el árbol un mismo código.

Dejamos como ejercicio para el lector la realización de mejoras adicionales en el programa en los Ejercicios 12.14–12.16.

## 12.2 Un generador de referencias cruzadas

Un generador de referencias cruzadas enumera los identificadores y sus números de línea. Se trata de una aplicación común, porque es similar al proceso de creación de un índice.

En esta sección, vamos a diseñar un programa denominado *generador de referencias cruzadas* que analiza un archivo fuente Java, ordena los identificadores e imprime todos los identificadores junto con los números de línea en los que aparecen. Una aplicación para compiladores sería enumerar, para cada método, el nombre de todos los demás métodos a los que invoca.

Sin embargo, este es un problema de carácter general que se presenta en muchos otros contextos. Por ejemplo, puede utilizarse para generalizar la creación del índice de un libro. Otro uso, la comprobación ortográfica se describe en el Ejercicio 12.21. A medida que un corrector ortográfico detecta palabras mal escritas en un documento, esas palabras se recopilan junto con las líneas en las que aparecen. Este proceso evita imprimir repetidamente la misma palabra mal escrita e indica dónde se encuentran los errores.

### 12.2.1 Ideas básicas

Nuestra principal idea algorítmica consiste en utilizar un mapa para almacenar cada identificador y los números de línea en los que aparece. En el mapa, el identificador es la clave, y la lista de números

de línea es el valor. Después de haber leído el archivo fuente y de haber construido el mapa, podemos iterar a través de la colección, imprimiendo los identificadores y sus correspondientes números de línea.

Utilizamos un mapa para almacenar identificadores y sus números de línea. Almacenamos los números de línea correspondientes a cada identificador en una lista.

### 12.2.2 Implementación java

El esqueleto de la clase `Xref` se muestra en la Figura 12.26. Es similar a (pero más simple que) la clase `Balance` mostrada en la Figura 11.3, que era parte de un programa de equilibrado de símbolos. Como esa clase, hace uso de la clase `Tokenizer` definida en la Figura 11.2.

```
1 import java.io.InputStreamReader;
2 import java.io.IOException;
3 import java.io.FileReader;
4 import java.io.Reader;
5 import java.util.Set
6 import java.util.TreeMap;
7 import java.util.List;
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.Map;
11
12 // Interfaz de la clase Xref: generar referencia cruzada
13 //
14 // CONSTRUCCIÓN: con un objeto Reader
15 //
16 // *****OPERACIONES PÚBLICAS*****
17 // void generateCrossReference( ) --> Genera referencia cruzada
18 // *****ERRORES*****
19 // Se comprueban errores relativos a comentarios y comillas
20
21 public class Xref
22 {
23     public Xref( Reader inStream )
24     { tok = new Tokenizer( inStream ); }
25
26     public void generateCrossReference( )
27     { /* Figura 12.30 */ }
28
29     private Tokenizer tok;    // objeto analizador sintáctico
30 }
```

Figura 12.26 El esqueleto de la clase `Xref`.

Las rutinas de análisis sintáctico son sencillas, aunque como es habitual, requieren algo de esfuerzo.

Ahora podemos analizar la implementación de las dos rutinas restantes de la clase `Tokenizer`: `getNextID` y `getRemainingString`. Estas nuevas rutinas de análisis se ocupan del reconocimiento de un identificador.

La rutina mostrada en la Figura 12.27 comprueba si un carácter forma parte de un identificador. En la rutina `getRemainingString`, mostrada en la Figura 12.28, asumimos que el primer carácter de un identificador ya ha sido leído y está almacenado en el miembro de datos `ch` de la clase `Tokenizer`. Se leen repetidamente caracteres hasta que aparezca uno que no forme parte de un identificador. En este punto, volvemos a poner el carácter en su sitio (en la línea 12) y luego devolvemos un `String`. `StringBuilder` se utiliza para evitar repetidas concatenaciones de objetos `String`, que son costosas. En la Sección 15.4 se describen los problemas relacionados con esto.

La rutina `getNextID` mostrada en la Figura 12.29 es similar a la rutina mostrada en la Figura 11.7. La diferencia es que aquí, en la línea 17, si nos encontramos el primer carácter de un identificador, invocamos a `getRemainingString` para devolver el símbolo sintáctico. El hecho de que `getNextID`

```

1  /**
2   * Devolver true si ch puede ser parte de un identificador Java
3   */
4  private static final boolean isIdChar( char ch )
5  {
6      return Character.isJavaIdentifierPart( ch );
7 }
```

**Figura 12.27** Una rutina para comprobar si un carácter puede ser parte de un identificador.

```

1  /**
2   * Devolver un identificador leído del flujo de datos de entrada
3   * El primer carácter ya ha sido leído en ch
4   */
5  private String getRemainingString( )
6  {
7      StringBuilder result = new StringBuilder( ch );
8
9      for( ; nextChar( ); result.append( ch ) )
10         if( !isIdChar( ch ) )
11         {
12             putBackChar( );
13             break;
14         }
15
16     return new String( result );
17 }
```

**Figura 12.28** Una rutina para devolver un `String` a partir de la entrada.

```

1  /**
2   * Devuelve el siguiente identificador, saltando los comentarios, las
3   * constantes de cadena y las constantes de caracteres.
4   * Colocar el identificador en currentIdNode.word y devolver false
5   * solo si se alcanza el final del flujo de datos.
6   */
7  public String getNextID( )
8  {
9      while( nextChar( ) )
10     {
11         if( ch == '/' )
12             processSlash( );
13         else if( ch == '\\' )
14             nextChar( );
15         else if( ch == '\'' || ch == '"' )
16             skipQuote( ch );
17         else if( !Character.isDigit( ch ) && isIdChar( ch ) )
18             return getRemainingString( );
19     }
20     return null; // Fin de archivo
21 }

```

**Figura 12.29** Una rutina para devolver el siguiente identificador.

y `getNextOpenClose` sean tan similares sugiere que habría merecido la pena escribir una función miembro privada que llevara a cabo sus tareas comunes.

Habiendo escrito todas las rutinas de soporte, consideremos el único método, `generateCrossReference`, mostrado en la Figura 12.30. Las líneas 6 y 7 crean un mapa vacío. Leemos la entrada y construimos el mapa en las líneas 11 a 20. En cada iteración, tenemos el identificador actual `current`. Veamos cómo funciona el cuerpo del bucle. Hay dos casos:

1. El identificador `current` se encuentra en el mapa. En este caso, `lines` proporciona una referencia a la `List` de números de línea y se añade el nuevo número de línea al final de la `List`.
2. El identificador `current` no está en el mapa. En este caso, las líneas 16 y 17 añaden `current` al mapa con una `List` vacía. Así, la llamada a `add` añade el nuevo número de línea a la lista, y como resultado, la `List` contiene ese único número de línea que es lo que queremos.

Una vez que hemos construido el mapa, simplemente iteramos a través de él utilizando un bucle `for` avanzado sobre el conjunto de entradas subyacente. El mapa se visita en el orden que marcan las claves, porque el mapa es un `TreeMap`. Cada vez que aparece una entrada de mapa, necesitamos imprimir información relativa al identificador que está siendo examinado por el iterador del mapa.

La salida se obtiene recorriendo el mapa y empleando un bucle `for` avanzado con el conjunto de entradas. Se emplea un iterador de lista para obtener los números de línea.

Recuerde que un iterador de un conjunto de entradas examina las distintas `Map.Entry`; en `Map.Entry`, la clave está dada por el método `getKey` y el valor es proporcionado por el método `getValue`. Por tanto, el identificador que estemos explorando estará dado por `thisNode.getKey()`, como se

```

1  /**
2  * Imprimir las referencias cruzadas
3  */
4  public void generateCrossReference( )
5  {
6      Map<String,List<Integer>> theIdentifiers =
7          new TreeMap<String,List<Integer>>();
8      String current;
9
10         // Insertar identificadores en el árbol de búsqueda
11     while( ( current = tok.getNextID( ) ) != null )
12     {
13         List<Integer> lines = theIdentifiers.get( current );
14         if( lines == null )
15         {
16             lines = new ArrayList<Integer>();
17             theIdentifiers.put( current, lines );
18         }
19         lines.add( tok.getLineNumber( ) );
20     }
21
22         // Iterar a través del árbol de búsqueda e imprimir
23         // los identificadores y su número de línea
24     Set entries = theIdentifiers.entrySet( );
25     for( Map.Entry<String,List<Integer>> thisNode : entries )
26     {
27         Iterator<Integer> lineItr = thisNode.getValue( ).iterator( );
28
29         // Imprimir el identificador y la primera linea en la que aparece
30         System.out.print( thisNode.getKey( ) + ": " );
31         System.out.print( lineItr.next( ) );
32
33         // Imprimir las demás líneas en las que aparece
34         while( lineItr.hasNext( ) )
35             System.out.print( ", " + lineItr.next( ) );
36         System.out.println( );
37     }
38 }
```

**Figura 12.30** El algoritmo principal de generación de referencias cruzadas.

muestra en la línea 30. Para acceder a las líneas individuales necesitamos un iterador de lista: el iterador de la línea 27 hace referencia a los números de linea correspondientes a la entrada actual.

Imprimimos la palabra y el primer número de línea en las líneas 30 y 31 (tenemos garantizado que la lista no esté vacía). Después, mientras que no alcancemos el final de la lista, imprimimos repetidamente los números de línea dentro del bucle que abarca las líneas 34 y 35. En la línea 36 imprimimos una nueva línea. No proporcionamos un programa `main` aquí, porque coincide esencialmente con el facilitado en la Figura 11.10.

El uso del mapa de esta forma, en el que la clave es algo sencillo y el valor es una lista o algún otro tipo de colección es bastante común. En los ejercicios veremos otros ejemplos en los que se hace uso de esta técnica.

## Resumen

En este capítulo hemos presentado implementaciones de dos utilidades importantes: compresión de texto y referencias cruzadas. La compresión de texto es una técnica de gran importancia que nos permite incrementar tanto la capacidad efectiva de los discos como la velocidad efectiva de los modems. Se trata de un área de investigación muy activa. El método simple descrito aquí; es decir, el algoritmo de Huffman, suele conseguir tasas de compresión del 25 por ciento con archivos de texto. Otros algoritmos y extensiones del algoritmo de Huffman funcionan mejor. La técnica de referencias cruzadas es una técnica general que tiene múltiples aplicaciones.



## Conceptos clave

**algoritmo de Huffman** Un algoritmo que construye un código prefijo óptimo, combinando repetidamente los dos árboles con menor peso. (466)

**árbol completo** Un árbol cuyos nodos son hojas o tienen dos hijos. (465)

**código prefijo** Código en el que ningún código de un carácter es prefijo de ningún otro código de carácter. Esta condición estará garantizada en un *trie* si los caracteres solo aparecen en las hojas. Un código prefijo puede ser decodificado de forma no ambigua. (465)

**compresión** El acto de reducir el número de bits requeridos para la representación de datos. El proceso tiene en la práctica dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión). (464)

**generador de referencias cruzadas** Un programa que enumera identificadores junto con sus números de línea. Es una aplicación común porque es similar al proceso de creación de un índice. (484)

**trie binario** Una estructura de datos en la que la rama izquierda representa un 0 y la rama derecha representa un 1. El camino recorrido hasta un nodo indica su representación. (464)



## Errores comunes

1. Cuando trabaje con la E/S de caracteres, a menudo necesitará utilizar un `int` para almacenar los caracteres, debido al símbolo `EOF` adicional. Hay varios otros problemas de codificación que tienen sus complicaciones.
2. Utilizar demasiada memoria para almacenar la tabla de compresión es un error bastante común. Si se hace así, se limita la cantidad de compresión que se puede conseguir.



## Internet

El programa de compresión y el generador de referencias cruzadas están disponibles en línea.

### **Hzip.java**

Contiene la fuente para el programa de compresión y descompresión. Véase también **HZIPInputStream.java**, **HZIPOutputStream.java** y **Tokenizer.java**.

### **Xref.java**

Contiene la fuente para el generador de referencias cruzadas.



## Ejercicios

### EN RESUMEN

- 12.1 ¿Qué sucede si se utiliza un archivo comprimido con el algoritmo de Huffman para transmitir datos a través de una línea telefónica y se pierde accidentalmente un único bit? ¿Qué se puede hacer en esta situación?
- 12.2 Muestre el árbol de Huffman resultante de la siguiente distribución de caracteres 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).
- 12.3 La mayoría de los sistemas incluyen un programa de compresión. Comprima varios tipos de archivos para determinar la tasa típica compresión en su sistema. ¿Cómo de grandes tienen que ser los archivos como para que la compresión merezca la pena? Compare su rendimiento con el del programa de codificación de Huffman (`Hzip`) proporcionado en el código fuente disponible en línea.

### EN TEORÍA

- 12.4 El algoritmo de Huffman genera ocasionalmente archivos comprimidos que no son más pequeños que el original. Demuestre que todos los algoritmos de compresión tienen que tener esta propiedad (es decir, independientemente del archivo de

compresión que diseñemos, siempre existirán algunos archivos de entrada para los que el algoritmo genera archivos comprimidos que no son más pequeños que los originales).

- 12.5** Demuestre que, si los símbolos ya han sido ordenados según su frecuencia, el algoritmo de Huffman puede implementarse en un tiempo lineal.
- 12.6** ¿En qué circunstancias podría un árbol de Huffman de caracteres ASCII generar un código de 2 bits para un cierto carácter? ¿En qué circunstancias podría generar un código de 20 bits?
- 12.7** Demuestre la corrección del algoritmo de Huffman desarrollando estos pasos.
- Demuestre que ningún nodo tiene un único hijo.
  - Demuestre que los dos caracteres menos frecuentes deben ser los dos nodos más profundos del árbol.
  - Demuestre que los caracteres en cualesquiera dos nodos situados a la misma profundidad pueden intercambiarse sin afectar a la optimidad.
  - Utilice la inducción: al combinar árboles, considere como nuevo conjunto de caracteres los caracteres en las raíces de los árboles.

## EN LA PRÁCTICA

- 12.8** Modifique el algoritmo de modo que si aparece una misma palabra en líneas consecutivas, se indique un rango. Por ejemplo,  
if: 2, 4, 6-9, 11
- 12.9** En el generador de referencias cruzadas, almacene los números de línea en una `LinkedList` en lugar de en un `ArrayList` y compare el rendimiento.
- 12.10** Si una palabra aparece dos veces en una línea, el generador de referencias cruzadas mostrará ese número de línea dos veces. Modifique el algoritmo para que los duplicados solo aparezcan una vez.
- 12.11** Se nos proporciona un `Map` que contiene una libreta de direcciones de una lista de correo electrónico. En el `Map`, las claves son alias y los correspondientes valores son listas compuestas por direcciones de correo electrónico y otros alias. Una dirección de correo electrónico tiene que contener obligatoriamente el signo @ y se garantiza que los alias no contienen el signo @. Un ejemplo de un `Map` sería

```
{ faculty=[fran48@fiu.edu,pat37@fiu.edu],  
  staff=[jls123@fiu.edu,moe45@cis.fiu.edu],  
  facstaff=[faculty,staff],  
  all=[facstaff,president@fiu.edu,provost@fiu.edu] }
```

Escriba la rutina `expandAliases` que tome un `Map` y un alias y devuelva el `Set` de todas las direcciones de correo electrónico a las que se expande el alias. Por ejemplo, expandir `all` nos da un `Set` que contiene seis direcciones de correo electrónico.

Observe que si el parámetro `alias` es una dirección de correo electrónico, `expandAliases` devuelve un `Set` de tamaño uno. Si el parámetro `alias` no es una

dirección de correo electrónico, pero es un alias inválido (es decir, que no está en el mapa), puede devolver un `Set` de tamaño cero. Al escribir su código, suponga primero que no existen ciclos que hagan que un alias termine por incluirse a sí mismo. Al final, trate también el caso en el que un alias aparezca más de una vez en una expansión (llevando la cuenta de los alias que ya han sido expandidos).

- 12.12** En la lista de alumnos de un curso, el número de identificación de nueve dígitos de un estudiante se imprime con un código formado por cinco X al principio y los últimos cuatro dígitos. Por ejemplo, el código para el ID de estudiante 999-44-8901 es XXX-XX-8901. Escriba un método que tome como entrada una matriz compuesta por números de identificación de estudiantes y devuelva una `List<String>` que contenga todos los códigos que se correspondan con dos o más estudiantes.
- 12.13** Escriba una rutina `groupWords` que tome una matriz de objetos `String` como parámetro y devuelva un `Map` en el que las claves sean números representando la longitud de un objeto `String` y el correspondiente valor sea una `List` de todos los objetos `String` de dicha longitud. El `Map` debe estar ordenado según la longitud de las cadenas de caracteres.

## PROYECTOS DE PROGRAMACIÓN

- 12.14** Añada las comprobaciones de error robustas para el programa de compresión que se sugieren al final de la Sección 12.1.3.
- 12.15** Almacenar los recuentos de caracteres en la tabla de codificación proporciona al algoritmo de descompresión la capacidad de realizar comprobaciones de coherencia adicionales. Añada código que verifique que el resultado de la descompresión tiene los mismos recuentos de caracteres que la tabla de codificación.
- 12.16** Describa e implemente un método para almacenar la tabla de codificación que utilice menos espacio que el método trivial consistente en almacenar recuentos de caracteres.
- 12.17** Genere un índice para un libro. El archivo de entrada está compuesto por un conjunto de entradas de índice. Cada línea consiste en la cadena `IX:`, seguida del nombre de una entrada de índice encerrado entre llaves y luego de un número de página encerrado también entre llaves. Cada `:` en el nombre de una entrada de índice representa un subnivel. Los caracteres `[` representan el índice de un rango y los caracteres `]` representan el final del rango. Ocasionalmente, este rango estará dentro de la misma página. En este caso, hay que imprimir un único número de página. Por lo demás, no contraiga o expanda los rangos a su libre albedrío. Como ejemplo, la Figura 12.31 muestra una entrada ejemplo y la Figura 12.32 muestra la salida correspondiente.
- 12.18** Utilice un mapa para implementar un corrector ortográfico. Suponga que el diccionario procede de dos fuentes: un archivo que contiene un diccionario de gran tamaño existente y un segundo archivo que contiene un diccionario personal. Imprima todas las palabras mal escritas y los números de línea en los que aparecen (observe que llevar la cuenta de las palabras mal escritas y sus números de línea es un problema idéntico al de generar una referencia cruzada). Asimismo, para cada

```

IX: {Series|() (2)
IX: {Series!geometric|() (4)
IX: {Euler's constant) (4)
IX: {Series!geometric|)) (4)
IX: {Series!arithmetic|() (4)
IX: {Series!arithmetic|)) (5)
IX: {Series!harmonic|() (5)
IX: {Euler's constant) (5)
IX: {Series!harmonic|)) (5)
IX: {Series|)) (5)

```

Figura 12.31 Entrada de ejemplo para el Ejercicio 12.17.

```

Euler's constant: 4, 5
Series: 2-5
arithmetic: 4-5
geometric: 4
harmonic: 5

```

Figura 12.32 Salida de ejemplo para el Ejercicio 12.17.

palabra mal escrita, enumere las palabras de diccionario que se puedan obtener aplicando cualquiera de las siguientes reglas:

- Añadiendo un carácter.
- Eliminando un carácter.
- Intercambiando caracteres adyacentes.

**12.19** Divida la clase `Tokenizer` en tres clases: una clase base abstracta que se encargue de la funcionalidad común y dos clases derivadas separadas (una que se ocupe de la simbolización para el programa de equilibrado de símbolos y otra que se encargue de la simbolización para el generador de referencias cruzadas).

**12.20** Analice empíricamente el rendimiento del programa de compresión y determine si su velocidad se puede mejorar significativamente. En caso afirmativo, haga los cambios necesarios.

**12.21** Suponga que tiene un `Map` en el que las claves son nombres de estudiantes (`String`) y que para cada estudiante, el valor es una `List` de asignaturas (cada nombre de asignatura es una `String`). Escriba una rutina que calcule el mapa inverso, en el que las claves sean los nombres de las asignaturas y los valores sean las listas de los estudiantes matriculados.

**12.22** Dos palabras son anagramas si contienen el mismo conjunto de letras (y con las mismas frecuencias). Por ejemplo, `least` y `steal` son anagramas. Utilice un mapa para implementar un programa que encuentre grandes grupos de palabras (cinco palabras o más) en el que cada palabra del grupo sea un anagrama de todas las

restantes palabras del grupo. Por ejemplo, `least`, `steal`, `tales`, `stale` y `slate` (un ejemplo de anagramas de cinco letras en español sería `parte`, `rapte`, `repta`, `trepá`, `petar`) son anagramas unas de otras y forman un gran grupo de anagramas. Asuma que existe una larga lista de palabras en un archivo. Para cada palabra, calcule su *representante*. El representante son los caracteres de la palabra en orden alfabético. Por ejemplo, el representante de la palabra `enraged` es `adeegnr`. Observe que las palabras que son anagramas, tendrán todas ellas el mismo representante. Por tanto, el representante de `grenade` es también `adeegnr`. Utilice un `Map` en el que la clave sea un objeto `String` que sea un representante y el valor sea una `List` de todas las palabras que tengan dicha clave como su representante. Después de construir el `Map`, simplemente necesitará encontrar todos los valores cuyas listas tengan un tamaño igual a cinco o superior, e imprimir dichas listas. Ignore la distinción entre mayúsculas y minúsculas.

- 12.23** Implemente un algoritmo de ordenación utilizando un `TreeMap`. Puesto que un `TreeMap` no permite duplicados, cada valor del `TreeMap` es una lista que contiene duplicados.
- 12.24** Un `MultiSet`, tal como se describe en el Ejercicio 6.29, es como un `Set`, pero permite la existencia de duplicados. El Ejercicio 6.29 sugería una implementación que utiliza un `Map`, en el que el valor representa un recuento del número de duplicados. Sin embargo, dicha implementación pierde información. Por ejemplo, si añadimos un `BigDecimal` que representa 4,0 y otro `BigDecimal` que representa 4,000 (observe que para estos dos objetos, `compareTo` da como resultado 0, mientras que `equals` da `false`), la primera ocurrencia se insertará con un recuento igual a 2 y `toString` perderá necesariamente información acerca de que 4,000 se encuentra en el `MultiSet`. En consecuencia, una implementación alternativa consistiría en utilizar un `Map`, en el que el valor represente una lista de todas las instancias adicionales de clave. Escriba una implementación completa del `MultiSet`, y compruébelo añadiendo varios valores `BigDecimal` que sean lógicamente iguales.
- 12.25** El método estático `computeCounts` toma como entrada una matriz de cadenas de caracteres y devuelve un mapa que almacena las cadenas como claves y el número de apariciones de cada cadena como valores.
- Implemente `computeCounts` y determine el tiempo de ejecución de su implementación.
  - Escriba una rutina, `mostCommonStrings`, que tome el mapa generado en la parte (a) y devuelva una lista de las cadenas de caracteres que aparecen más frecuentemente (es decir, si hay  $k$  cadenas de caracteres que estén empatadas como las más comunes, la lista de retorno tendrá tamaño  $k$ ), y determine el tiempo de ejecución de su rutina.



## Referencias

El artículo original sobre el algoritmo de Huffman es [3]. En [2] y [4] se analizan variantes del algoritmo. Otro esquema de compresión popular es la codificación *Ziv-Lempel*, descrita en [7] y [6]. Funciona generando una serie de códigos de longitud fija. Normalmente, generaríamos 4.096 códigos de 12 bits que representen las subcadenas más comunes del archivo. Las referencias [1] y [5] proporcionan buenas panorámicas sobre 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.



## Simulación

Un uso importante de las computadoras está en el campo de la *simulación*, en el que las computadoras se emplean para emular el funcionamiento de un sistema real y recopilar estadísticas. Por ejemplo, puede que sea conveniente simular el funcionamiento de un banco con  $k$  cajeros, con el fin de determinar el valor mínimo de  $k$  que proporciona un tiempo de servicio razonable. Utilizar una computadora para esta tarea tiene muchas ventajas. En primer lugar, la información se recopilará sin necesidad de implicar a los clientes reales. En segundo lugar, una simulación por computadora puede ser más rápida que la implementación real, debido a la velocidad de los equipos informáticos. En tercer lugar, la simulación se puede repetir fácilmente. En muchos casos, la apropiada elección de las estructuras de datos nos puede ayudar a mejorar la eficiencia de la simulación.

Un uso importante de la computadoras está en el campo de la simulación, donde la computadora se utiliza para emular el funcionamiento de un sistema real y recopilar estadísticas.

En este capítulo, veremos

- Cómo simular un juego modelado en torno al problema de Josefo.
- Cómo simular el funcionamiento de un servicio de atención de llamadas telefónicas.

### 13.1 El problema de Josefo

El llamado *problema de Josefo* es el siguiente juego:  $N$  personas, numeradas de 1 a  $N$ , están sentadas formando un círculo; comenzando por la persona 1, se pasan una patata caliente; después de haber pasado la patata  $M$  veces, la persona que la tiene es eliminada, el círculo se estrecha y el juego continúa teniendo la patata la persona que estaba sentada después de la persona eliminada; la última persona que quede, gana. Una suposición común en este juego es que  $M$  sea una constante, aunque también se podría emplear un generador de números aleatorios para cambiar el valor de  $M$  después de cada eliminación.

El problema de Josefo surgió en el primer siglo de nuestra era, en una cueva de una montaña en Israel en la que los zelotes judíos estaban sitiados por los soldados romanos. El historiador Josefo estaba entre ellos. Para consternación de Josefo, los zelotes votaron a favor de un pacto de suicidio, antes que rendirse a los romanos. Josefo sugirió el juego que ahora lleva su nombre. La patata caliente era la sentencia a muerte de la persona situada a continuación de la que tenía la patata. Josefo manipuló el juego para que la patata le llegara a él en último lugar y convenció a la víctima

En el problema de Josefo, se pasa repetidamente una patata caliente; cuando se termina de pasar, el jugador que la tiene en ese momento es eliminado; el juego continúa y el último jugador que queda gana.

que quedaba que los dos debían rendirse. Así es como conocemos hoy día este juego. De hecho, lo que hizo Josefo fue hacer trampas.<sup>1</sup>

Si  $M = 0$ , los jugadores se eliminan por orden, y el último jugador gana siempre. Para otros valores de  $M$ , las cosas no son tan obvias. La Figura 13.1 muestra que si  $N = 5$  y  $M = 1$ , los jugadores son eliminados según el orden 2, 4, 1, 5. En este caso, el jugador 3 gana. Los pasos son los siguientes:

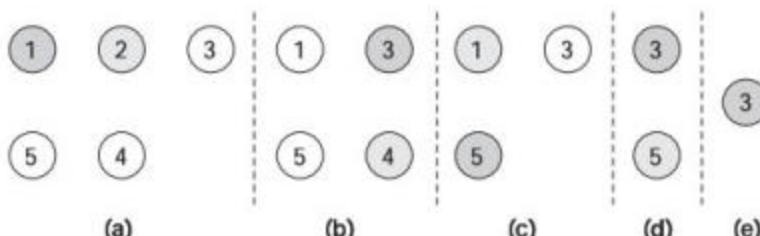
1. Al principio, la patata la tiene el jugador 1. Después de una pasada, la tiene el jugador 2.
2. El jugador 2 es eliminado. El jugador 3 toma la patata y después de una pasada se encontrará en poder del jugador 4.
3. El jugador 4 es eliminado. El jugador 5 toma la patata y se la pasa al jugador 1.
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, así que el jugador 3 gana.

En primer lugar, vamos a escribir un programa que simula, pasada a pasada, una de estas partidas para cualesquiera valores de  $N$  y  $M$ . El tiempo de ejecución de la simulación es  $O(MN)$ , que es aceptable si el número de pasadas es pequeño. Cada paso requiere un tiempo  $O(M)$ , porque se realizan  $M$  pasadas. Después, veremos cómo implementar cada paso en un tiempo  $O(\log N)$ , independientemente del número de veces que se pase la patata. El tiempo de ejecución de la simulación pasa a ser  $O(N \log N)$ .

### 13.1.1 La solución simple

Podemos representar a los jugadores mediante una lista enlazada y utilizar el iterador para simular el paso de la patata.

La etapa de paso de la patata en el problema de Josefo sugiere que representemos a los jugadores según una lista enlazada. Creamos una lista enlazada en la que se insertan por orden los elementos 1, 2, ...,  $N$ . Después, configuramos un iterador que apunte al primer elemento. Cada operación de pasar la patata se corresponde con una operación `next` del iterador. Al llegar



**Figura 13.1** El problema de Josefo: en cada paso, el círculo de color más oscuro representa al jugador que tiene la patata inicialmente y el círculo más claro representa al jugador que recibe la patata caliente (y es eliminado). El paso de la patata se realiza en el sentido de las agujas del reloj.

<sup>1</sup> Gracias a David Teague por enviarme esta historia. La versión que aquí resolvemos viene de la descripción histórica. En el Ejercicio 13.9 pedimos al lector que resuelva la versión histórica.

al último jugador (que queda actualmente) de la lista, implementamos el paso de la patata creando un nuevo iterador posicionado antes del primer elemento. Esta acción permite simular la existencia del círculo. Una vez que terminamos de pasar la patata, eliminamos el elemento en el que se encuentre el iterador.

En la Figura 13.2 se muestra una implementación. La lista enlazada y el iterador se declaran en las líneas 8 y 15. Construimos la lista inicial utilizando el bucle de las líneas 11 y 12.

En la Figura 13.2, el código de las líneas 18 a 25, implementa un paso el algoritmo, pasando la patata (líneas 18 a 24) y eliminando luego un jugador (línea 25). Este procedimiento se repite hasta que la comprobación de la línea 16 nos dice que solo queda un jugador. En dicho punto, devolvemos el número del jugador en la línea 30.

```
1  /**
2   * Devolver el ganador en el problema de Josefo.
3   * Implementación mediante una lista enlazada.
4   * (Se puede reemplazar por ArrayList o TreeSet).
5   */
6  public static int josephus( int people, int passes )
7  {
8      Collection<Integer> theList = new LinkedList<Integer>();
9
10     // Construir la lista
11     for( int i = 1; i <= people; i++ )
12         theList.add( i );
13
14     // Jugar la partida;
15     Iterator<Integer> itr = theList.iterator( );
16     while( people-- != 1 )
17     {
18         for( int i = 0; i <= passes; i++ )
19         {
20             if( !itr.hasNext( ) )
21                 itr = theList.iterator( );
22
23             itr.next( );
24         }
25         itr.remove( );
26     }
27
28     itr = theList.iterator( );
29
30     return itr.next( );
31 }
```

Figura 13.2 Implementación del problema de Josefo mediante una lista enlazada.

El tiempo de ejecución de esta rutina es  $O(MN)$ , porque ese es exactamente el número de pasos de la patata que se producen durante algoritmo. Para valores pequeños de  $M$ , este tiempo de ejecución es aceptable, aunque hay que recalcar que el caso  $M = 0$  no proporciona un tiempo de ejecución de  $O(0)$ ; obviamente, el tiempo de ejecución es  $O(N)$ . Simplemente, no podemos multiplicar meramente por cero al tratar de interpretar una expresión O mayúscula. Observe que podemos sustituir `LinkedList` por `ArrayList`, sin afectar al tiempo de ejecución. También podemos utilizar un `TreeSet`, pero el coste de construcción no será  $O(N)$ .

### 13.1.2 Un algoritmo más eficiente

Si implementamos cada ronda de paso de la patata mediante una única operación logarítmica, la simulación será más rápida.

total de jugadores y  $P$  será 1. Después de pasar  $M$  veces la patata, un cálculo nos dice que estaremos situados en el jugador  $((P + M) \bmod N)$ , contando desde el principio, salvo si eso nos da como resultado el jugador 0, en cuyo caso tenemos que situarnos en el jugador  $N$ . Este cálculo es algo complicado, aunque el concepto no lo es.

El cálculo es complicado, debido a la existencia del círculo.

Aplicando este cálculo a la Figura 13.1, observamos que  $M$  es 1,  $N$  es inicialmente 5 y  $P$  es inicialmente 1. Por tanto, el nuevo valor de  $P$  será 2. Después de la eliminación,  $N$  se reduce a 4, pero seguimos estando en la posición 2, como sugiere la parte (b) de la figura. El siguiente valor de  $P$  es 3, como también se muestra en la parte (b), por lo que se borra el tercer elemento de la lista y  $N$  pasa a valer 3. El siguiente valor de  $P$  es  $4 \bmod 3$ , o 1, por lo que estaremos de nuevo situados en el primer jugador de la lista restante, como se muestra en el parte (c). Este jugador se elimina y  $N$  pasa a ser 2. En este punto, sumamos  $M$  a  $P$ , obteniendo 2. Puesto que  $2 \bmod 2$  es 0, asignamos a  $P$  el valor  $N$ , por lo que será eliminado el último jugador de la lista. Esta acción concuerda con la parte (d) de la figura. Después de la eliminación,  $N$  es 1 y habremos terminado.

`findKth` puede ser soportada por un árbol de búsqueda.

Así, todo lo que necesitamos es una estructura de datos que soporte de manera eficiente la operación `findKth` de encontrar el  $k$ -ésimo elemento. La operación `findKth` devuelve el  $k$ -ésimo elemento (más pequeño), para cualquier parámetro  $k$ .<sup>2</sup> Lamentablemente, ninguna estructura de datos de la API de Colecciones soporta la operación `findKth`. Sin embargo, podemos utilizar una de las estructuras de datos genéricas que implementaremos en la Parte Cuatro. Recuerde de las explicaciones de la Sección 6.7 que las estructuras de datos que implementaremos en el Capítulo 19 se ajustan a un protocolo básico que emplea `insert`, `remove` y `find`. Podemos entonces añadir `findKth` a la implementación.

Existen varias alternativas similares. Todas ellas utilizan el hecho de que, como se explica en la Sección 6.7, `TreeSet` podría haber soportado la operación de encontrar el  $k$ -ésimo elemento en

<sup>2</sup> El parámetro  $k$  para `findKth` va de 1 a  $N$ , ambos inclusive, donde  $N$  es el número de elementos que forman la estructura de datos.

un tiempo logarítmico como promedio o en un tiempo logarítmico como caso peor, si hubiéramos usado un sofisticado árbol de búsqueda binaria. En consecuencia, podemos esperar obtener un algoritmo  $O(N \log N)$  si tenemos el cuidado suficiente.

El método más simple consiste en insertar los elementos secuencialmente en un árbol de búsqueda binaria eficiente en el caso peor, como por ejemplo un árbol rojo-negro, un árbol-AA o un árbol *splay* (hablaremos de estos árboles en capítulos posteriores). Podemos entonces invocar a `findKth` y `remove`, de la forma apropiada. Resulta que un árbol *splay* (un tipo de árbol autoajustable) constituye una excelente elección para esta aplicación, porque las operaciones `findKth` e `insert` son inusualmente eficientes y `remove` no es terriblemente difícil de codificar. Sin embargo, aquí utilizaremos una alternativa, porque las implementaciones de estas estructuras de datos que proporcionamos en capítulos posteriores dejan la implementación de `findKth` como ejercicio para el lector.

Vamos a utilizar la clase `BinarySearchTreeWithRank` que soporta la operación `findKth` y está completamente implementada en la Sección 19.2. Se basa en el árbol de búsqueda binaria simple y no tiene, por tanto, un rendimiento logarítmico en el caso peor, sino solo en el caso promedio. En consecuencia, no podemos limitarnos a insertar los elementos de forma meramente secuencial; eso haría que el árbol de búsqueda exhibiera su rendimiento de caso peor.

Tenemos varias opciones. Una consiste en insertar una permutación aleatoria de  $1, \dots, N$  en el árbol de búsqueda. La otra es construir un árbol de búsqueda binaria perfectamente equilibrado mediante un método de clase. Puesto que un método de clase tendría acceso a los entresijos internos del árbol de búsqueda, la operación se podría realizar en un tiempo lineal. Esta rutina se deja como ejercicio para el lector para cuando analicemos los árboles de búsqueda.

El método que utilizaremos consiste en escribir una rutina recursiva que inserte los elementos en orden equilibrado. Insertando el elemento central en la raíz y construyendo los dos subárboles recursivamente 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 la rutina de la clase con tiempo lineal, no afecta de manera adversa al tiempo asintótico de ejecución del algoritmo global. Entonces se garantiza que las operaciones `remove` sean logarítmicas. Esta rutina se denomina `buildTree`; en la Figura 13.3 se muestra el código correspondiente a esta rutina y al método `josephus`.

Un árbol de búsqueda equilibrado funcionaría, pero no es necesario si tenemos cuidado y construimos un árbol de búsqueda binaria simple que no esté desequilibrado al comienzo. Se puede emplear un método de clase para construir en un tiempo lineal un árbol perfectamente equilibrado.

Construimos el mismo árbol mediante inserciones recursivas, pero utilizando un tiempo  $O(N \log N)$ .

## 13.2 Simulación dirigida por sucesos

Volvamos al problema de la simulación bancaria descrita en la introducción. Aquí, tenemos un sistema en el que los clientes llegan y esperan en la cola hasta que está disponible uno de los  $k$  cajeros. La llegada de los clientes está gobernada por una función de distribución de probabilidad, como también lo está el *tiempo de servicio* (la cantidad de tiempo necesaria para ser atendido una vez que un cajero está disponible). Nos interesa obtener estadísticas tales como el tiempo medio que un cliente tiene que esperar y el porcentaje de tiempo que los cajeros están realmente atendiendo solicitudes. (Si hay demasiados cajeros, algunos no harán nada durante un largo periodo de tiempo.)

```

1  /**
2   * Construir recursivamente un BinarySearchTreeWithRank perfectamente
3   * equilibrado mediante inserciones repetidas en un tiempo  $O(N \log N)$ .
4   * t debe estar vacío en la llamada inicial.
5   */
6  public static void buildTree( BinarySearchTreeWithRank<Integer> t,
7                               int low, int high )
8  {
9      int center = ( low + high ) / 2;
10     if( low <= high )
11     {
12         t.insert( center );
13
14         buildTree( t, low, center - 1 );
15         buildTree( t, center + 1, high );
16     }
17 }
18 }
19
20 /**
21  * Devolver el ganador del problema de Josefo.
22  * Implementación del árbol de búsqueda.
23  */
24 public static int josephus( int people, int passes )
25 {
26     BinarySearchTreeWithRank<Integer> t =
27         new BinarySearchTreeWithRank<Integer>();
28
29     buildTree( t, 1, people );
30
31     int rank = 1;
32     while( people > 1 )
33     {
34         rank = ( rank + passes ) % people;
35         if( rank == 0 )
36             rank = people;
37
38         t.remove( t.findKth( rank ) );
39         people--;
40     }
41
42     return t.findKth( 1 );
43 }

```

---

Figura 13.3 Una solución  $O(N \log N)$  al problema de Josefo.

Con ciertas distribuciones de probabilidad y ciertos valores de  $k$ , podemos calcular estas respuestas de forma exacta. Sin embargo, a medida que  $k$  va aumentando de tamaño, el análisis se va haciendo considerablemente más difícil, por lo que resulta extremadamente útil emplear una computadora para simular el funcionamiento del banco. De esta forma, los directores pueden determinar cuántos cajeros hacen falta para garantizar un servicio razonablemente adecuado. La mayoría de las simulaciones requieren conocimientos de probabilidad, estadística y teoría de colas.

### 13.2.1 Ideas básicas

Una simulación de sucesos discretos consiste en procesar sucesos uno tras otro. Aquí, los dos sucesos son (1) la llegada de un cliente y (2) la marcha de un cliente, liberando así a un cajero.

Podemos utilizar una función de probabilidad para generar un flujo de datos de entrada de parejas de instantes de llegada y tiempos de servicio al cliente, ordenadas por el instante de llegada.<sup>3</sup> No necesitamos utilizar la hora exacta del día. En lugar de ello, podemos emplear un quantum temporal, al que denominaremos *tic*.

En una *simulación dirigida por tiempo discreto* podemos arrancar un reloj de simulación en el instante igual a cero tics y hacer avanzar el reloj un tic cada vez, comprobando si se produce algún suceso. En caso de que se produzca, procesamos el suceso o sucesos y recopilamos las estadísticas. Cuando no queden clientes en el flujo de datos de entrada y todos los cajeros estén libres, la simulación habrá terminado.

El problema con esta estrategia de simulación es que su tiempo de ejecución no depende ni del número de clientes ni de los sucesos (hay dos sucesos por cliente en este caso). En lugar de ello depende del número de tics, que no forman en realidad parte de la entrada. Para ver por qué esta condición es importante, cambiemos las unidades de reloj a microtics y multipliquemos todos los tiempos de la entrada por 1.000.000. La simulación tardaría entonces un tiempo 1.000.000 de veces más grande.

La clave para evitar este problema consiste en hacer avanzar el reloj en cada etapa hasta el tiempo correspondiente al siguiente suceso, lo cual es algo que se denomina *simulación dirigida por sucesos*, lo que es conceptualmente fácil de llevar a cabo. En cualquier punto, el siguiente suceso que puede tener lugar es la llegada del siguiente cliente del flujo de datos de entrada o la marcha de uno de los clientes de la ventanilla correspondiente a un cajero.

Están disponibles todos los instantes en los que se producen los sucesos, por lo que lo único que tenemos que encontrar es el suceso que se produzca antes y procesarlo (fijando como instante actual el momento en que ese suceso se produce).

En el caso de que un cliente se marche, el procesamiento incluye recopilar estadísticas relativas al cliente que se ha marchado y comprobar la cola para determinar si hay otro cliente esperando. En caso afirmativo, añadimos ese cliente, procesamos las estadísticas requeridas, calculamos el instante en el que el cliente se marchará y añadiremos dicha marcha al conjunto de sucesos que están esperando a ocurrir.

El *tic* es el quantum temporal en una simulación.

Una simulación dirigida por tiempo discreto procesa cada unidad de tiempo consecutivamente. Es inapropiada si el intervalo entre sucesos consecutivos es grande.

Una simulación dirigida por sucesos hace avanzar el instante actual hasta el correspondiente al siguiente suceso.

<sup>3</sup> La función de probabilidad genera *tiempos entre llegadas*, garantizando así que las llegadas se generen cronológicamente.

Si el suceso es una llegada de un cliente, comprobamos si hay un cajero disponible. Si no hay ninguno, colocamos a ese cliente en la cola. En caso contrario, asignamos un cajero al cliente, calculamos el instante en el que el cliente se marchará y añadiremos esa marcha al conjunto de sucesos que están esperando a ocurrir.

El conjunto de sucesos (sucesos que están esperando a ocurrir) se organiza en forma de cola con prioridad.

La cola de espera de clientes se puede implementar mediante una estructura de cola. Puesto que necesitamos encontrar el siguiente suceso que vaya a producirse, el conjunto de sucesos se debe organizar en una cola con prioridad. El siguiente suceso será, por tanto, una llegada o una marcha (lo que ocurra antes); ambos tipos de suceso están fácilmente disponibles. Una simulación dirigida por sucesos es apropiada si se espera que el número de ticks entre sucesos sea grande.

### 13.2.2 Ejemplo: una simulación de un servicio de atención telefónica

El principal elemento algorítmico en una simulación es la organización de los sucesos en una cola con prioridad. Para centrarnos en este requisito vamos a escribir una simulación simple. El sistema que simularemos es el servicio de atención telefónica en una gran empresa.

El servicio de atención telefónica elimina la cola de la simulación. Por tanto, solo hay una estructura de datos.

Un servicio de atención telefónica está compuesto por una serie de operadores que atienden llamadas telefónicas. Se accede al operador marcando un número de teléfono. Si está disponible alguno de los operadores, se conecta al usuario con él. Si todos los operadores ya están atendiendo una llamada, el teléfono comunicará. Esto puede contemplarse como el mecanismo utilizado por un servicio automatizado de atención al cliente.

Nuestra simulación modela el servicio proporcionado por el conjunto de operadores. Las variables son:

- El número de operadores de los que dispone el servicio.
- La distribución de probabilidad que gobierna los intentos de llamada.
- La distribución de probabilidad que gobierna el tiempo de atención.
- El periodo de tiempo durante el que hay que ejecutar la simulación.

La simulación del servicio de atención telefónica es una versión simplificada de la simulación de los cajeros, porque no existe cola de espera. Cada llamada entrante es una llegada y el tiempo total invertido una vez que se ha establecido una conexión es el tiempo de servicio. Eliminando la cola de espera, eliminamos la necesidad de mantener una estructura de datos de cola. Por tanto, solo tenemos una estructura de datos, la cola con prioridad. En el Ejercicio 13.16 le pediremos que

incorpore una cola, podrán ponerse hasta  $L$  llamadas en cola si todos los operadores están ocupados.

Informamos de cada suceso a medida que se produce; la recopilación de estadísticas es una simple extensión de este programa.

Para simplificar las cosas, no calculamos estadísticas, simplemente enumeramos cada suceso a medida que se procesa. También suponemos que los intentos de conexión tienen lugar a intervalos constantes; en una simulación precisa, modelaríamos este tiempo entre llegadas con un proceso aleatorio. La Figura 13.4 muestra la salida de una simulación.

```
1 User 0 dials in at time 0 and connects for 1 minute
2 User 0 hangs up at time 1
3 User 1 dials in at time 1 and connects for 5 minutes
4 User 2 dials in at time 2 and connects for 4 minutes
5 User 3 dials in at time 3 and connects for 11 minutes
6 User 4 dials in at time 4 but gets busy signal
7 User 5 dials in at time 5 but gets busy signal
8 User 6 dials in at time 6 but gets busy signal
9 User 1 hangs up at time 6
10 User 2 hangs up at time 6
11 User 7 dials in at time 7 and connects for 8 minutes
12 User 8 dials in at time 8 and connects for 6 minutes
13 User 9 dials in at time 9 but gets busy signal
14 User 10 dials in at time 10 but gets busy signal
15 User 11 dials in at time 11 but gets busy signal
16 User 12 dials in at time 12 but gets busy signal
17 User 13 dials in at time 13 but gets busy signal
18 User 3 hangs up at time 14
19 User 14 dials in at time 14 and connects for 6 minutes
20 User 8 hangs up at time 14
21 User 15 dials in at time 15 and connects for 3 minutes
22 User 7 hangs up at time 15
23 User 16 dials in at time 16 and connects for 5 minutes
24 User 17 dials in at time 17 but gets busy signal
25 User 15 hangs up at time 18
26 User 18 dials in at time 18 and connects for 7 minutes
```

Figura 13.4 Salida de ejemplo para la simulación del servicio de atención telefónica, para tres líneas de teléfono: entra una llamada por minuto; el tiempo medio de conexión es de 5 minutos y la simulación se ejecuta durante 18 minutos.

La clase de simulación requiere otra clase para representar los sucesos. La clase `Event` se muestra en la Figura 13.5. Los miembros de datos están compuestos por el número de cliente, el instante de tiempo en el que tendrá lugar el suceso y una indicación del tipo de suceso del que se trata (`DIAL_IN` o `HANG_UP` para indicar una llamada entrante o un fin de llamada). Si esta simulación fuera más compleja, con varios tipos de sucesos, haríamos de `Event` una clase abstracta y derivaríamos subclases a partir de ella. No hacemos esto aquí porque complicaría las cosas y oscurecería el funcionamiento básico del algoritmo de simulación. La clase `Event` contiene un constructor y una función de comparación utilizados por la cola con prioridad. La clase `Event` concede estatus de visibilidad de paquete a la clase de simulación del servicio de atención telefónica, de modo que los métodos de `CallSim` pueden acceder a los miembros internos de `Event`. La clase `Event` está anidada dentro de la clase `CallSim`.

La clase `Event` representa los sucesos. En una simulación compleja, derivaríamos todos los posibles tipos de sucesos como subclases. Sin embargo, utilizar la herencia para la clase `Event` complicaría el código.

```
1  /**
2   * La clase que representa los sucesos.
3   * Implementa la interfaz Comparable para
4   * ordenar los sucesos según el instante en el que ocurren.
5   * (anidada en CallSim)
6   */
7  private static class Event implements Comparable<Event>
8  {
9      static final int DIAL_IN = 1;
10     static final int HANG_UP = 2;
11
12     public Event( )
13     {
14         this( 0, 0, DIAL_IN );
15     }
16
17     public Event( int name, int tm, int type )
18     {
19         who = name;
20         time = tm;
21         what = type;
22     }
23
24     public int compareTo( Event rhs )
25     {
26         return time - rhs.time;
27     }
28
29     int who; // el número de usuario
30     int time; // cuándo tendrá lugar un suceso
31     int what; // DIAL_IN o HANG_UP
32 }
```

---

Figura 13.5 La clase Event utilizada para la simulación del servicio de atención telefónica.

El esqueleto de la clase para la simulación del servicio de atención telefónica, `CallSim`, se muestra en la Figura 13.6. Está compuesto por un montón de miembros de datos, un constructor y dos métodos. Los miembros de datos incluyen un objeto numérico aleatorio `r` mostrado en la línea 27. En la línea 28, el conjunto de sucesos `eventSet` se mantiene como una cola con prioridad de objetos `Event`. Los restantes miembros de datos son `availableOperators`, que es inicialmente el número de operadores de la simulación, pero va cambiando a medida que los usuarios conectan y desconectan para representar el número de operadores disponibles, y `avgCallLen` y `freqOfCalls`, que son parámetros de la simulación que indican la duración media de las llamadas y la frecuencia

```
1 import weiss.util.Random;
2 import java.util.PriorityQueue;
3
4 // Interfaz de la clase CallSim: ejecuta una simulación
5 //
6 // CONSTRUCCIÓN: con tres parámetros: el número de operadores,
7 // el tiempo medio de conexión y el
8 // tiempo entre llamadas.
9 //
10 // *****OPERACIONES PÚBLICAS*****
11 // void runSim( ) --> Ejecutar una simulación
12
13 public class CallSim
14 {
15     public CallSim( int operators, double avgLen, int callIntrvl )
16         { /* Figura 13.7 */ }
17
18     // Ejecutar la simulación.
19     public void runSim( long stoppingTime )
20         { /* Figura 13.9 */ }
21
22     // Añadir una llamada a eventSet en el instante actual y
23     // programar una para un cierto delta en el futuro.
24     private void nextCall( int delta )
25         { /* Figura 13.8 */ }
26
27     private Random r;                      // Un origen aleatorio
28     private PriorityQueue<Event> eventSet; // Sucesos pendientes
29
30     // Parámetros básicos de la simulación
31     private int availableOperators; // Número de operadores disponibles
32     private double avgCallLen;      // Duración de una llamada
33     private int freqOfCalls;        // Intervalo entre llamadas
34
35     private static class Event implements Comparable<Event>
36         { /* Figura 13.5 */ }
37 }
```

Figura 13.6 El esqueleto de la clase CallSim.

de las mismas. Recuerde que habrá un intento de conexión cada `freqOfCalls` ticks. El constructor, declarado en la línea 15 e implementado en la Figura 13.7, inicializa estos miembros de datos y coloca la primera llamada en la cola con prioridad `eventSet`.

El método `nextCall` añade una solicitud en forma de llamada entrante a un conjunto de sucesos.

La clase de simulación está compuesta solo por dos métodos. En primer lugar, `nextCall`, mostrado en la Figura 13.8, añade una solicitud en forma de llamada entrante al conjunto de sucesos. Mantiene dos variables privadas: el número del siguiente usuario que tratará de conectar y cuándo se producirá ese suceso. De nuevo, hemos hecho la simplificación de que las llamadas se producen a intervalos regulares. En la práctica, utilizaremos un generador de números aleatorios para modelar el flujo de llamadas entrantes.

```

1  /**
2   * Constructor.
3   * @param operators número de operadores.
4   * @param avgLen duración media de una llamada.
5   * @param callIntrvl tiempo medio entre llamadas.
6   */
7  public CallSim( int operators, double avgLen, int callIntrvl )
8  {
9      eventSet = new PriorityQueue<Event>();
10     availableOperators = operators;
11     avgCallLen = avgLen;
12     freqOfCalls = callIntrvl;
13     r = new Random();
14     nextCall( freqOfCalls ); // Programar primera llamada
15 }
```

Figura 13.7 El constructor `CallSim`.

```

1  private int userNum = 0;
2  private int nextCallTime = 0;
3
4  /**
5   * Colocar un nuevo suceso DIAL_IN en la cola de sucesos. Programar el
6   * instante en el que el siguiente suceso DIAL_IN tendrá lugar. En la
7   * práctica usaremos un número aleatorio para establecer el instante.
8   */
9  private void nextCall( int delta )
10 {
11     Event ev = new Event( userNum++, nextCallTime, Event.DIAL_IN );
12     eventSet.add( ev );
13     nextCallTime += delta;
14 }
```

Figura 13.8 El método `nextCall` coloca un nuevo suceso `DIAL_IN` en la cola de sucesos y programa el tiempo en el que tendrá lugar el siguiente suceso `DIAL_IN`.

El otro método es `runSim`, que se invoca para ejecutar la simulación completa. El método `runSim` realiza la mayor parte del trabajo y se muestra en la Figura 13.9. Se invoca con un único parámetro que indica cuándo debe terminar la simulación. Mientras que el conjunto de sucesos no esté vacío, nos dedicamos a procesar sucesos. Observe que nunca debería estar vacío, porque en el momento que llegamos a la línea 12 hay exactamente una solicitud de llamada entrante en la cola con prioridad y una solicitud de desconexión por cada llamante actualmente conectado. Cada vez que extraemos un suceso en la línea 12 y se confirma que es una llamada entrante, generamos un suceso sustituto de llamada entrante en la línea 40. También se genera un suceso de terminación de llamada en la línea 35 si la llamada entrante es aceptada. Por tanto, la única forma de finalizar la rutina es si se configura `nextCall` para que no genere un suceso a partir de un determinado momento o, más probablemente, ejecutando la instrucción `break` de la línea 15.

Resumamos cómo se procesan los distintos sucesos. Si el suceso es una desconexión, incrementamos `availableOperators` en la línea 19 e imprimimos un mensaje en las líneas 20 y 21. Si el suceso es una llamada entrante, generamos una línea parcial de salida que deja constancia del intento y luego, si hay algún operador disponible, conectamos al usuario con él. Para hacer esto, decrementamos `availableOperators` en la línea 29, generamos un tiempo de conexión (utilizando una distribución de Poisson en lugar de una distribución uniforme) en la línea 30, imprimimos el resto de la salida en las líneas 31–32 y añadimos una desconexión al conjunto de sucesos (líneas 33 a 35). Alternativamente, si no hay operadores disponibles, proporcionamos el mensaje de que el teléfono está comunicando. En cualquier caso, se genera un suceso adicional de llamada entrante. La Figura 13.10 muestra el estado de la cola con prioridad después de cada `deleteMin` para las primeras etapas de la salida de ejemplo mostrada en la Figura 13.4. El instante en el que tiene lugar cada suceso se muestra en negrita y el número de operadores libres (si hay alguno) se muestra a la derecha de la cola con prioridad. (Observe que la duración de la llamada no se almacena en la práctica en un objeto `Event`; la incluimos en los casos apropiados para conseguir que la figura se comprenda mejor. Un símbolo ‘?’ para la duración de la llamada indica un suceso de llamada entrante que dará como resultado una señal indicativa de que el teléfono está ocupado; sin embargo, dicho resultado no se conoce en el momento en que se añade el suceso a la cola con prioridad.) La secuencia de pasos en la cola con prioridad es la siguiente.

1. Se inserta la primera solicitud `DIAL_IN`.
2. Despues de eliminar `DIAL_IN`, se atiende la solicitud dando lugar a un suceso `HANG_UP` y a una solicitud sustituta `DIAL_IN`.
3. Se procesa una solicitud `HANG_UP`.
4. Se procesa una solicitud `DIAL_IN`, dando como resultado una conexión. Debido a ello, se añade (tres veces) tanto un suceso `HANG_UP` como un suceso `DIAL_IN`.
5. Una solicitud `DIAL_IN` falla, se genera una solicitud `DIAL_IN` sustituta (tres veces).
6. Se procesa una solicitud `HANG_UP` (dos veces).
7. Una solicitud `DIAL_IN` tiene éxito, añadiéndose `HANG_UP` y `DIAL_IN`.

El método `runSim` ejecuta la simulación.

La terminación de una llamada incrementa `availableOperators`. La llamada entrante comprueba si hay un operador disponible, y en caso afirmativo, decrementa `availableOperators`.

```

1  /**
2  * Ejecutar la simulación hasta el instante stoppingTime.
3  * Imprimir la salida como en la Figura 13.4.
4  */
5  public void runSim( long stoppingTime )
6  {
7      Event e = null;
8      int howLong;
9
10     while( !eventSet.isEmpty( ) )
11     {
12         e = eventSet.remove( );
13
14         if( e.time > stoppingTime )
15             break;
16
17         if( e.what == Event.HANG_UP ) // HANG_UP
18         {
19             availableOperators++;
20             System.out.println( "User " + e.who +
21                                 " hangs up at time " + e.time );
22         }
23         else // DIAL_IN
24         {
25             System.out.print( "User " + e.who +
26                               " dials in at time " + e.time + " " );
27             if( availableOperators > 0 )
28             {
29                 availableOperators--;
30                 howLong = r.nextPoisson( avgCallLen );
31                 System.out.println( "and connects for " +
32                                     " + howLong + " minutes" );
33                 e.time += howLong;
34                 e.what = Event.HANG_UP;
35                 eventSet.add( e );
36             }
37             else
38                 System.out.println( "but gets busy signal" );
39
40             nextCall( freqOfCalls );
41         }
42     }
43 }

```

Figura 13.9 La rutina básica de simulación.

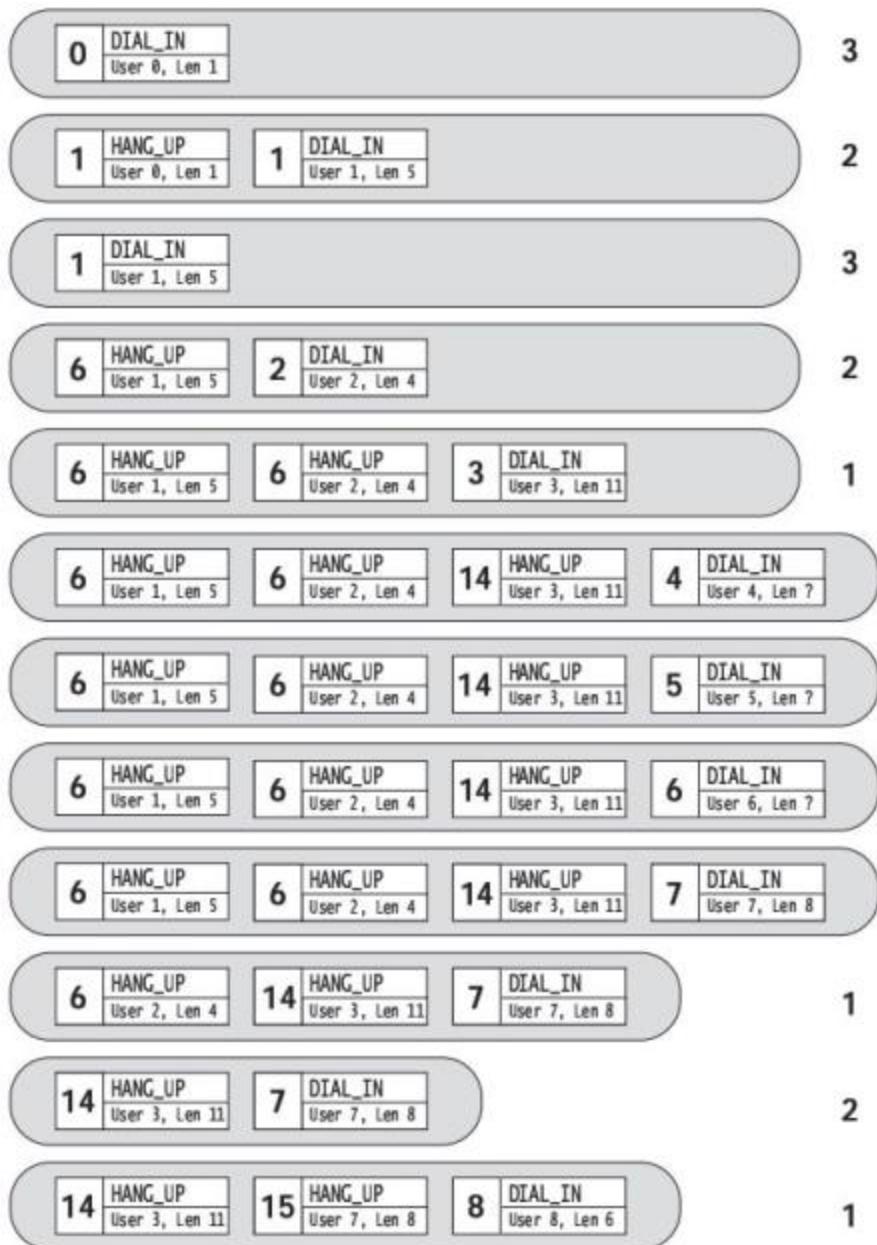


Figura 13.10 La cola con prioridad después de cada paso de la simulación del servicio de atención telefónica.

De nuevo, si Event fuera una clase base abstracta, esperaríamos que se definiera un procedimiento doEvent a todo lo largo de la jerarquía Event; entonces no necesitaríamos grandes cascadas de instrucciones if/else. Sin embargo, para acceder a la cola con prioridad, que se encuentra en la clase de simulación, necesitaríamos que Event almacenara una referencia a la clase CallSim de simulación como miembro de datos. La insertaríamos en el momento de la construcción.

```

1  /**
2  * Rutina main simple con propósitos de prueba.
3  */
4  public static void main( String [ ] args )
5  {
6      CallSim s = new CallSim( 3, 5.0, 1 );
7      s.runSim( 20 );
8  }

```

Figura 13.11 Una rutina `main` simple para probar la simulación.

La simulación utiliza un modelo muy pobre. Las distribuciones exponenciales negativas permitirían modelar con más precisión el tiempo transcurrido entre intentos de llamada y el tiempo total de conexión.

Para terminar de completar el ejemplo, en la Figura 13.11 se muestra una rutina `main` mínima (en el más estricto sentido). Observe que utilizar una distribución de Poisson para modelar el tiempo de conexión no resulta apropiado. Una mejor opción sería utilizar una distribución exponencial negativa (pero las razones para hacerlo quedan fuera del alcance de este libro). Además, tampoco es muy preciso suponer que transcurre un tiempo fijo entre intentos sucesivos de llamada. De nuevo, una distribución exponencial negativa sería un mejor modelo. Si cambiamos la simulación para emplear estas distribuciones, el reloj sería representado como un valor `double`. En el Ejercicio 13.8 le pediremos que implemente estos cambios.

## Resumen

La simulación es una importante área del campo de las Ciencias de la computación y su complejidad es mucho mayor de lo que podríamos analizar aquí. La precisión de una simulación está limitada por su modelo de aleatoriedad, por lo que se necesitan sólidos conocimientos de probabilidad, estadística y teoría de colas para que el modelador conozca qué tipos de distribuciones de probabilidad resulta razonable utilizar. La simulación es un área de aplicación importante para las técnicas de orientación a objetos.



## Conceptos clave

**problema de Josefo** Un juego en el que se va pasando de mano en mano repetidamente una patata caliente; cuando se termina de pasar, el jugador que tiene la patata es eliminado; el juego continúa y el último jugador que queda es el que gana. (498)

**simulación** Una aplicación importante de las computadoras, en la que se las emplea para emular el funcionamiento de un sistema real y recopilar estadísticas. (497)

**simulación dirigida por sucesos** Una simulación en la que se hace avanzar el tiempo actual hasta el siguiente suceso. (503)

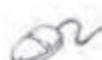
**simulación dirigida por tiempo discreto** Una simulación en la que cada unidad de tiempo se procesa de forma consecutiva. Resulta inapropiada si el intervalo entre sucesos consecutivos es muy grande. (503)

**tic** El quantum de tiempo en una simulación. (503)



## Errores comunes

1. El error más común en una simulación es utilizar un modelo inadecuado. La precisión de una simulación está limitada por la de su entrada aleatoria.



## Internet

Los dos ejemplos de este capítulo están disponibles en línea.

### **Josephus.java**

Contiene tanto implementaciones de `josephus` como una rutina `main` para probarlas.

### **CallSim.java**

Contiene el código para la simulación del servicio de atención telefónica.



## Ejercicios

### EN RESUMEN

- 13.1 Muestre el estado de la cola con prioridad después de cada una de las 10 primeras líneas de la simulación descrita en la Figura 13.4.
- 13.2 Si  $M = 1$ , ¿quién gana el juego de Josefo?
- 13.3 Muestre el funcionamiento del algoritmo de Josefo de la Figura 13.3 para el caso de siete personas con tres pases. Incluya el cálculo de `rank` y una imagen que contenga los elementos restantes después de cada iteración.
- 13.4 ¿Hay algún valor de  $M$  para el que el jugador 1 pueda ganar un juego de Josefo con 32 personas?

### EN TEORÍA

- 13.5 Sea  $J(N)$  el ganador de un juego de Josefo de  $N$  jugadores con  $M = 1$ . Demuestre que
  - a. Si  $N$  es par, entonces  $J(N) = 2 J(N/2) - 1$ .
  - b. Si  $N$  es impar y  $J(\lceil N/2 \rceil) \neq 1$ , entonces  $J(N) = 2 J(\lceil N/2 \rceil) - 3$ .
  - c. Si  $N$  es impar y  $J(\lceil N/2 \rceil) = 1$ , entonces  $J(N) = N$ .
- 13.6 Utilice los resultados del Ejercicio 13.5 para escribir un algoritmo que devuelva el ganador de un juego de Josefo de  $N$  jugadores con  $M = 1$ . ¿Cuál es el tiempo de ejecución de su algoritmo?
- 13.7 Sea  $N = 2^k$  para cualquier entero  $k$ . Demuestre que si  $M$  es 1, entonces el jugador 1 siempre gana el juego de Josefo.

### EN LA PRÁCTICA

- 13.8** Rehaga la simulación de modo que el reloj esté representado como un valor `double`, el tiempo entre intentos de conexión y el tiempo de conexión se modelen ambos mediante una distribución exponencial negativa.
- 13.9** Escriba un programa que resuelva la versión histórica del problema de Josefo. Proporcione los algoritmos tanto de la lista enlazada como del árbol de búsqueda.
- 13.10** Implemente el algoritmo de Josefo con una cola. Cada pase de la patata es una operación `dequeue`, seguida de una operación `enqueue`.
- 13.11** Suponga que implementamos el algoritmo de Josefo mostrado en la Figura 13.2 mediante un `TreeSet` en lugar de mediante una `LinkedList`. Si el cambio funcionara, ¿cuál sería el tiempo de ejecución?
- 13.12** Rehaga la simulación del servicio de atención telefónica de modo que `Event` sea una clase base abstracta y `DialInEvent` y `HangUpEvent` sean clases derivadas, utilizadas para modelar las llamadas entrantes y las terminaciones de llamada. La clase `Event` deberá almacenar una referencia a un objeto `CallSim` como miembro adicional de datos; esa referencia debe inicializarse durante la construcción. También debe proporcionar un método abstracto denominado `doEvent` que se implemente en las clases derivadas y que pueda ser llamado desde `runSim` para procesar el suceso.

### PROYECTOS DE PROGRAMACIÓN

- 13.13** Escriba de nuevo el algoritmo de Josefo mostrado en la Figura 13.3 para utilizar un *montículo mediano* (véase el Ejercicio 6.30). Utilice una implementación simple del montículo mediano; los elementos se mantienen ordenados. Compare el tiempo de ejecución de este algoritmo con el tiempo obtenido utilizando el árbol de búsqueda binaria.
- 13.14** Implemente el algoritmo de Josefo con árboles *splay* (véase el Capítulo 22) e inserción secuencial. (La clase de árboles *splay* está disponible en línea, pero necesitaría un método `findKth`.) Compare el rendimiento con el de la implementación del texto y con el de un algoritmo que utilice un método de tiempo lineal para construcción de un árbol equilibrado.
- 13.15** Escriba de nuevo la simulación del servicio de atención telefónica para recopilar estadísticas, en lugar de para imprimir cada suceso. A continuación, suponiendo que existen varios cientos de operadores y una simulación muy larga, compare la velocidad de la simulación con la de algunas otras colas con prioridad posibles (algunas de las cuales están disponibles en línea); en concreto, compare con las siguientes alternativas.
- Una representación de cola con prioridad asintóticamente ineficiente, descrita en el Ejercicio 6.25.
  - Árboles *splay* (véase el Capítulo 22).
- 13.16** Suponga que el servicio de atención telefónica tiene instalado un sistema que pone en cola las llamadas telefónicas cuando todos los operadores están ocupados. Escriba de nuevo la rutina de simulación para permitir el uso de colas de distintos tamaños. Tenga en cuenta la posibilidad de una cola “infinita”.

## Grafos y caminos

En este capítulo vamos a examinar los *grafos* y a mostrar cómo resolver un tipo particular de problema –el del cálculo de los caminos más cortos. El cálculo del camino más corto es una aplicación fundamental en el campo de las Ciencias de la computación, porque hay muchas situaciones que se pueden modelar mediante un grafo. Como ejemplos podríamos citar la determinación de las rutas más rápidas para un sistema de transporte público, o el enrutamiento de correo electrónico a través de una red de computadoras. Examinaremos variantes de problemas de cálculo del camino más corto que dependen de cuál sea la interpretación del término *más corto* y de las propiedades del grafo. Los problemas del camino más corto son interesantes, porque aunque los algoritmos son bastante simples, resultan muy lentos para grafos de gran tamaño, a menos que se preste una cuidadosa atención a la hora de elegir las estructuras de datos.

En este capítulo veremos

- Definiciones formales de un grafo y de sus componentes.
- Las estructuras de datos utilizadas para representar un grafo.
- Algoritmos para resolver diversas variantes del problema del camino más corto, con implementaciones completas en Java.

### 14.1 Definiciones

Un grafo está compuesto de un conjunto de vértices y de un conjunto de aristas que conectan los vértices. Es decir,  $G = (V, E)$ , donde  $V$  es el conjunto de vértices y  $E$  es el conjunto de aristas. Cada arista es una pareja  $(v, w)$ , donde  $v, w \in V$ . Los vértices en ocasiones se denominan *nodos* y las aristas a veces se denominan *arcos*. Si el par correspondiente a una arista está ordenado, decimos que el grafo es un *grafo dirigido*. Los grafos dirigidos en ocasiones se denominan *digraphos*. En un digrafo, el vértice  $w$  es adyacente al vértice  $v$  si y solo si  $(v, w) \in E$ . A veces, una arista tiene un tercer componente, llamado *coste de la arista* (o *peso*) que mide el coste de recorrer la arista. En este capítulo, todos los grafos son dirigidos.

El grafo mostrado en la Figura 14.1 tiene siete vértices.

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$$

Un grafo consta de un conjunto de vértices y de un conjunto de aristas que conectan dichos vértices. Si las parejas que definen las aristas están ordenadas, el grafo se denomina *grafo dirigido*.

El vértice  $w$  es adyacente al vértice  $v$  si existe una arista que va de  $v$  a  $w$ .

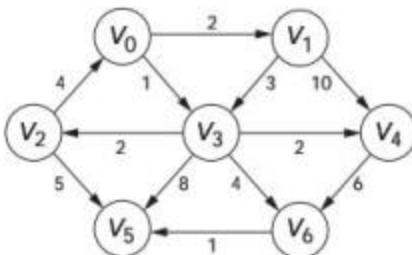


Figura 14.1 Un grafo dirigido.

y doce aristas

$$E = \left\{ \begin{array}{l} (V_0, V_1, 2), \quad (V_0, V_3, 1), \quad (V_1, V_3, 3), \quad (V_1, V_4, 10) \\ (V_3, V_4, 2) \quad (V_3, V_6, 4), \quad (V_3, V_5, 8), \quad (V_3, V_2, 2) \\ (V_2, V_0, 4), \quad (V_2, V_5, 5) \quad (V_4, V_6, 6), \quad (V_6, V_5, 1) \end{array} \right\}$$

Los siguientes vértices son adyacentes a  $V_3$ :  $V_2$ ,  $V_4$ ,  $V_5$  y  $V_6$ . Observe que  $V_0$  y  $V_1$  no son adyacentes a  $V_3$ . Para este grafo,  $|V| = 7$  y  $|E| = 12$ ; aquí,  $|S|$  representa el tamaño del conjunto  $S$ .

Un *camino* (a veces denominado también *ruta*) en un grafo es una secuencia de vértices conectados mediante aristas. En otras palabras, una secuencia de vértices  $w_1, w_2, \dots, w_N$  tal que  $(w_i, w_{i+1}) \in E$  para  $1 \leq i \leq N$ . La longitud de un camino es el número de aristas que forman el camino –concretamente,  $N - 1$ – denominándose también *longitud no ponderada de un camino*. La *longitud ponderada de un camino* es la suma de los costes de las aristas de un camino. Por ejemplo,  $V_0, V_3, V_5$  definen un camino que va del vértice  $V_0$  a  $V_5$ . La longitud del camino es de dos aristas –la ruta más corta entre  $V_0$  y  $V_5$ , mientras que la longitud ponderada del camino es igual a 9. Sin embargo, si el coste es importante, el camino ponderado más corto entre esos vértices tiene un coste igual a 6 y sería  $V_0, V_3, V_6, V_5$ . Es perfectamente posible que exista un camino que vaya desde un vértice hasta sí mismo. Si dicho camino no contiene ninguna arista, la longitud del camino es igual a 0, lo que es una forma bastante conveniente de definir un caso que de otro modo sería especial. Un *camino simple* es un camino en el que todos los vértices son distintos, salvo por la excepción de que el primer y el último vértices pueden coincidir.

Un *ciclo* en un grafo dirigido es un camino que comienza y termina en el mismo vértice y que contiene al menos una arista. Es decir, tiene una longitud de al menos 1, de modo que  $w_1 = w_N$ ; este ciclo será simple si el camino es simple. Un *grafo dirigido acíclico* (DAG, *Directed Acyclic Graph*) es un tipo de grafo dirigido que no tiene ningún ciclo.

Un ejemplo de situación de la vida real que podría modelarse mediante un grafo es el sistema aeroportuario. Cada aeropuerto es un vértice. Si existe un

Un camino es una secuencia de vértices conectados mediante aristas.

La longitud no ponderada de un camino mide el número de aristas que forman el camino.

La longitud ponderada de un camino es la suma de los costes de las aristas que forman el camino.

Un ciclo en un grafo dirigido es un camino que comienza y termina en el mismo vértice y que contiene al menos una arista.

Un grafo dirigido acíclico no tiene ningún ciclo. Dichos grafos constituyen una importante subclase de grafos.

vuelo directo entre dos aeropuertos, habrá dos vértices conectados por una arista. La arista podría tener un peso que represente el tiempo, la distancia o el coste del vuelo. En un grafo no dirigido, la existencia de una arista  $(v, w)$  implicaría que existe otra arista  $(w, v)$ . Sin embargo, el coste de cada una de esas dos aristas podría ser distinto, porque volar en direcciones diferentes podría requerir más tiempo (dependiendo de los vientos prevalentes) o costar más dinero (dependiendo de los impuestos locales). Por ello, usamos un grafo dirigido en el que se dibujan ambas aristas, posiblemente con diferentes pesos. Naturalmente, queríamos poder determinar de manera rápida cuál es el mejor vuelo posible entre dos aeropuertos determinados; aquí, la palabra *mejor* podría significar el camino con el menor número de aristas o aquel que tenga el mínimo para alguna de las medidas de peso existentes (distancia, coste, etc.).

Un segundo ejemplo de situación de la vida real que podría modelarse mediante un grafo es el enrutamiento del correo electrónico a través de redes de computadoras. Los vértices representan a las computadoras, las aristas representan los enlaces entre parejas de computadoras y los costes de las aristas representan los costes de comunicación (factura telefónica por megabyte), el retardo (segundos por megabyte) o una combinación de estos y otros factores.

Para la mayoría de los grafos, lo más probable es que exista, como máximo, una arista que vaya desde cualquier vértice  $v$  hasta cualquier otro vértice  $w$  (permitiéndose que exista una arista en cada una de las direcciones entre  $v$  y  $w$ ). En consecuencia,  $|E| \leq |V|^2$ . Cuando la mayoría de las aristas están presentes, tenemos que  $|E| = \Theta(|V|^2)$ . Dicho tipo de grafo se considera un *grafo denso*; es decir, que tiene un gran número de aristas y es generalmente cuadrático.

Un grafo es *denso* si el número de aristas es grande (generalmente cuadrático). Los grafos típicos no son densos, sino dispersos.

Sin embargo, en la mayoría de las aplicaciones, la norma es que exista un *grafo disperso*. Por ejemplo, en el modelo aeroportuario, no cabe esperar que existan vuelos directos entre cada pareja de aeropuertos. En lugar de ello, habrá unos pocos aeropuertos muy bien conectados, y la mayor parte de los otros tendrán muy pocos vuelos. En un sistema complejo de transporte público en el que haya trenes y autobuses, para cualquier estación dada solo habrá unas cuantas estaciones que sean directamente alcanzables y que tengan por tanto asociada una arista. Asimismo, en una red de computadoras, la mayoría de las máquinas están conectadas únicamente a otras pocas computadoras locales. Por tanto, en la mayoría de los casos, el grafo es relativamente disperso, cumpliéndose que  $|E| = \Theta(|V|)$  o quizás algo más (no existe una definición estándar del término disperso). Los algoritmos que desarrollaremos, por tanto, deberán ser eficientes para grafos dispersos.

### 14.1.1 Representación

Lo primero que tenemos que considerar es cómo representar un grafo internamente. Suponga que los vértices están numerados secuencialmente a partir de 0, como sugiere el grafo mostrado en la Figura 14.1. Una forma simple de representar un grafo consiste en utilizar una matriz bidimensional denominada *matriz de adyacencia*. Para cada arista  $(v, w)$ , hacemos que  $a[v][w]$  sea igual al coste de la arista; las aristas no existentes pueden inicializarse con un valor lógico **INFINITY**, para representar un coste infinito. La inicialización del grafo parece requerir que la matriz de adyacencia completa se inicialice con el valor **INFINITY**. Después, a medida que vamos encontrando aristas, configuraremos el valor de la entrada apropiada. En este escenario, la

Una matriz de adyacencia representa un grafo y utiliza un espacio cuadrático.

inicialización tarda un tiempo  $O(|V|^2)$ . Aunque se puede evitar el coste cuadrático de inicialización (véase el Ejercicio 14.7), el coste en términos de espacio seguirá siendo  $O(|V|^2)$ , lo cual es adecuado para grafos densos pero completamente inaceptable para grafos dispersos.

Una lista de adyacencia representa un grafo, utilizando un espacio lineal.

Puesto que cada arista aparece como un nodo de una lista, el número de nodos de lista es igual al número de aristas. En consecuencia, se emplea un espacio  $O(|E|)$  para almacenar los nodos de las listas. Tenemos  $|V|$  listas, por lo que también se necesita un espacio  $O(|V|)$  adicional. Si asumimos que todo vértice tiene al menos una arista asociada, el número de aristas será como mínimo  $\lceil |V|/2 \rceil$ . Por tanto, podemos despreciar todos los términos  $O(|V|)$  cuando haya un término  $O(|E|)$ . En consecuencia, diremos que el requisito de espacio es  $O(|E|)$ , es decir, que se requiere un espacio lineal que depende del tamaño del grafo.

Las listas de adyacencia se pueden construir en un tiempo lineal a partir de una lista de aristas.

La lista de adyacencia se puede construir en un tiempo lineal a partir de una lista de aristas. Comenzamos haciendo que todas las listas estén vacías. Cada vez que nos encontramos con una arista  $(v, w, c_{v,w})$ , añadimos una entrada compuesta por  $w$  y el coste  $c_{v,w}$  a la lista de adyacencia de  $v$ . La inserción se puede hacer en cualquier lugar; si realizamos la inserción al principio de la lista, podemos hacerlo en un tiempo constante. Cada arista se puede insertar en un tiempo constante, por lo que toda la estructura de la lista de adyacencia se puede construir en un tiempo lineal. Observe que, a la hora de insertar una arista, no comprobamos si ya se encuentra presente. Esto no se puede hacer en un tiempo constante (utilizando una simple lista enlazada), y si hicieramos la comprobación, destruiríamos la cota de tiempo lineal para la construcción. En la mayoría de los casos no tiene ninguna importancia ignorar esta comprobación. Si hay dos o más aristas con costes distintos que conectan una pareja de vértices, cualquier algoritmo de determinación del camino corto seleccionará la arista de menor coste sin necesidad de recurrir a ningún tipo de procesamiento especial. Observe también que se pueden utilizar `ArrayList` en lugar de listas enlazadas, con lo que la operación `add` de tiempo constante sustituiría a la operación de insertar un elemento al principio de la lista enlazada.

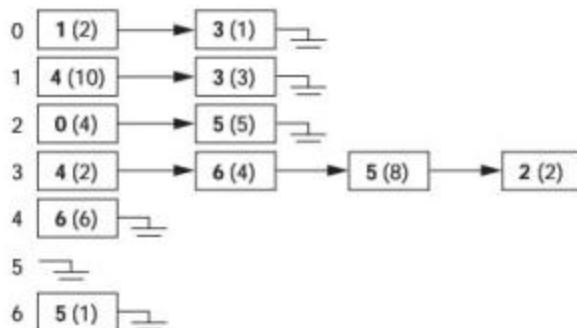


Figura 14.2 Representación mediante lista de adyacencia del grafo mostrado en la Figura 14.1; los nodos de la lista / representan los vértices adyacentes a / y el coste de la arista de conexión.

En la mayoría de las aplicaciones de la vida real, los vértices no tienen números sino nombres, que son desconocidos en tiempo de compilación. En consecuencia, debemos proporcionar una manera de transformar los nombres en números. La forma más fácil de hacerlo es proporcionar un *mapa* que nos permita asignar a cada nombre de vértice un número interno que vaya de 0 a  $|V| - 1$  (el número de vértices se determina a medida que se ejecuta el programa). Los números internos se asignan a medida que se lee el grafo. El primer número asignado es 0. Según se va añadiendo cada arista, comprobamos si se ha asignado ya un número a uno de los dos vértices, examinando para ello el mapa. Si ya se ha asignado un número interno, lo utilizamos. En caso contrario, asignamos al vértice el siguiente número disponible e insertamos en el mapa el nombre y el número del vértice. Con esta transformación, todos los algoritmos que trabajan con grafos únicamente utilizan los números internos. Al final, tendremos que imprimir los nombres reales de los vértices, no los números internos, por lo que necesitamos anotar el nombre de vértice correspondiente a cada número interno. Una forma de hacer esto es manteniendo una cadena de caracteres para cada vértice. Emplearemos esta técnica para implementar una clase *Graph*. La clase y los algoritmos de determinación del camino más corto requieren varias estructuras de datos –en concreto, una lista, una cola, un mapa y una cola con prioridad. Las directivas `import` se muestran en la Figura 14.3. La cola (implementada con una lista enlazada) y la cola con prioridad se utilizan en diversos cálculos del camino más corto. La lista de adyacencia se representa mediante un *LinkedList*. También se utiliza un *HashMap* para representar el grafo.

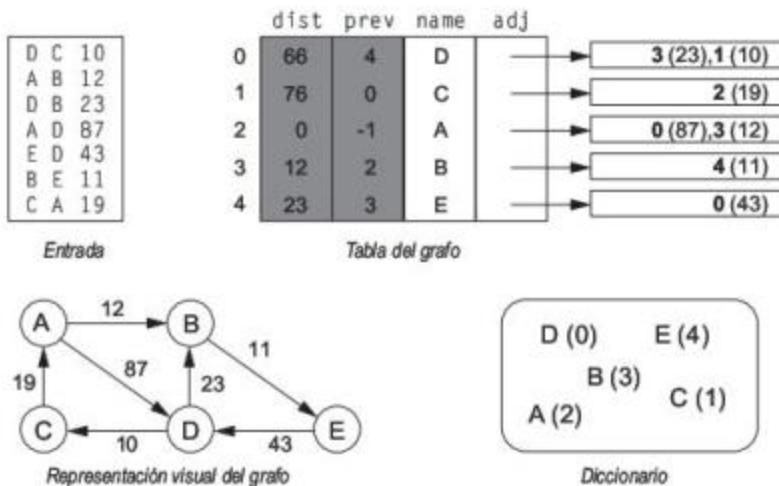
Al escribir una implementación Java real, no necesitamos los números internos de vértice. En lugar de ello, cada vértice se almacena en un objeto *Vertex* y en lugar de usar un número, podemos emplear como número una referencia al objeto *Vertex* (que es un número que identifica únicamente al objeto). Sin embargo, a la hora de describir los algoritmos, el asumir que los vértices están numerados suele ser bastante cómodo, por lo que en ocasiones lo haremos.

Antes de mostrar el esqueleto de la clase *Graph*, examinemos las Figuras 14.4 y 14.5, que indican cómo vamos a representar nuestro grafo. La Figura 14.4 muestra la representación en la que usamos

```
1 import java.io.FileReader;
2 import java.io.InputStreamReader;
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.util.StringTokenizer;
6
7 import java.util.Collection;
8 import java.util.List;
9 import java.util.LinkedList;
10 import java.util.Map;
11 import java.util.HashMap;
12 import java.util.Iterator;
13 import java.util.Queue;
14 import java.util.PriorityQueue;
15 import java.util.NoSuchElementException;
```

Se puede utilizar un mapa para asignar a cada nombre de vértice su número interno.

Figura 14.3 Las directivas `import` para la clase *Graph*.



**Figura 14.4** Un escenario abstracto de las estructuras de datos utilizadas en el cálculo del camino más corto, con un grafo de entrada tomado de un archivo. El camino ponderado más corto desde A a C es el que va de A a B, luego a E, después a D y luego a C (el coste es 76).

números internos. La Figura 14.5 sustituye los números internos por variables `Vertex`, como hacemos en el código. Aunque esto simplifica el código, complica enormemente el dibujo. Puesto que las dos figuras representan entradas idénticas, la Figura 14.4 puede utilizarse para tratar de entender las complejidades de la Figura 14.5.

Como se indica en la parte etiquetada como *Entrada*, podemos esperar que el usuario proporcione una lista de aristas, estando cada arista en una línea diferente. Al principio del algoritmo, no sabemos los nombres de los vértices, ni cuántos vértices hay, ni cuántas aristas existen. Utilizamos dos estructuras de datos básicas para representar el grafo. Como hemos dicho en el párrafo anterior, para cada vértice mantenemos un objeto `Vertex` que almacena una cierta información. Describiremos los detalles de `Vertex` (en particular, cómo interactúan entre sí los distintos objetos `Vertex`) al final.

Como hemos mencionado anteriormente, la primera de las estructuras de datos principales es un mapa que nos permite encontrar, para cada nombre de vértice, cuál es el objeto `Vertex` que lo representa. Este mapa se muestra en la Figura 14.5 como `vertexMap` (la Figura 14.4 asigna al nombre un valor `int` en el componente etiquetado como *Diccionario*).

La segunda de las estructuras de datos principales es el objeto `Vertex` que almacena información acerca de todos los vértices. De especial interés es cómo interactúa cada uno de estos objetos con otros objetos `Vertex`. Las Figuras 14.4 y 14.5 muestran que un objeto `Vertex` mantiene cuatro piezas de información para cada vértice.

- `name`: el nombre correspondiente a este vértice se define en el momento de introducir el vértice en el mapa y nunca varía. Ninguno de los algoritmos de determinación del camino más corto examina este miembro de datos. Se utiliza solo para imprimir un camino al final.
- `adj`: esta lista de vértices adyacentes se establece en el momento de leer el grafo. Ninguno de los algoritmos de determinación del camino más corto modifica la lista. En la representación abstracta de la Figura 14.4 se muestra que se trata de una lista de objetos `Edge`, cada uno de los cuales contiene un número interno de vértice y un coste de la arista. En la práctica, como

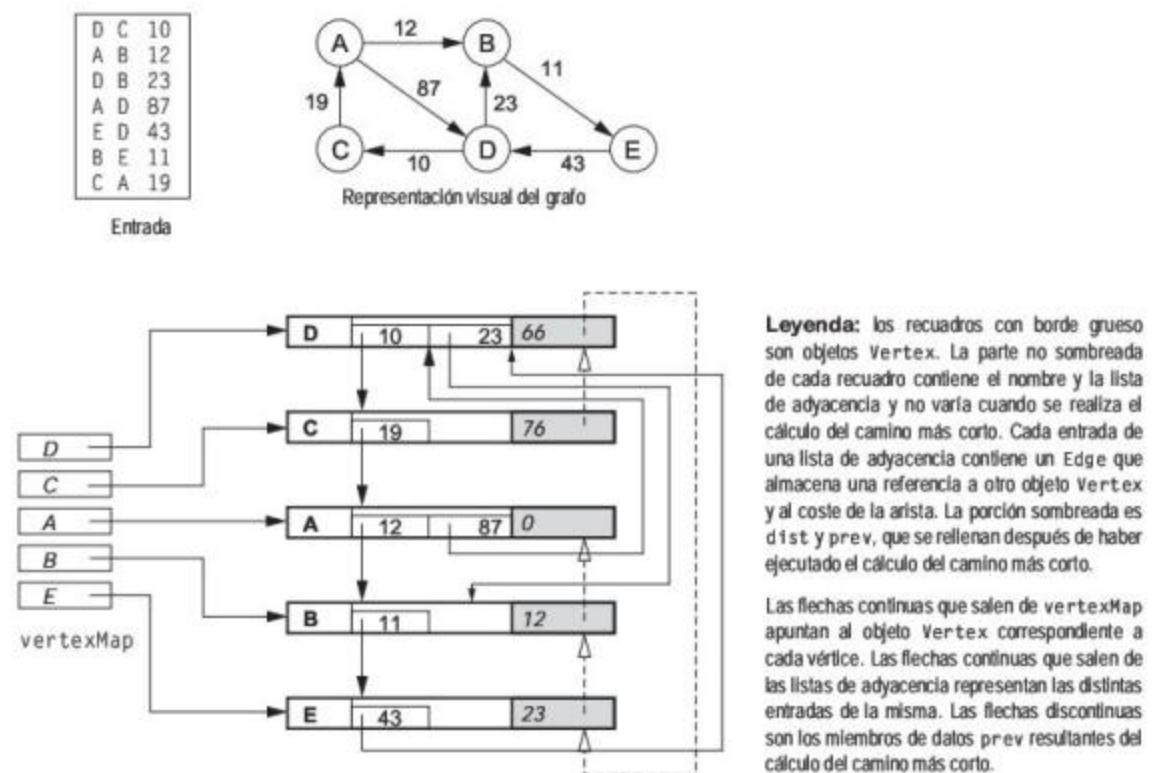


Figura 14.5 Estructuras de datos utilizadas en el cálculo del camino más corto, con un grafo de entrada tomado de un archivo; el camino ponderado más corto de A a C es el que va de A hasta B, luego a E, después a D y luego a C (el coste es 76).

muestra la Figura 14.5, cada objeto Edge contiene una referencia a un objeto Vertex y un coste de arista, y la lista se almacena en realidad utilizando un ArrayList o una LinkedList.

- **dist:** la longitud del camino más corto (ponderado o no ponderado, dependiendo del algoritmo) que va desde el vértice inicial hasta este vértice, tal como lo haya calculado el algoritmo de determinación del camino más corto.
- **prev:** el vértice anterior dentro del camino más corto que va hasta este vértice, que en la representación abstracta (Figura 14.4) es un valor int pero que en realidad (en el código y la Figura 14.5) es una referencia a un objeto Vertex.

Para ser más específicos, en las Figuras 14.4 y 14.5 los elementos no sombreados no se ven modificados por ninguno de los cálculos del camino más corto. Representan el grafo de entrada y no varían, a menos que varíe el propio grafo (quizá por adición o borrado de aristas en algún momento posterior). Los elementos sombreados se calculan mediante los algoritmos de determinación del camino más corto. Antes del cálculo podemos asumir que no están inicializados.<sup>1</sup>

<sup>1</sup> La información calculada (sombreada) podría separarse en una clase distinta, manteniéndose en Vertex una referencia a ella; eso haría el código más reutilizable, pero también más complejo.

Los algoritmos del camino más corto son *algoritmos de un único origen*, que calculan los caminos más cortos desde un cierto punto de partida hasta todos los vértices.

El miembro de datos `prev` puede utilizarse para extraer el camino real.

El elemento de una lista de adyacencia es una referencia a un objeto `Vertex` del vértice adyacente, junto con el coste de la arista.

Los algoritmos de determinación del camino más corto son todos ellos *algoritmos de un único origen*, que comienzan en un cierto punto inicial y calculan los caminos más cortos desde ese punto a todos los vértices. En este ejemplo, el punto inicial es A y consultando el mapa podemos encontrar su objeto `Vertex` correspondiente. Observe que el algoritmo del camino más corto declara que el camino más corto hasta A es 0.

El miembro de datos `prev` nos permite imprimir el camino más corto, en lugar de simplemente su longitud. Por ejemplo, consultando el objeto `Vertex` para C, vemos que el camino más corto desde el vértice inicial hasta C tiene un coste total de 76. Obviamente, el último vértice de este camino es C. El vértice anterior a C en este camino es D, antes de D está E, antes de E está B y antes de B está A –el vértice inicial. Por tanto, recorriendo el camino hacia atrás, gracias al miembro de datos `prev`, podemos construir el camino más corto. Aunque este procedimiento nos da el camino en orden inverso, el ordenarlo al revés es bastante sencillo. En el resto de esta sección vamos a describir cómo se construyen las partes no sombreadas de todos los objetos `Vertex` y vamos a proporcionar un método que imprime el camino más corto, asumiendo que los miembros de datos `dist` y `prev` ya han sido calculados. Analizaremos individualmente los algoritmos utilizados para llenar los datos correspondientes a los caminos más cortos.

La Figura 14.6 muestra la clase `Edge` que representa el elemento básico colocado en la lista de adyacencia. `Edge` consta de una referencia a un vértice `Vertex` y del coste de la arista. La clase `Vertex` se muestra en la Figura 14.7. Se proporciona un miembro adicional denominado `scratch`, el cual tiene diferentes usos en los distintos algoritmos. Todo lo demás procede de nuestra descripción anterior. El método `reset` se utiliza para inicializar los miembros de datos (sombreados) calculados por los algoritmos del camino más corto; se invoca cuando se reinicializa un cálculo del camino más corto.

Ahora estamos preparados para examinar el esqueleto de la clase `Graph`, que se muestra en la Figura 14.8. El campo `vertexMap` almacena el mapa. El resto de la clase proporciona métodos que

```

1 // Representa una arista del grafo.
2 class Edge
3 {
4     public Vertex dest; // Segundo vértice de la arista
5     public double cost; // Coste de la arista
6
7     public Edge( Vertex d, double c )
8     {
9         dest = d;
10        cost = c;
11    }
12 }
```

Figura 14.6 El elemento básico almacenado en una lista de adyacencia.

```
1 // Representa un vértice del grafo.
2 class Vertex
3 {
4     public String name;      // Nombre del vértice
5     public List<Edge> adj; // Vértices adyacentes
6     public double dist;     // Coste
7     public Vertex prev;    // Vértice anterior en el camino más corto
8     public int scratch;    // Variable extra usada por el algoritmo
9
10    public Vertex( String nm )
11        { name = nm; adj = new LinkedList<Edge>(); reset( ); }
12
13    public void reset( )
14    { dist = Graph.INFINITY; prev = null; pos = null; scratch = 0; }
15 }
```

Figura 14.7 La clase `Vertex` almacena información de cada vértice.

llevan a cabo la inicialización, añaden los vértices y aristas, imprimen el camino más corto y realizan diversos cálculos del camino más corto. Analizaremos cada rutina cuando examinemos su implementación.

En primer lugar, consideremos el constructor. El constructor predeterminado crea un mapa vacío inicializando el campo correspondiente; eso funciona perfectamente, por lo que aceptamos ese comportamiento predeterminado.

Ahora podemos examinar los métodos principales. El método `getVertex` se muestra en la Figura 14.9. Consultamos el mapa para obtener la entrada `Vertex` correspondiente. Si el objeto `Vertex` no existe, creamos un nuevo `Vertex` y actualizamos el mapa. El método `addEdge`, mostrado en la Figura 14.10, es corto. Obtenemos las entradas `Vertex` correspondientes y luego actualizamos una lista de adyacencia.

Los miembros que terminarán siendo calculados mediante el algoritmo del camino más corto se inicializan mediante la rutina `clearAll`, mostrada en la Figura 14.11. La siguiente rutina, `printPath`, imprime un camino más corto después de realizar el cálculo. Como hemos mencionado anteriormente, podemos utilizar el miembro `prev` para recorrer hacia atrás el camino, pero eso hace que obtengamos el camino en sentido inverso. Este orden no es un problema si lo que hacemos es emplear la recursión: los vértices que componen el camino hasta `dest` son los mismos que los que componen el camino hasta el vértice anterior a `dest` (dentro del camino), seguidos por `dest`. Esta estrategia se traduce directamente en la breve rutina recursiva mostrada en la Figura 14.12, asumiendo, por supuesto, que exista en realidad un camino. La rutina `printPath`, mostrada en la Figura 14.13, realiza primero esta comprobación y luego imprime un mensaje si el camino no existe. En caso contrario, invoca a la rutina recursiva e imprime el coste del camino.

Las aristas se añaden mediante inserciones en la lista de adyacencia apropiada.

La rutina `clearAll` borra los miembros de datos, para que puedan comenzar su procesamiento los algoritmos del camino más corto.

La rutina `printPath` imprime el camino más corto después de que el algoritmo se ha ejecutado.

```

1 // Clase Graph: evalúa los caminos más cortos.
2 //
3 // CONSTRUCCIÓN: sin parámetros.
4 //
5 // *****OPERACIONES PÚBLICAS*****
6 // void addEdge( String v, String w, double cvw )
7 //           --> Añadir arista adicional
8 // void printPath( String w ) --> Imprimir camino después de ejecutar alg.
9 // void unweighted( String s ) --> Origen Único no ponderado
10 // void dijkstra( String s ) --> Origen Único ponderado
11 // void negative( String s ) --> Origen Único ponderación negativa
12 // void acyclic( String s ) --> Origen Único acíclico
13 // *****ERRORES*****
14 // Se hacen algunas comprobaciones de errores para asegurarse de que el
15 // grafo es correcto y satisface las propiedades que cada algoritmo necesita.
16 // Se generan excepciones si se detectan errores.
17
18 public class Graph
19 {
20     public static final double INFINITY = Double.MAX_VALUE;
21
22     public void addEdge( String sourceName, String destName, double cost )
23         { /* Figura 14.10 */ }
24     public void printPath( String destName )
25         { /* Figura 14.13 */ }
26     public void unweighted( String startName )
27         { /* Figura 14.22 */ }
28     public void dijkstral( String startName )
29         { /* Figura 14.27 */ }
30     public void negative( String startName )
31         { /* Figura 14.29 */ }
32     public void acyclic( String startName )
33         { /* Figura 14.32 */ }
34
35     private Vertex getVertex( String vertexName )
36         { /* Figura 14.9 */ }
37     private void printPath( Vertex dest )
38         { /* Figura 14.12 */ }
39     private void clearAll( )
40         { /* Figura 14.11 */ }
41

```

*Continúa***Figura 14.8** El esqueleto de la clase Graph.

```

42     private Map<String,Vertex> vertexMap = new HashMap<String,Vertex>( );
43 }
44
45 // Utilizado para indicar las violaciones de las precondiciones para
46 // los diversos algoritmos del camino más corto.
47 class GraphException extends RuntimeException
48 {
49     public GraphException( String name )
50     { super( name ); }
51 }

```

**Figura 14.8 (Continuación)**

```

1  /**
2   * Si vertexName no está presente, añadirlo a vertexMap.
3   * En cualquiera de los casos, devolver el objeto Vertex.
4   */
5  private Vertex getVertex( String vertexName )
6  {
7      Vertex v = vertexMap.get( vertexName );
8      if( v == null )
9      {
10         v = new Vertex( vertexName );
11         vertexMap.put( vertexName, v );
12     }
13     return v;
14 }

```

**Figura 14.9** La rutina `getVertex` devuelve el objeto `Vertex` representado por `vertexName`, creando el objeto en caso necesario.

```

1  /**
2   * Añadir una nueva arista al grafo.
3   */
4  public void addEdge( String sourceName, String destName, double cost )
5  {
6      Vertex v = getVertex( sourceName );
7      Vertex w = getVertex( destName );
8      v.adj.add( new Edge( w, cost ) );
9 }

```

**Figura 14.10** Añade una arista al grafo.

```

1  /**
2   * Inicializa la información de salida de los vértices antes de
3   * ejecutar ningún algoritmo del camino más corto.
4   */
5  private void clearAll( )
6  {
7      for( Vertex v : vertexMap.values( ) )
8          v.reset( );
9  }

```

**Figura 14.11** Rutina privada para inicializar los miembros en los que los algoritmos de determinación del camino más corto almacenan sus resultados.

```

1 /**
2  * Rutina recursiva para imprimir el camino más corto hasta dest,
3  * después de ejecutar el algoritmo del camino más corto.
4  * Se sabe que ese camino existe.
5  */
6  private void printPath( Vertex dest )
7  {
8      if( dest.prev != null )
9      {
10         printPath( dest.prev );
11         System.out.print( " to " );
12     }
13     System.out.print( dest.name );
14 }

```

**Figura 14.12** Una rutina recursiva para imprimir el camino más corto.

La clase Graph es fácil de utilizar.

Proporcionamos un programa de ejemplo simple que lee un grafo de un archivo de entrada, pide que se identifique un vértice inicial y un vértice de destino y luego ejecuta uno de los algoritmos de determinación del camino más corto. La Figura 14.14 ilustra que para construir el objeto *Graph*, se lee repetidamente una línea de entrada, se asigna la línea a un objeto *StringTokenizer*, se analiza sintácticamente dicha línea y se invoca *addEdge*. Utilizar un *StringTokenizer* nos permite verificar que toda línea tiene los tres componentes que definen a una arista.

Una vez leído el grafo, invocamos repetidamente *processRequest*, mostrado en la Figura 14.15. Esta versión solicita un vértice inicial y otro final, y luego invoca a uno de los algoritmos de determinación del camino más corto. Este algoritmo genera una excepción *GraphException* si, por ejemplo, se le pide encontrar el camino entre vértices que no forman parte del grafo. Así, *processRequest* atrapa cualquier excepción *GraphException* que pueda generarse e imprime un mensaje de error apropiado.

```
1  /**
2   * Rutina de prep. para gestionar los vértices inalcanzables e imprimir
3   * el coste total. Invoca la rutina recursiva para imprimir el camino más
4   * corto a destNode después de ejecutar el algoritmo del camino más corto.
5   */
6  public void printPath( String destName )
7  {
8      Vertex w = vertexMap.get( destName );
9      if( w == null )
10         throw new NoSuchElementException();
11     else if( w.dist == INFINITY )
12         System.out.println( destName + " is unreachable" );
13     else
14     {
15         System.out.print( "(Cost is: " + w.dist + ") " );
16         printPath( w );
17         System.out.println();
18     }
19 }
```

Figura 14.13 Una rutina para imprimir el camino más corto consultando la tabla del grafo (véase la Figura 14.5).

## 14.2 Problema del camino más corto no ponderado

Recuerde que la longitud de camino no ponderada mide el número de aristas. En esta sección, vamos a considerar el problema de determinar la longitud de camino no ponderada más corta entre vértices especificados.

La longitud de camino no ponderada mide el número de aristas de un camino.

### Problema del camino más corto no ponderado con un único origen

Encontrar el camino más corto (medido según el número de aristas) entre un vértice designado  $S$  y todos los demás vértices.

El problema del camino más corto no ponderado es un caso especial del problema del camino más corto ponderado (en este caso especial, todos los pesos son iguales a 1). Por tanto, debería tener una solución más eficiente que el problema del camino más corto ponderado. Eso efectivamente es así, aunque los algoritmos para todos los problemas de cálculo de caminos son similares.

Todas las variantes del problema del camino más corto tienen soluciones similares.

### 14.2.1 Teoría

Para resolver el problema del camino más corto no ponderado utilizamos el grafo mostrado anteriormente en la Figura 14.1, empleando  $V_2$  como vértice inicial  $S$ . Por ahora, lo que nos interesa

```

1  /**
2  * Una rutina main que:
3  * 1. Lee un archivo (suministrado como un parámetro de la línea de
4  * comandos) que contiene aristas.
5  * 2. Construye el grafo;
6  * 3. Pide repetidamente dos vértices y ejecuta
7  * el algoritmo de determinación del camino más corto.
8  * El archivo de datos es una secuencia de líneas con el formato
9  * origen destino coste
10 */
11 public static void main( String [ ] args )
12 {
13     Graph g = new Graph( );
14     try
15     {
16         FileReader fin = new FileReader( args[0] );
17         Scanner graphFile = new Scanner( fin );
18
19         // Leer las aristas e insertar
20         String line;
21         while( graphFile.hasNextLine( ) )
22         {
23             line = graphFile.nextLine( );
24             StringTokenizer st = new StringTokenizer( line );
25
26             try
27             {
28                 if( st.countTokens( ) != 3 )
29                 {
30                     System.err.println( "Skipping bad line " + line );
31                     continue;
32                 }
33                 String source = st.nextToken( );
34                 String dest = st.nextToken( );
35                 int cost = Integer.parseInt( st.nextToken( ) );
36                 g.addEdge( source, dest, cost );
37             }
38             catch( NumberFormatException e )
39             { System.err.println( "Skipping bad line " + line ); }
40         }
41     }
42     catch( IOException e )
43     { System.err.println( e ); }
44

```

*Continúa***Figura 14.14** Una rutina main simple.

```
45     System.out.println( "File read..." );
46
47     Scanner in = new Scanner( System.in );
48     while( processRequest( in, g ) )
49     ;
50 }
```

Figura 14.14 (Continuación)

```
1  /**
2  * Procesar solicitud; devolver false si estamos en el final de archivo.
3  */
4 public static boolean processRequest( Scanner in, Graph g )
5 {
6     try
7     {
8         System.out.print( "Enter start node:" );
9         String startName = in.nextLine( );
10
11        System.out.print( "Enter destination node:" );
12        String destName = in.nextLine( );
13
14        System.out.print( "Enter algorithm (u, d, n, a): " );
15        String alg = in.nextLine( );
16
17        if( alg.equals( "u" ) )
18            g.unweighted( startName );
19        else if( alg.equals( "d" ) )
20            g.dijkstra( startName );
21        else if( alg.equals( "n" ) )
22            g.negative( startName );
23        else if( alg.equals( "a" ) )
24            g.acyclic( startName );
25
26        g.printPath( destName );
27    }
28    catch( NoSuchElementException e )
29    {
30        return false;
31    }
32    catch( GraphException e )
33    {
34        System.err.println( e );
35    }
36    return true;
37 }
```

Figura 14.15 Para propósitos de prueba, processRequest llama a uno de los algoritmos de determinación del camino más corto.

es determinar la longitud de todos los caminos más cortos. Posteriormente, veremos cómo almacenar los correspondientes caminos.

Podemos ver de forma inmediata que el camino más corto de  $S$  a  $V_2$  es un camino de longitud 0. Esta información nos da el grafo mostrado en la Figura 14.16. Ahora podemos comenzar a buscar todos los vértices que estén a una distancia 1 de  $S$ . Podemos encontrarlos buscando los vértices que sean adyacentes a  $S$ . Si lo hacemos así, vemos que  $V_0$  y  $V_5$  están a una arista de  $S$ , como se muestra en la Figura 14.17.

A continuación, localizamos todos los vértices cuyo camino más corto a partir de  $S$  es exactamente 2. Lo hacemos localizando todos los vértices que sean adyacentes a  $V_0$  o a  $V_5$  (los vértices que están a distancia 1) y cuyos caminos más cortos no sean ya conocidos. Esta búsqueda nos dice que el camino más corto hasta  $V_1$  y  $V_3$  es 2. La Figura 14.18 muestra el progreso que hemos hecho hasta el momento.

Finalmente, examinando los vértices adyacentes a los vértices  $V_1$  y  $V_3$  que acabamos de evaluar, encontramos que  $V_4$  y  $V_6$  tienen un camino más corto equivalente a 3 aristas. Con esto habremos calculado ya todos los vértices. La Figura 14.19 muestra el resultado final del algoritmo.

Esta estrategia para explorar un grafo se denomina *búsqueda en anchura*, que opera procesando los vértices capa a capa: los más próximos al inicio se evalúan primero y los más distantes se evalúan en último lugar.

La Figura 14.20 ilustra un principio fundamental: si un camino hasta un vértice  $v$  tiene un coste  $D_v$  y  $w$  es adyacente a  $v$ , entonces existe un camino hasta  $w$  de coste  $D_w = D_v + 1$ . Todos los algoritmos de determinación del camino más corto funcionan comenzando con  $D_w = \infty$  y reduciendo su valor después de analizar un  $v$  apropiado. Para llevar a cabo esta tarea de manera eficiente, debemos explorar los vértices  $v$  sistemáticamente. Cuando se explora un vértice  $v$ , actualizamos los vértices  $w$  adyacentes a  $v$ , recorriendo la lista de adyacencia de  $v$ .

A partir de las explicaciones anteriores, concluimos que un algoritmo para resolver el problema del camino más corto no ponderado sería el siguiente. Sea  $D_v$  la longitud del camino más corto que

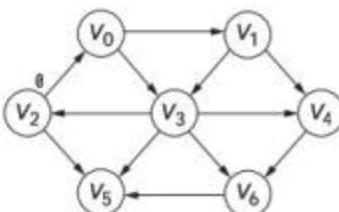


Figura 14.16 El grafo después de haber marcado el vértice inicial como alcanzable en cero aristas.

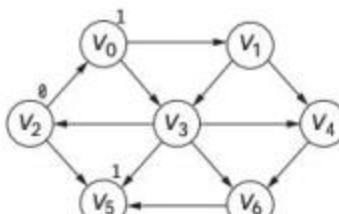


Figura 14.17 El grafo después de haber encontrado todos los vértices cuya longitud de camino a partir del vértice inicial es 1.

**La búsqueda en anchura**  
procesa los vértices capa a capa: los más próximos al inicio se evalúan primero y los más distantes se evalúan en último lugar.

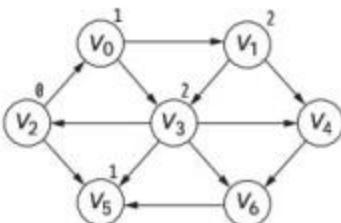


Figura 14.18 El grafo después de haber encontrado todos los vértices cuya longitud de camino a partir del vértice inicial es 2.

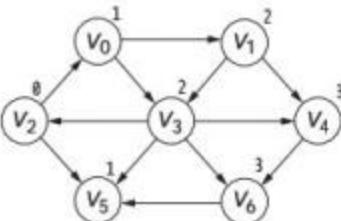


Figura 14.19 Los caminos finales más cortos.

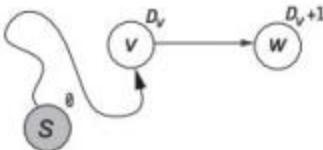


Figura 14.20 Si  $w$  es adyacente a  $v$  y existe un camino hasta  $v$ , también habrá un camino hasta  $w$ .

va de  $S$  a  $i$ . Sabemos que, inicialmente  $D_S = 0$  y que  $D_i = \infty$  para todo  $i \neq S$ . Lo que hacemos es mantener un *foco móvil* que va saltando de vértice en vértice y que se encuentra inicialmente en  $S$ . Si ves el vértice en el que se encuentra actualmente el foco, 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 hasta  $w$  siguiendo un camino hasta  $v$  y ampliando ese camino mediante la arista  $(v, w)$  –de nuevo, esto se ilustra en la Figura 14.20. Así que actualizamos los vértices  $w$  a medida que vamos viéndolos desde ese ángulo aventajado que nos proporcionada el foco móvil. Puesto que el foco móvil procesa cada vértice por orden de distancia con respecto al vértice inicial, y puesto que la arista añade exactamente 1 unidad a la longitud del camino hasta  $w$ , está garantizado que la primera vez que  $D_w$  baje de  $\infty$ , adoptará el valor correspondiente a la longitud del camino más corto a  $w$ . Estas acciones también nos dicen que el penúltimo vértice del camino que va hasta  $w$  es  $v$ , por lo que una sola línea adicional del código nos permite almacenar el camino completo.

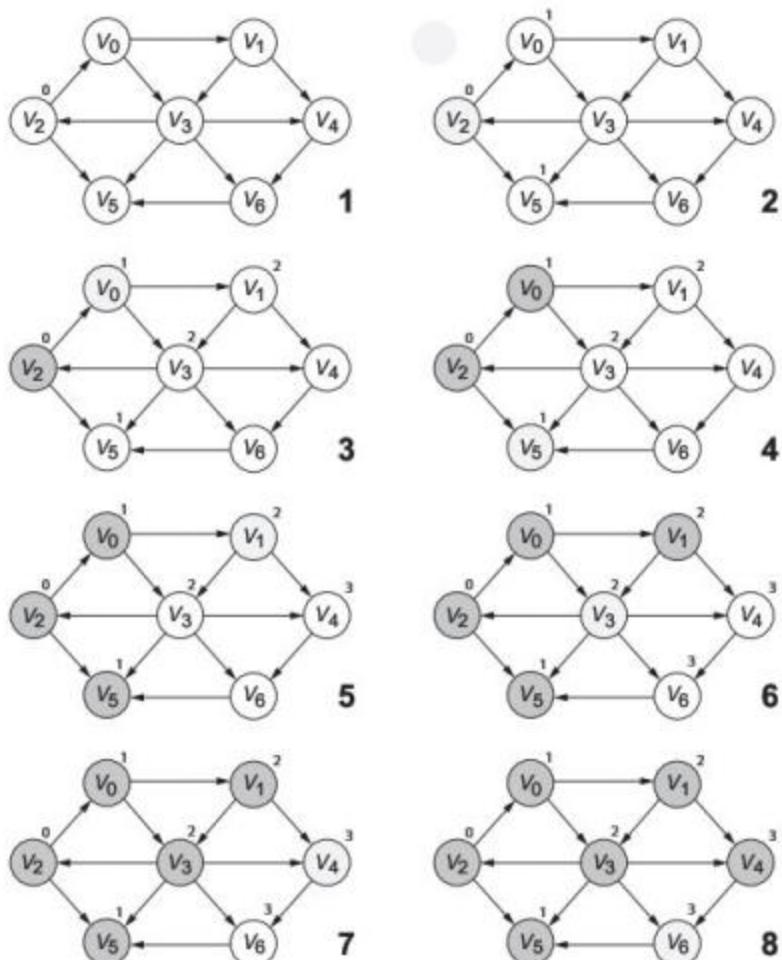
*El foco móvil* se va desplazando de vértice en vértice y actualiza las distancias correspondientes a los vértices adyacentes.

Después de haber procesado todos los vértices adyacentes a  $v$ , desplazamos el foco hasta otro vértice  $u$  (que no haya sido visitado por el foco) de modo que  $D_u \equiv D_v$ . Si eso no es posible, nos desplazamos hasta un  $u$  que satisfaga  $D_u = D_v + 1$ . Si esto no es posible, entonces habremos terminado de procesar el grafo. La Figura 14.21 muestra cómo el foco va visitando los vértices y

Todos los vértices adyacentes a  $v$  se pueden encontrar analizando la lista de adyacencia de  $v$ .

actualizando las distancias. El nodo ligeramente sombreado en cada etapa representa la posición del foco. En esta imagen y en las que siguen, las etapas se muestran de arriba a abajo y de izquierda a derecha.

El detalle que nos queda por comentar es el de la estructura de datos, y aquí hay dos acciones básicas que debemos tomar. En primer lugar, tenemos que ir encontrando repetidamente el vértice en el que colocar el foco. En segundo lugar, necesitamos comprobar todos los wadyacentes a  $v$  (el vértice actual) a lo largo de todo el algoritmo. La segunda acción puede implementarse fácilmente iterando a través de la lista de adyacencia de  $v$ . De hecho, como cada arista solo se procesa una vez, el coste total de todas las iteraciones será  $O(|E|)$ . La primera acción es más complicada: no podemos simplemente explorar la tabla del grafo (véase la Figura 14.4) buscando un vértice apropiado, porque cada exploración podría requerir un tiempo



**Figura 14.21** Exploración del grafo en el cálculo del camino más corto no ponderado. Los vértices con sombreado más oscuro ya han sido completamente procesados, los vértices más claros todavía no han sido utilizados como  $v$ , y el vértice con sombreado intermedio es el vértice actual,  $v$ . Las etapas van de izquierda a derecha y de arriba a abajo, tal como indican los números adjuntos.

$O(|V|)$  y tendríamos que realizarla  $|V|$  veces. Por tanto, el coste total sería  $O(|V|^2)$ , que es inaceptable para grafos dispersos. Afortunadamente, esta técnica no es necesaria.

Cuando el  $D_w$  de un vértice  $w$  baja de  $\infty$ , pasa a ser un candidato a ser visitado por el foco en algún momento futuro. Es decir, después de que el foco visite los vértices en el grupo actual de distancias  $D_v$ , visitará el siguiente grupo de distancias  $D_v + 1$ , que es el grupo que contiene  $w$ . Por tanto, lo único que tiene que hacer  $w$  es esperar en una cola a que le llegue el turno. Asimismo, puesto que no necesita ser visitado antes que ningún otro vértice cuya distancia ya haya sido reducida, habrá que colocar  $w$  al final de una cola de vértices que estén a la espera de que el foco los visite.

Para seleccionar un vértice  $v$  para que el foco lo visite, simplemente elegimos el vértice situado al principio de la cola. Comenzamos con una cola vacía y luego ponemos en cola el vértice inicial  $S$ . Cada vértice será puesto en cola y extraído de la cola como máximo una vez por cada cálculo del camino más corto, y las operaciones con colas tardan un tiempo constante, por lo que el coste de elegir el vértice que hay que seleccionar es solo  $O(|V|)$  para todo el algoritmo. Por tanto, el coste de la búsqueda en anchura está dominado por las exploraciones de las listas de adyacencia y es  $O(|E|)$ , es decir, lineal con respecto al tamaño del grafo.

Cuando se reduce la distancia de un vértice (lo que solo puede suceder una vez), se coloca en la cola para que el foco pueda visitarlo en el futuro. El vértice inicial se coloca en la cola en el momento de inicializar su distancia a cero.

#### 14.2.2 Implementación Java

El algoritmo de determinación del camino más corto no ponderado se implementa mediante el método `unweighted`, como se muestra en la Figura 14.22. El código es una traducción línea a línea del algoritmo que hemos descrito anteriormente. La inicialización en las líneas 6 a 13 hace que todas las distancias sean igual a infinito, asigna a  $D_S$  el valor 0 y luego introduce en la cola el vértice inicial. La cola se declara en la línea 12. Mientras que la cola no esté vacía, habrá vértices que visitar. Por ello, en la línea 17 nos desplazamos al vértice  $v$  que se encuentre al principio de la cola. La línea 19 itera a través de la lista de adyacencia y genera todos los  $w$ adyacentes a  $v$ . La comprobación  $D_w = \infty$  se lleva a cabo en la línea 23. Si devuelve `true`, se realiza la actualización  $D_w = D_v + 1$  en la línea 25 junto con la actualización del miembro de datos `prev` de  $w$  la puesta en cola de  $w$  en las líneas 26 y 27, respectivamente.

La implementación es mucho más simple de lo que parece. Sigue literalmente la descripción del algoritmo que hemos proporcionado.

### 14.3 Problema del camino más corto con ponderaciones positivas

Recuerde que la longitud ponderada de un camino es la suma de todos los costes de las aristas que forman el camino. En esta sección, vamos a considerar el problema de calcular el camino más corto con ponderación en un grafo cuyas aristas tengan un coste no negativo. Queremos encontrar el camino más corto con ponderación hasta todos los vértices a partir de un cierto vértice inicial. Como veremos en breve, la suposición de que los costes de las aristas sean no negativos es importante, porque permite obtener un algoritmo relativamente eficiente. El método

La longitud de un camino con ponderación es la suma de los costes de todas las aristas que forman el camino.

```
1  /**
2   * Algoritmo del camino más corto no ponderado con un único origen.
3   */
4  public void unweighted( String startName )
5  {
6      clearAll( );
7
8      Vertex start = vertexMap.get( startName );
9      if( start == null )
10         throw new NoSuchElementException( "Start vertex not found" );
11
12     Queue<Vertex> q = new LinkedList<Vertex>();
13     q.add( start ); start.dist = 0;
14
15     while( !q.isEmpty( ) )
16     {
17         Vertex v = q.remove( );
18
19         for( Edge e : v.adj )
20         {
21             Vertex w = e.dest;
22
23             if( w.dist == INFINITY )
24             {
25                 w.dist = v.dist + 1;
26                 w.prev = v;
27                 q.add( w );
28             }
29         }
30     }
31 }
```

---

Figura 14.22 Algoritmo de determinación del camino más corto no ponderado, utilizando una búsqueda en anchura.

utilizado para resolver el problema de la determinación del camino más corto con ponderación positiva se conoce con el nombre de *algoritmo de Dijkstra*. En la siguiente sección examinaremos otro algoritmo más lento que funciona incluso aunque haya aristas con un coste negativo.

#### Problema del camino más corto con ponderaciones positivas y un único origen

Encontrar el camino más corto (medido según el coste total) desde un vértice designado  $S$  hasta todos los demás vértices. Todos los costes de las aristas son no negativos.

### 14.3.1 Teoría: algoritmo de Dijkstra

El problema de la determinación del camino más corto con ponderación positiva se resuelve de forma bastante similar al problema en el caso de que no haya ponderación. Sin embargo, debido a los costes de las aristas, hay algunas cosas que cambian. Es necesario analizar las siguientes cuestiones:

1. ¿Cómo ajustamos  $D_w$ ?
2. ¿Cómo encontramos el vértice  $v$  que el foco tiene que visitar?

Comenzamos examinando cómo modificar  $D_w$ . Al resolver el problema del camino más corto sin ponderación, cuando  $D_w = \infty$ , hacíamos  $D_w = D_v + 1$  porque reducimos el valor de  $D_w$  si el vértice  $v$  ofrecía un camino más corto hasta  $w$ . La dinámica del algoritmo garantiza que solo necesitamos modificar  $D_w$  una vez. Sumamos 1 a  $D_v$  porque la longitud del camino hasta  $w$  es 1 unidad más que la longitud del camino hasta  $v$ . Si aplicamos esta misma lógica al caso de aristas con ponderación, deberíamos hacer  $D_w = D_v + c_{v,w}$  si este nuevo valor de  $D_w$  es mejor que el valor original. Sin embargo, ya no está garantizado que  $D_w$  solo se vea modificado una única vez. En consecuencia,  $D_w$  deberá ser modificado si su valor actual es mayor que  $D_v + c_{v,w}$  (en lugar de limitarnos a comprobar si es  $\infty$ ). Dicho de forma simple, el algoritmo decide si debería emplearse  $v$  como parte del camino a  $w$ . El coste original  $D_w$  es el coste sin utilizar  $v$ ; el coste  $D_v + c_{v,w}$  es el camino más barato utilizando  $v$  (hasta ahora).

La Figura 14.23 muestra una situación típica. Anteriormente en el algoritmo, se había reducido hasta 8 la distancia a  $w$  cuando el foco visitaba el vértice  $u$ . Sin embargo, cuando el foco visita el vértice  $v$ , se hace necesario reducir la distancia hasta el vértice  $w$  a 6, porque ahora disponemos de un nuevo camino más corto. Este resultado nunca puede presentarse en el caso del algoritmo sin ponderación, porque todas las aristas añaden 1 a la longitud del camino, por lo que  $D_u \leq D_v$  implica que  $D_u + 1 \leq D_v + 1$  y por tanto  $D_w \leq D_v + 1$ . Aquí, aunque  $D_u \leq D_v$ , podemos continuar mejorando el camino hasta  $w$ , tomando en consideración  $v$ .

La Figura 14.23 ilustra otro punto importante. Cuando se reduce la distancia hasta  $w$ , esto se produce únicamente porque dicho vértice es adyacente a algún otro vértice que ha sido visitado por el foco. Por ejemplo, después de que el foco visite  $v$  y se haya completado el procesamiento, el valor de  $D_w$  será 6 y el último vértice del camino será un vértice que ha sido visitado por el foco. De forma similar, el vértice anterior a  $v$  tiene que haber sido visitado también por el foco, etc. Por tanto, en

**El algoritmo de Dijkstra**  
se utiliza para resolver el problema de la determinación del camino más corto con ponderación positiva.

Utilizamos  $D_v + c_{v,w}$  como la nueva distancia y para decidir si la distancia se debe actualizar.

Ya no es apropiado utilizar una cola para almacenar los vértices que están esperando a que el foco los visite.

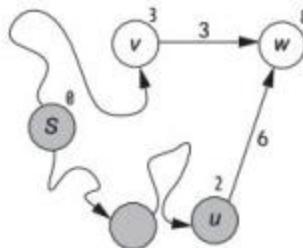


Figura 14.23 El foco se encuentra en  $v$  y  $w$  es adyacente, por lo que  $D_w$  debe reducirse a 6.

La distancia hasta los vértices no visitados representa un camino en el que los únicos nodos intermedios son vértices que ya han sido visitados.

cualquier instante, el valor de  $D_w$  representa un camino desde  $S$  a  $w$  utilizando únicamente como nodos intermedios vértices que ya han sido visitados por el foco. Este hecho crucial nos da el Teorema 14.1.

### Teorema 14.1

Si desplazamos el foco al vértice todavía no visitado con  $D_v$  mínimo, el algoritmo genera correctamente los caminos más cortos, siempre que no haya costes de aristas negativos.

### Demostración

Vamos a llamar "etapa" a cada visita del foco. Vamos a demostrar por inducción que, después de cualquier etapa, los valores de  $D_i$  para los vértices visitados por el foco forman el camino más corto y que los valores de  $D_i$  para los otros vértices forman el camino más corto empleando únicamente como nodos intermedios vértices visitados por el foco. Puesto que el primer vértice visitado es el vértice inicial, este enunciado es correcto para la primera etapa. Suponga que es correcto para las primeras  $k$  etapas. Sea  $v$  el vértice elegido por el foco en la etapa  $k+1$ . Suponga, con el objeto de demostrar una contradicción, que existe un camino que va desde  $S$  hasta  $v$  cuya longitud es inferior a  $D_v$ .

Este camino deberá pasar a través de un vértice intermedio que todavía no haya sido visitado por el foco. Vamos a llamar  $u$  al primer vértice intermedio todavía no visitado por el foco que haya en el camino. Esta situación se muestra en la Figura 14.24. El camino hasta  $u$  solo emplea vértices visitados por el foco como nodos intermedios, por lo que por inducción,  $D_u$  representa la distancia óptima hasta  $u$ . Además,  $D_u < D_v$ , porque  $u$  se encuentra en el camino hasta  $v$  que se supone que es más corto. Esta desigualdad es una contradicción, porque entonces teníamos que haber movido el foco a  $u$  en lugar de a  $v$ . La demostración se completa demostrando que todos los valores  $D_i$  continúan siendo correctos para los nodos no visitados, lo cual está claro por la regla de actualización.

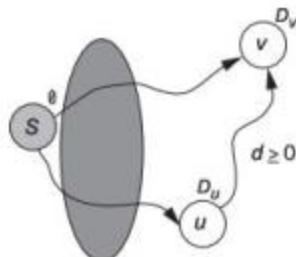


Figura 14.24 Si  $D_v$  es mínimo entre todos los vértices todavía no visitados y si todos los costes de las aristas son no negativos,  $D_v$  representa el camino más corto.

La Figura 14.25 muestra las etapas del algoritmo de Dijkstra. El problema que queda es seleccionar una estructura de datos apropiada. Para grafos densos, podemos explorar la tabla del grafo, buscando el vértice apropiado. Como sucede con el algoritmo del camino más corto sin ponderación, esta exploración requerirá un tiempo  $O(|V|^2)$ , que resulta óptimo para un grafo denso. Pero para un grafo disperso, queremos una solución mejor.

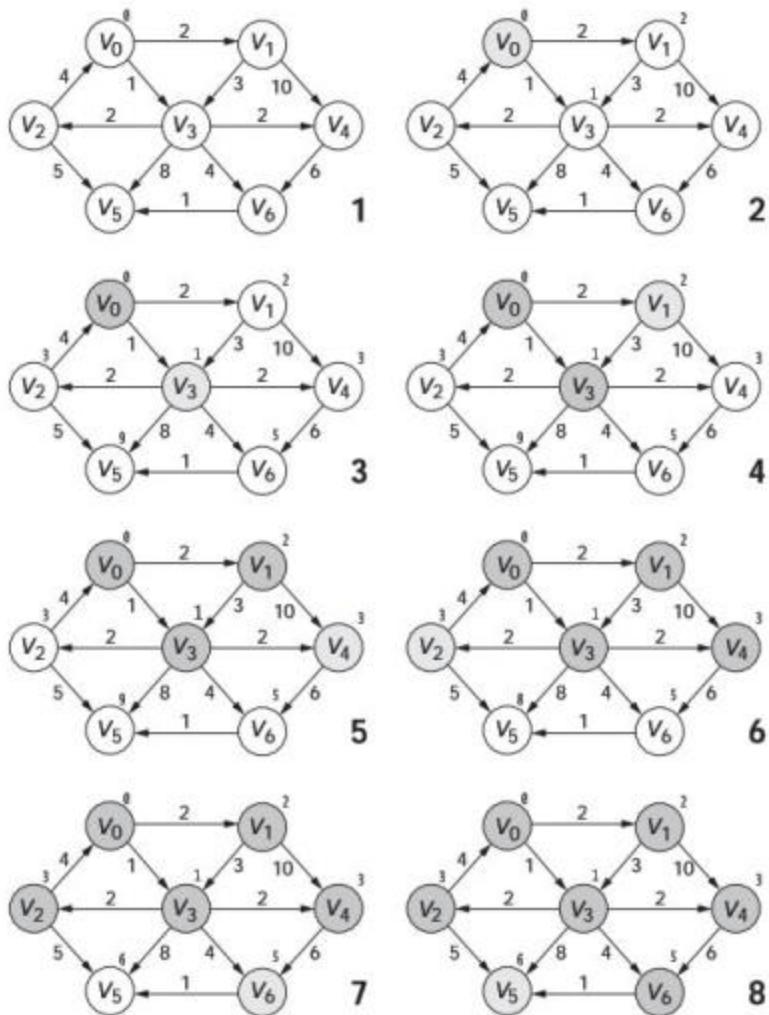


Figura 14.25 Etapas del algoritmo de Dijkstra. Los convenios son los mismos que en la Figura 14.21.

Ciertamente, una cola no funciona. El hecho de que necesitemos encontrar el vértice  $v$  con un valor  $D_v$  mínimo sugiere que lo que deberíamos utilizar es una cola con prioridad. Hay dos formas de emplear la cola con prioridad. Una consiste en almacenar cada vértice en la cola con prioridad y utilizar la distancia (obtenida consultando la tabla del grafo) como función de ordenación. Cuando modifiquemos cualquier  $D_w$ , tendremos que actualizar la cola con prioridad restableciendo la propiedad de ordenación. Esta acción equivale a una operación `decreaseKey`. Para llevarla a cabo, necesitaremos poder determinar la ubicación de  $w$  en la cola con prioridad. Muchas implementaciones de la cola con prioridad no soportan `decreaseKey`. Una que sí lo hace es el *montículo de emparejamiento*; hablaremos del uso del montículo de emparejamiento para esta aplicación en el Capítulo 23.

La cola con prioridad es una estructura de datos apropiada. El método más simple consiste en añadir a la cola con prioridad una nueva entrada, formada por un vértice y una distancia, cada vez que se reduce la distancia de un vértice. Podemos encontrar el nuevo vértice al que hay que desplazar el foco extrayendo rápidamente el vértice de distancia mínima de la cola con prioridad, hasta que aparezca un vértice no visitado.

En lugar de utilizar una sofisticada cola con prioridad, empleamos un método que funciona con una cola con prioridad simple, como el montículo binario, del que hablaremos en el Capítulo 21. Nuestro método implica insertar un objeto compuesto por  $w$  y  $D_w$  en la cola con prioridad cada vez que reduzcamos el valor de  $D_w$ . Para seleccionar un nuevo vértice  $v$  con el fin de visitarlo, vamos extrayendo repetidamente el método mínimo (basado en la distancia) de la cola con prioridad, hasta que aparezca un vértice no visitado. Puesto que el tamaño de cola con prioridad podría ser de hasta  $|E|$  y hay como máximo  $|E|$  inserciones y borrados en la cola con prioridad, el tiempo de ejecución es  $O(|E| \log |E|)$ . Puesto que  $|E| \leq |V|^2$  implica que  $\log |E| \leq 2 \log |V|$ , tenemos el mismo algoritmo  $O(|E| \log |V|)$  que tendríamos si usáramos el primer método (en el que el tamaño de la cola con prioridad es como máximo  $|V|$ ).

### 14.3.2 Implementación Java

De nuevo, la implementación sigue de forma bastante fiel la descripción que hemos proporcionado.

El objeto insertado en la cola con prioridad se muestra en la Figura 14.26. Está compuesto por  $w$  y  $D_w$  y una función de comparación definida sobre la base de  $D_w$ . La Figura 14.27 muestra la rutina `dijkstra` que calcula los caminos más cortos.

La línea 6 declara la cola con prioridad `pq`. Declaramos `vrec` en la línea 18 para almacenar el resultado de cada `deleteMin`. Como sucede con el algoritmo del camino más corto sin ponderación, comenzamos asignando a todas las distancias un valor infinito, haciendo  $D_s = 0$  y colocando el vértice inicial en nuestra estructura de datos.

```

1 // Representa un elemento de la cola con prioridad para el algoritmo de Dijkstra.
2 class Path implements Comparable<Path>
3 {
4     public Vertex dest; // w
5     public double cost; // d(w)
6
7     public Path( Vertex d, double c )
8     {
9         dest = d;
10        cost = c;
11    }
12
13    public int compareTo( Path rhs )
14    {
15        double otherCost = rhs.cost;
16
17        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
18    }
19 }
```

Figura 14.26 Elemento básico almacenado en la cola con prioridad.

```
1  /**
2  * Algoritmo del camino más corto con ponderación y un único origen.
3  */
4  public void dijkstra( String startName )
5  {
6      PriorityQueue<Path> pq = new PriorityQueue<Path>();
7
8      Vertex start = vertexMap.get( startName );
9      if( start == null )
10         throw new NoSuchElementException( "Start vertex not found" );
11
12     clearAll();
13     pq.add( new Path( start, 0 ) ); start.dist = 0;
14
15     int nodesSeen = 0;
16     while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
17     {
18         Path vrec = pq.remove();
19         Vertex v = vrec.dest;
20         if( v.scratch != 0 ) // v ya procesado
21             continue;
22
23         v.scratch = 1;
24         nodesSeen++;
25
26         for( Edge e : v.adj )
27         {
28             Vertex w = e.dest;
29             double cvw = e.cost;
30
31             if( cvw < 0 )
32                 throw new GraphException( "Graph has negative edges" );
33
34             if( w.dist > v.dist + cvw )
35             {
36                 w.dist = v.dist + cvw;
37                 w.prev = v;
38                 pq.add( new Path( w, w.dist ) );
39             }
40         }
41     }
42 }
```

Figura 14.27 Un algoritmo de determinación del camino más corto con ponderaciones positivas: algoritmo de Dijkstra.

Cada iteración del bucle `while` que comienza en la línea 16, sitúa el foco en un vértice  $v$  y lo procesa examinando los vértices  $w$  adyacentes.  $v$  se selecciona extrayendo repetidamente elementos de la cola con prioridad (en la línea 18) hasta encontrar un vértice que no haya sido procesado. Usamos la variable `scratch` para anotar este hecho. Inicialmente, `scratch` es 0. Por tanto, si el vértice no ha sido procesado, la comprobación falla en la línea 20 y alcanzamos la línea 23. Entonces, al procesar el vértice, `scratch` se pone a 1 (en la línea 23). La cola con prioridad podría estar vacía si, por ejemplo, algunos de los vértices fueran inalcanzables. En ese caso, podríamos volver inmediatamente. El bucle de las líneas 26 a 40 es bastante similar al del algoritmo sin ponderación. La diferencia es que en la línea 29, debemos extraer `cvw` del elemento de la lista de adyacencia, asegurarnos de que la arista es no negativa (en caso contrario, nuestro algoritmo podría producir respuestas incorrectas), sumar `cvw` en lugar de 1 en las líneas 34 y 36 e insertar en la cola con prioridad en la línea 38.

## 14.4 Problema del camino más corto con ponderaciones negativas

Las aristas negativas hacen que el algoritmo de Dijkstra deje de funcionar. Nos hace falta un algoritmo alternativo.

El algoritmo de Dijkstra exige que el coste de las aristas sea no negativo. Este requisito es razonable para la mayoría de las aplicaciones de los grafos, pero en ocasiones es demasiado restrictivo. En esta sección vamos a analizar brevemente el caso más general: el algoritmo del camino más corto con ponderaciones negativas.

### Problema del camino más corto con ponderaciones negativas y un único origen

Encontrar el camino más corto (medido según el coste total) desde un vértice designado  $S$  hasta todos los demás vértices. Los costes de las aristas pueden ser negativos.

#### 14.4.1 Teoría

La demostración del algoritmo de Dijkstra requería la condición de que el coste de las aristas y por tanto de los caminos fueran no negativos. De hecho, si el grafo tiene costes de aristas negativos, el algoritmo de Dijkstra no funciona. El problema es que, una vez que un vértice  $v$  ha sido procesado, puede haber para algún otro vértice  $u$  no procesado, un camino de coste negativo que vuelva a  $v$ . En tal caso, tomar un camino desde  $S$  hasta  $u$  y luego hasta  $v$  es mejor que ir desde  $S$  hasta  $v$  sin utilizar  $u$ . Si hiciéramos esto último, tendríamos problemas. No solo es que el camino hasta  $v$  sería incorrecto, sino que también tendríamos que revisitar  $v$  porque las distancias de los vértices alcanzables desde  $v$  podrían verse afectadas. (En el Ejercicio 14.9 le pediremos que construya un ejemplo explícito; basta con cuatro vértices.)

Tenemos otro problema más del que preocuparnos. Considere el grafo mostrado en la Figura 14.28. El camino que va desde  $V_3$  a  $V_4$  tiene un coste igual a 2. Sin embargo, existe un camino más corto describiendo el bucle  $V_3, V_4, V_1, V_3, V_4$ , que tiene un coste de  $-3$ . Este camino sigue, incluso, sin ser el más corto, porque podríamos permanecer en el bucle un tiempo arbitrariamente largo. Por tanto, el camino más corto entre esos dos puntos no está definido.

Este problema no está restringido a los nodos que componen el ciclo. El camino más corto desde  $V_2$  a  $V_5$  tampoco está definido, porque existe una forma de entrar y salir del bucle. Este tipo de bucle

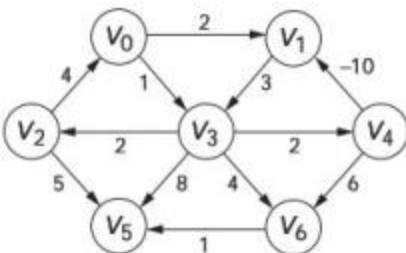


Figura 14.28 Un grafo con un ciclo de coste negativo.

se denomina *ciclo de coste negativo*, y si está presente en un grafo hace que la mayoría de los caminos cortos, si es que no todos, sean indefinidos. Las aristas con coste negativo no son necesariamente perjudiciales por sí mismas; son los ciclos los que son perjudiciales. Nuestro algoritmo encontrará los caminos más cortos o informará de la existencia de un ciclo con coste negativo.

Una combinación de los algoritmos con ponderación y sin ponderación permitirá resolver el problema, pero a cambio habrá que pagar el coste de un enorme incremento potencial en el tiempo de ejecución. Como hemos sugerido anteriormente, cuando  $D_v$  se modifica, debemos revisitarla en algún momento futuro. En consecuencia, usamos la cola como en el algoritmo no ponderado, pero empleamos  $D_v + c_{v,w}$  como medida de distancia (como en el algoritmo de Dijkstra). El algoritmo que se utiliza para resolver el problema del camino más corto con ponderaciones negativas se conoce con el nombre de *algoritmo de Bellman–Ford*.

Cuando el foco se detiene en el vértice  $v$  por  $i$ ésima vez, el valor de  $D_v$  es la longitud del camino ponderado más corto compuesto de  $i$  o menos aristas. Dejamos la demostración para el lector como Ejercicio 14.14. En consecuencia, si no hay ciclos con coste negativo, un vértice podrá ser extraído de la cola como máximo  $|V|$  veces y el algoritmo necesitará como máximo un tiempo  $O(|E||V|)$ . Además, si se extrae un vértice de la cola más de  $|V|$  veces, habremos detectado un ciclo de coste negativo.

Un ciclo de coste negativo hace que la mayoría de los caminos más cortos, si es que no todos, sean indefinidos, porque podemos permanecer en el ciclo un tiempo arbitrariamente largo y obtener una longitud ponderada de camino arbitrariamente negativa.

Cada vez que se reduce la distancia de un vértice, es necesario insertarlo en una cola. Esto puede suceder rápidamente para cada vértice.

El tiempo de ejecución puede ser grande, especialmente si hay un ciclo de coste negativo.

#### 14.4.2 Implementación Java

La implementación del algoritmo de determinación del camino más corto con ponderaciones negativas se proporciona en la Figura 14.29. Hacemos un pequeño cambio en la descripción del algoritmo: en concreto, no insertamos un vértice en la cola si ya se encuentra en la misma. Este cambio implica utilizar el miembro de datos `scratch`. Cuando se pone un vértice en la cola, incrementamos `scratch` (en la línea 31). Cuando se extrae de la cola, lo volvemos a incrementar (en la línea 18). Por tanto, `scratch` es impar si el vértice se encuentra en la cola y `scratch/2` nos dice cuántas veces ha salido de la cola (lo que explica la

La parte complicada de la implementación es la manipulación de la variable `scratch`. Tratamos de evitar que ningún vértice aparezca en la cola dos veces en ningún momento.

```

1  /**
2   * Algoritmo del camino más corto con ponderaciones negat. y único origen
3   */
4  public void negative( String startName )
5  {
6      clearAll( );
7
8      Vertex start = vertexMap.get( startName );
9      if( start == null )
10         throw new NoSuchElementException( "Start vertex not found" );
11
12     Queue<Vertex> q = new LinkedList<Vertex>();
13     q.add( start ); start.dist = 0; start.scratch++;
14
15     while( !q.isEmpty( ) )
16     {
17         Vertex v = q.removeFirst( );
18         if( v.scratch++ > 2 * vertexMap.size( ) )
19             throw new GraphException( "Negative cycle detected" );
20
21         for( Edge e : v.adj )
22         {
23             Vertex w = e.dest;
24             double cvw = e.cost;
25
26             if( w.dist > v.dist + cvw )
27             {
28                 w.dist = v.dist + cvw;
29                 w.prev = v;
30                 // Insertar en cola solo si no se encuentra ya en ella.
31                 if( w.scratch++ % 2 == 0 )
32                     q.add( w );
33                 else
34                     w.scratch--; // Deshacer el incremento de puesta en cola
35             }
36         }
37     }
38 }

```

**Figura 14.29** Un algoritmo de determinación del camino más corto con ponderaciones negativas: se permiten las aristas negativas.

comprobación de la línea 18). Cuando se modifica la distancia de algún  $w$  pero este ya se encuentra en la cola (porque `scratch` es impar), no lo insertamos en la cola. Sin embargo, no sumamos 2 para

indicar que ha entrado en la cola (y salido de ella); de eso se encargan los incrementos y decrementos de las líneas 31 y 34. El resto del algoritmo utiliza código que ya hemos empleado anteriormente tanto en el algoritmo del camino más corto sin ponderación (Figura 14.22) como en el algoritmo de Dijkstra (Figura 14.27).

## 14.5 Problemas de caminos en grafos acíclicos

Recuerde que un grafo acíclico dirigido no tiene ningún ciclo. Esta importante clase de grafos simplifica la solución del problema del camino más corto. Por ejemplo, no tenemos que preocuparnos por los ciclos de coste negativo porque no existen ciclos. Por tanto, vamos a considerar el siguiente problema.

### **Problema del camino más corto con ponderaciones negativas y un único origen para grafos acíclicos**

Encontrar el camino más corto (medido según el coste total) desde un vértice designado  $S$  hasta todos los demás vértices de un grafo acíclico. Los costes de las aristas no están restringidos.

### 14.5.1 Ordenación topológica

Antes de considerar el problema del camino más corto, examinemos otro problema relacionado: el de la ordenación topológica. Una *ordenación topológica* ordena los vértices de un grafo acíclico dirigido de forma tal que, si existe un camino desde  $u$  hasta  $v$ , entonces  $v$  aparece después de  $u$  en la ordenación. Por ejemplo, se utiliza típicamente un grafo para representar cuáles son los requisitos de los cursos en las universidades. Una arista  $(v, w)$  indicará que la asignatura  $v$  debe completarse antes de poder matricularse de la asignatura  $w$ . Una ordenación topológica de las asignaturas es cualquier secuencia que no viole los requisitos establecidos para las mismas.

Una ordenación topológica ordena los vértices de un grafo acíclico dirigido de forma tal que, si existe un camino que va desde  $u$  hasta  $v$ , entonces  $v$  aparece después de  $u$  en la ordenación. Un grafo que tenga un ciclo no puede tener una ordenación topológica.

Claramente, no se puede realizar una ordenación topológica si un grafo tiene un ciclo, porque para cualesquiera dos vértices  $v$  y  $w$  pertenecientes al ciclo, existe un camino que va de  $v$  a  $w$  y otro que va de  $w$  a  $v$ . Por tanto, cualquier ordenación de  $v$  y  $w$  entraría en contradicción con uno de los dos caminos. Un grafo puede tener varias ordenaciones topológicas, y en la mayoría de los casos nos valdrá con cualquier ordenación que sea legal.

En un algoritmo simple para llevar a cabo una ordenación topológica, lo primero que hacemos es encontrar algún vértice  $v$  que no tenga ninguna arista entrante. A continuación, imprimimos el vértice y lo eliminamos desde el punto de vista lógico del grafo, junto con sus aristas. Finalmente, aplicamos la misma estrategia al resto del grafo. De manera más formal, decimos que el *grado entrante* de un vértice  $v$  es el número de aristas entrantes ( $u, v$ ).

El grado entrante de un vértice es el número de aristas entrantes en el mismo. Se puede realizar una ordenación topológica en un tiempo lineal eliminando repetidamente, desde el punto de vista lógico, los vértices que no tengan aristas entrantes.

Calculamos el grado entrante para todos los vértices del grafo. En la práctica, cuando decimos que lo que hacemos es *eliminar desde el punto de vista lógico* un vértice, lo que queremos decir es que reducimos el número de aristas entrantes para cada vértice adyacente a  $v$ . La Figura 14.30 muestra el algoritmo aplicado a un grafo acíclico. Se calcula el grado entrante para cada vértice. El vértice  $V_2$  tiene un grado entrante igual a 0, por lo que

será el primero en la ordenación topológica. Si hubiera varios vértices con grado entrante igual a 0, podríamos seleccionar cualquiera de ellos. Si eliminamos  $V_2$  y sus aristas del grafo, el grado entrante de  $V_0$ ,  $V_3$  y  $V_5$  se reducen en 1. Ahora  $V_0$  tendrá un grado entrante igual a 0, por lo que será el siguiente vértice en la ordenación topológica, y  $V_1$  y  $V_3$  verán reducido su grado entrante. El algoritmo continúa y los restantes vértices se examinan en el orden  $V_1$ ,  $V_3$ ,  $V_4$ ,  $V_6$  y  $V_5$ . Por dejarlo claro otra vez, no borramos físicamente aristas del grafo, el eliminarlas en la figura es simplemente para que resulte más fácil ver cómo se reduce el grado entrante de cada vértice.

El algoritmo produce la respuesta correcta y detecta los ciclos si el grafo no es acíclico.

Dos cuestiones importantes que hay que considerar son la *corrección* y la *eficiencia*. Claramente, cualquier ordenación producida por el algoritmo es una ordenación topológica. La pregunta es si todo grafo acíclico tiene una ordenación topológica y, en caso afirmativo, si está garantizado que nuestro algoritmo encuentre una. La respuesta es sí a ambas preguntas.

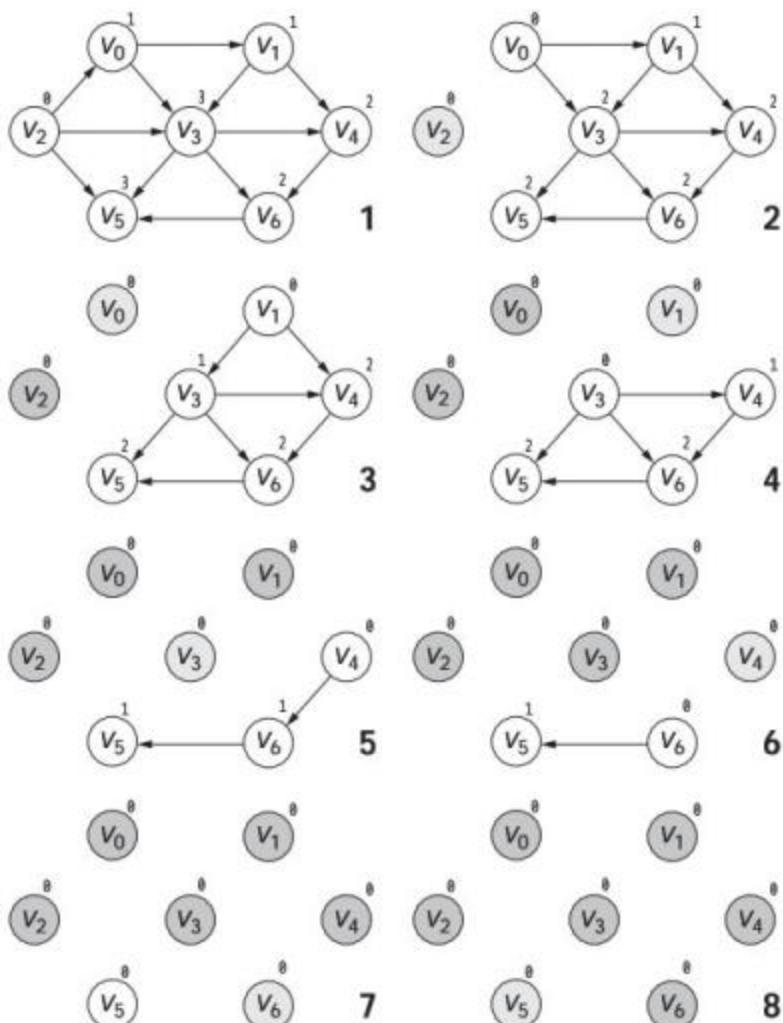


Figura 14.30 Una ordenación topológica. Los convenios son los mismos que los de la Figura 14.21.

Si en cualquier punto hay vértices todavía no visitados, pero ninguno de ellos tiene un grado entrante igual a 0, está garantizado que existe un ciclo. Para ilustrarlo, podemos seleccionar cualquier vértice  $A_0$ . Puesto que  $A_0$  tiene una arista entrante, sea  $A_1$  el vértice conectado a ella. Y como  $A_1$  tiene una arista entrante, sea  $A_2$  el vértice conectado a ella. Repetimos este proceso  $N$  veces, donde  $N$  es el número de vértices no procesados que quedan en el grafo. Entre  $A_0, A_1, \dots, A_N$  deberá haber dos vértices idénticos (puesto que hay  $N$  vértices pero  $N + 1$  aristas distintas). Si nos desplazamos hacia atrás entre esos  $A_i$  y  $A_j$  idénticos, tendremos el ciclo que buscamos.

Podemos implementar el algoritmo en un tiempo lineal insertando en una cola todos los vértices de grado entrante igual a 0 no procesados. Inicialmente, se colocan en la cola todos los vértices de grado entrante igual a 0. Para encontrar el siguiente vértice en el orden topológico, simplemente extraemos el elemento situado al principio de la cola. Cada vez que un vértice ve su grado entrante reducido a 0, se coloca en la cola. Si la cola se vacía antes de haber ordenado topológicamente todos los vértices, querrá decir que el grafo tiene un ciclo. El tiempo de ejecución es claramente lineal, por el mismo tipo de razonamiento utilizado en el algoritmo del camino más corto sin ponderación.

El tiempo de ejecución es lineal si se utiliza una cola.

### 14.5.2 Teoría del algoritmo del camino más corto para grafos acíclicos

Una aplicación importante de la ordenación topológica es a la hora de resolver el problema del camino más corto para grafos acíclicos. La idea es que el foco vaya visitando los vértices en orden topológico.

En un grafo acíclico, el foco simplemente visita los vértices en orden topológico.

Esta idea funciona porque, cuando el foco visita el vértice  $v$ , tenemos garantizado que  $D_v$  ya no puede reducirse más; por la regla de la ordenación topológica, no tiene aristas entrantes que salgan de nodos aun no visitados. La Figura 14.31 muestra las etapas del algoritmo de determinación del camino más corto, utilizando la ordenación topológica para guiar las visitas a los vértices. Observe que la secuencia de vértices visitados no coincide con la del algoritmo de Dijkstra. Observe también que los vértices visitados por el foco antes de alcanzar el vértice inicial son inalcanzables desde el vértice inicial, y no influyen en la distancia existente hasta ningún vértice.

El resultado es un algoritmo de tiempo lineal aun cuando existan aristas con un peso negativo.

No necesitamos una cola con prioridad. En lugar de ello, lo único que tenemos que hacer es incorporar la ordenación topológica en el cálculo del camino más corto. Por tanto, encontramos que el algoritmo se ejecuta en un tiempo lineal, incluso con pesos de arista negativos.

### 14.5.3 Implementación Java

La implementación del algoritmo del camino más corto para grafos acíclicos se muestra en la Figura 14.32. Utilizamos una cola para realizar la ordenación topológica y mantenemos la información del grado entrante en el miembro de datos `scratch`. Las líneas 15 a 18 calculan los grados entrantes y en las líneas 21 a 23 colocamos en la cola todos los vértices que tengan un grado entrante igual a 0.

La implementación combina el cálculo de una ordenación topológica y el cálculo del camino más corto. La información sobre el grado entrante de cada vértice se almacena en el miembro de datos `scratch`.

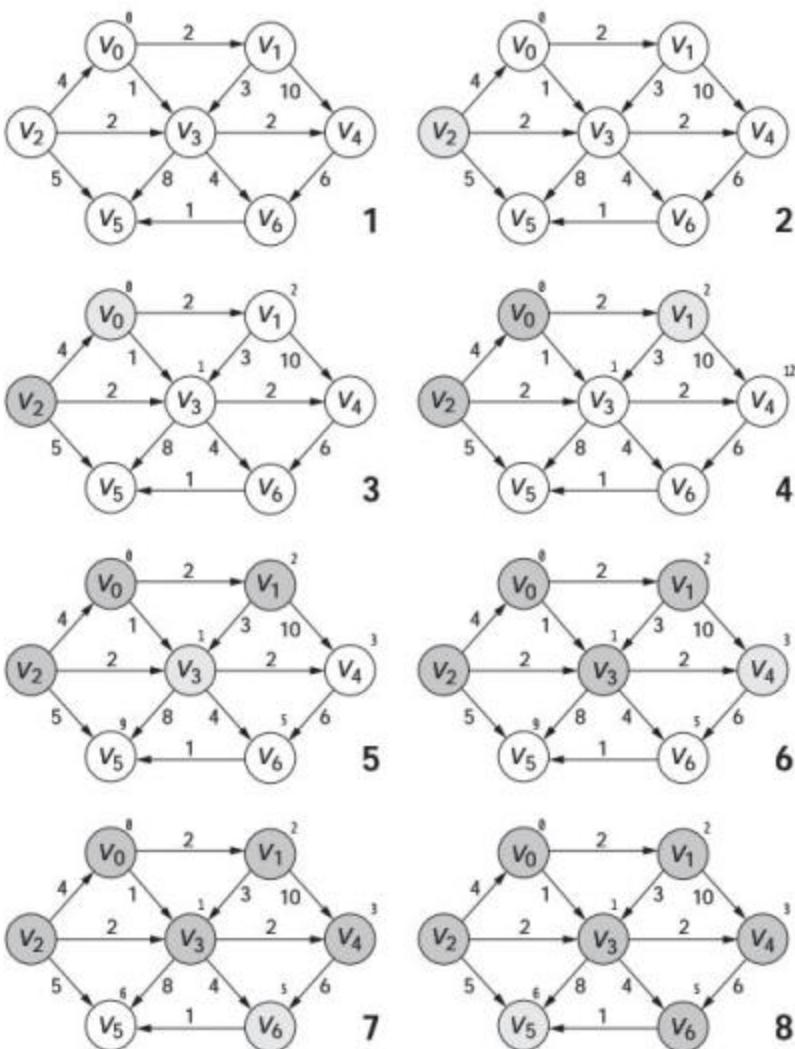


Figura 14.31 Las etapas del algoritmo para un grafo acíclico. Los convenios son los mismos que los de la Figura 14.21.

```

1  /**
2   * Alg. camino más corto con ponderac. neg. y único origen para grafos acíccl.
3   */
4  public void acyclic( String startName )
5  {
6      Vertex start = vertexMap.get( startName );
7      if( start == null )
8          throw new NoSuchElementException( "Start vertex not found" );
9

```

Continúa

Figura 14.32 Un algoritmo de determinación del camino más corto para grafos acíclicos.

```
10    clearAll( );
11    Queue<Vertex> q = new LinkedList<Vertex>();
12    start.dist = 0;
13
14    // Calcular los grados entrantes
15    Collection<Vertex> vertexSet = vertexMap.values();
16    for( Vertex v : vertexSet )
17        for( Edge e : v.adj )
18            e.dest.scratch++;
19
20    // Poner en cola los vértices con grado entrante cero
21    for( Vertex v : vertexSet )
22        if( v.scratch == 0 )
23            q.add( v );
24
25    int iterations;
26    for( iterations = 0; !q.isEmpty(); iterations++ )
27    {
28        Vertex v = q.remove();
29
30        for( Edge e : v.adj )
31        {
32            Vertex w = e.dest;
33            double cvw = e.cost;
34
35            if( --w.scratch == 0 )
36                q.add( w );
37
38            if( v.dist == INFINITY )
39                continue;
40
41            if( w.dist > v.dist + cvw )
42            {
43                w.dist = v.dist + cvw;
44                w.prev = v;
45            }
46        }
47    }
48
49    if( iterations != vertexMap.size() )
50        throw new GraphException( "Graph has a cycle!" );
51 }
```

Figura 14.32 (Continuación)

Después extraemos repetidamente un vértice de la cola en la línea 28. Observe que, si la cola está vacía, el bucle `for` se termina mediante la comprobación de la línea 26. Si el bucle se termina debido a un ciclo, este hecho se indica en la línea 50. En caso contrario, el bucle de la línea 30 recorre la lista de adyacencia y obtiene un valor de `w` en la línea 32. Inmediatamente reducimos el grado entrante de `w` en la línea 35 y, si ha pasado a valer 0, colocamos dicho vértice en la cola en la línea 36.

Recuerde que si el vértice actual `v` aparece antes de `S` en la ordenación topológica, `v` tiene que ser inalcanzable desde `S`. En consecuencia, seguirá teniendo  $D_v \equiv \infty$  y por tanto no cabe esperar que permita obtener ningún camino hacia ningún vértice adyacente `w`. Realizamos una comprobación en la línea 38 y si no se puede proporcionar ningún camino, no intentamos realizar ningún cálculo de distancia. En caso contrario, en las líneas 41 a 45, usamos los mismos cálculos que en el algoritmo de Dijkstra para actualizar  $D_w$  en caso necesario.

Los vértices que aparecen antes de `S` en la ordenación topológica son inalcanzables.

#### 14.5.4 Una aplicación: análisis del camino crítico

El análisis del camino crítico se utiliza para planificar tareas asociadas con un proyecto.

Un grafo de nodos de actividad representa las actividades como vértices y las relaciones de precedencia como aristas.

Un uso importante de los grafos acíclicos es el *análisis del camino crítico*, una forma de análisis utilizada para planificar las tareas asociadas con un proyecto. El grafo mostrado en la Figura 14.33 proporciona un ejemplo. Cada vértice representa una actividad que debe completarse, junto con el tiempo necesario para completarla. El grafo se denomina por tanto, *grafo de nodos de actividad*, en el que los vértices representan actividades y las aristas representan relaciones de precedencia. Una arista  $(v, w)$  indica que la actividad  $v$  tiene que completarse antes de que la actividad  $w$  pueda comenzar, lo que implica que el grafo tiene que ser acíclico. Asumimos que cualesquier actividades que no dependan (directa o indirectamente) la una de la otra pueden realizarse en paralelo, utilizándose diferentes servidores.

Este tipo de grafo podría ser utilizado (y frecuentemente lo es) para modelar proyectos de construcción. Hay dos preguntas importantes que deben responderse. En primer lugar, ¿cuál es el momento más temprano de terminación de un proyecto? La respuesta, como muestra el grafo, es 10 unidades de tiempo –requeridas a lo largo del camino  $A, C, F, H$ . En segundo lugar, ¿Qué actividades pueden ser retrasadas, y durante cuánto tiempo, sin afectar al tiempo mínimo de terminación? Por ejemplo, retardar cualquiera de las actividades  $A, C, F$  o  $H$  haría que el tiempo de terminación fuera superior a las 10 unidades de tiempo. Sin embargo, la actividad  $B$  es menos crítica y puede retrasarse hasta 2 unidades temporales sin afectar al momento de terminación del proyecto.

Para realizar estos cálculos, convertimos el grafo de nodos de actividad en un *grafo de nodos de sucesos*, en el que cada suceso se corresponde con la terminación de una actividad y de todas sus actividades dependientes.

Un grafo de nodos de sucesos está compuesto por vértices de sucesos que se corresponden con la terminación de una actividad y de todas sus actividades dependientes.

Los sucesos alcanzables desde un nodo  $v$  en el grafo de nodos de sucesos no puede comenzar hasta que se haya completado el suceso  $v$ . Este grafo se puede construir automáticamente o de forma manual (a partir del grafo de nodos de actividad). Puede que sea necesario insertar aristas y vértices ficticios para evitar introducir falsas dependencias (o falsas carencias de dependencia). En la Figura 14.34 se muestra el grafo de nodos de sucesos correspondiente al grafo de nodos de actividad de la Figura 14.33.

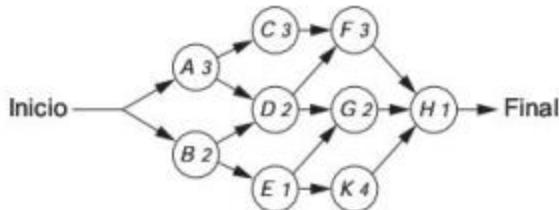


Figura 14.33 Un grafo de nodos de actividad.

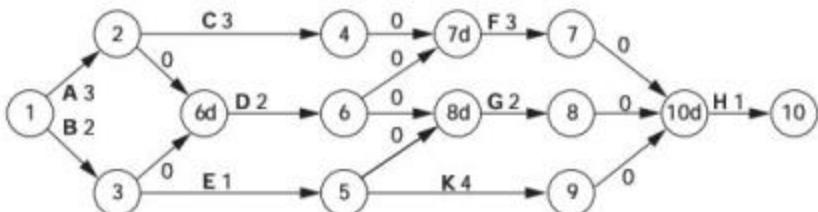


Figura 14.34 Un grafo de nodos de sucesos.

Para encontrar el instante más temprano de terminación del proyecto, simplemente necesitamos determinar la longitud del camino *más largo* que va del primer suceso al último. Para grafos de tipo general, el problema del camino más largo no suele tener sentido, debido a la posibilidad de que exista un *ciclo de coste positivo*, que es equivalente a un ciclo de coste negativo en los problemas de determinación del camino más corto. Si existe algún ciclo de coste positivo, podríamos tratar de determinar el camino simple más largo. Sin embargo, no se conoce ninguna solución satisfactoria a este problema. Afortunadamente, el grafo de nodos de sucesos es acíclico, por lo que no necesitamos preocuparnos por los ciclos. Podemos adaptar fácilmente el algoritmo del camino más corto para calcular el instante más temprano de terminación para todos los nodos del grafo. Si  $EC_i$  es el instante más temprano de terminación para el nodo  $i$ , las reglas aplicables son

$$EC_i = 0 \quad \text{y} \quad EC_w = \max_{\{v, w\} \in E} (EC_v + c_{v,w})$$

La Figura 14.35 muestra el instante más temprano de terminación para cada suceso de nuestro grafo de nodos de sucesos de ejemplo. También podemos calcular el instante más tardío,  $LC$ , en que puede finalizar cada suceso sin afectar al instante final de terminación. Las fórmulas para hacer esto son

$$LC_N = EC_N \quad \text{y} \quad LC_v = \min_{\{v, w\} \in E} (LC_w - c_{v,w})$$

Estos valores se pueden calcular en un tiempo lineal, manteniendo para cada vértice una lista de todos los vértices adyacentes y precedentes. Los instantes más tempranos de terminación se calculan para los vértices según su ordenación topológica, y los instantes más tardíos de terminación se calculan mediante una ordenación topológica inversa. Los instantes de terminación más tardíos se muestran en la Figura 14.36.

Las aristas muestran qué actividad debe completarse para avanzar de un vértice al siguiente. El instante más temprano de terminación es el camino más largo.

También se puede calcular el momento más tardío en el que un suceso puede terminar sin retardar el proyecto.

La tolerancia temporal es la cantidad de tiempo que una actividad se puede retrasar sin retrasar la terminación del proyecto global.

Las actividades con tolerancia cero son críticas y no se pueden retrasar. Un camino formado por aristas con tolerancia cero será un *caminó crítico*.

La *tolerancia temporal* para cada arista del grafo de nodos de sucesos es la cantidad de tiempo que puede retrasarse la terminación de la actividad correspondiente sin retrasar la finalización del proyecto global o

$$\text{Tolerancia}_{(v, w)} = LC_w - EC_v - c_{v, w}$$

La Figura 14.37 muestra la tolerancia (como tercer atributo) de cada actividad en el grafo de nodos de sucesos. Para cada nodo, el número de la parte superior es el instante más temprano de terminación y el número de la parte inferior es el instante más tardío de terminación.

Algunas actividades tienen una tolerancia cero. Se trata de actividades críticas que se deben finalizar de acuerdo con lo programado. Un camino compuesto enteramente por aristas con tolerancia igual a cero será un *caminó crítico*.

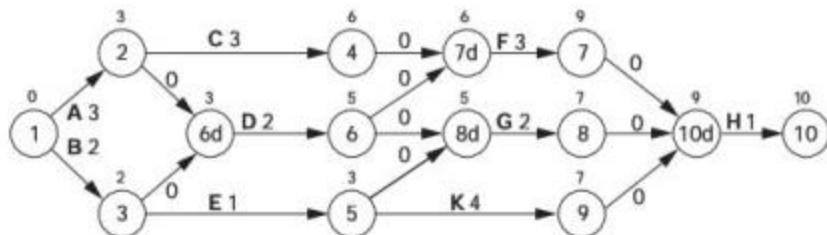


Figura 14.35 Instantes más tempranos de terminación.

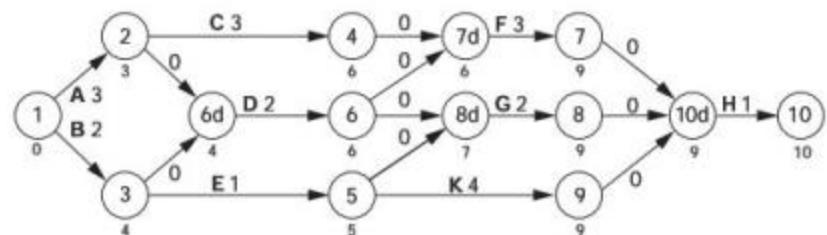


Figura 14.36 Instantes más tardíos de terminación.

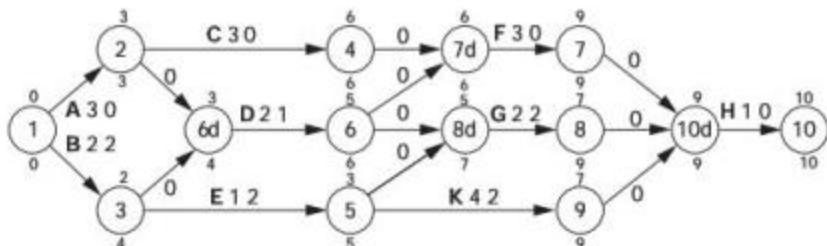


Figura 14.37 Instante más temprano de terminación, instante más tardío de terminación y tolerancia (atributo adicional de la arista).

## Resumen

En este capítulo hemos mostrado cómo se pueden utilizar los grafos para modelar muchos problemas del mundo real y, en particular, para calcular el camino más corto en una amplia variedad de circunstancias. Muchos de los grafos con que nos encontramos suelen ser bastante dispersos, por lo que es importante seleccionar estructuras de datos apropiadas para implementarlos.

Para los grafos no ponderados, el camino más corto puede calcularse en un tiempo lineal, utilizando la búsqueda en anchura. Para grafos con ponderación positiva hace falta un tiempo ligeramente mayor, utilizando el algoritmo de Dijkstra y una cola con prioridad eficiente. Para grafos con ponderación negativa, el problema es más difícil. Finalmente, para los grafos acíclicos, el tiempo de ejecución pasa a ser de nuevo lineal con la ayuda de una ordenación topológica.

La Figura 14.38 resume dichas características para estos algoritmos.



### Conceptos clave

**algoritmo de Bellman-Ford** Un algoritmo que se utiliza para resolver el problema del camino más corto con ponderaciones negativas. (541)

**algoritmo de Dijkstra** Un algoritmo que se emplea para resolver el problema del camino más corto con ponderaciones positivas. (535)

**algoritmos de un único origen** Algoritmos que calculan los caminos más cortos desde un cierto punto inicial hasta todos los vértices de un grafo. (522)

**análisis del camino crítico** Una forma de análisis utilizada para planificar tareas asociadas con un proyecto. (548)

**búsqueda en anchura** Un procedimiento de búsqueda que procesa los vértices por capas: se evalúa primero los más próximos al inicio y en último lugar los más distantes. (530)

**camino** Una secuencia de vértices conectados mediante aristas. (516)

**camino simple** Un camino en el que todos los vértices son distintos, salvo porque el primer y el último vértices pueden coincidir. (549)

**ciclo** En un grafo dirigido, un camino que comienza y termina en el mismo vértice y que contiene al menos una arista. (516)

Tipo de grafo	Tiempo de ejecución	Comentarios
No ponderado	$O( E )$	Búsqueda en anchura
Ponderado, sin aristas negativas	$O( E  \log  V )$	Algoritmo de Dijkstra
Ponderado, aristas negativas	$O( E  \cdot  V )$	Algoritmo de Bellman-Ford
Ponderado, acíclico	$O( E )$	Usa ordenación topológica

Figura 14.38 Tiempos de ejecución de caso peor para diversos algoritmos para grafos.

**ciclo de coste negativo** Un ciclo cuyo coste es inferior a cero y que hace que la mayoría de los caminos, si no todos ellos, estén indefinidos, porque podemos recorrer el ciclo un número arbitrario de veces y obtener una longitud ponderada de camino arbitrariamente negativa. (541)

**ciclo de coste positivo** En un problema de determinación del camino más largo es el equivalente de un ciclo de coste negativo en un problema de determinación del camino más corto. (549)

**coste (peso) de la arista** El tercer componente de una arista, que mide el coste de recorrer la arista. (515)

**grado entrante** El número de aristas entrantes en un vértice. (543)

**grafo** Un conjunto de vértices y un conjunto de aristas que conectan esos vértices. (515)

**grafo de nodos de actividad** Un grafo en el que los vértices representan actividades y las aristas representan relaciones de precedencia. (548)

**grafo de nodos de sucesos** Un grafo que está compuesto por vértices de suceso que se corresponden con la terminación de una actividad y de todas sus actividades dependientes. Las aristas muestran qué actividades deben completarse para pasar de un vértice al siguiente. El instante más temprano de terminación es el camino más largo. (548)

**grafo dirigido** Un grafo en el que las aristas son parejas ordenadas de vértices. (515)

**grafo dirigido acíclico** Un tipo de grafo dirigido que no tiene ningún ciclo. (516)

**grafos densos y dispersos** Un grafo denso tiene un gran número de aristas (generalmente cuadrático). Los grafos típicos no son densos, sino dispersos. (517)

**listas de adyacencia** Una matriz de listas utilizadas para representar un grafo, empleando un espacio lineal. (518)

**longitud de camino** El número de aristas que forman un camino. (516)

**longitud de camino no ponderada** El número de aristas que forman un camino. (516)

**longitud de camino ponderada** La suma de los costes de las aristas que forman un camino. (516)

**matriz de adyacencia** Una representación matricial de un grafo que utiliza un espacio cuadrático. (517)

**ordenación topológica** Un proceso que ordena los vértices en un grafo dirigido acíclico, de forma tal que si existe un camino desde  $u$  hasta  $v$ , entonces  $v$  aparece después de  $u$  en la ordenación. Un grafo que tenga un ciclo no puede tener una ordenación topológica. (543)

**tolerancia temporal** La cantidad de tiempo que una actividad se puede retrasar sin retrasar la terminación del proyecto global. (550)

**vértices adyacentes** El vértice  $w$  es adyacente al vértice  $v$  si existe una arista desde  $v$  a  $w$ . (515)



## Errores comunes

1. Un error común es olvidarse de garantizar que el grafo de entrada satisfaga los requisitos para el algoritmo que se esté utilizando (es decir, que sea cíclico o que tenga ponderación positiva).
2. Para `Path`, la función de comparación compara únicamente el miembro de datos `cost`. Si se utiliza el miembro de datos `dest` para regular la función de comparación, el algoritmo podría parecer que funciona para grafos de pequeño tamaño, pero para grafos de mayor tamaño sería incorrecto y proporcionaría respuestas ligeramente subóptimas. Sin embargo, nunca llegaría a generar como respuesta un camino que no exista. Por ello, este error es difícil de localizar.
3. El algoritmo del camino más corto para grafos con ponderaciones negativas debe incluir una comprobación de la existencia de ciclos negativos; en caso contrario, podría estar ejecutándose eternamente.



## Internet

Todos los algoritmos de este capítulo están disponibles en línea en un único archivo. La clase `Vertex` tiene un miembro de datos adicional que se utiliza en la implementación alternativa del algoritmo de Dijkstra mostrado en la Sección 23.2.3.

### **Graph.java**

Contiene todo en un archivo con la rutina `main` simple mostrada en la Figura 14.14.



## Ejercicios

### EN RESUMEN

- 14.1** Determinar el camino ponderado más corto desde  $V_2$  a todos los restantes vértices del grafo mostrado en la Figura 14.1.
- 14.2** En la Figura 14.5, invierta la dirección de las aristas  $(B, E)$  y  $(C, A)$ . Muestre los cambios resultantes en la figura y el resultado de ejecutar el algoritmo de ordenación topológica.
- 14.3** Determine el camino más corto sin ponderación desde  $V_3$  a todos los restantes vértices del grafo mostrado en la Figura 14.1.
- 14.4** Suponga que añadimos al final de la entrada de la Figura 14.5 las aristas  $(C, B)$  con un coste de 5 y  $(B, F)$  con un coste de 7. Muestre los cambios resultantes en la figura y calcule de nuevo el camino más corto que sale del vértice  $A$ .

## EN TEORÍA

- 14.5** Explique cómo modificar el algoritmo del camino más corto sin ponderación de modo que, si hay más de un camino mínimo (en términos del número de aristas), se rompa el empate en favor del camino con un peso total más pequeño.
- 14.6** Explique cómo modificar el algoritmo de Dijkstra para obtener un recuento del número de caminos mínimos diferentes existentes entre  $v$  y  $w$ .
- 14.7** Muestre cómo evitar la inicialización cuadrática inherente a las matrices de adyacencia, al mismo tiempo que se mantiene el acceso en tiempo constante a cualquier arista.
- 14.8** Suponga que, en un grafo dirigido, el coste del camino es la suma de los costes de las aristas que forman un camino MÁS el número de aristas del camino. Muestre cómo resolver esta versión del problema del camino más corto.
- 14.9** Detalle un ejemplo de situación en la que el algoritmo de Dijkstra proporcione la respuesta incorrecta en presencia de una arista negativa pero sin que existan ciclos de coste negativo.
- 14.10** Explique cómo modificar el algoritmo de Dijkstra de modo que, si hay más de un camino mínimo desde  $v$  hasta  $w$ , se elija un camino con el menor número de aristas.
- 14.11** Proporcione un algoritmo de tiempo lineal para determinar el camino ponderado más largo en un grafo acíclico. ¿Puede ampliarse el algoritmo para que sea aplicable a grafos que tengan ciclos?
- 14.12** Considere el siguiente algoritmo para resolver el problema del camino más corto con ponderaciones negativas: sume una constante  $c$  al coste de la arista, eliminando así las aristas de coste negativo; calcule el camino más corto en el nuevo grafo y después utilice ese resultado en el original. ¿Qué es lo que no funciona con este algoritmo?
- 14.13** Demuestre que si los pesos de las aristas son exclusivamente 0 o 1, se puede implementar el algoritmo de Dijkstra en un tiempo lineal utilizando una cola de doble terminación (Sección 16.5).
- 14.14** Demuestre la corrección del algoritmo del camino más corto con ponderaciones negativas. Para hacerlo, demuestre que cuando el foco visita el vértice  $v$  por  $i$ -ésima vez, el valor de  $D_v$  es la longitud del camino ponderado más corto compuesto por  $i$  o menos aristas.
- 14.15** Sea  $G$  un grafo (dirigido) y sean  $u$  y  $v$  dos vértices distintos de  $G$ . Demuestre la verdad o falsedad de cada una de las siguientes afirmaciones.
- Si  $G$  es acíclico, se puede añadir al grafo la arista  $(u, v)$  o  $(v, u)$  sin crear un ciclo.
  - Si es imposible añadir  $(u, v)$  o  $(v, u)$  a  $G$  sin crear un ciclo, entonces es que  $G$  ya tiene un ciclo.
- 14.16** Para cualquier camino de un grafo, el *coste de cuello de botella* está dado por el peso de la arista más corta del camino. Por ejemplo, en la Figura 14.4, el coste de cuello de botella del camino,  $E, D, B$  es 23 y el coste de cuello de botella del camino

*E, D, C, A, B* es 10. El problema del cuello de botella máximo consiste en encontrar el camino entre dos vértices especificados que tenga un coste de cuello de botella máximo. Así el camino de cuello de botella máximo de *E* y *B* sería el camino *E, D, B*. Proporcione un algoritmo eficiente para resolver el problema del cuello de botella máximo.

- 14.17** Suponga que todos los costes de las aristas de un grafo son 1 o 2. Demuestre que en esas condiciones el algoritmo de Dijkstra se puede implementar de modo que se ejecute en un tiempo lineal.

### EN LA PRÁCTICA

- 14.18** En este capítulo afirmamos que, para la implementación de los algoritmos de grafos que se ejecutan con grandes conjuntos de datos de entrada, las estructuras de datos son cruciales para garantizar un rendimiento razonable. Para cada una de las siguientes instancias en las que se utiliza una estructura de datos o un algoritmo inadecuados, proporcione un análisis O mayúscula del resultado y compare el rendimiento real con los algoritmos y estructuras de datos presentados en el texto. Implemente solo un cambio cada vez. Debería ejecutar sus pruebas con un grafo aleatorio relativamente disperso y razonablemente grande. A continuación, haga lo siguiente.

- Cuando se lee una arista, determine si ya se encuentra en el grafo.
- Implemente el “diccionario” utilizando una exploración secuencial de la tabla de vértices.
- Implemente la cola utilizando el algoritmo del Ejercicio 6.26 (que debería afectar al algoritmo del camino más corto sin ponderación).
- En el algoritmo del camino más corto sin ponderación, implemente la búsqueda del vértice de coste mínimo en forma de una exploración secuencial de la tabla de vértices.

### PROYECTOS DE PROGRAMACIÓN

- 14.19** Una palabra puede transformarse en otra mediante sustitución de un único carácter. Suponga que existe un diccionario de palabras de cinco letras. Proporcione un algoritmo para determinar si una palabra *A* se puede transformar en una palabra *B* mediante una serie de sustituciones de un único carácter *y*, en caso afirmativo, imprimir la correspondiente secuencia de palabras. Por ejemplo, *b1eed* se convierte en *blood* mediante la secuencia *bleed*, *blend*, *blond*, *blood*.

- 14.20** La entrada es una colección de monedas nacionales y de sus tasas de cambio. ¿Existe una secuencia de cambios que permita generar dinero instantáneamente? Por ejemplo, si las monedas son *X*, *Y* y *Z* y las tasas de cambio son  $1X$  es igual a  $2Y$ ,  $1Y$  es igual a  $2Z$  y  $1X$  es igual a  $3Z$ , entonces  $300Z$  comprarían  $100X$ , que a su vez permitirían comprar  $200Y$ , que a su vez comprarían  $400Z$ . Así habríamos obtenido un beneficio del 33 por ciento.

- 14.21** Un grafo dirigido está fuertemente conectado si existe un camino desde cada vértice hasta cualquier otro vértice. Haga lo siguiente:

- Seleccione cualquier vértice  $S$ . Demuestre que, si el grafo está fuertemente conectado, un algoritmo de determinación del camino más corto declarará que todos los nodos son alcanzables desde  $S$ .
- Demuestre que, si el grafo está fuertemente conectado y luego se invierten las direcciones de todas las aristas y se ejecuta un algoritmo de determinación del camino más corto a partir de  $S$ , todos los nodos serán alcanzables desde  $S$ .
- Demuestre que las comprobaciones de los apartados (a) y (b) son suficientes para decidir si un grafo está fuertemente conectado (es decir, un grafo que pase ambas comprobaciones deberá estar fuertemente conectado).
- Escriba un programa que compruebe si un grafo está fuertemente conectado. ¿Cuál es el tiempo de ejecución de su algoritmo?

*Explique cómo puede resolverse cada uno de los siguientes problemas aplicando un algoritmo de determinación del camino más corto. Después diseñe un mecanismo para representar una entrada y escriba un programa que resuelva el problema.*

**14.22** La entrada es una lista de puntuaciones de partidos de la liga (y no existen empates). Si todos los equipos tienen al menos una victoria y una derrota, podemos “demostrar” con carácter general, mediante un sencillo argumento de transitividad, que cualquier equipo es mejor que cualquier otro. Por ejemplo, en una liga con seis equipos en la que todos jueguen tres partidos, suponga que obtenemos los siguientes resultados:  $A$  gana a  $B$  y a  $C$ ,  $B$  gana a  $C$  y a  $F$ ,  $C$  gana a  $D$ ,  $D$  gana a  $E$ ,  $E$  gana a  $A$ ; y  $F$  gana a  $D$  y a  $E$ . Entonces, podemos demostrar que  $A$  es mejor que  $F$  porque  $A$  gana a  $B$  que a su vez gana a  $F$ . De forma similar, podemos demostrar que  $F$  es mejor que  $A$  porque  $F$  gana a  $E$  y  $E$  gana a  $A$ . Dada una lista de puntuaciones de partidos y dos equipos  $X$  e  $Y$ , encuentre una prueba (si existe) de que  $X$  es mejor que  $Y$  o indique que no puede encontrarse una prueba así.

**14.23** Un estudiante necesita aprobar una serie de cursos para graduarse y esos cursos tienen una serie de prerequisitos que hay que cumplir. Suponga que se ofrecen todos los cursos cada semestre y que el estudiante se puede matricular de un número ilimitado de cursos. A partir de una lista de cursos y sus prerequisitos, calcule un plan que requiera el número mínimo de semestres.

**14.24** Modifique el Ejercicio 14.19 para permitir palabras de longitud arbitraria y transformaciones en las que se añada o se elimine un carácter. El coste de añadir o eliminar un carácter es igual a la longitud de la cadena más larga en la transformación, mientras que una sustitución de un único carácter solo cuesta 1. Por tanto, `ark`, `as`, `was` es una transformación válida de `ark` a `was` y el coste es 7 ( $1+3+3$ ).

**14.25** Suponga que tiene un grafo en el que cada vértice representa una computadora y en el que cada arista representa una conexión directa entre dos computadoras. Cada arista  $(v, w)$  tiene un peso  $p_{v,w}$  que representa la probabilidad de que una transferencia de red entre  $v$  y  $w$  tenga éxito ( $0 < p_{v,w} \leq 1$ ). Escriba un programa que encuentre la forma más fiable de transferir datos desde una computadora inicial designada  $sa$  todas las demás computadoras de la red.

**14.26** La entrada es un laberinto bidimensional con paredes y el problema consiste en atravesar el laberinto, utilizando la ruta más corta desde la esquina superior izquierda hasta la esquina inferior derecha. Se pueden derribar paredes, pero cada pared que se derriba implica una penalización  $p$  (que se especifica como parte de la entrada).

**14.27** El objeto del juego *Kevin Bacon* consiste en vincular a un actor con Kevin Bacon a través de películas en las que hayan trabajado juntos. El número mínimo de vínculos es el *número de Bacon* de un actor. Por ejemplo, Tom Hanks tiene un número de Bacon de 1. Trabajó en Apolo 13 con Kevin Bacon. Sally Field tiene un número de Bacon de 2 porque trabajó en *Forest Gump* con Tom Hanks, que a su vez trabajó en Apolo 13 con Kevin Bacon. Casi todos los actores más conocidos tienen un número de Bacon de 1 o 2. Suponga que tiene una lista completa de actores con sus papeles y haga lo siguiente.

- Explique cómo encontrar el número de Bacon de un actor.
- Explique cómo encontrar el actor con el mayor número de Bacon.
- Explique cómo encontrar el número mínimo de saltos de conexión entre dos actores arbitrarios.



## Referencias

El uso de listas de adyacencia para representar grafos fue defendido por primera vez en [3]. El algoritmo del camino más corto de Dijkstra fue descrito originalmente en [2]. El algoritmo para costes de arista negativos está tomado de [1]. En [6] se describe una comprobación más eficiente de terminación; allí se muestra también cómo las estructuras de datos desempeñan un papel importante en una amplia gama de algoritmos de la teoría de grafos. El algoritmo de ordenación topológica está tomado de [4]. En [5] se presentan muchas aplicaciones de la vida real para los algoritmos de grafos, junto con referencias para obtener más información.

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, MA, 1993.
6. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1985.



# Implementaciones

Capítulo 15 Clases internas e  
implementación de ArrayList

Capítulo 16 Pilas y colas

Capítulo 17 Listas enlazadas

Capítulo 18 Árboles

Capítulo 19 Árboles de búsqueda binaria

Capítulo 20 Tablas hash



## Clases internas e implementación de ArrayList

En este capítulo iniciamos el análisis de la implementación de las estructuras de datos estándar. Una de las estructuras de datos más simple es `ArrayList`, que forma parte de la API de Colecciones. En la Parte Uno (específicamente en la Figura 3.17 y la Figura 4.24) ya hemos visto esqueletos de la implementación, así que en este capítulo vamos a concentrarnos en los detalles de cómo implementar la clase completa con los iteradores asociados. Al hacerlo, utilizaremos una interesante creación sintáctica de Java, la *clase interna*. Hablamos de la clase interna en este capítulo, en lugar de en la Parte Uno (donde se han presentado otros elementos sintácticos), porque consideramos la clase interna como una técnica de implementación de Java, más que como una característica fundamental del lenguaje.

En este capítulo veremos

- Los usos y la sintaxis de la clase interna.
- Una implementación de una nueva clase denominada `AbstractCollection`.
- Una implementación de la clase `ArrayList`.

### 15.1 Iteradores y clases anidadas

Comenzamos repasando la implementación de un iterador simple que hemos descrito por primera vez en la Sección 6.2. Recuerde que allí definimos una interfaz de iterador simple, que se asemeja al `Iterator` estándar (no genérico) de la API de Colecciones; esta interfaz se muestra en la Figura 15.1.

Después definimos dos clases: el contenedor y su iterador. Cada clase contenedora es responsable de proporcionar una implementación de la interfaz iteradora. En nuestro caso, la interfaz iteradora es proporcionada por la clase `MyContainerIterator`, mostrada en la Figura 15.2. La clase `MyContainer` mostrada en la Figura 15.3 proporciona un método factoría que crea una instancia de `MyContainerIterator` y devuelve esta instancia utilizando el tipo de interfaz `Iterator`. La Figura 15.4 proporciona una rutina `main` que ilustra el uso de la combinación contenedor/iterador. Las Figuras 15.1 a 15.4 simplemente replican las Figuras 6.5 a 6.8 contenidas en las explicaciones originales sobre iteradores de la Sección 6.2.

```

1 package weiss.ds;
2
3 public interface Iterator
4 {
5     boolean hasNext( );
6     Object next( );
7 }

```

**Figura 15.1** La interfaz Iterator de la Sección 6.2.

```

1 // Una clase iteradora que recorre un contenedor MyContainer.
2
3 package weiss.ds;
4
5 class MyContainerIterator implements Iterator
6 {
7     private int current = 0;
8     private MyContainer container;
9
10    MyContainerIterator( MyContainer c )
11        { container = c; }
12
13    public boolean hasNext( )
14        { return current < container.size; }
15
16    public Object next( )
17        { return container.items[ current++ ]; }
18 }

```

**Figura 15.2** Implementación de MyContainerIterator de la Sección 6.2.

```

1 package weiss.ds;
2
3 public class MyContainer
4 {
5     Object [ ] items;
6     int size;
7
8     public Iterator iterator( )
9         { return new MyContainerIterator( this ); }
10
11    // No se muestran otros métodos.
12 }

```

**Figura 15.3** La clase MyContainer de la Sección 6.2.

```
1 public static void main( String [ ] args )
2 {
3     MyContainer v = new MyContainer( );
4
5     v.add( "3" );
6     v.add( "2" );
7
8     System.out.println( "Container contents: " );
9     Iterator itr = v.iterator( );
10    while( itr.hasNext( ) )
11        System.out.println( itr.next( ) );
12 }
```

Figura 15.4 El método `main` para ilustrar el diseño de iterador de la Sección 6.2.

Este diseño oculta la implementación de la clase iteradora, porque `MyContainerIterator` no es una clase pública. Por tanto, el usuario está obligado a programar según la interfaz `Iterator` y no tiene acceso a los detalles de cómo está implementado el iterador –el usuario ni siquiera puede declarar objetos de tipo `weiss.ds.MyContainerIterator`. Sin embargo, sigue dejando expuestos más detalles de lo que normalmente nos gustaría. En la clase `MyContainer`, los datos no son privados, y la correspondiente clase iteradora, aunque no es pública, sigue teniendo visibilidad de paquete. Podemos resolver ambos problemas utilizando clases anidadas: lo que hacemos es simplemente mover la clase iteradora dentro de la clase contenedora. En ese punto, la clase iteradora pasará a ser un miembro de la clase contenedora, y podrá ser así declarada como clase privada y sus métodos podrán acceder a los datos privados de `MyContainer`. El código revisado se muestra en la Figura 15.5, con un único cambio de carácter estilístico, consistente en renombrar `MyContainerIterator` como `LocalIterator`. No son necesarios más cambios; sin embargo, el constructor `LocalIterator` puede hacerse privado y se podrá seguir invocando desde `MyContainer`, ya que `LocalIterator` forma parte de `MyContainer`.

## 15.2 Iteradores y clases internas

En la Sección 15.1 hemos utilizado una clase anidada para ocultar aun más los detalles. Además de las clases anidadas, Java proporciona las denominadas clases internas. Una *clase interna* es similar a una clase anidada, en el sentido de que se trata de una clase dentro de otra clase y de que es tratada como miembro de la clase externa en todo lo que se refiere a visibilidad. Una clase interna se declara utilizando la misma sintaxis que una clase anidada, pero con la diferencia de que no se trata de una clase estática. En otras palabras, en la declaración de la clase interna falta el cualificador `static`.

Antes de entrar en los detalles específicos de la clase interna, examinemos el problema que este tipo de clases pretende resolver. La Figura 15.6 ilustra la relación entre las clases iteradora y contenedora escritas en la sección anterior. Cada instancia de `LocalIterator` mantiene una referencia al conte-

Una clase interna es similar a una clase anidada, en el sentido de que se trata de una clase contenida dentro de otra clase y que se emplea la misma sintaxis que para una clase anidada, con la única diferencia de que no es una clase `static`. Una clase interna siempre contiene una referencia implícita al objeto externo que la ha creado.

```
1 package weiss.ds;
2
3 public class MyContainer
4 {
5     private Object [ ] items;
6     private int size = 0;
7     // No se muestran otros métodos de MyContainer
8
9     public Iterator iterator( )
10    { return new LocalIterator( this ); }
11
12 // La clase iteradora como clase anidada
13 private static class LocalIterator implements Iterator
14 {
15     private int current = 0;
16     private MyContainer container;
17
18     private LocalIterator( MyContainer c )
19     { container = c; }
20
21     public boolean hasNext( )
22     { return current < container.size; }
23
24     public Object next( )
25     { return container.items[ current++ ]; }
26 }
27 }
```

---

Figura 15.5 Diseño de iterador utilizando una clase anidada.

nedor a través del cual está iterando, junto con una noción de cuál es la posición actual del iterador. La relación que tenemos es que cada `LocalIterator` debe estar asociado con exactamente una instancia de `MyContainer`. Es imposible que la referencia `container` en cualquier iterador sea `null`, y la existencia del iterador no tienen ningún sentido sin conocer qué objeto `MyContainer` ha provocado su creación.

Puesto que sabemos que `itr1` debe estar asociada a un iterador y solo a un iterador, parece que la expresión `container.items` es redundante: si el iterador pudiera recordar el contenedor que lo ha construido, no tendríamos por qué preocuparnos nosotros de recordarlo. Y si lo recordara, cabría esperar que al hacer referencia a `items` desde dentro de `LocalIterator`, puesto que `LocalIterator` no tiene ningún campo `items`, el compilador (y el sistema de tiempo de ejecución) fueran lo suficientemente inteligentes como para deducir que estamos hablando del campo `items` del objeto `MyContainer` que provocó la construcción de este `LocalIterator` concreto. Esto es lo que hace exactamente una clase interna, y lo que la distingue de una clase anidada.

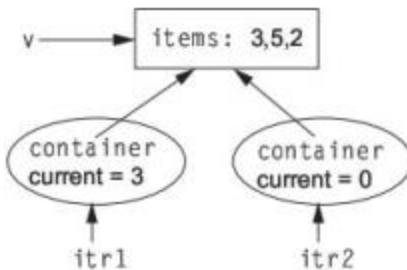


Figura 15.6 Relación iterador/contenedor.

La gran diferencia entre una clase interna y una clase anidada es que, cuando se construye una instancia de un objeto de la clase interna, existe una referencia implícita al objeto de la clase externa que ha provocado su construcción. Esto implica que un objeto de una clase interna no puede existir sin un objeto de una clase externa al que estar asociado, con la única excepción de si se la declara en un método estático (porque las clases locales y anónimas son técnicamente hablando clases internas), lo cual es un detalle del que hablaremos posteriormente.

Si el nombre de la clase externa es `Outer`, entonces la referencia implícita es `Outer.this`. Por tanto, si se declarara `LocalIterator` como una instancia de una clase interna (es decir, si eliminamos la palabra clave `static`), entonces se podría utilizar la referencia `MyContainer.this` para sustituir a la referencia `container` que el iterador almacena. La Figura 15.7 ilustra que la estructura sería idéntica. En la Figura 15.8 se muestra una clase revisada.

En la implementación revisada, observe que `LocalIterator` ya no dispone de ninguna referencia explícita a un `MyContainer`, y observe también que su constructor ya no es necesario, puesto que solo inicializaba la referencia `MyContainer`. Por último, la Figura 15.9 ilustra que, al igual que el uso de `this` es opcional en un método de instancia, la referencia `Outer.this` también es opcional si no existe ningún conflicto de nombres. Por tanto, `MyContainer.this.size` puede abreviarse a `size`, siempre que no haya ninguna otra variable denominada `size` con un ámbito más restringido.

Las clases locales y las clases anónimas no especifican si son de tipo `static`, y siempre se las considera técnicamente como clases internas. Sin embargo, si una de esas clases se declara en un

La gran diferencia entre una clase interna y una clase anidada es que, cuando se construye una instancia de un objeto de la clase interna, existe una referencia implícita al objeto de la clase externa que ha provocado su construcción.

Si el nombre de la clase externa es `Outer`, entonces la referencia implícita es `Outer.this`.

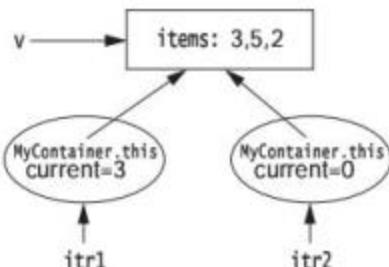


Figura 15.7 Iterador/contenedor con clases internas.

método estático, carece de referencia externa implícita (y por tanto se comporta como una clase anidada), mientras que si se la declara dentro de un método de instancia, su referencia externa implícita será el invocador del método.

```

1 package weiss.ds;
2
3 public class MyContainer
4 {
5     private Object [ ] items;
6     private int size = 0;
7
8     // No se muestran otros métodos de MyContainer
9
10    public Iterator iterator( )
11        { return new LocalIterator( ); }
12
13    // La clase iteradora como clase interna.
14    private class LocalIterator implements Iterator
15    {
16        private int current = 0;
17
18        public boolean hasNext( )
19            { return current < MyContainer.this.size; }
20
21        public Object next( )
22            { return MyContainer.this.items[ current++ ]; }
23    }
24 }
```

**Figura 15.8** Diseño de un iterador utilizando una clase interna.

```

1 // La clase iteradora como clase interna
2 private class LocalIterator implements Iterator
3 {
4     private int current = 0;
5
6     public boolean hasNext( )
7         { return current < size; }
8
9     public Object next( )
10        { return items[ current++ ]; }
11 }
```

**Figura 15.9** Clase interna: Outer.this puede ser opcional.

La adición de las clases internas requiere un significativo conjunto de reglas, muchas de las cuales tratan de regular los casos especiales del lenguaje y las prácticas de codificación dudosas. Por ejemplo, suponga que nos olvidamos de todo por un momento e imaginamos que `LocalIterator` es pública. Vamos a hacerlo únicamente para ilustrar las complicaciones con las que se enfrentan los diseñadores del lenguaje a la hora de añadir una nueva característica a este. Con esta suposición, el tipo del iterador será `MyContainer.LocalIterator`, y puesto que es visible, cabría esperar que

```
MyContainer.LocalIterator itr = new MyContainer.LocalIterator();
```

fuerá legal, ya que como todas las clases, dispone de un constructor público predeterminado con cero parámetros. Sin embargo, esta operación no puede llegar a funcionar nunca, porque no hay ninguna manera de inicializar la referencia implícita. ¿A qué `MyContainer` estaría haciendo referencia `itr`? Necesitamos alguna sintaxis que no entre en conflicto con otras reglas del lenguaje. He aquí la regla: si existe un contenedor `c`, entonces `itr` podría construirse utilizando una compleja sintaxis inventada para este caso específico, en la que el objeto externo en cuestión invocara a `new`:

```
MyContainer.LocalIterator itr = c.new LocalIterator();
```

Observe que esto implica que un método factoría de instancia `this.new` sería legal y las abreviaturas al `new` más convencional serían visibles en un método factoría. Si el lector se descubre a sí mismo en algún momento empleando esta complicada sintaxis, probablemente es que tenga un mal diseño. En nuestro ejemplo, una vez que `LocalIterator` es privada, todo el problema se desvanece, y si `LocalIterator` no es privada, en realidad no existe ninguna razón para utilizar una clase interna.

Existen también otras reglas, algunas de las cuales son arbitrarias. Los miembros privados de la clase interna o anidada son públicos para la clase externa. Para acceder a cualquier miembro de una clase interna, la clase externa solo tiene que proporcionar una referencia a una instancia de la clase interna y utilizar el operador punto, como se hace normalmente con otras clases. Por ello, las clases internas y anidadas se consideran parte de la clase externa.

Tanto las clases internas como las anidadas pueden ser finales o pueden ser abstractas, o pueden ser interfaces (pero las interfaces son siempre estáticas, porque no pueden tener ningún dato, incluyendo las referencias implícitas), o bien pueden no ser ninguna de estas cosas. Las clases internas no pueden tener campos o métodos estáticos, salvo campos estáticos finales. Las clases internas sí que pueden tener interfaces o clases anidadas. Finalmente, al compilar el ejemplo anterior, podrá ver que el compilador genera un archivo de clase denominado `MyContainer$LocalIterator.class`, que habría que incluir en cualquier distribución que se efectúe a los clientes. En otras palabras, cada clase interna y anidada es una clase y dispone de su correspondiente archivo de clase. Las clases anónimas emplean números en lugar de nombres.

## 15.3 La clase AbstractCollection

Antes de implementar la clase `ArrayList`, observe que algunos de los métodos de la interfaz `Collection` pueden implementarse fácilmente en términos de otros. Por ejemplo, `isEmpty` se puede implementar fácilmente comprobando si el tamaño es 0. En lugar de hacer esto en `ArrayList`, `LinkedList`, y en todas las implementaciones concretas, sería preferible hacerlo una sola vez y utilizar el mecanismo de herencia para obtener `isEmpty`. Podríamos incluso sustituir `isEmpty` si resultara que para algunas colecciones existiera una forma más rápida de realizar

`AbstractCollection`  
implementa algunos de  
los métodos de la interfaz  
`Collection`.

la comprobación `isEmpty` que calcular el tamaño actual. Sin embargo, no podemos implementar `isEmpty` en la interfaz `Collection`; esto solo se puede hacer en una clase abstracta. Lo haremos por tanto en la clase `AbstractCollection`. Para simplificar las implementaciones, los programadores que diseñan nuevas clases de colecciones pueden ampliar la clase `AbstractCollection` en lugar de implementar la interfaz `Collection`. En las Figuras 15.10 a 15.12 se muestra una implementación de ejemplo de `AbstractCollection`.

```
1 package weiss.util;
2
3 /**
4  * AbstractCollection proporciona implementaciones predeterminadas para
5  * algunos de los métodos sencillos de la interfaz Collection.
6  */
7 public abstract class AbstractCollection<AnyType> implements Collection<AnyType>
8 {
9     /**
10      * Comprueba si esta colección está vacía.
11      * @return true si el tamaño de esta colección es cero.
12      */
13     public boolean isEmpty( )
14     {
15         return size( ) == 0;
16     }
17
18     /**
19      * Cambiar el tamaño de esta colección a cero.
20      */
21     public void clear( )
22     {
23         Iterator<AnyType> itr = iterator( );
24         while( itr.hasNext( ) )
25         {
26             itr.next( );
27             itr.remove( );
28         }
29     }
30
31     /**
32      * Añadir x a esta colección.
33      * Esta implementación predeterminada siempre genera una excepción.
34      * @param x el elemento que hay que añadir.
35      * @throws UnsupportedOperationException siempre.
36      */
37     public boolean add( AnyType x )
38     {
39         throw new UnsupportedOperationException( );
40     }
```

---

Figura 15.10 Implementación de ejemplo de `AbstractCollection` (parte 1).

```

41  /**
42   * Devuelve true si esta colección contiene x.
43   * Si x es null, devuelve false.
44   * (Este comportamiento puede no ser siempre apropiado.)
45   * @param x el elemento que hay que buscar.
46   * @return true si x no es null y se encuentra en
47   * esta colección.
48   */
49 public boolean contains( Object x )
50 {
51     if( x == null )
52         return false;
53
54     for( AnyType val : this )
55         if( x.equals( val ) )
56             return true;
57
58     return false;
59 }
60
61 /**
62  * Elimina un x no-null de esta colección.
63  * (Este comportamiento puede no ser siempre apropiado.)
64  * @param x el elemento que hay que eliminar.
65  * @return true si la eliminación tiene éxito.
66  */
67 public boolean remove( Object x )
68 {
69     if( x == null )
70         return false;
71
72     Iterator<AnyType> itr = iterator( );
73     while( itr.hasNext( ) )
74         if( x.equals( itr.next( ) ) )
75         {
76             itr.remove( );
77             return true;
78         }
79
80     return false;
81 }

```

Figura 15.11 Implementación de ejemplo de AbstractCollection (parte 2).

```

82  /**
83   * Obtiene una vista de matriz primitiva de la colección.
84   * @return la vista de matriz primitiva.
85   */
86  public Object [ ] toArray( )
87  {
88      Object [ ] copy = new Object[ size( ) ];
89      int i = 0;
90
91      for( AnyType val : this )
92          copy[ i++ ] = val;
93
94      return copy;
95  }
96
97  public <OtherType> OtherType [ ] toArray( OtherType [ ] arr )
98  {
99      int theSize = size( );
100
101     if( arr.length < theSize )
102         arr = ( OtherType [ ] ) java.lang.reflect.Array.newInstance(
103                         arr.getClass( ).getComponentType( ), theSize );
104     else if( theSize < arr.length )
105         arr[ theSize ] = null;
106
107     Object [ ] copy = arr;
108     int i = 0;
109
110     for( AnyType val : this )
111         copy[ i++ ] = val;
112
113     return copy;
114 }
115
116 /**
117  * Devuelve una representación en forma de cadena de esta colección.
118  */
119 public String toString( )
120 {
121     StringBuilder result = new StringBuilder( "[ " );
122

```

*Continúa*

Figura 15.12 Implementación de ejemplo de AbstractCollection (parte 3).

```

123     for( AnyType obj : this )
124         result.append( obj + " " );
125
126     result.append( "]" );
127
128     return result.toString();
129 }
130 }
```

**Figura 15.12 (Continuación).**

La API de Colecciones también define clases adicionales tales como `AbstractList`, `AbstractSequentialList` y `AbstractSet`. Hemos decidido no implementar esas clases, de acuerdo con nuestra intención de proporcionar un subconjunto simplificado de la API de Colecciones. Si está implementando por alguna razón sus propias colecciones y ampliando la API de Colecciones de Java, debería ampliar la clase abstracta que sea más específica.

En la Figura 15.10, podemos ver las implementaciones de `isEmpty`, `clear` y `add`. Los dos primeros métodos tienen implementaciones muy sencillas. Ciertamente, la implementación de `clear` es utilizable, ya que elimina todos los elementos de la colección, pero es posible que existan formas más eficientes de llevar a cabo la operación `clear`, dependiendo del tipo de colección que se esté manipulando. Por tanto, esta implementación de `clear` sirve como implementación predeterminada, aunque lo más probable es que sea sustituida. No existe ninguna forma razonable de proporcionar una implementación utilizable para `add`. Así que las dos alternativas que tenemos es hacer `add` abstracta (lo que claramente se puede hacer ya que `AbstractCollection` es abstracta) o proporcionar una implementación que genere una excepción de tiempo de ejecución. Hemos optado por hacer esto último, lo que se ajusta al comportamiento de `java.util`. (Adelantándonos a los acontecimientos, diremos que esta decisión también hace un poco más fácil crear la clase necesaria para expresar los valores de un mapa). La Figura 15.11 proporciona implementaciones predeterminadas de `contains` y `remove`. Ambas implementaciones utilizan una búsqueda secuencial, así que no son eficientes y habrá que sustituirlas por implementaciones razonables de la interfaz `Set`.

La Figura 15.12 contiene las implementaciones de dos métodos `toArray`. El `toArray` de cero parámetros es bastante fácil de implementar. El `toArray` de un parámetro hace uso de una característica de Java conocida con el nombre de reflexión para crear un objeto matriz que se ajuste al tipo de parámetro, en caso de que el parámetro no sea lo suficientemente grande como para almacenar la colección subyacente.

## 15.4 StringBuilder

La Figura 15.12 también muestra una razonable implementación con tiempo lineal de `toString`, utilizando un `StringBuilder` para evitar el tiempo de ejecución cuadrático. (`StringBuilder` fue añadido en Java 5 y es ligeramente más rápido que `StringBuffer`; es preferible para aplicaciones monohebra). Para ver por qué es necesario `StringBuilder`, considere el siguiente fragmento de código que construye un `String` con  $N$  letras A:

```
String result = "";
for( int i = 0; i < N; i++ )
    result += 'A';
```

Aunque sin duda este fragmento funciona perfectamente porque los objetos `String` son inmutables, cada llamada a `result += 'A'` se describe como `result = result + 'A'`, y una vez que vemos eso, resulta obvio que cada concatenación de objetos `String` crea un nuevo objeto `String`. A medida que vamos progresando a lo largo del bucle, estos objetos `String` van siendo cada vez más caros de crear. Podemos estimar el coste de la  $i$ -ésima concatenación `String` como  $i$ , por lo que el coste total es  $1 + 2 + 3 + \dots + N$ , o  $O(N^2)$ . Si  $N$  es 100.000, es fácil escribir el código y comprobar que el tiempo de ejecución es significativo. Sin embargo, una simple re-escritura

```
char [ ] theChars = new char[ N ];
for( int i = 0; i < N; i++ )
    theChars[ i ] = 'A';
String result = new String( theChars );
```

da como resultado un algoritmo de tiempo lineal que se ejecuta en un abrir y cerrar de ojos.

El uso de una matriz de caracteres funciona solo si conocemos el tamaño final del objeto `String`. En caso contrario, tenemos que utilizar algo como `ArrayList<char>`. Un `StringBuilder` es similar en cuanto a concepto a un `ArrayList<char>`, con duplicación del tamaño de la matriz, pero con nombres de método que son específicos para operaciones `String`. Utilizando un `StringBuilder`, el código tendría el siguiente aspecto

```
StringBuilder sb = new StringBuilder();
for( int i = 0; i < N; i++ )
    sb.append( 'A' );
String result = new String( sb );
```

Este código es de tiempo lineal y se ejecuta rápidamente. Algunas concatenaciones de objetos `String`, como las que se producen dentro de una misma expresión, son optimizadas por el compilador para evitar la creación repetida de objetos `String`. Pero si las concatenaciones están entremezcladas con otras instrucciones, como es el caso aquí, entonces a menudo es posible utilizar un `StringBuilder` para obtener un código más eficiente.

## 15.5 Implementación de ArrayList con un iterador

Las diversas clases `ArrayList` mostradas en la Parte Uno no estaban preparadas para iteradores. Esta sección proporciona una implementación de `ArrayList` que incluiremos en `weiss.util` y que incluye soporte para iteradores bidireccionales. Para que la cantidad de código sea manejable hemos eliminado la mayor parte de los comentarios javadoc. Puede encontrarlos en el código en línea.

La implementación se muestra en las Figuras 15.13 a 15.16. En la línea 3 vemos que `ArrayList` amplia la clase abstracta `AbstractCollection` y en la línea 4 `ArrayList` declara que implementa la interfaz `List`.

La matriz interna, `theItems`, y el tamaño de la colección, `theSize`, se declaran en las líneas 9 y 10, respectivamente. Más interesante es `modCount`, que se declara en la línea 11. `modCount` representa el número de modificaciones estructurales (operaciones `add`, `remove`) hechas a `ArrayList`. La idea es que cuando se construye un iterador, el iterador guarda este valor en su miembro de datos `expectedModCount`. Cuando se lleva a cabo cualquier operación del iterador, se compara el miembro `expectedModCount` del iterador con el miembro `modCount` del `ArrayList`, y si no concuerdan, puede generarse una excepción `ConcurrentModificationException`.

La línea 16 ilustra el constructor típico que realiza una copia superficial de los miembros en otra colección, simplemente recorriendo la colección e invocando el método `add`. El método `clear`, que se inicia en la línea 26, inicializa el `ArrayList` y puede invocarse desde el constructor. También reinicializa `theItems`, lo que permite al recolector de basura reclamar todos los objetos que ya no estén referenciados y que se encuentren en el `ArrayList`. Las restantes rutinas de la Figura 15.13 son relativamente sencillas.

La Figura 15.14 implementa los restantes métodos que no dependen de los iteradores. `findPos` es una rutina privada auxiliar que devuelve la posición de un objeto que está siendo eliminado o que esté sujeto a una llamada `contains`. Existe código adicional porque es legal añadir `null` al `ArrayList` y si no tuviéramos cuidado, la llamada a `equals` en la línea 60 podría haber generado una excepción `NullPointerException`. Observe que tanto `add` como `remove` provocan una modificación de `modCount`.

En la Figura 15.15 vemos dos métodos factoría que devuelven iteradores, y también el comienzo de la implementación de la interfaz `ListIterator`. Observe que `ArrayListIterator` ES-UN `ListIterator` y `ListIterator` ES-UN `Iterator`. Por tanto, puede devolverse un `ArrayListIterator` en las líneas 103 y 106.

En la implementación de `ArrayListIterator`, realizada como una clase interna privada, mantenemos la posición actual en la línea 111. La posición actual representa el índice del elemento que se devolvería llamando a `next`. En la línea 112 declaramos el miembro `expectedModCount`. Al igual que todos los miembros de clase, se inicializa cuando se crea una instancia del iterador (inmediatamente antes de invocar al constructor); `modCount` es una abreviatura para `ArrayList.this.modCount`. Los dos miembros de instancia booleanos que siguen son indicadores utilizados para verificar que una llamada a `remove` es legal.

El constructor `ArrayListIterator` se declara con visibilidad de paquete; por ello, es utilizado por el `ArrayList`. Por supuesto, podría declararse público, pero no hay ninguna razón para hacerlo e incluso si fuera privado, seguiría siendo utilizable por el `ArrayList`. Sin embargo, la visibilidad de paquete parece más natural en este paquete. Tanto `hasNext` como `hasPrevious` verifican que no haya habido modificaciones estructurales externas desde que se creó el iterador, generando una excepción si el miembro `modCount` del `ArrayList` no se corresponde con el valor de `expectedModCount` del `ArrayListIterator`.

La clase `ArrayListIterator` se completa en la Figura 15.16. Las rutinas `next` y `previous` son simétricas. Examinando `next`, vemos primero una comprobación en la línea 138, para asegurarnos de que no hemos acabado la iteración (implícitamente, esto comprueba también la existencia de modificaciones estructurales). Después configuramos `nextCompleted` como `true` para permitir que `remove` pueda llevarse a cabo, y después devolvemos el elemento de la matriz que `current` esté examinando, haciendo avanzar `current` después de que su valor haya sido utilizado.

```
1 package weiss.util;
2
3 public class ArrayList<AnyType> extends AbstractCollection<AnyType>
4         implements List<AnyType>
5 {
6     private static final int DEFAULT_CAPACITY = 10;
7     private static final int NOT_FOUND = -1;
8
9     private AnyType [ ] theItems;
10    private int theSize;
11    private int modCount = 0;
12
13    public ArrayList( )
14        { clear( ); }
15
16    public ArrayList( Collection<? extends AnyType> other )
17    {
18        clear( );
19        for( AnyType obj : other )
20            add( obj );
21    }
22
23    public int size( )
24    { return theSize; }
25
26    public void clear( )
27    {
28        theSize = 0;
29        theItems = (AnyType []) new Object[ DEFAULT_CAPACITY ];
30        modCount++;
31    }
32
33    public AnyType get( int idx )
34    {
35        if( idx < 0 || idx >= size( ) )
36            throw new ArrayIndexOutOfBoundsException( );
37        return theItems[ idx ];
38    }
39
40    public AnyType set( int idx, AnyType newVal )
41    {
```

Continúa

---

Figura 15.13 Implementación de ArrayList (parte 1).

```

42     if( idx < 0 || idx >= size( ) )
43         throw new ArrayIndexOutOfBoundsException( );
44     AnyType old = theItems[ idx ];
45     theItems[ idx ] = newVal;
46
47     return old;
48 }
49
50 public boolean contains( Object x )
51     { return findPos( x ) != NOT_FOUND; }

```

Figura 15.13 (Continuación).

El método `previous` es similar, salvo porque debemos decrementar primero el valor de `current`. Esto es porque al recorrer la colección en sentido inverso, si `current` es igual al tamaño del contenedor, no habremos todavía iniciado la iteración, y cuando `current` sea igual a cero, ya habremos completado la iteración (pero podemos eliminar el elemento en esta posición si la operación anterior fue `previous`). Observe que `next` seguido de `previous` nos da elementos idénticos.

Finalmente, llegamos a `remove`, que es extremadamente complejo porque la semántica de `remove` depende de que en qué dirección se esté recorriendo la colección. De hecho, esto probablemente sugiere un defecto de diseño en la API de Colecciones: la semántica de los métodos no debería depender tan fuertemente de qué métodos han sido invocados antes que ellos, pero `remove` es lo que es, así que tenemos que implementarlo.

La implementación de `remove` comienza comprobando la modificación estructural en la línea 156. Si la operación anterior de cambio de estado del iterador fue `next`, como evidencia la comprobación en la línea 159 que muestra que `nextCompleted` es `true`, entonces invocamos el método `remove` de `ArrayList` (que se inicia en la línea 93 de la Figura 15.14) que acepta un índice como parámetro. El uso de `ArrayList.this.remove` es obligatorio porque la versión local de `remove` oculta la versión correspondiente a la clase externa. Puesto que ya hemos avanzado más allá del elemento que hay que eliminar, debemos eliminar el elemento situado en la posición `current - 1`.

```

52     private int findPos( Object x )
53     {
54         for( int i = 0; i < size( ); i++ )
55             if( x == null )
56             {
57                 if( theItems[ i ] == null )
58                     return i;
59             }
60             else if( x.equals( theItems[ i ] ) )

```

Continúa

Figura 15.14 Implementación de ArrayList (parte 2).

```
61         return i;
62
63     return NOT_FOUND;
64 }
65
66 public boolean add( AnyType x )
67 {
68     if( theItems.length == size( ) )
69     {
70         AnyType [ ] old = theItems;
71         theItems = (AnyType []) new Object[ theItems.length * 2 + 1 ];
72         for( int i = 0; i < size( ); i++ )
73             theItems[ i ] = old[ i ];
74     }
75     theItems[ theSize++ ] = x;
76     modCount++;
77     return true;
78 }
79
80 public boolean remove( Object x )
81 {
82     int pos = findPos( x );
83
84     if( pos == NOT_FOUND )
85         return false;
86     else
87     {
88         remove( pos );
89         return true;
90     }
91 }
92
93 public AnyType remove( int idx )
94 {
95     AnyType removedItem = theItems[ idx ];
96     for( int i = idx; i < size( ) - 1; i++ )
97         theItems[ i ] = theItems[ i + 1 ];
98     theSize--;
99     modCount++;
100    return removedItem;
101 }
```

---

Figura 15.14 (Continuación).

```
102     public Iterator<AnyType> iterator( )
103         { return new ArrayListIterator( 0 ); }
104
105     public ListIterator<AnyType> listIterator( int idx )
106         { return new ArrayListIterator( idx ); }
107
108     // Esta es la implementación del ArrayListIterator
109     private class ArrayListIterator implements ListIterator<AnyType>
110     {
111         private int current;
112         private int expectedModCount = modCount;
113         private boolean nextCompleted = false;
114         private boolean prevCompleted = false;
115
116         ArrayListIterator( int pos )
117         {
118             if( pos < 0 || pos > size( ) )
119                 throw new IndexOutOfBoundsException( );
120             current = pos;
121         }
122
123         public boolean hasNext( )
124         {
125             if( expectedModCount != modCount )
126                 throw new ConcurrentModificationException( );
127             return current < size( );
128         }
129
130         public boolean hasPrevious( )
131         {
132             if( expectedModCount != modCount )
133                 throw new ConcurrentModificationException( );
134             return current > 0;
135         }
}
```

Figura 15.15 Implementación de ArrayList (parte 3).

Esto hace que el siguiente elemento se desplace de `current` a `current - 1` (ya que la anterior posición `current - 1` ahora ha sido eliminada), por lo que utilizamos la expresión `--current` en la línea 160.

Al recorrer en la otra dirección, estamos situados en el último elemento devuelto, por lo que simplemente pasamos `current` como parámetro al método `remove` externo. Después de que este vuelve, los elementos situados en índices superiores se desplazan una unidad de índice hacia abajo, por lo que `current` estará situado sobre el elemento correcto y podrá ser utilizado en la expresión de la línea 162.

```
136     public AnyType next( )
137     {
138         if( !hasNext( ) )
139             throw new NoSuchElementException( );
140         nextCompleted = true;
141         prevCompleted = false;
142         return theItems[ current++ ];
143     }
144
145     public AnyType previous( )
146     {
147         if( !hasPrevious( ) )
148             throw new NoSuchElementException( );
149         prevCompleted = true;
150         nextCompleted = false;
151         return theItems[ --current ];
152     }
153
154     public void remove( )
155     {
156         if( expectedModCount != modCount )
157             throw new ConcurrentModificationException( );
158
159         if( nextCompleted )
160             ArrayList.this.remove( --current );
161         else if( prevCompleted )
162             ArrayList.this.remove( current );
163         else
164             throw new IllegalStateException( );
165
166         prevCompleted = nextCompleted = false;
167         expectedModCount++;
168     }
169 }
170 }
```

Figura 15.16 Implementación de ArrayList (parte 4).

En cualquiera de los dos casos, no podemos hacer otra operación `remove` hasta que llevemos a cabo una operación `next` o `previous`, por lo que en la línea 166 borramos ambos indicadores. Por último, en la línea 167, incrementamos el valor de `expectedModCount` para ajustarlo al del contenedor. Observe que este valor solo se incrementa para este iterador, por lo que cualquier otro iterador quedará ahora invalidado.

Esta clase, que es quizás la más simple de las clases de la API de Colecciones que contienen iteradores, ilustra por qué hemos elegido en la Parte Cuatro comenzar con un protocolo simple y luego proporcionar implementaciones más completas al final del capítulo.

## Resumen

En este capítulo se ha presentado el concepto de clase interna, que es una técnica Java utilizada para implementar clases iteradoras. Cada instancia de la clase interna se corresponde exactamente con una instancia de una clase externa y mantiene de forma automática una referencia al objeto de la clase externa que ha provocado su construcción. Una clase anidada relaciona dos tipos entre sí, mientras que una clase interna relaciona entre sí dos objetos. La clase interna se utiliza en este capítulo para implementar `ArrayList`.

El siguiente capítulo ilustra implementaciones de pilas y colas.



### Conceptos clave

`AbstractCollection` Implementa algunos de los métodos de la interfaz `Collection`. (567)

**clase interna** Una clase dentro de otra clase que resulta útil para implementar el patrón iterador. La clase interna siempre contiene una referencia implícita al objeto externo que la ha creado. (563)

`StringBuilder` Utilizado para construir objetos `String` sin crear repetidamente un gran número de objetos `String` intermedios. (571)



### Errores comunes

1. No se puede construir una clase interna de instancia sin un objeto externo. La manera más fácil de hacer esto es mediante un método factoría en la clase externa. Es bastante común olvidarse de la palabra `static` al declarar una clase anidada, lo que a menudo generará un error difícil de comprender relacionado con esta regla.
2. El excesivo número de concatenaciones de objetos `String` puede transformar un programa de tiempo lineal en un programa de tiempo cuadrático.



### Internet

Hay disponibles los siguientes archivos.

#### **MyContainerTest.java**

El programa de prueba para el ejemplo final de iterador que utiliza clases internas, como se muestra en la Sección 15.2. **Iterator.java** y **MyContainer.java** se encuentran ambos en el paquete `weiss.ds` disponible en línea.

#### **AbstractCollection.java**

Contiene el código de las Figuras 15.10 a 15.12.

#### **ArrayList.java**

Contiene el código de las Figuras 15.13 a 15.16.



## Ejercicios

### EN RESUMEN

- 15.1** ¿Qué es un `StringBuilder`?
- 15.2** ¿Cuál es la diferencia entre una clase anidada y una clase interna?
- 15.3** Los miembros privados de una clase interna (o anidada), ¿son visibles para los métodos de la clase externa?
- 15.4** En la Figura 15.17, ¿son legales las declaraciones de `a` y `b`? ¿Por qué o por qué no?
- 15.5** En la Figura 15.17 (suponiendo que se haya corregido el código ilegal), ¿cómo se crean los objetos de tipo `Inner1` e `Inner2` (puede sugerir otros miembros adicionales)?

### EN TEORÍA

- 15.6** ¿Cuál es el tiempo de ejecución de `clear`, tal como está implementado para `ArrayList`? ¿Cuál sería el tiempo de ejecución si se utilizara en su lugar la versión heredada de `AbstractCollection`?
- 15.7** Suponga que una clase interna `I` se declara pública en su clase externa `O`. ¿Por qué podría ser necesario una sintaxis inusual para declarar una clase `E` que amplíe `I` pero que se declare como clase de nivel superior? (La sintaxis requerida es todavía más complicada que la que vimos para `new`, pero a menudo hace falta un mal diseño para que esta sintaxis llegue a ser necesaria.)

### EN LA PRÁCTICA

- 15.8** Suponga que queremos un iterador que implemente el conjunto de métodos `isValid`, `advance` y `retrieve`, pero todo lo que tenemos es la interfaz estándar `java.util.Iterator`.
  - a. ¿Qué patrón describe el problema que estamos tratando de resolver?
  - b. Diseñe una clase `BetterIterator` y luego impleméntela en términos de `java.util.Iterator`.
- 15.9** La Figura 15.18 contiene dos implementaciones propuestas de `clear` para `AbstractCollection`. ¿Funciona alguna de ellas?

### PROYECTOS DE PROGRAMACIÓN

- 15.10** `Collections.unmodifiableCollection` toma una `Collection` y devuelve una `Collection` inmutable. Implemente este método. Para ello, necesitará utilizar una clase local (una clase dentro de un método). La clase implementa la interfaz `Collection` y genera una excepción `UnsupportedOperationException` para todos los métodos mutadores. Para otros métodos, reenvía la solicitud a la `Collection` que esté siendo envuelta. También tendrá que ocultar un iterador no modifiable.

```

1 class Outer
2 {
3     private int x = 0;
4     private static int y = 37;
5
6     private class Inner1 implements SomeInterface
7     {
8         private int a = x + y;
9     }
10
11    private static class Inner2 implements SomeInterface
12    {
13        private int b = x + y;
14    }
15 }

```

Figura 15.17 Código para los Ejercicios 15.4 y 15.5.

```

1 public void clear() // Versión #1
2 {
3     Iterator<AnyType> itr = iterator();
4     while( !isEmpty() )
5         remove( itr.next() );
6 }
7
8 public void clear() // Versión #2
9 {
10    while( !isEmpty() )
11        remove( iterator().next() );
12 }

```

Figura 15.18 Implementaciones propuestas de `clear` para `AbstractCollection`.

- 15.11** Dos objetos `Collection` son iguales si ambos implementan la interfaz `List` y contienen los mismos elementos en el mismo orden, o si ambos implementan la interfaz `Set` y contienen los mismos elementos en cualquier orden. En caso contrario, los objetos `Collection` no son iguales. Proporcione, en `AbstractCollection`, una implementación de `equals` que se ajuste a esta especificación general. Además, proporcione un método `hashCode` en `AbstractCollection` que se ajuste a la especificación general de `hashCode`. (Haga esto utilizando un iterador y añadiendo los códigos hash de todas las entradas. Tenga cuidado con la entradas `null`.)

**15.12** La interfaz Collection de la API de Colecciones de Java define métodos removeAll, addAll y containsAll. Añada estos métodos a la interfaz Collection y proporcione implementaciones en AbstractCollection.

## Pilas y colas

En este capítulo vamos a analizar la implementación de las estructuras de datos denominadas pila y cola. Recuerde del Capítulo 6 que las operaciones básicas con estas estructuras de datos consumen un tiempo esperado constante. Tanto para la pila como para la cola hay dos formas básicas de conseguir operaciones en tiempo constante. La primera consiste en almacenar los elementos de manera contigua en una matriz, mientras que la segunda consiste en almacenar los elementos de forma no contigua en una lista enlazada. En este capítulo presentaremos implementaciones para ambas estructuras de datos utilizando ambos métodos de almacenamiento.

En este capítulo veremos

- Una implementación de la pila y de la cola basada en matriz.
- Una implementación de la pila y de la cola basada en lista enlazada.
- Una breve comparación de ambos métodos.
- Una ilustración de la implementación de la pila en la API de Colecciones.

### 16.1 Implementaciones basadas en matriz dinámica

En esta sección vamos a utilizar una matriz simple para implementar la pila y la cola. Los algoritmos resultantes son extremadamente eficientes y también resultan bastante fáciles de programar. Recuerde que hemos estado utilizando `ArrayList` en lugar de matrices. El método `add` de `ArrayList` es, en la práctica, lo mismo que `push`. Sin embargo, puesto que estamos interesados en un análisis general de los algoritmos, implementaremos la pila basada en matrices utilizando matrices básicas, duplicando parte del código que hemos visto anteriormente en las implementaciones con `ArrayList`.

#### 16.1.1 Pilas

Como muestra la Figura 16.1, una pila se puede implementar con una matriz y un entero. El entero `tos` (*top of stack*, cima de la pila) proporciona el índice en la matriz del elemento superior de la pila. Así, cuando `tos` es `-1`, la pila está vacía. Para introducir un elemento con la operación `push`,

Una pila se puede implementar con una matriz y un entero, indicando este último el índice del elemento superior.

La mayoría de las rutinas de la pila son aplicaciones de ideas que ya hemos presentado anteriormente.

Recuerde que el mecanismo de duplicación del tamaño de la matriz no afecta a largo plazo al rendimiento del algoritmo.

incrementamos `tos` y colocamos el nuevo elemento en la posición `tos` de la matriz. Acceder al elemento superior es por tanto trivial, y podemos realizar la extracción `pop` decrementando `tos`. En la Figura 16.1 partimos de una pila vacía. A continuación, mostramos la pila después de tres operaciones: `push(a)`, `push(b)` y `pop()`.

La Figura 16.2 muestra el esqueleto de la clase `Stack` basada en matriz. Especifica dos miembros de datos: `theArray`, que es una matriz que se expande según sea necesario y que almacena los elementos contenidos en la pila; y `topOfStack` que proporciona el índice del elemento que está actualmente en la cima de la pila. Para una pila vacía este índice será  $-1$ . El constructor se muestra en la Figura 16.3.

Los métodos públicos se indican en las líneas 22 y 23. La mayoría de estas rutinas tienen implementaciones simples. Las rutinas `isEmpty` y `makeEmpty` son rutinas de una única línea, como se muestra en la Figura 16.4 y sirven para ver si la pila está vacía y para vaciarla, respectivamente. El método `push` se muestra en la Figura 16.5. Si no fuera por el mecanismo de duplicación del tamaño de la matriz, la rutina `push` estaría dada por la única línea de código mostrada en la línea 9. Recuerde que el uso del operador prefijo `++` significa que se incrementa primero `topOfStack` y luego se utiliza su nuevo valor para indexar la matriz `theArray`. Las restantes rutinas son igualmente cortas, como se muestra en las Figuras 16.6 y 16.7. El operador postfijo `--` utilizado en la Figura 16.7 indica que, aunque se decremente `topOfStack`, es su valor anterior el que se utiliza para indexar `theArray`.

Si no hay mecanismo de duplicación del tamaño de la matriz, toda operación tarda un tiempo constante. Una operación `push` que exija una duplicación del tamaño de la matriz requerirá un tiempo  $O(N)$ . Si esto ocurriera de manera frecuente, tendríamos motivos para preocuparnos. Sin embargo, resulta infrecuente porque una duplicación de la matriz que implique a  $N$  elementos, deberá estar precedida por al menos  $N/2$  inserciones con `push` que no exijan una duplicación del tamaño de la matriz. En consecuencia, podemos cargar el coste  $O(N)$  de la duplicación a esas  $N/2$  inserciones anteriores con `push`, lo que en la práctica equivale a incrementar el coste de cada operación `push` en una cierta constante pequeña. Esta técnica se denomina *amortización*.

Un ejemplo de amortización extraído de la vida real sería el pago de los impuestos de la renta. En lugar de pagar todos los impuestos de una vez, el gobierno exige que paguemos la mayor parte de nuestros impuestos a través de retenciones de los salarios. El importe total del impuesto es siempre

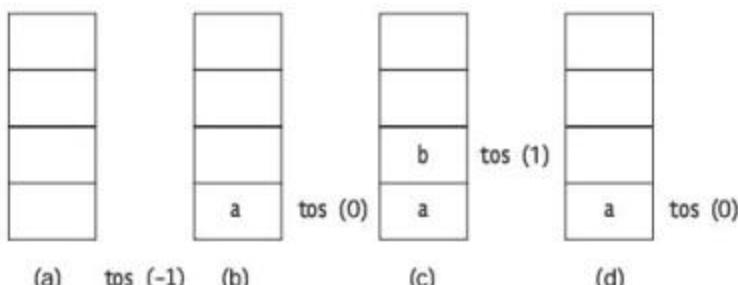


Figura 16.1 Cómo funcionan las rutinas de la pila. (a) Pila vacía. (b) `push('a')`. (c) `push('b')`. (d) `pop()`.

```

1 package weiss.nonstandard;
2
3 // Clase ArrayStack
4 //
5 // CONSTRUCCIÓN: sin inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void push( x )      --> Insertar x
9 // void pop( )         --> Extraer elemento más recientemente insertado
10 // AnyType top( )    --> Devolver elemento más recientemente insertado
11 // AnyType topAndPop( ) --> Devolver y extraer elemento más reciente
12 // boolean isEmpty( ) --> Devolver true si vacía; en caso contrario, false
13 // void makeEmpty( )  --> Extraer todos los elementos
14 // *****ERRORES*****
15 // top, pop, o topAndPop con una pila vacía
16
17 public class ArrayStack<AnyType> implements Stack<AnyType>
18 {
19     public ArrayStack( )
20     { /* Figura 16.3 */ }
21
22     public boolean isEmpty( )
23     { /* Figura 16.4 */ }
24     public void makeEmpty( )
25     { /* Figura 16.4 */ }
26     public AnyType top( )
27     { /* Figura 16.6 */ }
28     public void pop( )
29     { /* Figura 16.6 */ }
30     public AnyType topAndPop( )
31     { /* Figura 16.7 */ }
32     public void push( AnyType x )
33     { /* Figura 16.5 */ }
34
35     private void doubleArray( )
36     { /* Implementación en el código en línea */ }
37
38     private AnyType [ ] theArray;
39     private int topOfStack;
40
41     private static final int DEFAULT_CAPACITY = 10;
42 }

```

Figura 16.2 Esqueleto para la clase de pila basada en matriz.

```

1 /**
2 * Construir la pila.
3 */
4 public ArrayStack( )
5 {
6 theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
7 topOfStack = -1;
8 }

```

**Figura 16.3** El constructor de cero parámetros para la clase `ArrayStack`.

```

1 /**
2 * Comprobar si la pila está lógicamente vacía.
3 * @return true si está vacía, false en caso contrario.
4 */
5 public boolean isEmpty( )
6 {
7     return topOfStack == -1;
8 }
9
10 /**
11 * Hacer que la pila pase a estar lógicamente vacía.
12 */
13 public void makeEmpty( )
14 {
15     topOfStack = -1;
16 }

```

**Figura 16.4** Las rutinas `isEmpty` y `makeEmpty` para la clase `ArrayStack`.

```

1 /**
2 * Insertar un nuevo elemento en la pila.
3 * @param x el elemento que hay que insertar.
4 */
5 public void push( AnyType x )
6 {
7     if( topOfStack + 1 == theArray.length )
8         doubleArray( );
9     theArray[ ++topOfStack ] = x;
10 }

```

**Figura 16.5** El método `push` para la clase `ArrayStack`.

```

1  /**
2   * Obtener el elemento más recientemente insertado en la pila.
3   * No modifica la pila.
4   * @return el elemento más recientemente insertado en la pila.
5   * @throws UnderflowException si la pila está vacía.
6   */
7  public AnyType top( )
8  {
9      if( isEmpty( ) )
10         throw new UnderflowException( "ArrayStack top" );
11         return theArray[ topOfStack ];
12     }
13
14 /**
15  * Extraer de la pila el elemento más recientemente insertado.
16  * @throws UnderflowException si la pila está vacía.
17  */
18 public void pop( )
19 {
20     if( isEmpty( ) )
21         throw new UnderflowException( "ArrayStack pop" );
22     topOfStack--;
23 }
```

Figura 16.6 Los métodos top y pop para la clase ArrayStack.

```

1 /**
2  * Devolver y extraer el elemento más recientemente insertado
3  * en la pila.
4  * @return el elemento más recientemente insertado en la pila.
5  * @throws UnderflowException si la pila está vacía.
6  */
7  public AnyType topAndPop( )
8  {
9      if( isEmpty( ) )
10         throw new UnderflowException( "ArrayStack topAndPop" );
11         return theArray[ topOfStack-- ];
12 }
```

Figura 16.7 El método topAndpop para la clase ArrayStack.

el mismo, lo único que varía es *cuándo* se paga. Esto mismo es aplicable a las operaciones push. Podemos cargar el coste de la duplicación de la matriz al instante en que tiene lugar, o podemos

distribuir ese coste de manera equitativa entre todas las operaciones `push`. Una cota amortizada exige que carguemos a cada operación de una secuencia una parte equitativa del coste total. En nuestro ejemplo, el coste de la duplicación del tamaño de la matriz no sería, por tanto, excesivo.

### 16.1.2 Colas

Almacenar los elementos de la cola comenzando al principio de una matriz hace que la operación de extracción de un elemento de la cola sea muy costosa.

La operación de extracción `dequeue` se implementa incrementando la posición de `front`.

La forma más fácil de implementar la cola es almacenar los elementos en una matriz, con el elemento inicial en la primera posición (es decir, con el índice de matriz 0). Si `back` representa la posición del último elemento de la cola, entonces para introducir un elemento en la cola con la operación `enqueue` simplemente tenemos que incrementar `back` y colocar ahí el elemento. El problema es que la operación `dequeue` es muy cara. La razón es que, al exigir que los elementos estén colocados al principio de la matriz, forzamos a `dequeue` a desplazar todos los elementos una posición después de extraer el elemento inicial.

La Figura 16.8 muestra que podemos resolver este problema al realizar una operación `dequeue` incrementando `front` en lugar de desplazando todos los elementos. Cuando la cola tiene un único elemento, tanto `front` como `back` representan el índice en la matriz de dicho elemento. Por tanto, para una cola vacía, `back` se debe inicializar con el valor `front - 1`.

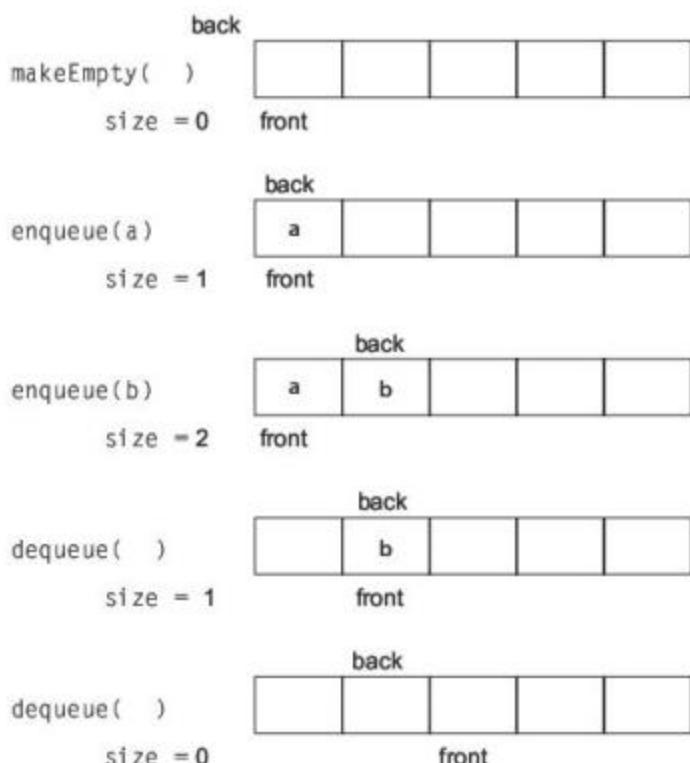


Figura 16.8 Implementación matricial básica de la cola.

Esta implementación garantiza que tanto enqueue como dequeue puedan realizarse en un tiempo constante. El problema fundamental de este enfoque se muestra en la primera línea de la Figura 16.9. Después de tres operaciones enqueue adicionales, ya no podemos añadir más elementos, a pesar de que la cola no está realmente llena. La técnica de duplicación de tamaño de la matriz no resuelve el problema, porque si incluso el tamaño de la matriz fuera 1.000, después de 1.000 operaciones enqueue no habría espacio en la cola, independientemente de cuál fuera su tamaño real. Aunque se hubieran realizado 1.000 operaciones dequeue, lo que hace que la cola pase a estar vacía desde el punto de vista abstracto, nos veríamos imposibilitados para añadir ningún otro elemento.

Sin embargo, como se muestra en la Figura 16.9, existe una gran cantidad de espacio adicional: todas las posiciones anteriores a front no están utilizadas y pueden por tanto reciclarse. Para ello, utilizamos la técnica del *encadenamiento circular*, es decir, cuando back o front alcancen el final de la matriz, volvemos a colocarlos al principio. Esta operación para implementar una cola se denomina *implementación mediante matriz circular*. Solo necesitaremos duplicar el tamaño de la matriz cuando el número de elementos de la cola sea igual al número de posiciones de la matriz. Para efectuar la operación enqueue(f), lo que hacemos, por tanto, es volver a situar back al principio de la matriz y colocar f allí. Después de tres operaciones dequeue, front también volverá a ser colocado al principio de la matriz.

El encadenamiento circular devuelve front o back al principio de la matriz cuando alguno de ellos alcanza el final de la misma. La utilización del encadenamiento circular para implementar la cola se denomina *implementación mediante matriz circular*.

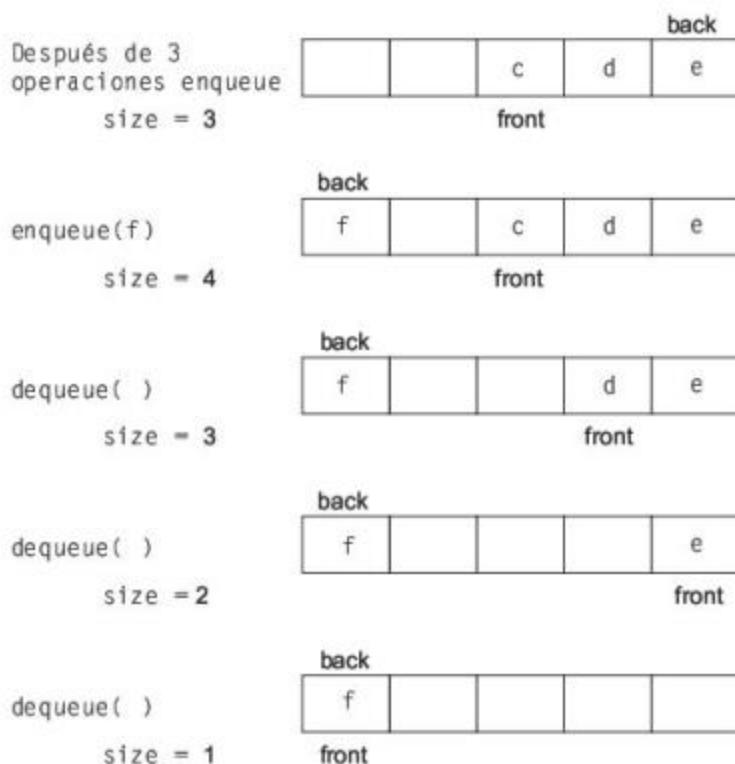


Figura 16.9 Implementación matricial de la cola con encadenamiento circular.

```
1 package weiss.nonstandard;
2
3 // Clase ArrayQueue
4 //
5 // CONSTRUCCIÓN: sin inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void enqueue( x ) --> Insertar x
9 // AnyType getFront( ) --> Devolver elemento menos recientemente insertado
10 // AnyType dequeue( ) --> Devolver y extraer elemento menos reciente
11 // boolean isEmpty( ) --> Devolver true si vacía; false en caso contrario
12 // void makeEmpty( ) --> Extraer todos los elementos
13 // *****ERRORES*****
14 // getFront o dequeue con cola vacía
15
16 public class ArrayQueue<AnyType>
17 {
18     public ArrayQueue( )
19         { /* Figura 16.12 */ }
20
21     public boolean isEmpty( )
22         { /* Figura 16.13 */ }
23     public void makeEmpty( )
24         { /* Figura 16.17 */ }
25     public AnyType dequeue( )
26         { /* Figura 16.16 */ }
27     public AnyType getFront( )
28         { /* Figura 16.16 */ }
29     public void enqueue( AnyType x )
30         { /* Figura 16.14 */ }
31
32     private int increment( int x )
33         { /* Figura 16.11 */ }
34     private void doubleQueue( )
35         { /* Figura 16.15 */ }
36
37     private AnyType [ ] theArray;
38     private int currentSize;
39     private int front;
40     private int back;
41
42     private static final int DEFAULT_CAPACITY = 10;
43 }
```

---

Figura 16.10 Esqueleto para la clase de cola basada en matriz.

El esqueleto de la clase `ArrayQueue` se muestra en la Figura 16.10. La clase `ArrayQueue` tiene cuatro miembros de datos: una matriz que se expande de forma dinámica, el número de elementos que hay actualmente en la cola, el índice matricial del elemento inicial y el índice matricial del elemento final.

Declaramos dos métodos en la sección privada. Estos dos métodos son utilizados internamente por los métodos de `ArrayQueue`, pero no se ponen a disposición del usuario de la clase. Uno de estos métodos es la rutina `increment`, que suma 1 a su parámetro y devuelve el nuevo valor. Puesto que este método implementa el encadenamiento circular, si el resultado fuera igual al tamaño de la matriz, se reinicializaría con el valor cero. Esta rutina se muestra en la Figura 16.11. La otra rutina es `doubleQueue`, que se invoca si una operación `enqueue` requiere duplicar el tamaño de la matriz. Es ligeramente más compleja que la duplicación de tamaño usual, porque los elementos de la cola no están necesariamente almacenados dentro de la matriz a partir de la posición 0. Por tanto, los elementos deben copiarse con cuidado. Analizaremos `doubleQueue` cuando hablamos de `enqueue`.

Si la cola está llena,  
debemos implementar el  
mecanismo de duplicación  
de la matriz con cuidado.

Muchos de los métodos públicos se asemejan a los correspondientes métodos utilizados con las pilas, incluyendo el constructor mostrado en la Figura 16.12 e `isEmpty`, mostrado en la Figura 16.13. Este constructor no tiene nada especialmente particular, salvo que debemos cerciorarnos de disponer de los valores iniciales correctos tanto para `front` como para `back`. Esto se hace invocando a `makeEmpty`.

```

1  /**
2   * Método interno para incrementar con encadenamiento circular.
3   * @param x cualquier índice del rango de theArray.
4   * @return x+1, o 0 si x está al final de theArray.
5   */
6  private int increment( int x )
7  {
8      if( ++x == theArray.length )
9          x = 0;
10     return x;
11 }
```

Figura 16.11 La rutina de encadenamiento circular.

```

1  /**
2   * Construir la cola.
3   */
4  public ArrayQueue()
5  {
6      theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
7      makeEmpty();
8 }
```

Figura 16.12 El constructor para la clase `ArrayQueue`.

```

1  /**
2   * Comprobar si la cola está lógicamente vacía.
3   * @return true si está vacía, false en caso contrario.
4   */
5  public boolean isEmpty( )
6  {
7      return currentSize == 0;
8  }

```

Figura 16.13 La rutina isEmpty para la clase ArrayQueue.

Cuando duplicamos la matriz utilizada para la cola, no podemos limitarnos a copiar directamente toda la matriz.

La rutina enqueue se muestra en la Figura 16.14. La estrategia básica es bastante simple, como se ilustra en las líneas 9 a 11 de la rutina enqueue. La rutina doubleQueue, mostrada en la Figura 16.15, comienza cambiando el tamaño de la matriz. Tenemos que desplazar los elementos a partir de la posición front, en lugar de a partir de 0.

Así, doubleQueue recorre la matriz antigua y copia cada elemento en la nueva la matriz en las líneas 11 y 12. A continuación, volvemos a configurar el valor de back en la línea 16. Las rutinas dequeue y getFront se muestran en la Figura 16.16; ambas son rutinas cortas. Finalmente, la rutina makeEmpty se muestra en la Figura 16.17. Las rutinas de la cola son, claramente, operaciones de tiempo constante, por lo que el coste de duplicación de la matriz se puede amortizar entre toda la secuencia de operaciones enqueue, igual que sucedía con la pila.

La implementación mediante matriz circular de la cola puede realizarse incorrectamente con mucha facilidad, cuando se hacen intentos de abreviar el código. Por ejemplo, si intentamos evitar emplear el miembro size utilizando front y back para deducir el tamaño, la matriz deberá ser redimensionada cuando el número de elementos de la cola sea 1 menos que el tamaño de la matriz.

```

1  /**
2   * Insertar un nuevo elemento en la cola.
3   * @param x el elemento que hay que insertar.
4   */
5  public void enqueue( AnyType x )
6  {
7      if( currentSize == theArray.length )
8          doubleQueue( );
9      back = increment( back );
10     theArray[ back ] = x;
11     currentSize++;
12 }

```

Figura 16.14 La rutina enqueue para la clase ArrayQueue.

```

1  /**
2  * Método interno para expandir theArray.
3  */
4 private void doubleQueue( )
5 {
6     AnyType [ ] newArray;
7
8     newArray = (AnyType [ ]) new Object[ theArray.length * 2 ];
9
10    // Copiar los elementos que están lógicamente dentro de la cola
11    for( int i = 0; i < currentSize; i++, front = increment( front ) )
12        newArray[ i ] = theArray[ front ];
13
14    theArray = newArray;
15    front = 0;
16    back = currentSize - 1;
17 }

```

Figura 16.15 Expansión dinámica para la clase ArrayQueue.

```

1 /**
2 * Devolver y extraer el elemento menos recientemente insertado
3 * en la cola.
4 * @return el elemento menos recientemente insertado en la cola.
5 * @throws UnderflowException si la cola está vacía.
6 */
7 public AnyType dequeue( )
8 {
9     if( isEmpty( ) )
10         throw new UnderflowException( "ArrayQueue dequeue" );
11     currentSize--;
12
13     AnyType returnValue = theArray[ front ];
14     front = increment( front );
15     return returnValue;
16 }
17
18 /**
19 * Obtener el elemento menos recientemente insertado en la cola.
20 * No modifica la cola.
21 * @return el elemento menos recientemente insertado en la cola.
22 * @throws UnderflowException si la cola está vacía.
23 */
24 public AnyType getFront( )
25 {
26     if( isEmpty( ) )
27         throw new UnderflowException( "ArrayQueue getFront" );
28     return theArray[ front ];
29 }

```

Figura 16.16 Las rutinas dequeue y getFront para la clase ArrayQueue.

```
1  /**
2  * Hacer que la cola pase a estar lógicamente vacía.
3  */
4  public void makeEmpty( )
5  {
6      currentSize = 0;
7      front = 0;
8      back = -1;
9 }
```

Figura 16.17 La rutina makeEmpty para la clase ArrayQueue.

## 16.2 Implementaciones con lista enlazada

Una alternativa a la implementación mediante una matriz contigua es la lista enlazada. Recuerde de la Sección 6.5 que, en una lista enlazada, cada elemento se almacena en un objeto separado, que contiene también una referencia al siguiente elemento de la lista.

La ventaja de la lista enlazada es que la cantidad adicional de memoria requerida es solamente de una referencia por cada elemento. Por el contrario, una implementación basada en una matriz contigua utiliza una memoria adicional igual al número de elementos de la matriz vacíos (más algo de memoria adicional durante la fase de duplicación del tamaño). La ventaja de la lista enlazada puede ser significativa en otros lenguajes, si los elementos vacíos de la matriz almacenan instancias no inicializadas de objetos que consumen un espacio significativo. En Java esta ventaja es mínima. Aun así, vamos a presentar las implementaciones basadas en lista enlazada por tres razones.

La ventaja de una implementación basada en lista enlazada es que la memoria adicional requerida es de solo una referencia por cada elemento. La desventaja es que la gestión de memoria puede ser costosa en términos de tiempo.

1. Porque es importante comprender aquellas implementaciones que pueden ser útiles en otros lenguajes.
2. Porque las implementaciones que utilizan listas enlazadas pueden ser más cortas para la cola que las versiones comparables basadas en matriz.
3. Porque estas implementaciones ilustran los principios que subyacen a las operaciones más generales con listas enlazadas de las que hablaremos en el Capítulo 17.

Para que la implementación sea competitiva con las implementaciones basadas en matriz contigua, debemos ser capaces de realizar las operaciones básicas con la lista enlazada en un tiempo constante. Resulta sencillo conseguirlo, porque los cambios en la lista enlazada están restringidos a los elementos situados en los dos extremos (anterior y posterior) de la lista.

### 16.2.1 Pilas

La clase para las pilas puede implementarse como una lista enlazada en la que el elemento superior de la pila está representado por el primer elemento de la lista, como se muestra en la Figura 16.18.

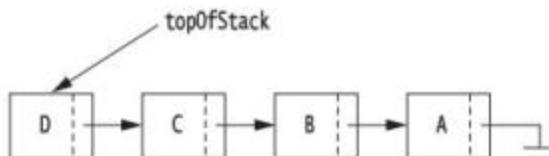


Figura 16.18 Implementación de la clase Stack mediante lista enlazada.

Para implementar una operación de inserción `push`, creamos un nuevo nodo en la lista y lo conectamos como nuevo primer elemento de la misma. Para implementar una operación de extracción `pop`, simplemente hacemos avanzar la cima de la pila hasta el segundo elemento de la lista (si existe). Una pila vacía estará representada por una lista enlazada vacía. Claramente, cada operación se lleva a cabo en un tiempo constante, porque al restringir las operaciones al primer nodo, hemos hecho que todos los cálculos sean independientes del tamaño de la lista. Lo único que nos queda por hacer es la implementación en Java.

La Figura 16.19 proporciona el esqueleto de la clase. Las líneas 39 a 49 proporcionan la declaración de tipos para los nodos de la lista. Un nodo `ListNode` estará compuesto por dos miembros de datos: `element` que almacena el elemento y `next` que almacena una referencia al siguiente `ListNode` de la lista enlazada. Proporcionamos constructores para `ListNode` que se pueden utilizar para ejecutar tanto

```
ListNode<AnyType> p1 = new ListNode<AnyType>( x );
```

como

```
ListNode<AnyType> p2 = new ListNode<AnyType>( x, ptr2 );
```

Una opción consiste en anidar `ListNode` en la clase `Stack`. Aquí, hemos utilizado la alternativa, ligeramente peor, de hacerlo una clase de nivel superior pero que solo tiene visibilidad de paquete, permitiendo así la reutilización de la clase para la implementación de la cola. La propia pila está representada por un único miembro de datos, `topOfStack`, que es una referencia al primer `ListNode` de la lista enlazada.

El constructor no está escrito explícitamente, ya que de manera predeterminada obtenemos una pila vacía configurando `topOfStack` como `NULL`. `makeEmpty` y `isEmpty` son por tanto triviales y se muestran en las líneas 19 a 22.

En la Figura 16.20 se muestran dos rutinas. La operación `push` es esencialmente una línea de código en la que asignamos un nuevo `ListNode` cuyo miembro de datos contiene el elemento `x` que queremos insertar. La referencia `next` para este nuevo nodo es el valor `topOfStack` original. Este nodo pasa entonces a convertirse en el nuevo `topOfStack`. Hacemos todo esto en la línea 7.

La operación `pop` también es simple. Después de la comprobación obligatoria de que la pila no está vacía, hacemos que `topOfStack` apunte al segundo nodo de la lista.

Finalmente, `top` y `topAndPop` son rutinas muy sencillas y se implementan como se muestra en la Figura 16.21.

Al implementar la clase de pila, la cima de la pila se representa mediante el primer elemento de una pila enlazada.

La declaración de `ListNode` tiene visibilidad de paquete, pudiendo así ser utilizada por la implementación de la cola contenida en el mismo paquete.

Las rutinas para la pila constan esencialmente de una única línea.

```

1 package weiss.nonstandard;
2
3 // Clase ListStack
4 //
5 // CONSTRUCCIÓN: sin inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void push( x )    --> Insertar x
9 // void pop( )       --> Extraer elemento más recientemente insertado
10 // AnyType top( )   --> Devolver elemento más recientemente insertado
11 // AnyType topAndPop( ) --> Devolver y extraer elemento más reciente
12 // boolean isEmpty( ) --> Devolver true si está vacía; false en otro caso
13 // void makeEmpty( ) --> Extraer todos los elementos
14 // *****ERRORES*****
15 // top, pop o topAndPop con una pila vacía
16
17 public class ListStack<AnyType> implements Stack<AnyType>
18 {
19     public boolean isEmpty( )
20     { return topOfStack == null; }
21     public void makeEmpty( )
22     { topOfStack = null; }
23
24     public void push( AnyType x )
25     { /* Figura 16.20 */ }
26     public void pop( )
27     { /* Figura 16.20 */ }
28     public AnyType top( )
29     { /* Figura 16.21 */ }
30     public AnyType topAndPop( )
31     { /* Figura 16.21 */ }
32
33     private ListNode<AnyType> topOfStack = null;
34 }
35
36 // Nodo básico almacenado en la una lista enlazada.
37 // Observe que esa clase no es accesible desde fuera
38 // del paquete weiss.nonstandard
39 class ListNode<AnyType>
40 {
41     public ListNode( AnyType theElement )

```

Continúa

Figura 16.19 Esqueleto para la clase de pila basada en lista enlazada.

```

42     { this( theElement, null ); }
43
44     public ListNode( AnyType theElement, ListNode<AnyType> n )
45     { element = theElement; next = n; }
46
47     public AnyType element;
48     public ListNode next;
49 }
```

Figura 16.19 (Continuación)

```

1  /**
2  * Insertar un nuevo elemento en la pila.
3  * @param x el elemento que hay que insertar.
4  */
5  public void push( AnyType x )
6  {
7      topOfStack = new ListNode<AnyType>( x, topOfStack );
8  }
9
10 /**
11 * Extraer el elemento más recientemente insertado en la pila.
12 * @throws UnderflowException si la pila está vacía.
13 */
14 public void pop( )
15 {
16     if( isEmpty( ) )
17         throw new UnderflowException( "ListStack pop" );
18     topOfStack = topOfStack.next;
19 }
```

Figura 16.20 Las rutinas push y pop para la clase ListStack.

## 16.2.2 Colas

La cola puede implementarse mediante una lista enlazada, siempre y cuando mantengamos referencias tanto a `front` como a `back`, es decir, al primer y el último elemento de la lista. La Figura 16.22 muestra la idea general.

La clase `ListQueue` es similar a la clase `ListStack`. En la Figura 16.23 se muestra el esqueleto de la clase `ListQueue`. El único aspecto nuevo aquí es que mantenemos dos referencias en lugar de una. La Figura 16.24 muestra los constructores de la clase `ListQueue`.

La Figura 16.25 implementa tanto `enqueue` como `dequeue`. La rutina `dequeue` es lógicamente idéntica a una operación `pop` en una pila (en realidad, a `popAndTop`). La rutina `enqueue` tiene dos

Para implementar una cola con operaciones de tiempo constante se puede utilizar una lista enlazada en la que mantengamos una referencia al primer y al último elemento.

```

1  /**
2  * Obtener el elemento más recientemente insertado en la pila.
3  * No modifica la pila.
4  * @return el elemento más recientemente insertado en la pila.
5  * @throws UnderflowException si la pila está vacía.
6  */
7  public AnyType top( )
8  {
9      if( isEmpty( ) )
10         throw new UnderflowException( "ListStack top" );
11         return topOfStack.element;
12     }
13
14 /**
15 * Devolver y extraer el elemento más recientemente insertado
16 * en la pila.
17 * @return el elemento más recientemente insertado en la pila.
18 * @throws UnderflowException si la pila está vacía.
19 */
20 public AnyType topAndPop( )
21 {
22     if( isEmpty( ) )
23         throw new UnderflowException( "ListStack topAndPop" );
24
25     AnyType topItem = topOfStack.element;
26     topOfStack = topOfStack.next;
27     return topItem;
28 }

```

Figura 16.21 Las rúbricas `top` y `topAndPop` para la clase `ListStack`.

Introducir en la cola el primer elemento constituye un caso especial, porque no hay ninguna referencia `next` a la que pueda asociarse un nuevo nodo.

casos distintos. Si la cola está vacía, creamos una cola de un solo elemento invocando `new` y haciendo que tanto `front` como `back` hagan referencia al único nodo existente. En caso contrario, creamos un nuevo nodo con un valor de datos `x`, lo añadimos al final de la lista y luego reconfiguramos el final de la lista para que apunte a este nuevo nodo, como se ilustra en la Figura 16.26. Observe que introducir en la cola el primer elemento es un caso especial, porque no hay ninguna referencia `next` a la que pueda asociarse un nuevo nodo. Hacemos todo esto en la línea 10 de la Figura 16.25.

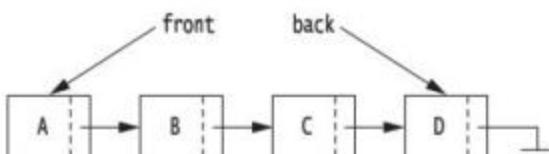


Figura 16.22 Implementación mediante lista enlazada de la clase de cola.

```
1 package weiss.nonstandard;
2
3 // Clase ListQueue
4 //
5 // CONSTRUCCIÓN: sin inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void enqueue( x ) --> Insertar x
9 // AnyType getFront( ) --> Devolver elemento menos recientemente insertado
10 // AnyType dequeue( ) --> Devolver y extraer menos elemento reciente
11 // boolean isEmpty( ) --> Devolver true si está vacía; false en otro caso
12 // void makeEmpty( ) --> Extraer todos los elementos
13 // *****ERRORES*****
14 // getFront o dequeue con una cola vacía
15
16 public class ListQueue<AnyType>
17 {
18     public ListQueue( )
19         { /* Figura 16.24 */ }
20     public boolean isEmpty( )
21         { /* Figura 16.27 */ }
22     public void enqueue( AnyType x )
23         { /* Figura 16.25 */ }
24     public AnyType dequeue( )
25         { /* Figura 16.25 */ }
26     public AnyType getFront( )
27         { /* Figura 16.27 */ }
28     public void makeEmpty( )
29         { /* Figura 16.27 */ }
30
31     private ListNode<AnyType> front;
32     private ListNode<AnyType> back;
33 }
```

Figura 16.23 Esqueleto para la clase de cola basada en lista enlazada.

```
1 /**
2 * Construir la cola.
3 */
4 public ListQueue( )
5 {
6     front = back = null;
7 }
```

Figura 16.24 Constructor para la clase ListQueue basada en lista enlazada.

```

1  /**
2   * Insertar un nuevo elemento en la cola.
3   * @param x el elemento que hay que insertar.
4   */
5  public void enqueue( AnyType x )
6  {
7      if( isEmpty( ) ) // Construir una cola de un solo elemento
8          back = front = new ListNode<AnyType>( x );
9      else           // Caso normal
10         back.next = new ListNode<AnyType>( x );
11    }
12
13 /**
14  * Devolver y extraer de la cola el elemento menos
15  * recientemente insertado.
16  * @return el elemento menos recientemente insertado en la cola.
17  * @throws UnderflowException si la cola está vacía.
18  */
19  public AnyType dequeue( )
20  {
21      if( isEmpty( ) )
22          throw new UnderflowException( "ListQueue dequeue" );
23
24      AnyType returnValue = front.element;
25      front = front.next;
26      return returnValue;
27  }

```

Figura 16.25 Las rutinas enqueue y dequeue para la clase ListQueue.

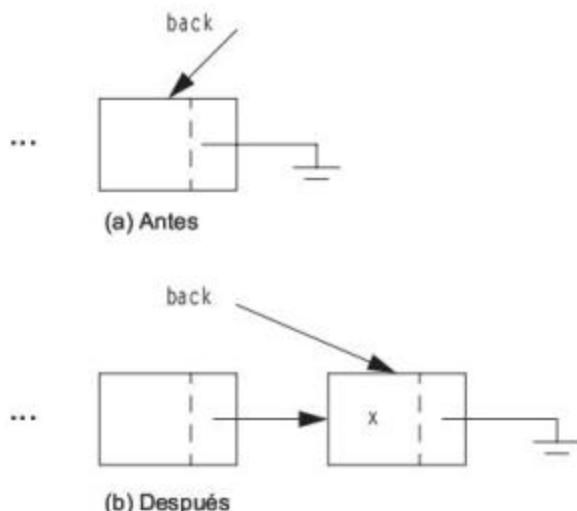


Figura 16.26 La operación enqueue para la implementación basada en lista enlazada.

```
1  /**
2  * Obtener el elemento menos recientemente insertado en la cola.
3  * No modifica la cola.
4  * @return el elemento menos recientemente insertado en la cola.
5  * @throws UnderflowException si la cola está vacía.
6  */
7  public AnyType getFront( )
8  {
9      if( isEmpty( ) )
10         throw new UnderflowException( "ListQueue getFront" );
11         return front.element;
12     }
13
14 /**
15 * Hacer que la cola pase a estar lógicamente vacía.
16 */
17 public void makeEmpty( )
18 {
19     front = null;
20     back = null;
21 }
22
23 /**
24 * Comprobar si la cola está lógicamente vacía.
25 */
26 public boolean isEmpty( )
27 {
28     return front == null;
29 }
```

Figura 16.27 Rutinas de soporte para la clase ListQueue.

Los métodos restantes para la clase ListQueue son idénticos a las rutinas correspondientes de ListStack. Se muestran en la Figura 16.27.

### 16.3 Comparación de los dos métodos

Tanto las versiones basadas en matriz como las basadas en lista enlazada requieren un tiempo constante por cada operación. Por tanto, son tan rápidas que es poco probable que lleguen a constituir el cuello de botella de cualquier algoritmo, así que, a ese respecto, raramente importa qué versión utilicemos.

Las versiones basadas en matriz de estas estructuras de datos serán probablemente más rápidas que sus equivalentes basadas en lista enlazada, especialmente si existe una estimación precisa

La comparación entre las implementaciones basada en matriz y en lista enlazada representa un compromiso clásico entre tiempo y espacio.

de la capacidad requerida. Si se proporciona un constructor adicional para especificar la capacidad inicial (véase el Ejercicio 16.4) y la estimación es correcta, no llega a realizarse ninguna operación de duplicación de tamaño. Asimismo, el acceso secuencial proporcionado por un matriz suele ser más rápido que el acceso potencialmente no secuencial ofrecido por el mecanismo de asignación dinámica de memoria.

Sin embargo, la implementación basada en matriz presenta dos desventajas. En primer lugar, para las colas, la implementación basada en matriz es algo más compleja que la implementación basada en lista enlazada, debido a la combinación del mecanismo de encadenamiento circular y de duplicación del tamaño de la matriz. Nuestra implementación del mecanismo de duplicación del tamaño de la matriz no es lo más eficiente posible (véase el Ejercicio 16.7), por lo que una implementación más rápida de la cola requeriría unas cuantas líneas adicionales de código. Incluso la implementación matricial de la pila utiliza unas cuantas líneas más de código que su equivalente basada en lista enlazada.

La segunda desventaja afecta a otros lenguajes, aunque no a Java. Al duplicar el tamaño de la matriz, requerimos temporalmente el triple de espacio que el número de elementos de datos sugiere. La razón es que, al duplicar la matriz, necesitamos disponer de un medio para almacenar tanto la matriz antigua como la nueva (de tamaño doble). Además, para un tamaño pico de la cola, la matriz estará entre un 50 y un 100 por ciento llena, en promedio, estará llena al 75 por ciento, de modo que por cada tres elementos de la matriz habrá una posición vacía. El espacio desperdiciado será por tanto de un 33 por ciento como promedio, y del 100 por cien cuando la matriz solo esté llena a la mitad. Como hemos explicado anteriormente, en Java, cada elemento de la matriz es simplemente una referencia. En otros lenguajes, como C++, los objetos se almacenan directamente, en lugar de mediante referencia. En estos lenguajes, el espacio desperdiciado podría ser significativo, si lo comparamos con el de la versión basada en lista enlazada, que solo utiliza una referencia adicional por cada elemento.

## 16.4 La clase `java.util.Stack`

La API de Colecciones proporciona una clase `Stack`. La clase `Stack` de `java.util` se considera una clase heredada y no se utiliza ampliamente. La Figura 16.28 proporciona una implementación.

```

1 package weiss.util;
2
3 /**
4 * Clase Stack. A diferencia de java.util.Stack, no es una ampliación de
5 * Vector. Este es el mínimo conjunto utilizable de operaciones.
6 */
7 public class Stack<AnyType> implements java.io.Serializable
8 {
9     public Stack( )

```

Continúa

Figura 16.28 Una clase `Stack` simplificada, en el estilo de la API de Colecciones, basada en la clase `ArrayList`.

```
10  {
11      items = new ArrayList<AnyType>();
12  }
13
14 public AnyType push( AnyType x )
15 {
16     items.add( x );
17     return x;
18 }
19
20 public AnyType pop( )
21 {
22     if( isEmpty( ) )
23         throw new EmptyStackException( );
24     return items.remove( items.size( ) - 1 );
25 }
26
27 public AnyType peek( )
28 {
29     if( isEmpty( ) )
30         throw new EmptyStackException( );
31     return items.get( items.size( ) - 1 );
32 }
33
34 public boolean isEmpty( )
35 {
36     return size( ) == 0;
37 }
38
39 public int size( )
40 {
41     return items.size( );
42 }
43
44 public void clear( )
45 {
46     items.clear( );
47 }
48
49 private ArrayList<AnyType> items;
50 }
```

Figura 16.28 (Continuación)

## 16.5 Colas de doble terminación

Una cola de doble terminación permite acceder a la cola por ambos extremos.

Una *cola de doble terminación* (también llamada *deque*) es como una cola, salvo porque se permite acceder a la misma por ambos extremos. El Ejercicio 14.15 describe una aplicación de la cola de doble terminación. En lugar de los términos *enqueue* y *dequeue*, los términos utilizados en la literatura son *addFront*, *addRear*, *removeFront* y *removeRear*, que designan operaciones para añadir y extraer un elemento en la parte frontal o posterior.

Se puede implementar una cola de doble terminación utilizando una matriz de forma bastante similar a como se hace con una cola normal. Dejamos la implementación para el lector como Ejercicio 16.2. Sin embargo, no se puede conseguir una implementación limpia empleando una lista simplemente enlazada, porque en este tipo de lista es difícil extraer el último elemento.

Java 6 añade la interfaz *Deque* al paquete *java.util*. Esta interfaz amplía *Queue*, proporcionando así automáticamente métodos tales como *add*, *remove*, *element*, *size* y *isEmpty*. También añade los métodos familiares *getFirst*, *getLast*, *addFirst*, *addLast*, *removeFirst* y *removeLast* que ya sabemos que forman parte de *java.util.LinkedList*. De hecho, en Java 6, *LinkedList* amplía *Deque*. Adicionalmente, Java 6 proporciona una nueva clase, *ArrayDeque*, que implementa *Deque*. La clase *ArrayDeque* utiliza una eficiente implementación matricial del tipo descrito en este capítulo, y puede ser algo más rápida para operaciones con colas que *LinkedList*.

## Resumen

En este capítulo hemos descrito la implementación de las clases de pila y de cola. Ambas clases se pueden implementar utilizando una matriz contigua o una lista enlazada. En cada caso, todas las operaciones utilizan un tiempo constante, por lo que todas las operaciones son rápidas.



### Conceptos clave

**cola de doble terminación (deque)** Una cola a la que se puede acceder por ambos extremos. (604)

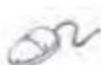
**encadenamiento circular** Se produce cuando *front* o *back* vuelven al principio de la matriz después de alcanzar el final. (589)

**implementación mediante matriz circular** El uso de la técnica de encadenamiento circular para implementar una cola. (589)



### Errores comunes

- Utilizar una implementación que no proporciona un acceso de tiempo constante es un error grave. No existe ninguna justificación para esta ineficiencia.



## Internet

Los siguientes archivos están disponibles.

<b>ArrayStack.java</b>	Contiene la implementación de una pila basada en matriz.
<b>ArrayQueue.java</b>	Contiene la implementación de una cola basada en matriz.
<b>ListStack.java</b>	Contiene la implementación de una pila basada en lista enlazada.
<b>ListQueue.java</b>	Contiene la implementación de una cola basada en lista enlazada.
<b>Stack.java</b>	Contiene la implementación de una pila de la API de Colecciones.



## Ejercicios

### EN RESUMEN

- 16.1** Dibuje las estructuras de datos de pila y de cola (tanto para la implementación basada en matriz como para la basada en lista enlazada) para cada paso de la siguiente secuencia: *add(1)*, *add(2)*, *remove*, *add(3)*, *add(4)*, *remove*, *remove*, *add(5)*. Suponga un tamaño inicial igual a 4 para la implementación basada en matriz.

### EN LA PRÁCTICA

- 16.2** En el paquete `weiss.util`, proporcione una interfaz `Deque`, una clase `ArrayList` y modifique `LinkedList` para implementar `Deque`.
- 16.3** Escriba una rutina `main` que declare y utilice una pila de `Integer` y una pila de `Double` simultáneamente.
- 16.4** Añada constructores a las clases `ArrayStack` y `ArrayQueue` que permitan al usuario especificar una capacidad inicial.
- 16.5** Compare los tiempos de ejecución para las versiones de la clase de pila basadas en matriz y en lista enlazada. Utilice objetos `Integer`.
- 16.6** Implemente la clase de cola basada en matriz con un `ArrayList`. ¿Cuáles son las ventajas y desventajas de esta solución?
- 16.7** Para la implementación de cola presentada en la Sección 16.1.2, muestre cómo copiar los elementos de la cola en la operación `doubleQueue` sin efectuar llamadas a `increment`.
- 16.8** Implemente la clase de pila basada en matriz con un `ArrayList`. ¿Cuáles son las ventajas y desventajas de esta solución?

## PROYECTOS DE PROGRAMACIÓN

- 16.9** Suponga que deseamos añadir la operación `findMin` (pero no `deleteMin`) al repertorio de `Deque`. Implemente este clase utilizando cuatro pilas. Si un borrado vacía una pila, tendrá que reorganizar equitativamente los elementos restantes.
- 16.10** Una cola de doble terminación con salida restringida admite inserciones en ambos extremos, pero los accesos y los borrados solo pueden tener lugar en la parte frontal de la cola. Implemente esta estructura de datos con una lista simplemente enlazada.
- 16.11** Suponga que desea añadir la operación `findMin` (pero no `deleteMin`) al repertorio de operaciones de la pila. Implemente esta clase mediante dos pilas como se describe en el Ejercicio 6.6.

## Listas enlazadas

En el Capítulo 16 hemos visto que se pueden utilizar las listas enlazadas para almacenar elementos de manera no contigua. Las listas enlazadas utilizadas en ese capítulo estaban simplificadas, realizándose todos los accesos a través de uno de los extremos de la lista.

En este capítulo veremos

- Cómo permitir el acceso a cualquier elemento utilizando una lista enlazada de carácter general.
- Los algoritmos generales para las operaciones con listas enlazadas.
- Cómo la clase iteradora proporciona un mecanismo seguro para recorrer las listas enlazadas y acceder a ellas.
- Variantes de las listas, como las listas doblemente enlazadas y las listas circularmente enlazadas.
- Cómo utilizar la herencia para obtener una clase de lista enlazada ordenada.
- Cómo implementar la clase `LinkedList` de la API de Colecciones.

### 17.1 Ideas básicas

En este capítulo vamos a implementar la lista enlazada y a permitir un acceso general (operaciones arbitrarias de inserción, borrado y consulta) a la lista. La lista enlazada básica está formada por una colección de nodos conectados, asignados dinámicamente. En una *lista simplemente enlazada*, cada nodo está compuesto por cada elemento de datos y un enlace al siguiente nodo de la lista. El último nodo de la lista tiene un enlace `next` con el valor `null`. En esta sección vamos a asumir que el nodo está dado por la siguiente declaración de `ListNode`, que no utiliza genéricos:

```
class ListNode
{
    Object element;
    ListNode next;
}
```

El primer nodo de la lista enlazada es accesible mediante una referencia, como se muestra en la Figura 17.1. Podemos imprimir en la lista enlazada o realizar búsquedas en la misma comenzando por el primer elemento y siguiendo la cadena de enlaces `next`. Las dos operaciones básicas que hay que realizar son la inserción y el borrado de un elemento arbitrario `x`.

La inserción consiste en introducir un nodo en la lista y se puede llevar a cabo con una única instrucción.

Para la inserción, debemos definir dónde debe tener lugar la inserción. Si tenemos una referencia a algún nodo de la lista, el lugar más fácil en el que realizar la inserción es inmediatamente después de dicho elemento. Por ejemplo, la Figura 17.2 muestra cómo insertar `x` después del elemento `a` en una lista enlazada. Tenemos que llevar a cabo los siguientes pasos:

```
tmp = new ListNode( );           // Crear un nuevo nodo
tmp.element = x;                // Colocar x en el miembro element
tmp.next = current.next;         // El nodo siguiente a x es b
current.next = tmp;              // El nodo siguiente a a es x
```

Como resultado de estas instrucciones, la antigua lista `... a, b, ...` aparecerá ahora como `... a, x, b, ...`. Podemos simplificar el código si el `ListNode` dispone de un constructor que inicialice los miembros de datos directamente. En ese caso, obtenemos

```
tmp = new ListNode( x, current.next ); // Crear un nuevo nodo
current.next = tmp;                  // El nodo siguiente a a es x
```

Como vemos ahora, `tmp` ya no hace falta. Por tanto, podemos limitarnos a ejecutar la única línea de código

```
current.next = new ListNode( x, current.next );
```

La extracción se puede realizar soslayando el nodo que queramos extraer. Necesitamos una referencia al nodo anterior al que vamos a extraer de la lista.

Las operaciones con listas enlazadas utilizan únicamente un número constante de movimientos de datos.

El comando de extracción se puede ejecutar modificando un único enlace. En la Figura 17.3 se muestra que, para eliminar el elemento `x` de la lista enlazada, basta con hacer que `current` sea el nodo anterior a `x` y luego hacer que el enlace al siguiente nodo de `current` soslaye a `x`. Esta operación se expresa mediante la instrucción

```
current.next = current.next.next;
```

La lista `... a, x, b, ...` ahora aparecerá como `... a, b, ...`

La discusión anterior resume los principios básicos de la inserción y extracción de elementos en lugares arbitrarios de una lista enlazada. La propiedad fundamental de una lista enlazada es que los cambios en la misma pueden realizarse utilizando únicamente un número constante de movimientos de

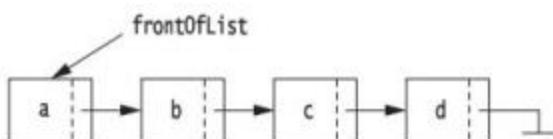
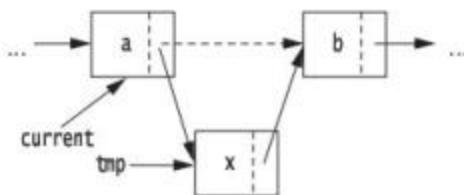
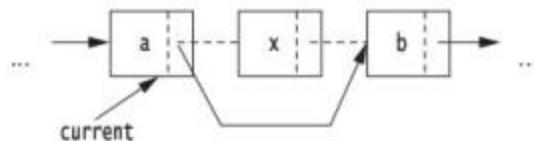


Figura 17.1 Lista enlazada básica.



**Figura 17.2** Inserción en una lista enlazada: crear un nuevo nodo (`tmp`), copiar en él `x`, configurar el enlace al siguiente nodo de `tmp` y configurar el enlace al siguiente nodo de `current`.



**Figura 17.3** Borrado en una lista enlazada.

datos, lo que constituye una mejora significativa con respecto a una implementación basada en matriz. Mantener la contigüidad en una matriz significa que cada vez que se añade o borra un elemento, todos los elementos situados tras él en la lista deben ser desplazados.

### 17.1.1 Nodos de cabecera

Hay un problema con esta descripción básica: se supone que cada vez que eliminemos un elemento `x`, siempre habrá un elemento anterior que permita soslayar `x`. En consecuencia, la eliminación del primer elemento de la lista enlazada se convierte en un caso especial. De forma similar, la rutina de inserción no nos permite insertar un elemento como nuevo primer elemento de la lista. La razón es que las inserciones deben realizarse después de algún elemento existente. Por tanto, aunque el algoritmo básico funciona correctamente, es preciso resolver algunos casos especiales que resultan molestos.

Los casos especiales siempre resultan problemáticos en el diseño de algoritmos y, frecuentemente, provocan la aparición de errores en el código. En consecuencia, generalmente es preferible escribir código que evite la existencia de casos especiales. Una forma de hacer esto, en este caso, consiste en introducir un nodo de cabecera.

Un *nodo de cabecera* es un nodo extra en una lista enlazada que no almacena ningún dato, pero sirve para satisfacer el requisito de que todo nodo que contenga un elemento disponga de un nodo anterior en la lista. El nodo de cabecera para la lista `a`, `b`, `c` se indica en la Figura 17.4. Observe que, a ya no es un caso especial. Se puede borrar al igual que cualquier otro nodo, haciendo que `current` haga referencia al nodo situado antes de él. También podemos añadir un nuevo primer elemento a la lista haciendo que `current` sea igual al nodo de cabecera e invocando a la rutina de inserción. Utilizando el nodo de cabecera, simplificamos enormemente el código –con un coste adicional despreciable en términos de espacio. En aplicaciones más

Un nodo de cabecera no alberga ningún dato, pero sirve para satisfacer el requisito de que todo nodo tenga un nodo anterior. Un nodo de cabecera nos permite evitar casos especiales como la inserción de un nuevo primer elemento y la extracción del primer elemento.

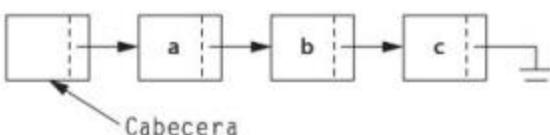


Figura 17.4 Utilización de un nodo de cabecera para la lista enlazada.

complejas, los nodos de cabecera no solo simplifican el código, sino que también mejoran la velocidad, ya que, después de todo, la realización de menos comprobaciones implica un menor tiempo de ejecución.

El uso de un nodo de cabecera es en cierto modo controvertido. Algunas personas argumentan que tratar de evitar casos especiales no es razón suficiente para añadir posiciones ficticias. Esas personas contemplan el uso de nodos de cabecera como poco más que un truco de programación de estilo anticuado. Aun así, nosotros los usamos aquí precisamente porque nos permiten ilustrar las manipulaciones básicas de los enlaces, sin oscurecer el código con casos especiales. El usar o no un nodo de cabecera es cuestión de preferencias personales. Además, en una implementación de clase, el uso de esos nodos de cabecera sería completamente transparente para el usuario. Sin embargo, debemos tener cuidado: la rutina de impresión debe saltarse el nodo de cabecera, al igual que deben hacerlo todas las rutinas de búsqueda. Situarse al principio de la lista significa ahora hacer que la posición actual sea `header.next`, etc. Además, como se muestra en la Figura 17.5, con un nodo de cabecera ficticio, una lista estará vacía si `header.next` es `null`.

### 17.1.2 Clases iteradoras

Almacenando una posición actual en una clase de lista, nos cercioramos de que el acceso esté controlado.

La estrategia primitiva típica identifica una lista enlazada mediante una referencia al nodo de cabecera. Se puede entonces acceder a cada elemento individual de la lista proporcionando una referencia al nodo que lo almacena. El problema con esta estrategia es que la comprobación de errores es complicada. Por ejemplo, un usuario podría pasar una referencia a algo que sea un nodo contenido en una lista diferente. Una manera de garantizar que esto no suceda consiste en almacenar una posición actual como parte de una clase de lista. Para hacer eso, añadimos un nuevo miembro de datos, `current`. Después, como todos los accesos a la lista pasan a través de los métodos de la clase, podemos cerciorarnos de que `current` represente siempre a un nodo de la lista, al nodo de cabecera o `null`.

Este esquema presenta un problema: disponiendo de una única posición, no proporcionamos ningún tipo de soporte para el caso de que haya dos iteradores que necesiten acceder a la lista independientemente. Una forma de evitar este problema consiste en definir una *clase iteradora*

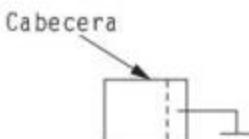


Figura 17.5 Lista vacía cuando se utiliza un nodo de cabecera.

separada, que mantendrá una noción de su posición actual. Una clase de lista, entonces, no mantendría ninguna noción de una posición actual y solo dispondría de métodos que trataran la lista como una unidad, tal como `isEmpty` y `makeEmpty`, o que acepten un iterador como parámetro, como por ejemplo `insert`. Las rutinas que dependan solo del propio iterador, como por ejemplo la rutina `advance` que hace avanzar al iterador a la siguiente posición, residirían en la clase iteradora. El acceso a la lista estaría garantizado haciendo que la clase iteradora fuera una clase interna o tuviera visibilidad de paquete. Podemos contemplar cada instancia de una clase iteradora como algo en lo que solo están permitidas las operaciones legales con las listas como avanzar a través de la lista.

En la Sección 17.2 definimos una clase de lista genérica `LinkedList` y una clase iteradora `LinkedListIterator`. La clase `LinkedList` no tiene la misma semántica que `java.util.LinkedList`. Sin embargo, más adelante en el capítulo definiremos una versión que sí la tiene. Para ilustrar cómo funciona la versión no estándar, examinemos un método estático que devuelve el tamaño de una lista enlazada, como se muestra en la Figura 17.6. Declaramos `itr` como un iterador que puede acceder a la lista enlazada `theList`.

Inicializamos `itr` haciendo referencia al iterador proporcionado por `theList` (saltándonos la cabecera, por supuesto) haciendo referencia al nodo dado por `theList.first()`.

La comprobación `itr.isValid()` trata de asemejarse a la comprobación `p!=null` que llevaremos a cabo si `p` fuera una referencia visible a un nodo. Por último, la expresión `itr.advance()` se asemeja a la instrucción convencional `p=p.next`.

Así, mientras que la clase iteradora defina unas cuantas operaciones simples, podemos iterar a través de la lista de manera natural. En la Sección 17.2 proporcionaremos su implementación en Java. Las rutinas son sorprendentemente simples.

Existe un paralelismo natural entre los métodos definidos en las clases `LinkedList` y `LinkedListIterator` y los contenidos en la clase `LinkedList` de la API de Colecciones. Por ejemplo, el método `advance` de `LinkedListIterator` es aproximadamente equivalente a `hasNext` en los iteradores de la API de Colecciones. La clase de lista de la Sección 17.2 es más simple que la clase `LinkedList` de la API de Colecciones; por ello, permite ilustrar diversos puntos básicos

```

1 // En esta rutina, LinkedList y LinkedListIterator son las
2 // clases escritas en la Sección 17.2.
3 public static <AnyType> int listSize( LinkedList<AnyType> theList )
4 {
5     LinkedListIterator<AnyType> itr;
6     int size = 0;
7
8     for( itr = theList.first(); itr.isValid(); itr.advance() )
9         size++;
10
11    return size;
12 }
```

Una clase iteradora mantiene una posición actual y suele tener visibilidad de paquete o ser una clase interna de una clase de lista (o de otro tipo de contenedor).

Figura 17.6 Un método estático que devuelve el tamaño de una lista.

y merece la pena examinarla. En la Sección 17.5 implementaremos la mayor parte de la clase `LinkedList` de la API de Colecciones.

## 17.2 Implementación Java

Como se sugiere en la descripción anterior, una lista se implementa mediante tres clases genéricas separadas: una clase es la propia lista (`LinkedList`), otra representa el nodo (`ListNode`) y la tercera representa la posición (`LinkedListIterator`).

`ListNode` se ha presentado en el Capítulo 16. A continuación, la Figura 17.7 presenta la clase que implementa el concepto de posición, es decir `LinkedListIterator`. La clase almacena una referencia a un `ListNode`, que representa la posición actual del iterador. El método `isValid` devuelve `true` si la posición no está situada más allá del extremo de la lista; `retrieve` devuelve el elemento almacenado en la posición actual y `advance` hace avanzar la posición actual hasta la siguiente posición. El constructor de `LinkedListIterator` requiere una referencia a un nodo que deba ser contemplado como nodo actual. Observe que este constructor tiene visibilidad de paquete y no puede, por tanto, ser utilizado por los métodos cliente. En lugar de ello, la idea general es que la clase `LinkedList` devuelva objetos preconstruidos `LinkedListIterator`, según sea apropiado; `LinkedList` se encuentra en el mismo paquete que `LinkedListIterator`, así que puede invocar al constructor de `LinkedListIterator`.

El esqueleto de la clase `LinkedList` se muestra en la Figura 17.8. El único miembro de datos es una referencia al nodo de cabecera asignado por el constructor. `isEmpty` es una corta rutina de una sola línea que se implementa fácilmente. Los métodos `zeroth` y `first` devuelven iteradores correspondientes a la cabecera y al primer elemento, respectivamente, como se muestra en la Figura 17.9. Otras rutinas buscan en la lista algún elemento o modifican la lista mediante inserción o borrado, y las veremos más adelante.

La Figura 17.10 ilustra cómo interactúan las clases `LinkedList` y `LinkedListIterator`. El método `printList` imprime el contenido de una lista. `printList` utiliza únicamente métodos públicos y una secuencia típica de iteración consistente en obtener un punto de partida (mediante `first`), comprobar que no hemos ido más allá del final (mediante `isValid`) y avanzar en cada iteración (mediante `advance`).

Volvemos a analizar la cuestión de si son necesarias las tres clases. Por ejemplo, ¿no podríamos limitarnos a hacer que la clase `LinkedList` mantuviera una noción de la posición actual? Aunque esta opción es factible y funciona para muchas aplicaciones, la utilización de una clase iteradora separada expresa la abstracción de que la posición y la lista son, en realidad, objetos separados. Además, permite acceder simultáneamente a múltiples posiciones de la lista. Por ejemplo, para eliminar una sublista de una lista, podemos añadir fácilmente una operación `remove` a la clase de lista que utilice dos iteradores para especificar los puntos inicial y final de la sublista que hay que eliminar. Sin la clase iteradora, esta acción sería más difícil de expresar.

Ahora podemos implementar los restantes métodos de `LinkedList`. El primero es `find`, mostrado en la Figura 17.11, que devuelve la posición de un cierto elemento dentro de la lista. La línea 10 aprovecha el hecho de que la operación `and` (`&&`) está cortocircuitada. Si la primera mitad de la operación `and` da un resultado falso, el resultado global será automáticamente falso y la segunda mitad no se evaluará.

Se utiliza el mecanismo de cortocircuitado en la línea 10 dentro de la rutina `find` y en la parte correspondiente de la rutina `remove`.

```

1 package weiss.nonstandard;
2
3 // Clase LinkedListIterator; mantiene una "posición actual"
4 //
5 // CONSTRUCCIÓN: solo visibilidad de paquete, con un ListNode
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void advance( ) --> Avanzar
9 // boolean isValid( ) --> True si está en una posición válida de la lista
10 // AnyType retrieve --> Devuelve el elemento en la posición actual
11
12 public class LinkedListIterator<AnyType>
13 {
14     /**
15      * Construir el iterador de la lista
16      * @param theNode cualquier nodo de la lista enlazada.
17      */
18     LinkedListIterator( ListNode<AnyType> theNode )
19     {
20         current = theNode;
21     }
22
23     /**
24      * Comprobar si la posición actual en la lista es una posición válida.
25      * @return true si la posición actual es válida.
26      */
27     public boolean isValid( )
28     {
29         return current != null;
30     }
31
32     /**
33      * Devuelve el elemento almacenado en la posición actual.
34      * @return el elemento almacenado o null si la posición actual
35      * no está en la lista.
36      */
37     public AnyType retrieve( )
38     {
39         return isValid( ) ? current.element : null;
40     }
41
42     /**
43      * Avanzar la posición actual hasta el siguiente nodo de la lista.
44      * Si la posición actual es null, no hacer nada.
45      */
46     public void advance( )
47     {
48         if( isValid( ) )
49             current = current.next;
50     }
51
52     ListNode<AnyType> current; // Posición actual
53 }
```

Figura 17.7 La clase LinkedListIterator.

```
1 package weiss.nonstandard;
2
3 // Clase LinkedList
4 //
5 // CONSTRUCCIÓN: sin ningún inicializador
6 // El acceso es a través de la clase LinkedListIterator
7 //
8 // *****OPERACIONES PÚBLICAS*****
9 // boolean isEmpty( ) --> Devolver true si está vacía; false, en otro caso
10 // void makeEmpty( ) --> Extraer todos los elementos
11 // LinkedListIterator zeroth( )
12 // --> Devolver posición al nodo anterior al primero
13 // LinkedListIterator first( )
14 // --> Devolver primera posición
15 // void insert( x, p ) --> Insertar x después de la pos. actual p del itera.
16 // void remove( x ) --> Extraer x
17 // LinkedListIterator find( x )
18 // --> Devolver posición correspondiente a x
19 // LinkedListIterator findPrevious( x )
20 // --> Devolver posición anterior a x
21 // *****ERRORES*****
22 // No hay errores especiales
23
24 public class LinkedList<AnyType>
25 {
26     public LinkedList( )
27         { /* Figura 17.9 */ }
28
29     public boolean isEmpty( )
30         { /* Figura 17.9 */ }
31     public void makeEmpty( )
32         { /* Figura 17.9 */ }
33     public LinkedListIterator<AnyType> zeroth( )
34         { /* Figura 17.9 */ }
35     public LinkedListIterator<AnyType> first( )
36         { /* Figura 17.9 */ }
37     public void insert( AnyType x, LinkedListIterator<AnyType> p )
38         { /* Figura 17.14 */ }
39     public LinkedListIterator<AnyType> find( AnyType x )
40         { /* Figura 17.11 */ }
41     public LinkedListIterator<AnyType> findPrevious( AnyType x )
42         { /* Figura 17.13 */ }
43     public void remove( Object x )
44         { /* Figura 17.12 */ }
45
46     private ListNode<AnyType> header;
47 }
```

---

Figura 17.8 Esqueleto de la clase `LinkedList`.

```
1  /**
2   * Construir la lista
3   */
4  public LinkedList( )
5  {
6      header = new ListNode<AnyType>( null );
7  }
8
9 /**
10  * Comprobar si la lista está lógicamente vacía.
11  * @return true si está vacía, false en caso contrario.
12  */
13 public boolean isEmpty( )
14 {
15     return header.next == null;
16 }
17
18 /**
19  * Hacer que la lista esté lógicamente vacía.
20  */
21 public void makeEmpty( )
22 {
23     header.next = null;
24 }
25
26 /**
27  * Devolver un iterador que represente el nodo de cabecera.
28  */
29 public LinkedListIterator<AnyType> zeroth( )
30 {
31     return new LinkedListIterator<AnyType>( header );
32 }
33
34 /**
35  * Devolver un iterador que represente el primer nodo de la lista.
36  * Esta operación es válida para listas vacías.
37  */
38 public LinkedListIterator<AnyType> first( )
39 {
40     return new LinkedListIterator<AnyType>( header.next );
41 }
```

Figura 17.9 Algunas rutinas de una sola línea de la clase `LinkedList`.

```

1 // Un método simple de impresión
2 public static <AnyType> void printList( LinkedList<AnyType> theList )
3 {
4     if( theList.isEmpty( ) )
5         System.out.print( "Empty list" );
6     else
7     {
8         LinkedListIterator<AnyType> itr = theList.first( );
9         for( ; itr.isValid( ); itr.advance( ) )
10            System.out.print( itr.retrieve( ) + " " );
11     }
12
13     System.out.println( );
14 }
```

Figura 17.10 Un método para imprimir el contenido de una `LinkedList`.

```

1 /**
2 * Devolver el iterador correspondiente al primer nodo que contenga x.
3 * @param x el elemento que hay que buscar.
4 * @return un iterator; iterador será null si no se encuentra el elemento.
5 */
6 public LinkedListIterator<AnyType> find( AnyType x )
7 {
8     ListNode<AnyType> itr = header.next;
9
10    while( itr != null && !itr.element.equals( x ) )
11        itr = itr.next;
12
13    return new LinkedListIterator<AnyType>( itr );
14 }
```

Figura 17.11 La rutina `find` para la clase `LinkedList`.

Este código no está hecho a prueba de errores: puede que existan dos iteradores, y uno de ellos podría quedar colgando si el otro elimina un nodo.

Nuestra siguiente rutina elimina algún elemento `x` de la lista. Tenemos que decidir qué hacer en caso de que `x` aparezca más de una vez o no aparezca en absoluto. Nuestra rutina elimina la primera aparición de `x` y no hace nada si no está en la lista. Para que eso suceda, localizamos `p`, que es la celda anterior a la que contiene a `x`, mediante una llamada a `findPrevious`. El código para implementar la rutina `remove` se muestra en la Figura 17.12. Este código no está hecho a prueba de errores: puede que existan dos iteradores, y que uno de ellos quede lógicamente en un limbo si el otro elimina un nodo. La rutina `findPrevious` es similar a la rutina `find` y se muestra en la Figura 17.13.

```

1  /**
2   * Eliminar la primera aparición de un elemento.
3   * @param x el elemento que hay que eliminar.
4   */
5  public void remove( AnyType x )
6  {
7      LinkedListIterator<AnyType> p = findPrevious( x );
8
9      if( p.current.next != null )
10         p.current.next = p.current.next.next; // Soslayar el nodo eliminado
11 }

```

Figura 17.12 La rutina `remove` para la clase `LinkedList`.

```

1 /**
2  * Devolver un iterador anterior al primer nodo que contenga un elemento.
3  * @param x el elemento que hay que buscar.
4  * @return un iterador si se encuentra el elemento. En caso contrario, se
5  * devuelve el iterador correspondiente al último elemento de la lista.
6  */
7 public LinkedListIterator<AnyType> findPrevious( AnyType x )
8 {
9     ListNode<AnyType> itr = header;
10
11    while( itr.next != null && !itr.next.element.equals( x ) )
12        itr = itr.next;
13
14    return new LinkedListIterator<AnyType>( itr );
15 }

```

Figura 17.13 La rutina `findPrevious` (similar a la rutina `find`) para utilización con `remove`.

La última rutina que vamos a escribir aquí es una rutina de inserción. Pasamos a la rutina el elemento que haya que insertar y una posición  $p$ . Esta rutina de inserción concreta inserta un elemento después de la posición  $p$ , como se muestra en la Figura 17.14. Observe que la rutina `insert` no hace ningún uso de la lista en la que se encuentra; solo depende de  $p$ .

La rutina `insert` tarda un tiempo constante.

Con la excepción de las rutinas `find` y `findPrevious` (y de `remove`, que llama a `findPrevious`), todas las operaciones que hemos codificado hasta el momento tardan un tiempo  $O(1)$ . Las rutinas `find` y `findPrevious` tardan un tiempo  $O(N)$  en el caso peor, porque puede que haga falta recorrer la lista completa si el elemento no se encuentra en la misma o está en la última posición de la lista. En promedio, el tiempo de ejecución es  $O(N)$ , porque como media habrá que recorrer la mitad de la lista.

Las rutinas `find` y `findPrevious` requieren un tiempo  $O(N)$ .

```

1  /**
2  * Insertar después de p.
3  * @param x el elemento que hay que insertar.
4  * @param p la posición anterior al elemento que se quiere insertar.
5  */
6  public void insert( AnyType x, LinkedListIterator<AnyType> p )
7  {
8      if( p != null && p.current != null )
9          p.current.next = new ListNode<AnyType>( x, p.current.next );
10 }

```

Figura 17.14 La rutina de inserción para la clase `LinkedList`.

El método `retreat` no está soportado de manera eficiente. Si esto es un problema se emplea una lista doblemente enlazada.

Obviamente, podríamos haber añadido más operaciones, pero este conjunto básico es bastante potente. Algunas operaciones, como `retreat`, no son soportadas de manera eficiente por esta versión de la lista enlazada; posteriormente en el capítulo analizaremos variantes de la lista enlazada que permiten implementar en un tiempo constante esa y otras operaciones.

### 17.3 Listas doblemente enlazadas y listas circularmente enlazadas

Una lista doblemente enlazada permite un recorrido bidireccional de la lista almacenando dos enlaces por cada nodo.

Como hemos mencionado en la Sección 17.2, la lista simplemente enlazada no soporta de manera eficiente algunas operaciones importantes. Por ejemplo, aunque es fácil desplazarse hasta el inicio de la lista, es muy costoso en términos de tiempo desplazarse hasta el final. Aunque podemos avanzar fácilmente mediante la rutina `advance`, la implementación de `retreat` (operación contraria a `advance`) no se puede realizar de manera eficiente si solo

se dispone de un enlace `next`. En algunas aplicaciones, esto puede ser un problema. Por ejemplo, al diseñar un editor de textos podemos mantener la imagen interna del archivo en forma de una lista enlazada de líneas. Queremos poder movernos hacia arriba con la misma facilidad que nos desplazamos hacia abajo en la lista; queremos poder insertar texto tanto antes como después de una línea, en lugar de limitarnos a insertar solo después; y queremos poder desplazarnos hasta la última línea rápidamente. Un momento de reflexión nos sugiere que para implementar estos procedimientos de manera eficiente, deberíamos hacer que cada nodo mantenga dos enlaces: uno al siguiente nodo de la lista y otro al nodo anterior. Entonces, para que todo sea simétrico, deberíamos disponer no solo de una cabecera, sino también de una cola. Una lista enlazada que permite un recorrido bidireccional de la misma, almacenando dos enlaces por nodo se denomina *lista doblemente enlazada*. La Figura 17.15 muestra la lista doblemente enlazada que representa a la lista formada por `a` y `b`. Cada nodo tiene ahora dos enlaces (`next` y `prev`) y las búsquedas y los desplazamientos pueden realizarse fácilmente en ambas direcciones. Obviamente, hay algunos cambios importantes con respecto a la lista simplemente enlazada.

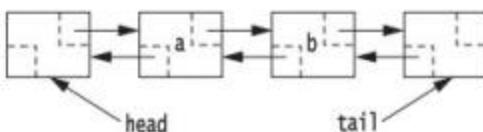


Figura 17.15 Una lista doblemente enlazada.

En primer lugar, una lista ahora estará formada ahora por una cabecera `head` y una cola `tail`, conectadas como se muestra en la Figura 17.16. Observe que `head.prev` y `tail.next` no son necesarias en los algoritmos y que ni siquiera se los inicializa. La comprobación de que la lista está vacía será ahora

```
head.next = tail
```

O

```
tail.prev = head
```

Ya no utilizamos `null` para decidir si una operación de avance nos ha llevado más allá del final de la lista. En lugar de ello, habremos ido más allá del final si `current` es `head` o `tail` (recuerde que podemos desplazarnos en cualquiera de las dos direcciones). La operación `retreat` se puede implementar mediante

```
current = current.prev;
```

Antes de describir algunas de las operaciones adicionales disponibles, consideremos cómo varían las operaciones de inserción y eliminación. Naturalmente, ahora podemos hacer tanto `insertBefore` como `insertAfter`, para insertar antes o después de un elemento. Hará falta el doble de modificaciones de enlace para `insertAfter`, ahora que empleamos listas doblemente enlazadas en lugar de simplemente enlazadas. Si escribimos cada instrucción explícitamente, obtenemos

```
newNode = new DoublyLinkedListNode( x );
newNode.prev = current;                      // Establecer enlace prev de x
newNode.next = current.next;                  // Establecer enlace next de x
newNode.prev.next = newNode;                  // Establecer enlace next de a
newNode.next.prev = newNode;                  // Establecer enlace prev de b
current = newNode;
```

La simetría exige que usemos tanto una cabecera `head` como una cola `tail` y que soportemos aproximadamente el doble de operaciones.

Cuando avanzamos más allá del extremo de la lista, nos topamos ahora con el nodo `tail` en lugar de con `null`.

La inserción y eliminación requieren el doble de modificaciones de enlaces que en una lista simplemente enlazada.

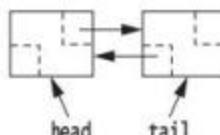


Figura 17.16 Una lista doblemente enlazada vacía.

Como mostramos anteriormente, las dos primeras modificaciones de enlace se pueden combinar en la operación de construcción de `DoublyLinkedListNode` realizada por `new`. Los cambios (en el orden 1, 2, 3, 4) se ilustran en la Figura 17.17.

La operación `remove` se puede llevar a cabo desde el nodo actual, porque podemos obtener el nodo anterior instantáneamente.

La Figura 17.17 también se puede utilizar como guía para el algoritmo de eliminación. A diferencia de la lista simplemente enlazada, podemos eliminar el nodo actual, porque tenemos disponible el nodo previo de manera automática. Por tanto, para efectuar la operación `remove` x tenemos que cambiar el enlace `next` de a y el enlace `prev` de b. Las modificaciones básicas son

```
current.prev.next = current.next;           // Establecer el enlace next de a
current.next.prev = current.prev;           // Establecer el enlace prev de b
current = head;                           // Para que current no quede colgando
```

Para realizar una implementación completa de una lista doblemente enlazada, necesitamos decidir qué operaciones soportar. Es razonable suponer que tendremos el doble de operaciones que en la lista simplemente enlazada. Cada procedimiento individual es similar a las rutinas para la lista simplemente enlazada; solo las operaciones dinámicas implican modificaciones de enlaces adicionales. Además, para muchas de las rutinas, el código está dominado por las comprobaciones de error. Aunque algunas de las comprobaciones cambiarán (por ejemplo, no tenemos que comprobar la existencia de `null`), no son en absoluto más complejas. En la Sección 17.5, utilizaremos una lista doblemente enlazada para implementar la clase de lista enlazada de la API de Colecciones, junto con sus iteradores asociados. Hay un montón de rutinas, pero la mayoría son cortas.

En una lista circularmente enlazada, el enlace `next` de la última celda hace referencia a `first`. Esta técnica es útil cuando se necesita un encadenamiento circular.

Un convenio bastante popular consiste en crear una *lista circularmente enlazada*, en la que el enlace `next` de la última celda hace referencia a `first`, lo que se puede realizar con o sin cabecera. Típicamente se lleva a cabo sin una cabecera, porque el propósito principal de la cabecera es garantizar que todo nodo tenga un nodo anterior, lo cual ya será cierto de manera natural para una lista circularmente enlazada no vacía. Sin una cabecera, el único caso especial que nos queda es el de la lista vacía. Mantenemos una referencia al primer nodo, pero eso no es lo mismo que un nodo de cabecera.

Podemos utilizar listas circularmente enlazadas y listas doblemente enlazadas simultáneamente, como se muestra en la Figura 17.18. La lista circular resulta útil cuando queremos que la búsqueda esté enlazada circularmente, como suele suceder en algunos editores de texto. En el Ejercicio 17.19 le pediremos que implemente una lista doblemente enlazada y circularmente enlazada.

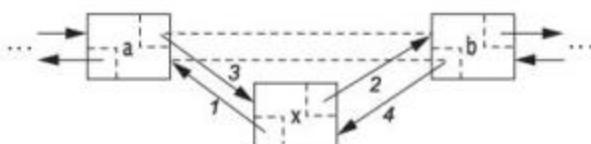


Figura 17.17 Inserción en una lista doblemente enlazada, obteniendo el nuevo nodo y cambiando luego los punteros en el orden indicado.

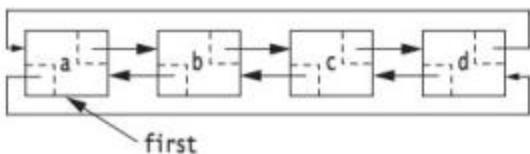


Figura 17.18 Una lista circular y doblemente enlazada.

## 17.4 Listas enlazadas ordenadas

En ocasiones, queremos mantener los elementos de una lista enlazada ordenados, lo cual es algo que podemos llevar a cabo con una *lista enlazada ordenada*. La diferencia fundamental entre una lista enlazada ordenada y una lista enlazada desordenada es la rutina de inserción. Ciertamente, podemos obtener una clase de lista ordenada simplemente modificando la rutina de inserción de nuestra clase de lista ya escrita. Puesto que la rutina `insert` es parte de la clase `LinkedList`, deberíamos ser capaces de definir una nueva clase derivada, `SortedLinkedList`, a partir de `LinkedList`. Efectivamente podemos definirla, y esa clase se muestra en la Figura 17.19.

Podemos mantener los elementos ordenados derivando una clase `SortedLinkedList` a partir de `LinkedList`.

La nueva clase tiene dos versiones de `insert`. Una versión toma una posición y luego la ignora; el punto de inserción está determinado exclusivamente por el orden de los elementos. La otra versión de `insert` requiere algo más de código.

La rutina `insert` de un parámetro utiliza dos objetos `LinkedListIterator` para recorrer la lista correspondiente hasta encontrar el punto de inserción correcto. En dicho punto, podemos aplicar la rutina `insert` de la clase base.

## 17.5 Implementación de la clase `LinkedList` de la API de Colecciones

En esta sección vamos a implementar la clase `LinkedList` de la API de Colecciones, de la que hemos hablado en la Sección 6.5. Aunque presentamos una gran cantidad de código, ya hemos descrito la mayor parte de las técnicas anteriormente en este capítulo.

Como hemos indicado previamente, necesitamos una clase para almacenar el nodo básico de la lista, otra clase para el iterador y otra clase para la propia lista. El esqueleto para la clase `LinkedList` se muestra en la Figura 17.20. `LinkedList` implementa las interfaces `List` y `Queue` y, como suele ser habitual, amplía `AbstractCollection`. En la línea 5 comienza la declaración de la clase `Node`, que es anidada y privada. En la línea 7 comienza la declaración de la clase `LinkedListIterator`, que es una clase interna privada. El patrón iterador ya se ha descrito en el Capítulo 6. Este mismo patrón ya se utilizó en la implementación de `ArrayList` con clases internas en el Capítulo 15.

La clase de lista lleva la cuenta de su tamaño en un miembro de datos declarado en la línea 54. Utilizamos este enfoque para que el método `size` pueda ejecutarse en un tiempo constante. `modCount` es utilizado por los iteradores para determinar si la lista ha sido modificada mientras se está llevando a cabo una iteración. Esa misma idea se ha utilizado en `ArrayList`. `beginMarker` y

```
1 package weiss.nonstandard;
2
3 // Clase SortedLinkedList
4 //
5 // CONSTRUCCIÓN: sin ningún inicializador
6 // Acceso a través de la clase LinkedListIterator
7 //
8 // *****OPERACIONES PÚBLICAS*****
9 // void insert( x ) --> Insertar x
10 // void insert( x, p ) --> Insertar x (ignorar p)
11 // Todas las operaciones restantes de LinkedList
12 // *****ERRORES*****
13 // Sin errores especiales
14
15 public class SortedLinkedList<AnyType extends Comparable<? super AnyType>>
16                     extends LinkedList<AnyType>
17 {
18     /**
19      * Insertar después de p.
20      * @param x el elemento que hay que insertar.
21      * @param p este parámetro se ignora.
22      */
23     public void insert( AnyType x, LinkedListIterator<AnyType> p )
24     {
25         insert( x );
26     }
27
28     /**
29      * Insertar en el orden correcto.
30      * @param x el elemento que hay que insertar.
31      */
32     public void insert( AnyType x )
33     {
34         LinkedListIterator<AnyType> prev = zeroth( );
35         LinkedListIterator<AnyType> curr = first( );
36
37         while( curr.isValid( ) && x.compareTo( curr.retrieve( ) ) > 0 )
38         {
39             prev.advance( );
40             curr.advance( );
41         }
42
43         super.insert( x, prev );
44     }
45 }
```

---

Figura 17.19 La clase SortedLinkedList, en la que las inserciones están restringidas, realizándose según el orden de la lista.

```
1 package weiss.util;
2 public class LinkedList<AnyType> extends AbstractCollection<AnyType>
3                     implements List<AnyType>, Queue<AnyType>
4 {
5     private static class Node<AnyType>
6         { /* Figura 17.21 */ }
7     private class LinkedListIterator<AnyType> implements ListIterator<AnyType>
8         { /* Figura 17.30 */ }
9
10    public LinkedList( )
11        { /* Figura 17.22 */ }
12    public LinkedList( Collection<? extends AnyType> other )
13        { /* Figura 17.22 */ }
14
15    public int size( )
16        { /* Figura 17.23 */ }
17    public boolean contains( Object x )
18        { /* Figura 17.23 */ }
19    public boolean add( AnyType x )
20        { /* Figura 17.24 */ }
21    public void add( int idx, AnyType x )
22        { /* Figura 17.24 */ }
23    public void addFirst( AnyType x )
24        { /* Figura 17.24 */ }
25    public void addLast( AnyType x )
26        { /* Figura 17.24 */ }
27    public AnyType element( )
28        { /* Añadido en Java 5; igual que getFirst */ }
29    public AnyType getFirst( )
30        { /* Figura 17.25 */ }
31    public AnyType getLast( )
32        { /* Figura 17.25 */ }
33    public AnyType remove( )
34        { /* Añadido en Java 5; igual que removeFirst */ }
35    public AnyType removeFirst( )
36        { /* Figura 17.27 */ }
37    public AnyType removeLast( )
38        { /* Figura 17.27 */ }
39    public boolean remove( Object x )
40        { /* Figura 17.28 */ }
```

*Continúa*

Figura 17.20 El esqueleto de clase para la clase `LinkedList` estándar.

```
41 public AnyType get( int idx )
42     { /* Figura 17.25 */ }
43 public AnyType set( int idx, AnyType newVal )
44     { /* Figura 17.25 */ }
45 public AnyType remove( int idx )
46     { /* Figura 17.27 */ }
47 public void clear( )
48     { /* Figura 17.22 */ }
49 public Iterator<AnyType> iterator( )
50     { /* Figura 17.29 */ }
51 public ListIterator<AnyType> listIterator( int idx )
52     { /* Figura 17.29 */ }
53
54 private int theSize;
55 private Node<AnyType> beginMarker;
56 private Node<AnyType> endMarker;
57 private int modCount = 0;
58
59 private static final Node<AnyType> NOT_FOUND = null;
60 private Node<AnyType> findPos( Object x )
61     { /* Figura 17.23 */ }
62 private AnyType remove( Node<AnyType> p )
63     { /* Figura 17.27 */ }
64 private Node<AnyType> getNode( int idx )
65     { /* Figura 17.26 */ }
66 }
```

---

Figura 17.20 (Continuación).

`endMarker` corresponden a `head` y `tail` de la Sección 17.3. Todos los métodos utilizan signaturas que ya hemos mostrado anteriormente.

La Figura 17.21 muestra la clase `Node`, que es similar a la clase `ListNode`. La principal diferencia es que, como utilizamos una lista doblemente enlazada, disponemos de sendos enlaces `prev` y `next`.

Observe que las clases internas y anidadas se consideran parte de la clase externa. Por tanto, independientemente de si los campos de datos de `Node` son públicos o privados, esos campos serán visibles para la clase `LinkedList`. Puesto que la propia clase `Node` es privada, solo la clase `LinkedList` será capaz de ver que `Node` es un tipo válido. En consecuencia, en esta instancia, no importa si los campos de datos del nodo son públicos y privados. Podría importar si `Node` fuera ampliado dentro de `LinkedList`, y este sería un argumento en contra de hacer los datos privados. Por otro lado, puesto que en nuestra implementación, `LinkedList` está accediendo directamente a los campos de datos de `Node`, en lugar de invocando métodos, parece más apropiado marcar los datos como públicos.

```
1  /**
2   * Este es el nodo de la lista doblemente enlazada.
3   */
4  private static class Node<AnyType>
5  {
6      public Node( AnyType d, Node<AnyType> p, Node<AnyType> n )
7      {
8          data = d; prev = p; next = n;
9      }
10
11     public AnyType data;
12     public Node<AnyType> prev;
13     public Node<AnyType> next;
14 }
```

Figura 17.21 Clase anidada `Node` para la clase `LinkedList` estándar.

La implementación de `LinkedList` comienza en la Figura 17.22, en la que mostramos los constructores y el método `clear`. En conjunto, hay pocas cosas nuevas aquí. Hemos combinado bastante del código de la `LinkedList` no estándar con los conceptos presentados en la Sección 17.3.

La Figura 17.23 muestra `size`, que es trivial y `contains`, que también es trivial porque invoca a la rutina `findPos` privada que realiza todo el trabajo. `findPos` trata con los valores `null` en las líneas 30 a 34; si no fuera por eso, estaría compuesta por cuatro líneas de código.

La Figura 17.24 muestra los diversos métodos `add`. Todos ellos terminan confluyendo en el último método `add` en las líneas 39 a 47, que inserta en la lista doblemente enlazada, tal como se hacía en la Sección 17.3. Requiere una rutina privada, `getNode`, cuya implementación analizaremos enseguida. `getNode` devuelve una referencia al nodo situado en el índice `idx`. Para que esto sea adecuado para `addLast`, `getNode` comienza su búsqueda a partir del extremo más próximo al nodo objetivo.

La Figura 17.25 detalla los diversos métodos `get` más un método `set`. Pocas cosas hay que sean especiales en cualquiera de estas rutinas. El método `element` de la interfaz `Queue` no se muestra. La Figura 17.26 contiene el método `getNode` privado que hemos mencionado anteriormente. La versión de tres parámetros se necesita específicamente para `add` y el constructor `LinkedListIterator`; la versión más común de un parámetro se emplea para todas las restantes llamadas a `getNode`. Si el índice representa un nodo en la primera mitad de la lista, entonces en las líneas 19 a 21 recorremos la lista enlazada, hacia adelante. En caso contrario, vamos hacia atrás, partiendo del final, como se muestra en las líneas 25 a 27.

En las Figuras 17.27 y 17.28 se muestran los diversos métodos `remove` y esos métodos confluyen en un método `remove` privado, mostrado en las líneas 40 a 48 (en la Figura 17.27), que se ajusta al algoritmo de la Sección 17.3.

Las factorías de iteradores se muestran en la Figura 17.29. Ambas devuelven un objeto `LinkedListIterator` recién construido. Por último, el `LinkedListIterator`, que quizás sea la parte más complicada de toda la implementación, se muestra en la Figura 17.30.

```

1  /**
2   * Construir una LinkedList vacía.
3   */
4  public LinkedList( )
5  {
6      clear( );
7  }
8
9 /**
10  * Construir una LinkedList con los mismos elementos que otra Collection.
11  */
12 public LinkedList( Collection<? extends AnyType> other )
13 {
14     clear( );
15     for( AnyType val : other )
16         add( val );
17 }
18
19 /**
20  * Cambiar el tamaño de esta colección a cero.
21  */
22 public void clear( )
23 {
24     beginMarker = new Node<AnyType>( null, null, null );
25     endMarker = new Node<AnyType>( null, beginMarker, null );
26     beginMarker.next = endMarker;
27
28     theSize = 0;
29     modCount++;
30 }

```

Figura 17.22 Constructores y método clear para la clase LinkedList estándar.

El iterador mantiene una posición actual, como se muestra en la línea 8. `current` representa el nodo que contiene el elemento que debe ser devuelto por una llamada a `next`. Observe que cuando `current` está posicionado en el marcador final, una llamada a `next` será ilegal, pero la llamada a `previous` debería dar el primer elemento yendo hacia atrás. Como en `ArrayList`, el iterador también mantiene el contador `modCount` de la lista a través de la que está iterando, inicializado en el momento de construirse el iterador. Esta variable, `expectedModCount`, solo puede cambiar si el iterador lleva a cabo una operación `remove`. `lastVisited` se utiliza para representar el último nodo que fue visitado; este dato es empleado por `remove`. Si `lastVisited` es `null`, la operación `remove` es ilegal. Por último, `lastMoveWasPrev` será `true` si el último movimiento del iterador hace que la operación `remove` sea a través de `previous`; será `false` si el último movimiento fue a través de `next`.

```
1  /**
2   * Devuelve el número de elementos de esta colección.
3   * @return el número de elementos de esta colección.
4   */
5  public int size( )
6  {
7      return theSize;
8  }
9
10 /**
11  * Comprueba si algún elemento se encuentra en esta colección.
12  * @param x cualquier objeto.
13  * @return true si esta colección contiene un elemento igual a x.
14  */
15 public boolean contains( Object x )
16 {
17     return findPos( x ) != NOT_FOUND;
18 }
19
20 /**
21  * Devuelve la posición del primer elemento que coincide con x
22  * en esta colección, o NOT_FOUND si no lo encuentra.
23  * @param x cualquier objeto.
24  * @return la posición del primer elemento que coincide con x
25  * en esta colección, o NOT_FOUND si no lo encuentra.
26  */
27 private Node<AnyType> findPos( Object x )
28 {
29     for( Node<AnyType> p = beginMarker.next; p != endMarker; p = p.next )
30         if( x == null )
31         {
32             if( p.data == null )
33                 return p;
34         }
35         else if( x.equals( p.data ) )
36             return p;
37
38     return NOT_FOUND;
39 }
```

Figura 17.23 `size` y `contains` para la clase `LinkedList` estándar.

```
1  /**
2   * Añade un elemento a esta colección, al final.
3   * @param x cualquier objeto.
4   * @return true.
5   */
6  public boolean add( AnyType x )
7  {
8      addLast( x );
9      return true;
10 }
11
12 /**
13  * Añade un elemento a esta colección, al principio.
14  * Otros elementos se desplazan una posición hacia arriba.
15  * @param x cualquier objeto.
16  */
17 public void addFirst( AnyType x )
18 {
19     add( 0, x );
20 }
21
22 /**
23  * Añade un elemento a esta colección, al final.
24  * @param x cualquier objeto.
25  */
26 public void addLast( AnyType x )
27 {
28     add( size(), x );
29 }
30
31 /**
32  * Añade un elemento a esta colección, en una posición especificada.
33  * Los elem. situados en esa pos. o después se desplazan una pos. arriba
34  * @param x cualquier objeto.
35  * @param idx posición en la que añadir el elemento.
36  * @throws IndexOutOfBoundsException si idx no está comprendido
37  * entre 0 y size(), ambos inclusive.
38  */
39 public void add( int idx, AnyType x )
40 {
41     Node<AnyType> p = getNode( idx, 0, size );
42     Node<AnyType> newNode = new Node<AnyType>( x, p.prev, p );
43     newNode.prev.next = newNode;
44     p.prev = newNode;
45     theSize++;
46     modCount++;
47 }
```

---

Figura 17.24 Métodos add para la clase LinkedList estándar.

```
1  /**
2  * Devuelve el primer elemento de la lista.
3  * @throws NoSuchElementException si la lista está vacía.
4  */
5  public AnyType getFirst( )
6  {
7      if( isEmpty( ) )
8          throw new NoSuchElementException( );
9      return getNode( 0 ).data;
10 }
11
12 /**
13 * Devuelve el último elemento de la lista.
14 * @throws NoSuchElementException si la lista está vacía.
15 */
16 public AnyType getLast( )
17 {
18     if( isEmpty( ) )
19         throw new NoSuchElementException( );
20     return getNode( size( ) - 1 ).data;
21 }
22
23 /**
24 * Devuelve el elemento situado en la posición idx.
25 * @param idx el índice que hay que buscar.
26 * @throws IndexOutOfBoundsException si el índice está fuera de rango.
27 */
28 public AnyType get( int idx )
29 {
30     return getNode( idx ).data;
31 }
32
33 /**
34 * Cambia el elemento situado en la posición idx.
35 * @param idx el índice que hay que cambiar.
36 * @param newVal el nuevo valor.
37 * @return el valor antiguo.
38 * @throws IndexOutOfBoundsException si el índice está fuera de rango.
39 */
40 public AnyType set( int idx, AnyType newVal )
41 {
42     Node<AnyType> p = getNode( idx );
43     AnyType oldVal = p.data;
44
45     p.data = newVal;
46     return oldVal;
47 }
```

Figura 17.25 Métodos `get` y `set` para la clase `LinkedList` estándar.

```

1  /**
2   * Obtiene el Node en la pos. idx, que tiene que estar entre lower y upper.
3   * @param idx el indice que hay que buscar.
4   * @param lower indice válido mínimo.
5   * @param upper indice válido máximo.
6   * @return nodo interno correspondiente a idx.
7   * @throws IndexOutOfBoundsException si idx no está entre
8   * lower y upper, ambos incluidos.
9   */
10 private Node<AnyType> getNode( int idx, int lower, int upper )
11 {
12     Node<AnyType> p;
13
14     if( idx < lower || idx > upper )
15         throw new IndexOutOfBoundsException( );
16
17     if( idx < size( ) / 2 )
18     {
19         p = beginMarker.next;
20         for( int i = 0; i < idx; i++ )
21             p = p.next;
22     }
23     else
24     {
25         p = endMarker;
26         for( int i = size( ); i > idx; i-- )
27             p = p.prev;
28     }
29
30     return p;
31 }
32
33 /**
34  * Obtiene el Node en la pos. idx, que tiene que estar entre 0 y size( ) - 1.
35  * @param idx el indice que hay que buscar.
36  * @return nodo interno correspondiente a idx.
37  * @throws IndexOutOfBoundsException si idx no se está entre
38  * 0 y size()-1, ambos incluidos.
39  */
40 private Node<AnyType> getNode( int idx )
41 {
42     return getNode( idx, 0, size( ) - 1 );
43 }

```

Figura 17.26 Rutina getNode privada para la clase `LinkedList` estándar.

```
1  /**
2   * Elimina el primer elemento de la lista.
3   * @return el elemento eliminado de la colección.
4   * @throws NoSuchElementException si la lista está vacía.
5   */
6  public AnyType removeFirst( )
7  {
8      if( isEmpty( ) )
9          throw new NoSuchElementException( );
10     return remove( getNode( 0 ) );
11 }
12
13 /**
14  * Elimina el último elemento de la lista.
15  * @return el elemento eliminado de la colección.
16  * @throws NoSuchElementException si la lista está vacía.
17  */
18 public AnyType removeLast( )
19 {
20     if( isEmpty( ) )
21         throw new NoSuchElementException( );
22     return remove( getNode( size( ) - 1 ) );
23 }
24
25 /**
26  * Elimina un elemento de esta colección.
27  * @param idx el índice del objeto.
28  * @return el elemento eliminado de la colección.
29  */
30 public AnyType remove( int idx )
31 {
32     return remove( getNode( idx ) );
33 }
34
35 /**
36  * Elimina el objeto contenido en el nodo p.
37  * @param p el nodo que contiene el objeto.
38  * @return el elemento eliminado de la colección.
39  */
40 private AnyType remove( Node<AnyType> p )
41 {
42     p.next.prev = p.prev;
43     p.prev.next = p.next;
44     theSize--;
45     modCount++;
46
47     return p.data;
48 }
```

Figura 17.27 Métodos `remove` para la clase `LinkedList` estándar.

```

1  /**
2   * Elimina un elemento de esta colección.
3   * @param x cualquier objeto.
4   * @return true si este elemento ha sido eliminado de la colección.
5   */
6  public boolean remove( Object x )
7  {
8      Node<AnyType> pos = findPos( x );
9
10     if( pos == NOT_FOUND )
11         return false;
12     else
13     {
14         remove( pos );
15         return true;
16     }
17 }

```

---

Figura 17.28 Método remove adicional para la clase `LinkedList` estándar.

```

1  /**
2   * Obtiene un objeto Iterator usado para recorrer la colección.
3   * @return un iterador posicionado antes del primer elemento.
4   */
5  public Iterator<AnyType> iterator( )
6  {
7      return new LinkedListIterator( 0 );
8  }
9
10 /**
11  * Obtiene un objeto ListIterator usado para recorrer la colección
12  * bidireccionalmente.
13  * @return un iterador posicionado antes del elemento solicitado.
14  * @param idx índice para iniciar el iterador. Use size() para recorrido
15  * inverso completo. Use 0 para un recorrido completo hacia adelante.
16  * @throws IndexOutOfBoundsException si idx no está comprendido
17  * entre 0 y size(), ambos incluidos.
18  */
19 public ListIterator<AnyType> listIterator( int idx )
20 {
21     return new LinkedListIterator( idx );
22 }

```

---

Figura 17.29 Métodos factoría de iteradores para la clase `LinkedList` estándar.

```
1  /**
2  * Esta es la implementación de LinkedListIterator.
3  * Mantiene una noción de una posición actual y, por supuesto,
4  * la referencia implícita a LinkedList.
5  */
6  private class LinkedListIterator implements ListIterator<AnyType>
7  {
8      private Node<AnyType> current;
9      private Node<AnyType> lastVisited = null;
10     private boolean lastMoveWasPrev = false;
11     private int expectedModCount = modCount;
12
13     public LinkedListIterator( int idx )
14     {
15         current = getNode( idx, 0, size( ) );
16     }
17
18     public boolean hasNext( )
19     {
20         if( expectedModCount != modCount )
21             throw new ConcurrentModificationException( );
22         return current != endMarker;
23     }
24
25     public AnyType next( )
26     {
27         if( !hasNext( ) )
28             throw new NoSuchElementException( );
29
30         AnyType nextItem = current.data;
31         lastVisited = current;
32         current = current.next;
33         lastMoveWasPrev = false;
34         return nextItem;
35     }
36     public void remove( )
37     {
38         if( expectedModCount != modCount )
39             throw new ConcurrentModificationException( );
40         if( lastVisited == null )
41             throw new IllegalStateException( );
```

Continúa

Figura 17.30 Implementación de la clase interna iteradora para la clase LinkedList estándar.

```
42
43     LinkedList.this.remove( lastVisited );
44     lastVisited = null;
45     if( lastMoveWasPrev )
46         current = current.next;
47     expectedModCount++;
48 }
49
50 public boolean hasPrevious( )
51 {
52     if( expectedModCount != modCount )
53         throw new ConcurrentModificationException( );
54     return current != beginMarker.next;
55 }
56
57 public AnyType previous( )
58 {
59     if( !hasPrevious( ) )
60         throw new NoSuchElementException( );
61
62     current = current.prev;
63     lastVisited = current;
64     lastMoveWasPrev = true;
65     return current.data;
66 }
67 }
```

---

Figura 17.30 (Continuación).

Los métodos `hasNext` y `hasPrevious` son bastante rutinarios. Ambos generan una excepción en caso de detectarse una modificación externa de la lista.

El método `next` hace avanzar `current` (línea 32) después de obtener el valor en el nodo (línea 30) que hay que devolver (línea 34). Los campos de datos `lastVisited` y `lastMoveWasPrev` se actualizan en las líneas 31 y 33, respectivamente. La implementación de `previous` no es exactamente simétrica, porque para `previous`, hacemos avanzar `current` antes de obtener el valor. Esto resulta evidente si consideramos que el estado inicial para una iteración hacia atrás es que `current` se encuentre en el marcador final.

Por último, `remove` se muestra en las líneas 36 a 48. Después de las comprobaciones de error obligatorias, utilizamos el método `remove` de `LinkedList` para eliminar el nodo `lastVisited`. La referencia explícita a la clase externa es necesaria, porque la operación `remove` del iterador oculta la operación `remove` de la lista. Después de hacer `lastVisited` igual a `null`, para evitar una segunda operación `remove`, comprobamos si la última operación fue `next` o `previous`. En este último caso, ajustamos `current`, como se muestra en la línea 46, al estado que tenía antes de la combinación `previous/remove`.

En conjunto, hay una gran cantidad de código, aunque simplemente sirve para embellecer los conceptos básicos presentados en la implementación original de la clase `LinkedList` no estándar en la Sección 17.2.

## Resumen

En este capítulo hemos descrito por qué y cómo se implementan las listas enlazadas, ilustrando las interacciones entre las clases de lista, de iterador y de nodo. Hemos examinado variantes de la lista enlazada, incluyendo las listas doblemente enlazadas. La lista doblemente enlazada permite recorrer bidireccionalmente la lista. También hemos mostrado cómo puede derivarse fácilmente una lista enlazada ordenada a partir de la lista enlazada básica. Finalmente, hemos proporcionado una implementación de la mayor parte de la clase `LinkedList` de la API de Colecciones.



### Conceptos clave

**clase iteradora** Una clase que mantiene una posición actual en un contenedor, tal como una lista. Una clase iteradora suele encontrarse en el mismo paquete que una clase de lista, o suele ser una clase interna de esta última. (611)

**lista circularmente enlazada** Una lista enlazada en la que el enlace `next` de la última celda hace referencia a `first`. Esta técnica resulta útil cuando hace falta un encadenamiento circular. (620)

**lista doblemente enlazada** Una lista enlazada que permite el recorrido bidireccional, almacenando dos enlaces por nodo. (618)

**lista enlazada ordenada** Una lista en la que los elementos se almacenan por orden. Puede derivarse una clase de lista enlazada ordenada a partir de una clase de lista. (621)

**nodo de cabecera** Un nodo adicional en una lista enlazada, que no almacena ningún dato pero sirve para satisfacer el requisito de que todo nodo disponga de un nodo anterior. Un nodo de cabecera nos permite evitar casos especiales como la inserción de un nuevo primer elemento y la eliminación del primer elemento. (609)



### Errores comunes

1. El error más común en una lista enlazada es insertar los nodos incorrectamente durante una operación de inserción. Este procedimiento es especialmente complicado con las listas doblemente enlazadas.
2. No se debe dejar que los métodos accedan a los campos a través de una referencia `null`. Debemos realizar comprobaciones de error para detectar este tipo de fallo y generar excepciones de la forma apropiada.
3. Cuando varios iteradores acceden a una lista simultáneamente, pueden producirse problemas. Por ejemplo, ¿qué pasa si un iterador borra el nodo al que el otro iterador

está a punto de acceder? Resolver este tipo de problemas requiere trabajo adicional, como el uso de un contador de modificaciones concurrentes.



## Internet

La clase de lista simplemente enlazada, incluida la lista enlazada ordenada, está disponible en línea, al igual que nuestra implementación de la lista de la API de Colecciones.

### **LinkedList.java**

Contiene la implementación para `weiss.nonstandard.LinkedList`.

### **LinkedListIterator.java**

Contiene la implementación para `LinkedListIterator`.

### **SortedListList.java**

Contiene la implementación para `SortedListList`.

### **LinkedList.java**

Contiene la implementación de la clase `LinkedList` de la API de Colecciones y su iterador.



## Ejercicios

### EN RESUMEN

- 17.1** Dibuje una lista doblemente enlazada vacía que utilice tanto una cabecera como una cola.
- 17.2** Dibuje una lista enlazada vacía con una cabecera implementada.

### EN TEORÍA

- 17.3** Suponga que tiene una referencia a un nodo en una lista simplemente enlazada y que está garantizado que no es el último nodo de la lista. No tenemos referencias a ningún otro nodo (excepto siguiendo enlaces). Describa un algoritmo  $O(1)$  que elimine lógicamente de la lista enlazada el valor almacenado en ese nodo, manteniendo la integridad de la lista enlazada. (*Pista:* utilice el siguiente nodo.)
- 17.4** Suponga que implementamos una lista simplemente enlazada con un nodo de cabecera y otro de cola. Utilizando las ideas expuestas en el Ejercicio 17.3, describa algoritmos de tiempo constante para
- Insertar el elemento  $x$  antes de la posición  $p$ .
  - Eliminar el elemento almacenado en la posición  $p$ .
- 17.5** Una lista enlazada contiene un ciclo si, comenzando a partir de un nodo  $p$ , el seguir un número de enlaces `next` nos vuelve a llevar al nodo  $p$ . El nodo  $p$  no tiene por qué ser el primer nodo de la lista. Suponga que tenemos una lista enlazada con  $N$  nodos. Sin embargo, el valor de  $N$  es desconocido.
- Diseñe un algoritmo  $O(N)$  para determinar si la lista contiene un ciclo. Puede utilizar  $O(N)$  espacio adicional.

- b. Repita el apartado (a), pero utilice únicamente  $O(1)$  espacio adicional. (*Pista:* utilice dos iteradores que estén inicialmente al principio de la lista, pero que avancen a diferentes velocidades.)
- 17.6** Escriba un algoritmo para imprimir a la inversa una lista simplemente enlazada, utilizando únicamente un espacio adicional constante. Esta restricción implica que no puede utilizar recursión, pero sí que puede asumir que su algoritmo es un método de la lista.
- 17.7** Una forma de implementar una cola consiste en implementar una lista doblemente enlazada. Suponga que la lista no contiene una cabecera y que se puede mantener un iterador para la lista. ¿Para cuáles de las siguientes representaciones pueden realizarse todas las operaciones de la cola en un tiempo constante de caso peor? Justifique sus respuestas.
- Mantener un iterador que corresponda al primer elemento de la lista.
  - Mantener un iterador que corresponda al último elemento de la lista.

## EN LA PRÁCTICA

- 17.8** Modifique `remove` en la clase `LinkedList` no estándar para eliminar todas las apariciones de `x`.
- 17.9** Suponga que queremos insertar parte de una lista enlazada en otra (una operación de cortar y pegar). Suponga que tres parámetros `LinkedListIterator` representan el punto inicial del *corte*, el punto final del *corte* y el punto en el que hay que *pegarla* información cortada. Suponga que todos los iteradores son válidos y que el número de elementos cortados es distinto de cero.
- Escriba un método para cortar y pegar que sea parte de `weiss.nonstandard`. ¿Cuál es el tiempo de ejecución del algoritmo?
  - Escriba un método en la clase `LinkedList` para cortar y pegar. ¿Cuál es el tiempo de ejecución del algoritmo?
- 17.10** Modifique la rutina `find` en la clase `LinkedList` no estándar para devolver la última aparición del elemento `x`.
- 17.11** Implemente una clase eficiente `Stack` utilizando una `LinkedList` (estándar o no estándar) como miembro de datos. Necesitará emplear un iterador, pero este puede ser un miembro de datos o una variable local de cualquier rutina que lo necesite.
- 17.12** Implemente una clase `Queue` eficiente utilizando (como en el Ejercicio 17.11) una lista simplemente enlazada e iteradores apropiados. ¿Cuántos de estos iteradores deben ser miembros de datos para conseguir una implementación eficiente?
- 17.13** Implemente la clase `LinkedList` no estándar sin el nodo de cabecera.
- 17.14** El método `insert` de `SortedLinkedList` utiliza únicamente métodos de iterador públicos. ¿Puede acceder a miembros privados del iterador?
- 17.15** Implemente `retreat` para listas simplemente enlazadas. Observe que requerirá un tiempo lineal.

## PROYECTOS DE PROGRAMACIÓN

- 17.16** Proporcione un método `add` para el `ListIterator` que sea coherente con la especificación de la API de Colecciones.
- 17.17** Reimplemente la clase estándar `LinkedList`
- Sin cabecera y sin cola.
  - Con cola pero sin cabecera.
  - Con cabecera pero sin cola.
- 17.18** Proporcione un método `set` para el `ListIterator` que sea coherente con la especificación de la API de Colecciones.
- 17.19** Implemente una lista circular y doblemente enlazada.
- 17.20** Escriba sendas rutinas `makeUnion` e `intersect` que devuelvan la unión y la intersección de dos listas enlazadas ordenadas.
- 17.21** Si el orden en el que los elementos de una lista están almacenados no es importante, a menudo se puede acelerar la búsqueda con el heuristic conocido con el nombre de *desplazamiento al frente*: cada vez que se accede a un elemento, se desplaza al principio de la lista. Esta acción suele proporcionar una mejora, porque los elementos a los que se accede frecuentemente tenderán a migrar al principio de la lista, mientras que los elementos a los que se accede de manera menos frecuente tenderán a migrar hacia el final de la lista. En consecuencia, los elementos a los que se accede más frecuentemente tenderán a requerir menos tarea de búsqueda. Implemente el heuristic de desplazamiento al frente para listas enlazadas.
- 17.22** Escriba un editor de textos basado en líneas. La sintaxis de comandos es similar a la del editor de líneas `ed` de Unix. La copia interna del archivo se mantiene como una lista enlazada de líneas. Para poder desplazarse hacia arriba y hacia abajo por el archivo, será necesario mantener una lista doblemente enlazada. La mayoría de los comandos están representados por una cadena de un único carácter. Otros tienen dos caracteres y requieren un argumento (o dos). Soporta los comandos mostrados en la Figura 17.31.

Comando	Función
l	Ir a la parte superior.
a	Añadir texto después de la línea actual hasta que introduzca un punto(.) en su propia línea.
d	Borrar línea actual.
dr num num	Borrar varias líneas.
f name	Cambiar el nombre del archivo actual (para la siguiente escritura).
g num	Ir a una línea numerada.
?	Obtener ayuda.
i	Como la operación de adición, pero añade líneas antes de la línea actual.
m num	Mover línea actual, situándola después de alguna otra línea.
mr num num num	Mover varias líneas como una unidad, situándolas después de otra línea.
v	Activar o desactivar la visualización de los números de línea.
p	Imprimir la línea actual.
pr num num	Imprimir varias líneas.
q!	Abortar sin escribir.
z name	Leer y pegar otro archivo en el archivo actual.
s text text	Sustituir un texto por otro texto.
t num	Copiar la línea actual, insertando la copia después de alguna otra línea.
tr num num num	Copiar varias líneas, insertando la copia después de alguna otra línea.
w	Escribir un archivo en disco.
r!	Salir después de escribir.
#	Ir a la última línea.
-	Subir una línea.
+	Bajar una línea.
=	Imprimir el número de la línea actual.
? text	Buscar un patrón hacia adelante.
/ text	Buscar un patrón hacia atrás.
\$	Imprimir los números de línea y de caracteres del archivo.

Figura 17.31 Comandos para el editor del Ejercicio 17.22.



## Árboles

El *árbol* es una estructura fundamental en las Ciencias de la computación. Casi todos los sistemas operativos almacenan los archivos en árboles o estructuras similares a árboles. Los árboles también se emplean en el diseño de compiladores, el procesamiento de textos y los algoritmos de búsqueda. Veremos esta última aplicación en el Capítulo 19.

En este capítulo veremos

- Una definición de un árbol general, analizando cómo se utiliza en un sistema de archivos.
- Un examen del árbol binario.
- La implementación de las operaciones con árboles utilizando la recursión.
- Cómo se recorre un árbol de forma no recursiva.

### 18.1 Árboles generales

Los árboles pueden definirse de dos maneras: de forma recursiva o no recursiva. La definición no recursiva es la técnica más directa, así que vamos a comenzar con ella. La formulación recursiva nos permitirá escribir algoritmos simples para la manipulación de árboles.

#### 18.1.1 Definiciones

Dicho de forma no recursiva, un *árbol* está compuesto por un conjunto de nodos y un conjunto de aristas dirigidas que conectan parejas de nodos. A lo largo de este texto solo vamos a considerar los árboles que tienen raíz. Un árbol con raíz tiene las siguientes propiedades:

- Uno de los nodos se distingue de los demás por estar designado como raíz.
- Todo nodo  $c$ , excepto la raíz, está conectado mediante una arista a exactamente un único otro nodo  $p$ . El nodo  $p$  es el padre de  $c$  y  $c$  es uno de los hijos de  $p$ .
- Existe un camino único que recorre el árbol desde la raíz hasta cada nodo. El número de aristas que hay que recorrer se denomina *longitud del camino*.

Un *árbol* se puede definir de manera no recursiva, como un conjunto de nodos y un conjunto aristas dirigidas que los conectan.

Los padres y los hijos se definen de forma natural. Una arista dirigida conecta al *padre* con el *hijo*.

Una hoja no tiene hijos.

La *profundidad* de un nodo es la longitud del camino que va desde la raíz hasta ese nodo. La *altura* de un nodo es la longitud del camino que va desde ese nodo hasta su hoja más profunda.

El *tamaño* de un nodo es el número de descendientes que tiene el nodo (incluyendo el propio nodo).

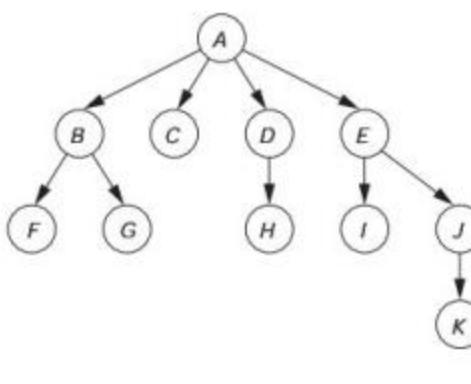
Los padres y los hijos se definen de forma natural. Una arista dirigida conecta al *padre* con el *hijo*.

La Figura 18.1 ilustra un árbol. El nodo raíz es *A*; los hijos de *A* son *B*, *C*, *D*, y *E*. Puesto que *A* es la raíz, no tiene padre; todos los demás nodos sí tienen padre. Por ejemplo, el padre de *B* es *A*. Un nodo que no tenga hijos se denomina *hoja*. Las hojas de este árbol son *C*, *F*, *G*, *H*, *I* y *K*. La longitud del camino que va desde *A* hasta *K* es 3 (aristas); la longitud del camino que va desde *A* hasta *A* es 0 (aristas).

Un árbol con  $N$  nodos debe tener  $N - 1$  aristas, porque todo nodo, salvo la raíz, tiene una arista entrante. La *profundidad* de un nodo de un árbol es la longitud del camino que va desde la raíz hasta ese nodo. Por tanto, la profundidad de la raíz es siempre 0, y la profundidad de cualquier nodo es 1 unidad más que la profundidad de su padre. La *altura* de un nodo es la longitud del camino que va desde el nodo hasta la hoja más profunda. Por tanto, la altura de *E*, en nuestro ejemplo, es 2. La altura de cualquier nodo es 1 más que la de su hijo de máxima altura. Por tanto, la altura de un árbol es la altura de su raíz.

Los nodos del mismo parentesco se denominan *hermanos*; por tanto, *B*, *C*, *D*, y *E* son hermanos en nuestro ejemplo. Si existe un camino desde el nodo *u* hasta el nodo *v*, entonces *u* es un *ancestro* de *v* y *v* es un *descendiente* de *u*. Si  $u \neq v$ , entonces *u* es un *ancestro propio* de *v* y *v* es un *descendiente propio* de *u*. El *tamaño* de un nodo es el número de descendientes que tiene el nodo (incluyendo el propio nodo). Así, el tamaño de *B* en nuestro ejemplo es 3, mientras que el tamaño de *C* es 1. El tamaño de un árbol es igual al tamaño de la raíz. Por tanto, el tamaño del árbol mostrado en la Figura 18.1 es el tamaño de su raíz *A*, es decir 11.

Existe una definición alternativa de árbol que es de carácter recursivo: o bien un árbol está vacío o está compuesto de una raíz y cero o más subárboles no vacíos  $T_1, T_2, \dots, T_k$ , cuyas raíces están conectadas con la raíz del árbol mediante una arista, como se ilustra en la Figura 18.2. En ciertos casos (y muy especialmente en el de los *árboles binarios* de los que hablaremos más adelante en el capítulo), podemos permitir que algunos de los subárboles estén vacíos.



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 18.1 Un árbol con la información de altura y profundidad.

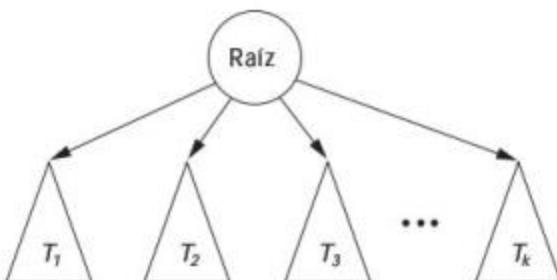


Figura 18.2 Un árbol visto recursivamente.

### 18.1.2 Implementación

Una forma de implementar un árbol sería tener en cada nodo un enlace a cada uno de los hijos del nodo, además de los datos correspondientes al nodo. Sin embargo, como el número de hijos por nodo puede variar enormemente y no se conoce de antemano, puede que no sea factible incluir en la estructura de datos enlaces directos a los hijos (se desperdiciaría demasiado espacio). La solución a este problema es muy simple y se denomina *método del primer hijo/siguiente hermano*: mantenemos los hijos de cada nodo en una lista enlazada de nodos de árbol, en la que cada nodo mantiene dos enlaces, uno a su hijo situado más a la izquierda (si el nodo no es una hoja) y otro al hermano situado a su derecha (si es que el propio nodo no es el hermano situado más a la derecha). Este tipo de implementación se ilustra en la Figura 18.3. Las flechas que apuntan hacia abajo son los enlaces `firstChild` al primer hijo, mientras que las flechas que apuntan de izquierda a derecha son los enlaces `nextSibling` al siguiente hermano. No hemos dibujado los enlaces `null` porque hay demasiados. En este árbol, el nodo *B* dispone de un enlace tanto a un hermano (*C*) como de un enlace al hijo situado más a la izquierda (*F*); algunos nodos solo disponen de uno de estos enlaces y otros no disponen de ninguno. Dada esta representación, implementar una clase de árbol es sencillo.

Los árboles generales se pueden implementar utilizando el *método del primer hijo/siguiente hermano*, que requiere dos enlaces por cada elemento.

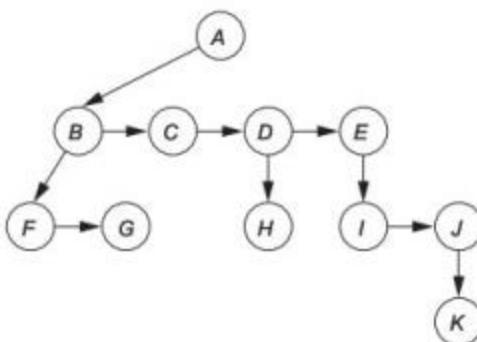


Figura 18.3 Representación del primer hijo/siguiente hermano, para el árbol de la Figura 18.1.

### 18.1.3 Una aplicación: sistemas de archivos

Los sistemas de archivos utilizan estructuras de tipo árbol.

Los árboles tienen muchas aplicaciones. Uno de sus usos más populares es la estructura de directorios en muchos sistemas operativos, incluyendo Unix, VAX/VMS y Windows/DOS. La Figura 18.4 muestra un directorio típico en el sistema de archivos Unix. La raíz de este directorio es `mark*`. (El asterisco situado a continuación del nombre indica que `mark` es un directorio.) Observe que `mark` tiene tres hijos: `books`, `courses` y `.login`, dos de los cuales son ellos mismos directorios. Por tanto, `mark` contiene dos directorios y un archivo normal. El nombre de archivo `mark/books/dsaa/ch1` se obtiene siguiendo tres veces el enlace al hijo situado más a la izquierda. Cada / después del primer nombre indica una arista; el resultado es un nombre de ruta. Si la ruta comienza en la raíz de todo el sistema de archivos, en lugar de en un directorio arbitrario del sistema de archivos, entonces será un nombre de ruta completo; en caso contrario, será un nombre de ruta relativo (relativo al directorio actual).

Este sistema de archivos jerárquico es popular porque permite a los usuarios utilizar sus datos desde el punto de vista lógico. Además, dos archivos situados en directorios diferentes, pueden compartir el mismo nombre, porque sus rutas desde la raíz son distintas y tienen, por tanto, diferentes nombres completos de ruta. Un directorio en el sistema de archivos Unix es simplemente un archivo con una lista de todos sus hijos,<sup>1</sup> de modo que los directorios se pueden recorrer mediante un esquema de iteración; es decir, podemos iterar secuencialmente a través de la serie de hijos. De hecho, en algunos sistemas, si aplicamos a un directorio el comando normal para imprimir un archivo, aparecen a la salida los nombres de archivo contenidos en el directorio (junto con otros tipos de información no-ASCII).

La forma más fácil de recorrer la estructura de archivos es utilizando la recursión.

Suponga que queremos enumerar los nombres de todos los archivos de un directorio (incluyendo sus subdirectorios) y que en nuestro formato de salida los archivos de profundidad  $d$  tienen sus nombres sangrados mediante  $d$  caracteres de tabulación. En la Figura 18.5 se proporciona un breve algoritmo para llevar a cabo esta tarea. La salida para el directorio presentado en la Figura 18.4 se muestra en la Figura 18.6.

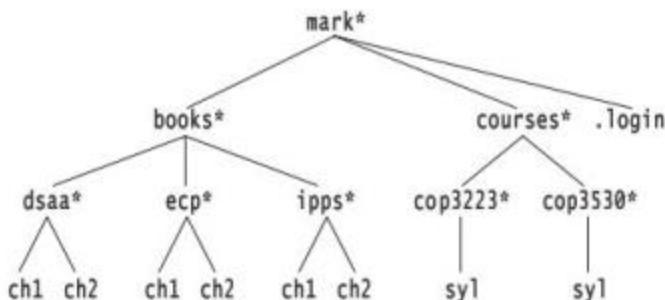


Figura 18.4 Un directorio Unix.

<sup>1</sup> Cada directorio de un sistema de archivos Unix también tiene una entrada (.) que apunta a sí mismo y otra entrada (..) que apunta al parent del directorio, lo que introduce un ciclo en el árbol. Por tanto, técnicamente, el sistema de archivos Unix no es un árbol sino una estructura similar al árbol. Lo mismo cabe decir para Windows/DOS.

```

1 void listAll( int depth = 0 ) // depth es inicialmente 0
2 {
3     printName( depth );           // Imprimir el nombre del objeto
4     if( isDirectory( ) )
5         for each file c in this directory (for each child)
6             c.listAll( depth + 1 );
7 }

```

Figura 18.5 Una rutina para obtener un listado de un directorio y sus subdirectorios en un sistema de archivos jerárquico.

```

mark
books
    dsaa
        ch1
        ch2
    ecp
        ch1
        ch2
    ipps
        ch1
        ch2
courses
    cop3223
        syll
    cop3530
        syll
.login

```

Figura 18.6 El listado del directorio para el árbol mostrado en la Figura 18.4.

Suponemos que existe la clase `FileSystem` y dos métodos, `printName` e `isDirectory`. `printName` imprime el objeto `FileSystem` actual, sangrado con `depth` tabulaciones; `isDirectory` comprueba si el objeto `FileSystem` actual es un directorio, devolviendo `true` en caso afirmativo. Entonces podemos escribir la rutina recursiva `listAll`. Necesitaremos pasarla el parámetro `depth`, que indica el nivel actual del directorio respecto de la raíz. La rutina `listAll` se inicia con `depth` igual a 0, para indicar que no hay sangrado con respecto a la raíz. Este valor de profundidad (`depth`) es una variable interna para el control de la impresión, y difícilmente cabría considerarla como un parámetro sobre el que la rutina que hace la invocación tenga que preocuparse. Por ello, el pseudocódigo especifica un valor predeterminado igual a 0 para `depth` (la especificación de un valor predeterminado no es legal en Java).

La lógica del algoritmo es sencilla de entender. Se imprime el objeto actual con el sangrado apropiado. Si la entrada es un directorio, procesamos recursivamente todos los hijos, uno por uno. Estos hijos están un nivel más abajo en el árbol y deben, por tanto, sangrarse con un carácter adicional de tabulación. Realizamos la llamada recursiva con `depth+1`. Resulta difícil imaginar un fragmento de código más corto que pudiera realizar lo que parece ser una tarea muy difícil.

En un recorrido de un árbol en preorden, el trabajo en un nodo se realiza antes de procesar a sus hijos. El recorrido requiere un tiempo constante por cada nodo.

En un recorrido de un árbol en postorden, el trabajo en un nodo se realiza después de evaluar a sus hijos. El recorrido requiere un tiempo constante por cada nodo.

En esta técnica algorítmica, conocida con el nombre de *recorrido del árbol en preorden*, el trabajo con un nodo se lleva a cabo antes (pre) de procesar a sus hijos. Además de ser un algoritmo compacto, el recorrido en preorden resulta eficiente porque requiere un tiempo constante por cada nodo. Explicaremos el por qué más adelante en el capítulo.

Otro método común de recorrer un árbol es el *recorrido del árbol en postorden*, en el que el trabajo en un nodo se lleva a cabo después (post) de evaluar a sus hijos. También requiere un tiempo constante por nodo. Por ejemplo, la Figura 18.7 representa la misma estructura de directorios que la que aparece en la Figura 18.4. Los números entre paréntesis representan el número de bloques de disco ocupados por cada archivo. Los propios directorios son archivos, así que también utilizan bloques de disco (para almacenar los nombres e información acerca de sus hijos).

Suponga que queremos calcular el número total de bloques utilizado por todos los archivos de nuestro árbol de ejemplo. La forma más natural de hacerlo consiste en calcular el número total de bloques contenidos en todos los hijos (que pueden ser directorios que habrá que evaluar recursivamente): books (41), courses (8) y .login (2). El número total de bloques será entonces el total de todos los hijos más los bloques utilizados por la raíz (1), lo que nos da un total de 52. La rutina `size` mostrada en la Figura 18.8 implementa esta estrategia. Si el objeto `FileSystem` actual no es un directorio, `size` simplemente devuelve el número de bloques que utiliza. En caso contrario, se suma el número de bloques del directorio actual al número de bloques calculado (recursivamente) para todos los hijos. Para ilustrar la diferencia entre el recorrido en postorden y el recorrido en preorden, en la Figura 18.9 mostramos cómo calcula el algoritmo el tamaño de cada directorio (o archivo). Obtenemos una estructura clásica en postorden porque el tamaño total de una entrada no se puede calcular hasta haber calculado la información correspondiente a sus hijos. Como hemos indicado anteriormente, el tiempo de ejecución es lineal. En la Sección 18.4 hablaremos más extensamente acerca del recorrido de los árboles.

## Implementación Java

Java proporciona una clase llamada `File` en el paquete `java.io` que se puede utilizar para recorrer jerarquías de directorios. Podemos utilizarla para implementar el pseudocódigo de la Figura 18.8.

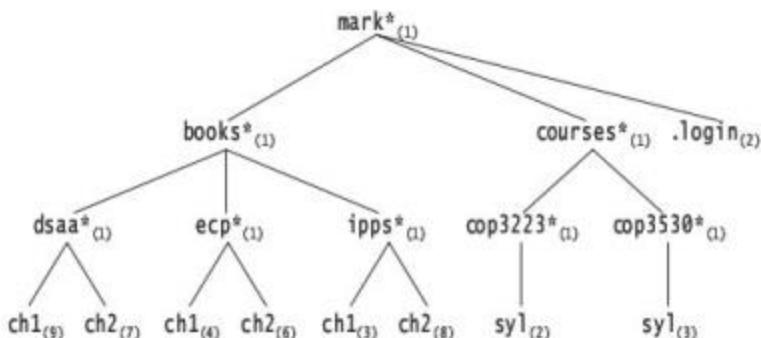


Figura 18.7 Un directorio Unix con tamaños de archivo.

```

1   int size( )
2   {
3       int totalSize = sizeOfFile( );
4
5       if(.isDirectory( ) )
6           for each file c in this directory (for each child)
7               totalSize += c.size( );
8
9       return totalSize;
10  }

```

Figura 18.8 Una rutina para calcular el tamaño total de todos los archivos de un directorio.

	ch1	9
	ch2	7
dsaa		17
	ch1	4
	ch2	6
ecp		11
	ch1	3
	ch2	8
ipps		12
books		41
	sy1	2
cop3223		3
	sy1	3
cop3530		4
courses		8
.login		2
mark		52

Figura 18.9 Una traza del método `size`.

El método `size` también se puede implementar; puede encontrar la implementación en el código disponible en línea. La clase `File` proporciona varios métodos útiles.

Se puede construir un objeto `File` proporcionando un nombre de archivo. `getName` proporciona el nombre de un objeto `File`. No incluye la parte de la ruta correspondiente al directorio; esa información se puede obtener mediante `getPath`. `isDirectory` devuelve `true` si el objeto `File` es un directorio y su tamaño en bytes se puede obtener mediante una llamada a `length`. Si el archivo es un directorio, el método `listFiles` devuelve una matriz de objetos `File` que representa los archivos del directorio (sin incluir `.` ni `..`).

Para implementar la lógica descrita en el pseudocódigo, simplemente utilizamos `printName`, `listAll` (tanto una rutina de preparación pública como una rutina recursiva privada) y `size`, como se muestra en la Figura 18.10. También proporcionamos una rutina `main` simple que permite comprobar la lógica en el directorio actual.

```
1 import java.io.File;
2
3 public class FileSystem
4 {
5     // Imprimir nombre de archivo con el sangrado adecuado
6     public static void printName( String name, int depth )
7     {
8         for( int i = 0; i < depth; i++ )
9             System.out.print( " " );
10        System.out.println( name );
11    }
12
13    // Rutina de preparación públ. para enumerar los archivos de un direct.
14    public static void listAll( File dir )
15    {
16        listAll( dir, 0 );
17    }
18
19    // Método recursivo para enumerar todos los archivos de un directorio
20    private static void listAll( File dir, int depth )
21    {
22        printName( dir.getName( ), depth );
23
24        if( dir.isDirectory( ) )
25            for( File child : dir.listFiles( ) )
26                listAll( child, depth + 1 );
27    }
28
29    public static long size( File dir )
30    {
31        long totalSize = dir.length( );
32
33        if( dir.isDirectory( ) )
34            for( File child : dir.listFiles( ) )
35                totalSize += size( child );
36
37        return totalSize;
38    }
39
40    // Rutina main simple para enumerar los archivos del direct. actual
41    public static void main( String [ ] args )
42    {
43        File dir = new File( "." );
44        listAll( dir );
45        System.out.println( "Total bytes: " + size( dir ) );
46    }
47 }
```

Figura 18.10 Implementación Java para obtener el listado de un directorio.

## 18.2 Árboles binarios

Un *árbol binario* es un árbol en el que ningún nodo puede tener más de dos hijos. Puesto que solo hay dos hijos, podemos llamarlos *left* y *right* (izquierdo y derecho). Recursivamente, un árbol binario o está vacío o está compuesto de una raíz, un árbol izquierdo y un árbol derecho. Los árboles izquierdo y derecho pueden ellos mismos estar vacíos; por ello, un nodo con un único hijo podría tener solo un hijo izquierdo o un hijo derecho. Utilizaremos la definición recursiva en varias ocasiones durante el diseño de algoritmos para árboles binarios. Los árboles binarios tienen muchas aplicaciones importantes, dos de las cuales se ilustran en la Figura 18.11.

Un uso de los árboles binarios es el denominado *árbol de expresión*, que es una estructura de datos fundamental en el diseño de compiladores. Las hojas de un árbol de expresión son operandos, como por ejemplo constantes o nombres de variables; los restantes nodos contienen operadores. Este árbol concreto es binario porque todas las operaciones son binarias. Aunque este caso es el más sencillo, los nodos pueden tener más de dos hijos (y en el caso de los operadores unarios, solo un hijo). Podemos evaluar un árbol de expresión *T* aplicando el operador de la raíz a los valores obtenidos al evaluar recursivamente los subárboles izquierdo y derecho. Al hacer esto, obtenemos la expresión  $(a + ((b - c) * d))$ . (Véase en la Sección 11.2 un análisis de la construcción y evaluación de árboles de expresión).

Un segundo uso del árbol binario es el *árbol de codificación de Huffman*, que se utiliza para implementar un algoritmo de compresión de datos simple, pero relativamente efectivo. Cada símbolo del alfabeto se almacena en una hoja. Su código se obtiene siguiendo el camino que va hasta esa hoja desde la raíz. Cada enlace izquierdo corresponde a un 0 y cada enlace derecho a un 1. Así, b se codificaría como 100. (Consulte la Sección 12.1 para ver una explicación acerca de la construcción del árbol óptimo, es decir, del mejor código posible.)

Otras aplicaciones de los árboles binarios son los árboles de búsqueda binaria (que se analizarán en el Capítulo 19), los cuales permiten insertar elementos y acceder a los mismos en un tiempo logarítmico. Los árboles binarios tienen aplicación también en las colas con prioridad, que soportan el acceso y borrado del elemento mínimo en una colección de elementos. Diversas implementaciones eficientes de las colas con prioridad utilizan árboles (como se explica en los Capítulos 21 a 23).

La Figura 18.12 proporciona el esqueleto para la clase *BinaryNode*. Las líneas 49 a 51 indican que cada nodo está compuesto de un elemento de datos y dos enlaces. El constructor, mostrado en

Un árbol binario no tiene ningún nodo con más de dos hijos.

Un árbol de expresión es un ejemplo de aplicación de los árboles binarios. Los árboles de expresión son estructuras de datos fundamentales en el diseño de compiladores.

Un uso importante de los árboles binarios es en otras estructuras de datos, especialmente el árbol de búsqueda binaria y la cola con prioridad.

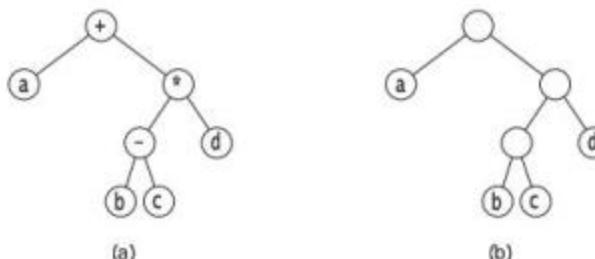


Figura 18.11 Aplicaciones de los árboles binarios. (a) Un árbol de expresión y (b) un árbol de codificación de Huffman.

```

1 // Clase BinaryNode; almacena un nodo en un árbol.
2 //
3 // CONSTRUCCIÓN: sin parámetros, o bien un Object,
4 // un hijo izquierdo y un hijo derecho.
5 //
6 // *****OPERACIONES PÚBLICAS*****
7 // int size( )      --> Devuelve el tamaño del subárbol corresp al nodo
8 // int height( )     --> Devuelve la altura del subárbol corresp al nodo
9 // void printPostOrder( ) --> Imprime un recorrido del árbol en postorden
10 // void printInOrder( )   --> Imprime un recorrido del árbol en orden
11 // void printPreOrder( )  --> Imprime un recorrido del árbol en preorder
12 // BinaryNode duplicate( )--> Devuelve un árbol duplicado
13
14 class BinaryNode<AnyType>
15 {
16     public BinaryNode( )
17         { this( null, null, null ); }
18     public BinaryNode( AnyType theElement,
19                         BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
20         { element = theElement; left = lt; right = rt; }
21
22     public AnyType getElement( )
23         { return element; }
24     public BinaryNode<AnyType> getLeft( )
25         { return left; }
26     public BinaryNode<AnyType> getRight( )
27         { return right; }
28     public void setElement( AnyType x )
29         { element = x; }
30     public void setLeft( BinaryNode<AnyType> t )
31         { left = t; }
32     public void setRight( BinaryNode<AnyType> t )
33         { right = t; }
34
35     public static <AnyType> int size( BinaryNode<AnyType> t )
36         { /* Figura 18.19 */ }
37     public static <AnyType> int height( BinaryNode<AnyType> t )
38         { /* Figura 18.21 */ }
39     public BinaryNode<AnyType> duplicate( )
40         { /* Figura 18.17 */ }
41
42     public void printPreOrder( )

```

Continúa

Figura 18.12 El esqueleto de la clase BinaryNode.

```

43     { /* Figura 18.22 */ }
44     public void printPostOrder( )
45     { /* Figura 18.22 */ }
46     public void printInOrder( )
47     { /* Figura 18.22 */ }
48
49     private AnyType element;
50     private BinaryNode<AnyType> left;
51     private BinaryNode<AnyType> right;
52 }

```

**Figura 18.12 (Continuación).**

líneas 18 a 20, inicializa todos los miembros de datos de la clase `BinaryNode`. Las líneas 22 a 33 proporcionan métodos accesores y mutadores para cada uno de los miembros de datos.

El método `duplicate`, declarado en la línea 39, se utiliza para realizar una copia del árbol que tiene su raíz en el nodo actual. Las rutinas `size` y `height`, declaradas en las líneas 35 y 37, calculan el tamaño y la altura para el nodo referenciado por el parámetro `t`. Implementaremos estas rutinas en la Sección 18.3. (Recuerde que los métodos estáticos no requieren un objeto controlador.) También proporcionamos en las líneas 42 a 47, rutinas para imprimir el contenido del árbol que tiene su raíz en el nodo actual, empleando diversas estrategias recursivas de recorrido. Hablaremos del recorrido de los árboles en la Sección 18.4. ¿Por qué pasamos un parámetro para `size` y `height` y hacemos que esas rutinas sean de tipo `static`, pero sin embargo utilizamos el objeto actual para los recorridos y para `duplicate`? No hay ninguna razón concreta; es cuestión de estilo, y hemos decidido mostrar aquí ambos estilos. Las implementaciones muestran que la diferencia entre esos dos estilos se presenta en el momento de realizar la requerida comprobación de si el árbol está vacío (lo que está indicado por una referencia `null`).

Muchas de las rutinas de `BinaryNode` son recursivas. Los métodos de `BinaryTree` utilizan las rutinas de `BinaryNode` sobre la raíz `root`.

En esta sección vamos a describir la implementación de la clase `BinaryTree`. La clase `BinaryNode` se implementa de forma separada, en lugar de como una clase anidada. El esqueleto de la clase `BinaryTree` se muestra en la Figura 18.13. En su mayor parte las rutinas son cortas porque invocan a los métodos de `BinaryNode`. La línea 44 declara el único miembro de datos –una referencia al nodo raíz `root`.

La clase `BinaryNode` se implementa de forma separada de la clase `BinaryTree`. El único miembro de datos de la clase `BinaryTree` es una referencia al nodo raíz.

Se proporcionan dos constructores básicos. El de las líneas 16 y 17 crea un árbol vacío, mientras que el de las líneas 18 y 19 crea un árbol de un solo nodo. En las líneas 28 a 33 se proporcionan rutinas para recorrer el árbol. Esas rutinas aplican un método de `BinaryNode` a la raíz `root`, después de verificar que el árbol no está vacío. Una estrategia de recorrido alternativa que se puede implementar es el recorrido por niveles. Hablaremos de estas rutinas de recorrido en la Sección 18.4. Las líneas 35 a 38 proporcionan rutinas para vaciar un árbol y para comprobar si un árbol está vacío, con sus implementaciones en línea, y también se ofrecen rutinas para calcular el tamaño y la altura del árbol. Observe que, como `size` y `height` son métodos estáticos en `BinaryNode`, podemos invocarlos utilizando simplemente `BinaryNode.size` y `BinaryNode.height`.

```
1 // Clase BinaryTree; almacena un árbol binario.
2 //
3 // CONSTRUCCIÓN: (a) sin parámetros o (b) con un objeto que hay colocado
4 // en la raíz de un árbol de un solo elemento.
5 //
6 // *****OPERACIONES PÚBLICAS*****
7 // Diversos recorridos del árbol, size, height, isEmpty, makeEmpty.
8 // También, el siguiente método más complejo:
9 // void merge( Object root, BinaryTree t1, BinaryTree t2 )
10 // --> Construir un nuevo árbol
11 // *****ERRORES*****
12 // Se imprimen mensajes de error para las mezclas ilegales de árboles.
13
14 public class BinaryTree<AnyType>
15 {
16     public BinaryTree( )
17     { root = null; }
18     public BinaryTree( AnyType rootItem )
19     { root = new BinaryNode<AnyType>( rootItem, null, null ); }
20
21     public BinaryNode<AnyType> getRoot( )
22     { return root; }
23     public int size( )
24     { return BinaryNode.size( root ); }
25     public int height( )
26     { return BinaryNode.height( root ); }
27
28     public void printPreOrder( )
29     { if( root != null ) root.printPreOrder( ); }
30     public void printInOrder( )
31     { if( root != null ) root.printInOrder( ); }
32     public void printPostOrder( )
33     { if( root != null ) root.printPostOrder( ); }
34
35     public void makeEmpty( )
36     { root = null; }
37     public boolean isEmpty( )
38     { return root == null; }
39
40     public void merge( AnyType rootItem,
41                         BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
42     { /* Figura 18.16 */ }
43
44     private BinaryNode<AnyType> root;
45 }
```

---

Figura 18.13 La clase BinaryTree, salvo en lo referido a merge.

El último método de la clase es la rutina de mezcla `merge`, que utiliza dos árboles, `t1` y `t2`, y un elemento para crear un nuevo árbol, colocando ese elemento en la raíz y los dos árboles existentes como subárboles izquierdo y derecho. En principio, esta rutina estaría compuesta por una única línea:

```
root = new BinaryNode<AnyType>( rootItem, t1.root, t2.root );
```

Pero si las cosas fueran tan simples, los programadores no tendríamos trabajo. Afortunadamente, para nuestras carreras profesionales, surgen varias complicaciones. La Figura 18.14 muestra el resultado de esa operación `merge` simple de una sola línea. Inmediatamente se ve que hay un problema: los nodos de los árboles `t1` y `t2` están ahora en dos árboles (sus arboles originales y el resultado de la mezcla). Esta compartición será un problema si queremos eliminar o modificar de algún modo los subárboles (porque podríamos eliminar o modificar múltiples subárboles de manera no intencionada).

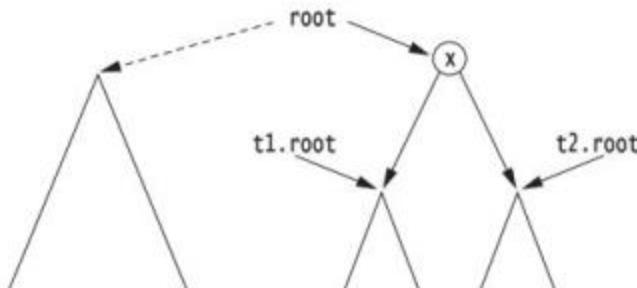
En principio, la solución parece simple. Podemos garantizar que los nodos no aparezcan en dos árboles, haciendo que `t1.root` y `t2.root` sean igual a `null` después de la operación `merge`.

Pero surgen complicaciones cuando consideramos algunas posibles llamadas que contienen nombres equivalentes entre sí:

```
t1.merge( x, t1, t2 );
t2.merge( x, t1, t2 );
t1.merge( x, t3, t3 );
```

Los dos primeros casos son similares, así que solo vamos a considerar el primero. En la Figura 18.15 se muestra un diagrama de la situación. Puesto que `t1` es un alias del objeto actual, `t1.root` y `root` son alias entre sí. Por tanto, después de la llamada a `new`, si ejecutamos `t1.root=null`, cambiaremos también `root` a la referencia `null`. En consecuencia, necesitamos tener mucho cuidado con los alias correspondientes a estos casos.

El tercer caso no se puede autorizar, porque colocaría todos los nodos pertenecientes al árbol `t3` en dos lugares de `t1`. Sin embargo, si `t3` representa un árbol vacío, se puede permitir esta operación. En conjunto, tenemos que hacer muchas más cosas de las que inicialmente esperábamos. El código resultante se muestra en la Figura 18.16. Lo que parecía ser una rutina de una sola línea ha resultado ser una rutina bastante larga.



La rutina `merge` es en principio una rutina de una sola línea. Sin embargo, también debemos tener en cuenta la posible presencia de alias, garantizar que no haya ningún nodo que pertenezca a dos árboles y comprobar los errores.

Hacemos la raíz `root` de los árboles originales igual a `null`, para que cada nodo esté en un solo árbol.

Si los dos árboles de entrada son alias entre sí, no debemos permitir la operación a menos que los árboles estén vacíos.

Si uno de los árboles de entrada es un alias del árbol de salida, tenemos que cerciorarnos de que la referencia `root` resultante no termine siendo asignada a un valor `null`.

Figura 18.14 Resultado de una operación `merge` incorrecta: los subárboles son compartidos.

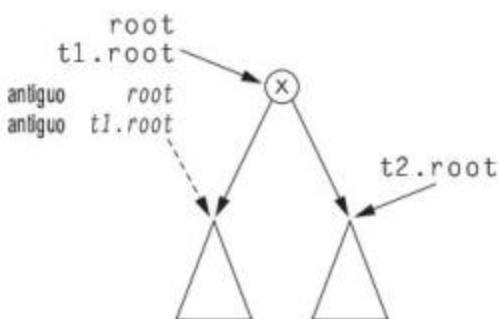


Figura 18.15 Problemas derivados de la existencia de alias en la operación merge; t1 es también el objeto actual.

```

1  /**
2  * Rutina de mezcla para clase BinaryTree.
3  * Forma un nuevo árbol a partir de rootItem, t1 y t2.
4  * No permite que t1 y t2 coincidan.
5  * Trata correctamente los restantes casos de posible existencia de alias.
6  */
7  public void merge( AnyType rootItem,
8                      BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
9  {
10     if( t1.root == t2.root && t1.root != null )
11         throw new IllegalArgumentException( );
12
13     // Asignar nuevo nodo
14     root = new BinaryNode<AnyType>( rootItem, t1.root, t2.root );
15
16     // Asegurarse de que cada nodo esté en un solo árbol
17     if( this != t1 )
18         t1.root = null;
19     if( this != t2 )
20         t2.root = null;
21 }

```

Figura 18.16 La rutina merge para la clase BinaryTree.

### 18.3 Recursión y árboles

Se utilizan rutinas recursivas para size y duplicate.

Puesto que los árboles se pueden definir recursivamente, no es sorprendente que la forma más fácil de implementar muchas rutinas para árboles sea utilizando la recursión. Aquí vamos a proporcionar implementaciones recursivas para casi todos los restantes métodos de `BinaryNode` y `BinaryTree`. Las rutinas resultantes son sorprendentemente compactas.

Comencemos con el método `duplicate` de la clase `BinaryNode`. Puesto que es un método de `BinaryNode`, tenemos garantizado que el árbol que vamos a duplicar no está vacío. El algoritmo recursivo es entonces bastante simple. En primer lugar, creamos un nuevo nodo con el mismo campo de datos que la raíz actual. Después, asociamos un árbol izquierdo invocando a `duplicate` recursivamente y asociamos un árbol derecho invocando también recursivamente `duplicate`. En ambos casos, hacemos la llamada recursiva después de verificar que efectivamente existe un árbol que copiar. Esta descripción está codificada de forma literal en la Figura 18.17.

El siguiente método que vamos a escribir es la rutina `size` de la clase `BinaryNode`. Esta rutina devuelve el tamaño del árbol que tiene su raíz en un nodo referenciado mediante `t`, que es un valor que se pasa como parámetro. Si dibujamos el árbol recursivamente, como se muestra en la Figura 18.18, vemos que el tamaño de un árbol es igual al tamaño del subárbol izquierdo más el tamaño del subárbol derecho más 1 (porque la raíz cuenta como un nodo). Toda rutina recursiva requiere un caso base que se pueda resolver sin recursión. El árbol más pequeño que `size` podría tener que tratar es el árbol vacío (si `t` es `null`) y el tamaño de un árbol vacío es obviamente 0. Hay que verificar que la recursión da la respuesta correcta para un árbol de tamaño 1. Hacer esto es sencillo, y la rutina recursiva se implementa como se ilustra en la Figura 18.19.

```

1  /**
2   * Devolver una referencia a un nodo que es la raíz de un duplicado
3   * del árbol binario cuya raíz está en el nodo actual.
4   */
5  public BinaryNode<AnyType> duplicate( )
6  {
7      BinaryNode<AnyType> root =
8          new BinaryNode<AnyType>( element, null, null );
9
10     if( left != null )           // Si hay un subárbol izquierdo
11         root.left = left.duplicate(); // Duplicar; asociar
12     if( right != null )          // Si hay un subárbol derecho
13         root.right = right.duplicate(); // Duplicar; asociar
14     return root;                // Devolver árbol resultante
15 }
```

Figura 18.17 Una rutina para devolver una copia del árbol cuya raíz es el nodo actual.

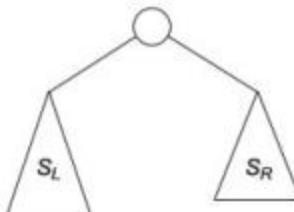


Figura 18.18 Vista recursiva utilizada para calcular el tamaño de un árbol:  $S_r = S_l + S_r + 1$ .

Puesto que `duplicate` es un método de `BinaryNode`, hacemos llamadas recursivas solo después de verificar que los subárboles no son `null`.

La rutina `size` se puede implementar fácilmente de forma recursiva después de analizar el problema con un dibujo.

```

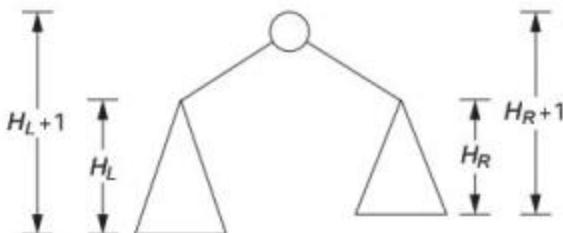
1  /**
2   * Devuelve el tamaño del árbol binario que tiene su raíz en t.
3   */
4  public static <AnyType> int size( BinaryNode<AnyType> t )
5  {
6      if( t == null )
7          return 0;
8      else
9          return 1 + size( t.left ) + size( t.right );
10 }

```

Figura 18.19 Una rutina para calcular el tamaño de un nodo.

La rutina `height` también se puede implementar fácilmente de forma recursiva. La altura de un árbol vacío es -1.

La última rutina recursiva que vamos a ver en esta sección calcula la altura de un nodo. Implementar esta rutina de forma no recursiva es complicado, mientras que la solución recursiva es trivial una vez que hemos hecho un dibujo. La Figura 18.20 muestra un árbol visto recursivamente. Suponga que el subárbol izquierdo tiene altura  $H_L$  y que el subárbol derecho tiene altura  $H_R$ . Cualquier nodo que tenga una profundidad de  $d$  niveles con respecto a la raíz del subárbol izquierdo, tendrá una profundidad de  $d + 1$  niveles con respecto a la raíz de todo el árbol. Lo mismo cabe decir con respecto al subárbol derecho. Por tanto, la longitud del camino hasta el nodo más profundo del árbol original es 1 más que su longitud de camino con respecto a la raíz de su subárbol. Si calculamos este valor para ambos subárboles, la respuesta será igual al máximo de esos dos valores más 1. En la Figura 18.21 se muestra el código para realizar estos cálculos.

Figura 18.20 Vista recursiva del cálculo de la altura de un nodo:  $H_T = \max(H_L + 1, H_R + 1)$ 

```

1  /**
2   * Devuelve la altura del árbol binario que tiene su raíz en t.
3   */
4  public static <AnyType> int height( BinaryNode<AnyType> t )
5  {
6      if( t == null )
7          return -1;
8      else
9          return 1 + Math.max( height( t.left ), height( t.right ) );
10 }

```

Figura 18.21 Una rutina para calcular la altura de un nodo.

## 18.4 Recorrido del árbol: clases iteradoras

En este capítulo hemos visto cómo se puede utilizar la recursión para implementar los métodos de un árbol binario. Cuando se aplica la recursión, calculamos la información no solo acerca de un nodo, sino también acerca de todos sus descendientes. Decimos entonces que estamos *recorriendo el árbol*. Dos métodos populares de recorrido que ya hemos mencionado son el recorrido en preorden y el recorrido en postorden.

En un recorrido en preorden, primero se procesa el nodo y luego se procesan recursivamente sus hijos. La rutina `duplicate` es un ejemplo de recorrido en preorden, porque lo primero que se hace es crear la raíz. Después se copia recursivamente en un subárbol izquierdo, para terminar copiando el subárbol derecho.

En un recorrido en postorden, el nodo se procesa después de haber procesado recursivamente ambos hijos. Dos ejemplos serían los métodos `size` y `height`. En ambos casos, la información acerca de un nodo (por ejemplo, su tamaño o su altura) se puede obtener únicamente después de conocer la correspondiente información para sus hijos.

Un tercer tipo común de recorrido recursivo es el *recorrido en orden*, en el que se procesa recursivamente el hijo izquierdo, luego se procesa el nodo actual y después se procesa recursivamente el hijo derecho. Este mecanismo se emplea, por ejemplo, para generar una expresión algebraica correspondiente a un árbol de expresión. Por ejemplo, en la Figura 18.11 el recorrido en orden nos da `(a+((b-c)*d))`.

En un recorrido en orden, el nodo actual se procesa entre ambas llamadas recursivas.

La Figura 18.22 ilustra una serie de rutinas que imprimen los nodos de un árbol binario utilizando cada uno de los tres algoritmos recursivos de recorrido del árbol. La Figura 18.23 muestra el orden en que se visitan los nodos para cada una de las tres estrategias. El tiempo de ejecución de cada algoritmo es lineal. En todos los casos, cada nodo se imprime una sola vez. En consecuencia, el coste total de una instrucción de salida para cualquier recorrido es  $O(N)$ . Como resultado, cada instrucción `if` se ejecuta también como máximo una vez por nodo, lo que nos da un coste total de  $O(N)$ . El número total de llamadas a métodos realizadas (lo que incluye el trabajo constante de las inserciones y extracciones de la pila interna de tiempo de ejecución) es también de una vez por nodo, es decir,  $O(N)$ . Por tanto, el tiempo total de ejecución es  $O(N)$ .

Un recorrido simple utilizando cualquiera de estas estrategias requiere un tiempo lineal.

¿Es obligatorio utilizar la recursión para implementar los recorridos? La respuesta es claramente no, porque como ya hemos explicado en la Sección 7.3, la recursión se implementa utilizando una pila. Por tanto, podríamos directamente mantener nuestra propia pila.<sup>2</sup> Cabría esperar que así se obtuviera un programa algo más rápido, porque podemos limitarnos a colocar en la pila únicamente la información esencial, en lugar de hacer que el compilador introduzca en la pila un registro de activación completo. La diferencia en cuanto a velocidad entre un algoritmo recursivo y otro no recursivo depende mucho de la plataforma, y en las computadoras modernas puede ser completamente despreciable. Es posible, por ejemplo, que si se utiliza una pila basada en matriz, las comprobaciones de límites que hay que realizar para que

Podemos realizar el recorrido de forma no recursiva manteniendo nosotros mismos la pila.

<sup>2</sup> También podemos añadir enlaces al padre en cada nodo del árbol, para evitar tanto la recursión como las pilas. Pero en este capítulo vamos a ilustrar la relación entre recursión y pilas, así que no utilizaremos los enlaces al nodo padre.

todos los accesos a la matriz tengan un coste significativo; la pila en tiempo de ejecución podría no estar sujeta a esas comprobaciones, si un compilador de optimización agresivo demuestra que es imposible que se produzca un subdesbordamiento de la pila. Por tanto, en muchas casos, la mejora de velocidad no justifica el esfuerzo realizado para eliminar la recursión. Aun así, saber cómo hacer esto merece la pena por si acaso su plataforma es una de las que sí se beneficiaría de eliminar la recursión, y también porque el ver cómo se implementa un programa de forma no recursiva puede en ocasiones ayudar a que la recursión esté más clara.

Una clase iteradora permite un recorrido paso a paso.

Vamos a escribir tres clases iteradoras, siguiendo todos ellos el espíritu de la lista enlazada. Cada una nos permitirá desplazarnos al primer nodo, avanzar hasta el siguiente nodo, comprobar si hemos ido más allá del último nodo y acceder al nodo actual. El orden en el que se acceda a los nodos estará

```

1 // Imprimir árbol cuya raíz está en nodo actual usando recorrido en preorden.
2 public void printPreOrder( )
3 {
4     System.out.println( element ); // Nodo
5     if( left != null )
6         left.printPreOrder( );      // Izquierda
7     if( right != null )
8         right.printPreOrder( );    // Derecha
9 }
10
11 // Impr. árbol cuya raíz está en nodo actual usando recorrido en postorden
12 public void printPostOrder( )
13 {
14     if( left != null )          // Izquierda
15         left.printPostOrder( );
16     if( right != null )         // Derecha
17         right.printPostOrder( );
18     System.out.println( element ); // Nodo
19 }
20
21 // Imprimir árbol cuya raíz está en nodo actual usando recorrido en orden.
22 public void printInOrder( )
23 {
24     if( left != null )          // Izquierda
25         left.printInOrder( );
26     System.out.println( element ); // Nodo
27     if( right != null )
28         right.printInOrder( );    // Derecha
29 }
```

Figura 18.22 Rutinas para imprimir nodos en preorden, en postorden y en orden.

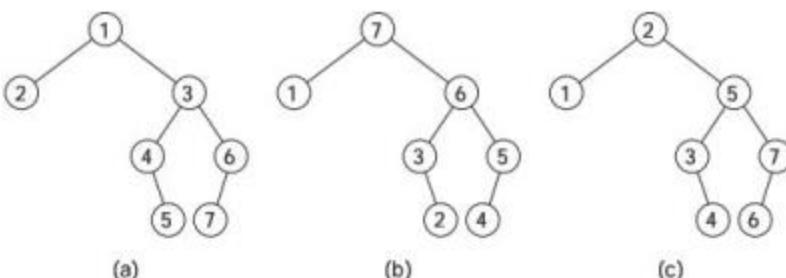


Figura 18.23 Rutas de visita en (a) preorden, (b) postorden y (c) orden.

determinado por el tipo de recorrido. También implementaremos un recorrido por niveles que es inherentemente no recursivo y que, de hecho, utiliza una cola en lugar de una pila y es similar al recorrido en preorden.

La Figura 18.24 proporciona una clase abstracta para la iteración a través de un árbol. Cada iterador almacena una referencia a la raíz del árbol y una indicación al nodo actual.<sup>3</sup> Estos miembros se declaran en las líneas 47 y 48, respectivamente, y se inicializan en el constructor. Son de tipo `protected` para permitir que las clases derivadas accedan a ellos.

En las líneas 22 a 42 se declaran cuatro métodos. Los métodos `isValid` y `retrieve` son invariantes para toda la jerarquía, de modo que se proporciona una implementación y se declaran como `final`. Los métodos abstractos `first` y `advance` deben ser proporcionados por cada tipo de iterador. Este iterador es similar al iterador de lista enlazada (`LinkedListIterator`, en la Sección 17.2), salvo porque aquí el método `first` forma parte del iterador del árbol, mientras que en la lista enlazada el método `first` era parte de la propia clase de lista.

La clase iteradora de árbol abstracta tiene métodos similares a los del iterador de lista enlazada. Cada tipo de recorrido está representado por una clase derivada.

#### 18.4.1 Recorrido en postorden

El recorrido en postorden se implementa utilizando una pila para almacenar el estado actual. La cima de la pila representará el nodo que estamos visitando en algún instante dentro del recorrido en postorden. Sin embargo, podemos estar en uno de tres lugares distintos dentro del algoritmo:

1. A punto de realizar una llamada recursiva al árbol izquierdo.
2. A punto de realizar una llamada recursiva al árbol derecho.
3. A punto de procesar el nodo actual.

El recorrido en postorden mantiene una pila que almacena los nodos que han sido visitados, pero cuyas llamadas recursivas no han sido todavía completadas.

En consecuencia, cada nodo será insertado tres veces en la pila durante el curso del recorrido. Si se extrae un nodo de la pila por tercera vez, podemos marcarlo como el nodo actual que hay que visitar.

<sup>3</sup> En estas implementaciones, una vez construidos los iteradores, no resulta seguro modificar estructuralmente el árbol durante una iteración, porque las referencias podrían quedar colgando.

```
1 import java.util.NoSuchElementException;
2
3 // Clase TreeIterator; mantiene la "posición actual"
4 //
5 // CONSTRUCCIÓN: con el árbol al que está asociado el iterador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // first y advance son abstractos; los otros son final
9 // boolean isValid( ) --> True si estamos en una posición válida del árbol
10 // AnyType retrieve( ) --> Devolver elemento de la posición actual
11 // void first( ) --> Hacer que la posición actual sea la del primer nodo
12 // void advance( ) --> Avanzar (prefijo)
13 // *****ERRORES*****
14 // Generar excepciones para operaciones ilegales de acceso o avance
15
16 abstract class TreeIterator<AnyType>
17 {
18     /**
19      * Construir el iterador. La posición actual se configura como null.
20      * @param theTree el árbol al que está asociado el iterador.
21      */
22     public TreeIterator( BinaryTree<AnyType> theTree )
23         { t = theTree; current = null; }
24
25     /**
26      * Comprobar si la pos. actual hace referencia a un elem. válido del árbol
27      * @return true si la posición actual no es null; false en caso contrario.
28      */
29     final public boolean isValid( )
30         { return current != null; }
31
32     /**
33      * Devolver el elemento almacenado en la posición actual.
34      * @return el elemento almacenado.
35      * @exception NoSuchElementException si la posición actual no es válida.
36      */
37     final public AnyType retrieve( )
38     {
39         if( current == null )
40             throw new NoSuchElementException( );
41         return current.getElement( );
42     }
43 }
```

Continúa

---

Figura 18.24 La clase base abstracta de iterador de árbol.

```
42     }
43
44     abstract public void first( );
45     abstract public void advance( );
46
47     protected BinaryTree<AnyType> t;           // La raíz del árbol
48     protected BinaryNode<AnyType> current; // La posición actual
49 }
```

Figura 18.24 (Continuación).

En caso contrario, el nodo estará siendo extraído por primera o por segunda vez. En este caso, todavía no estará listo para ser visitado, así que volvemos a introducirlo en la pila y simulamos una llamada recursiva. Si el nodo fue extraído por primera vez, necesitaremos meter en la pila el hijo izquierdo (si existe). En caso contrario, el nodo habrá sido extraído por segunda vez e insertaremos en la pila el hijo derecho (si existe). En cualquier caso, a continuación extraemos un nodo de la pila, aplicando la misma comprobación. Observe que, al extraer de la pila estamos simulando la llamada recursiva al hijo apropiado. Si el hijo no existe, y por tanto no fue introducido nunca en la pila, al extraer un nodo de la pila estaremos volviendo a extraer el nodo original.

Al final, o bien el proceso extrae un nodo por tercera vez o bien la pila se vacía. En este último caso, habremos ya iterado a lo largo de todo el árbol. Inicializaremos el algoritmo insertando en la pila una referencia a la raíz. En la Figura 18.25 se muestra un ejemplo de cómo se manipula la pila.

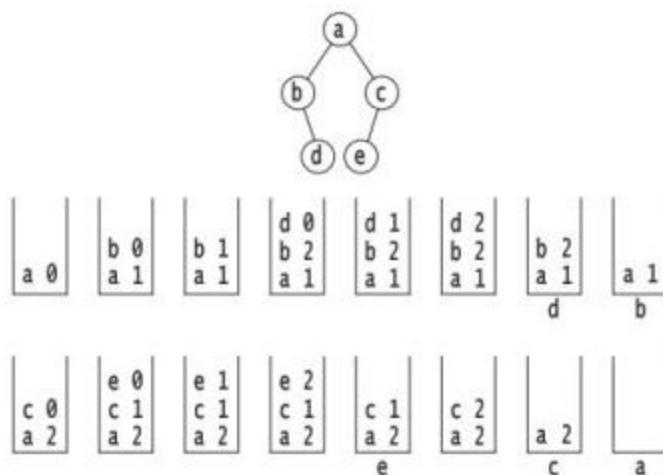
Un rápido resumen: la pila contiene nodos que hemos recorrido pero que aun no hemos completado. Cuando se introduce un nodo en la pila, el contador será 1, 2, o 3, de acuerdo con las siguientes reglas:

1. Si estamos a punto de procesar el subárbol izquierdo del nodo.
2. Si estamos a punto de procesar el subárbol derecho del nodo.
3. Si estamos a punto de procesar el propio nodo.

Hagamos una traza del recorrido postorden. Inicializamos el recorrido insertando la raíz a en la pila. La primera extracción hace que se visite a. Se trata de una primera extracción, por lo que se vuelve a colocar el nodo en la pila e insertamos en ella su hijo izquierdo, b. A continuación, se extrae b. Se trata de la primera extracción de b, por lo que se devuelve a la pila. Normalmente, a continuación sería insertado en la pila el hijo izquierdo de b, pero b no tiene ningún hijo izquierdo, por lo que no se inserta nada. Por tanto, la siguiente extracción hace que se extraiga b por segunda vez, así que b vuelve a insertarse en la pila y a continuación se introduce también en la pila su hijo derecho, d. La siguiente extracción nos da d por primera vez, por lo que se vuelve a insertar en la pila. No se realiza ninguna otra inserción, porque d no tiene ningún hijo izquierdo. Por tanto, d se extrae por segunda vez y se vuelve a insertar, pero como no tiene hijo derecho, no se inserta nada más. Por tanto, la

Cada nodo se inserta en la pila tres veces. La tercera vez que se extrae, el nodo se declara como visitado. Las veces anteriores, simulamos una llamada recursiva.

Cuando la pila esté vacía, se habrán visitado ya todos los nodos.



**Figura 18.25** Estados de la pila durante el recorrido en postorden.

siguiente extracción nos da d por tercera vez, así que d se marca como nodo visitado. El siguiente nodo extraído es b, y como es la tercera extracción de b, también se marca como visitado.

A continuación, se extrae a por segunda vez y se coloca de nuevo en la pila junto con su hijo derecho, c. Después, se extrae c por primera vez, por lo que se vuelve a insertar junto con su izquierdo, e. Ahora se extrae e, se inserta, se extrae, se inserta y finalmente se extrae por tercera vez (lo cual es una secuencia típica de los nodos hoja). Por tanto, se marca e como nodo visitado. A continuación, se extrae c por segunda vez y se devuelve a la pila. Sin embargo, no tiene hijo derecho, por lo que es inmediatamente extraído por tercera vez y se marca como visitado. Finalmente, se extrae a por tercera vez y se marca como visitado. En ese punto, la pila estará vacía y el recorrido en postorden terminará.

La clase `PostOrder` se implementa directamente a partir del algoritmo que acabamos de describir y se muestra en la Figura 18.26, salvo por el método `advance`. La clase anidada `StNode` representa los objetos insertados en la pila. Contiene una referencia a un nodo y un entero que almacena el número de veces que el elemento ha sido extraído de la pila. Cada objeto `StNode` se inicializa siempre, para reflejar el hecho de que todavía no ha sido extraído de la pila. (Utilizamos una clase `Stack` no estándar del Capítulo 16.)

La clase `PostOrder` deriva de `TreeIterator` y añade una pila interna a los miembros de datos heredados. La clase `PostOrder` se configura inicializando los miembros de datos de `TreeIterator` y luego insertando la raíz en la pila. Este proceso se ilustra en el constructor de las líneas 30 a 35. A continuación, se implementa `first` borrando la pila, insertando la raíz e invocando `advance`.

La Figura 18.27 implementa la rutina `advance`. Esta rutina sigue el esbozo que hemos comentado de forma casi literal. La línea 8 comprueba si la pila está vacía. Si lo está, habremos completado la iteración y podemos asignar a `current` el valor `null` y volver. (Si `current` ya es `null`, habremos ido más allá del final, por lo que se genera una excepción.) En caso contrario, realizamos repetidas inserciones y extracciones de la pila, hasta que salga de la pila un elemento por tercera vez. Cuando esto sucede, la comprobación de la línea 22 se evalúa como cierta y podemos volver. En caso

```

1 import weiss.nonstandard.Stack;
2 import weiss.nonstandard.ArrayStack;
3
4 // Clase PostOrder; mantiene la "posición actual"
5 // de acuerdo con un recorrido en postorden
6 //
7 // CONSTRUCCIÓN: con el árbol al que está asociado el iterador
8 //
9 // *****OPERACIONES PÚBLICAS*****
10 // boolean isValid( ) --> True si estamos en posición válida del árbol
11 // AnyType retrieve( ) --> Devolver el elemento de la posición actual
12 // void first( ) --> Hacer que la posición actual sea la del primer nodo
13 // void advance( ) --> Avanzar (prefijo)
14 // *****ERRORES*****
15 // Generar excepciones para operaciones ilegales de acceso o avance
16
17 class PostOrder<AnyType> extends TreeIterator<AnyType>
18 {
19     protected static class StNode<AnyType>
20     {
21         StNode( BinaryNode<AnyType> n )
22             { node = n; timesPopped = 0; }
23         BinaryNode<AnyType> node;
24         int timesPopped;
25     }
26
27 /**
28 * Construir el iterador. La posición actual se configura como null.
29 */
30 public PostOrder( BinaryTree<AnyType> theTree )
31 {
32     super( theTree );
33     s = new ArrayStack<StNode<AnyType>>();
34     s.push( new StNode<AnyType>( t.getRoot( ) ) );
35 }
36
37 /**
38 * Establecer la posición actual en el primer elemento.
39 */
40 public void first( )
41 {

```

*Continúa*

Figura 18.26 La clase PostOrder (clase completa excepto por advance).

```

42     s.makeEmpty( );
43     if( t.getRoot( ) != null )
44     {
45         s.push( new StNode<AnyType>( t.getRoot( ) ) );
46         advance( );
47     }
48 }
49
50 protected Stack<StNode<AnyType>> s; // La pila de objetos StNode
51 }

```

Figura 18.26 (Continuación).

contrario, en la línea 24 insertamos de nuevo el nodo en la pila (observe que el componente `timesPopped`, que indica el número de veces que se ha extraído el nodo de la pila, ya ha sido incrementado en la línea 22). A continuación implementamos la llamada recursiva. Si el nodo fue extraído por primera vez y tiene un hijo izquierdo, se inserta su hijo izquierdo en la pila. De forma similar, si el nodo fue extraído por segunda vez y tiene un hijo derecho, se inserta su hijo derecho en la pila. Observe que, en cualquiera de los dos casos, la construcción del objeto `StNode` implica que el nodo insertado se introduce en la pila con un número de extracciones igual a cero.

Al final, el bucle `for` terminará, porque algún nodo será extraído por tercera vez. A lo largo de toda la secuencia de iteración, puede haber como máximo  $3N$  inserciones y extracciones de la pila, lo cual es otra forma de establecer el carácter lineal del recorrido en postorden.

#### 18.4.2 Recorrido en orden

El recorrido en orden es similar al recorrido en postorden, salvo porque un nodo se declara como visitado después de haber sido extraído una segunda vez. Después de volver, el iterador inserta en la pila el hijo derecho (si existe) para que la siguiente llamada a `advance` pueda continuar recorriendo ahora el hijo derecho. Puesto que esta acción es tan similar a un recorrido en postorden, derivamos la clase `InOrder` de la clase `PostOrder` (aun cuando no exista una relación *ES-UM*). El único cambio es la ligera modificación de `advance`. La nueva clase se muestra en la Figura 18.28.

El recorrido en orden es igual que el recorrido en postorden, salvo porque un nodo se declara como visitado después de haber sido extraído una segunda vez. Después de volver, el iterador inserta en la pila el hijo derecho (si existe) para que la siguiente llamada a `advance` pueda continuar recorriendo ahora el hijo derecho. Puesto que esta acción es tan similar a un recorrido en postorden, derivamos la clase `InOrder` de la clase `PostOrder` (aun cuando no exista una relación *ES-UM*). El único cambio es la ligera modificación de `advance`. La nueva clase se muestra en la Figura 18.28.

#### 18.4.3 Recorrido en preorden

El recorrido en preorden es igual que el recorrido en orden, salvo porque un nodo se declara como visitado después de haber sido extraído por primera vez. Antes de volver, el iterador introduce el hijo derecho en la pila y luego introduce el hijo izquierdo. Observe el orden de introducción: queremos que el hijo izquierdo sea procesado antes que el hijo derecho, por lo que deberemos insertar primero el hijo derecho y después el hijo izquierdo.

El recorrido en preorden es igual que el recorrido en orden, salvo porque un nodo se declara como visitado después de haber sido extraído por primera vez. Antes de volver, el iterador introduce el hijo derecho en la pila y luego introduce el hijo izquierdo. Observe el orden de introducción: queremos que el hijo izquierdo sea procesado antes que el hijo derecho, por lo que deberemos insertar primero el hijo derecho y después el hijo izquierdo.

```
1  /**
2   * Avanzar la posición actual hasta el siguiente nodo del árbol,
3   * de acuerdo con el esquema de recorrido en postorden.
4   * @throws NoSuchElementException si la posición actual es null.
5   */
6  public void advance( )
7  {
8      if( s.isEmpty( ) )
9      {
10         if( current == null )
11             throw new NoSuchElementException( );
12         current = null;
13         return;
14     }
15
16     StNode<AnyType> cnode;
17
18     for( ; ; )
19     {
20         cnode = s.topAndPop( );
21
22         if( ++cnode.timesPopped == 3 )
23         {
24             current = cnode.node;
25             return;
26         }
27
28         s.push( cnode );
29         if( cnode.timesPopped == 1 )
30         {
31             if( cnode.node.getLeft( ) != null )
32                 s.push( new StNode<AnyType>( cnode.node.getLeft( ) ) );
33         }
34         else // cnode.timesPopped == 2
35         {
36             if( cnode.node.getRight( ) != null )
37                 s.push( new StNode<AnyType>( cnode.node.getRight( ) ) );
38         }
39     }
40 }
```

Figura 18.27 La rutina advance para la clase iteradora PostOrder.

```
1 // Clase InOrder; mantiene la "posición actual"
2 // de acuerdo con un recorrido en orden
3 //
4 // CONSTRUCCIÓN: con el árbol al que está asociado el iterador
5 //
6 // *****OPERACIONES PÚBLICAS*****
7 // Igual que TreeIterator
8 // *****ERRORES*****
9 // Generar excepciones para operaciones ilegales de acceso o avance
10
11 class InOrder<AnyType> extends PostOrder<AnyType>
12 {
13     public InOrder( BinaryTree<AnyType> theTree )
14     { super( theTree ); }
15
16     /**
17      * Avanzar la posición actual hasta el siguiente nodo del árbol,
18      * de acuerdo con el esquema de recorrido en orden.
19      * @throws NoSuchElementException si la iteración se ha
20      * terminado antes de la llamada.
21      */
22     public void advance( )
23     {
24         if( s.isEmpty( ) )
25         {
26             if( current == null )
27                 throw new NoSuchElementException( );
28             current = null;
29             return;
30         }
31
32         StNode<AnyType> cnode;
33         for( ; ; )
34         {
35             cnode = s.topAndPop( );
36
37             if( ++cnode.timesPopped == 2 )
38             {
39                 current = cnode.node;
40                 if( cnode.node.getRight( ) != null )
```

*Continúa*

---

Figura 18.28 La clase iteradora InOrder completa.

```

41         s.push( new StNode<AnyType>( cnode.node.getRight( ) ) );
42     return;
43 }
44 // Primera vez que procesamos
45 s.push( cnode );
46 if( cnode.node.getLeft( ) != null )
47     s.push( new StNode<AnyType>( cnode.node.getLeft( ) ) );
48 }
49 }
50 }

```

**Figura 18.28** (Continuación).

Podemos derivar la clase `PreOrder` de la clase `InOrder` o `PostOrder`, pero hacer esto sería un desperdicio, porque la pila ya no necesita mantener la cuenta del número de veces que un elemento ha sido extraído. En consecuencia, la clase `PreOrder` se deriva directamente de `TreeIterator`. La clase resultante, con sus implementaciones del constructor y del método `first` se muestra en la Figura 18.29.

En la línea 42, hemos añadido una pila de nodos de árbol a los campos de datos de `TreeIterator`. El constructor y el método `first` son similares a los que ya hemos presentado. Como se ilustra en la Figura 18.30, la operación `advance` es más simple. Ya no necesitamos un bucle `for`. En cuanto se extrae un nodo en la línea 17, se convierte en el nodo actual. Después, si existen, se insertan en la pila el hijo derecho y el hijo izquierdo.

El extraer solo una vez permite una cierta simplificación.

#### 18.4.4 Recorridos por niveles

Vamos cerrar el capítulo implementando un recorrido por niveles, que procesa los nodos comenzando por la raíz y yendo luego de arriba a abajo y después de izquierda a derecha. El nombre de este tipo de recorrido deriva del hecho de que vamos proporcionando como salida primero los nodos de nivel 0 (la raíz), luego los nodos de nivel 1 (los hijos de la raíz), después los nodos de nivel 2 (los nietos de la raíz), etc. El recorrido por niveles se implementa utilizando una cola en lugar de una pila. La cola almacena los nodos que todavía no han sido visitados. Cuando se visita un nodo, sus hijos se colocan al final de la cola, por lo que serán visitados después de que se hayan visitado los nodos que ya están en la cola. Este procedimiento garantiza que los nodos se visiten por orden de nivel. La clase `LevelOrder`, mostrada en las Figuras 18.31 y 18.32, se parece bastante a la clase `PreOrder`. Las únicas diferencias son que utilizamos una cola en lugar de una pila y que insertamos en la cola el hijo izquierdo y luego el hijo derecho, en lugar de a la inversa. Observe que la cola puede llegar a ser muy grande. En el caso peor, todos los nodos del último nivel (posiblemente  $N^2$ ) podrían estar en la cola simultáneamente.

En un recorrido por niveles, los nodos se visitan de arriba a abajo y de izquierda a derecha. El recorrido por niveles se implementa mediante una cola. Este recorrido es una búsqueda en anchura.

```
1 // Clase PreOrder; mantiene la "posición actual"
2 //
3 // CONSTRUCCIÓN: con el árbol al que está asociado el iterador
4 //
5 // *****OPERACIONES PÚBLICAS*****
6 // boolean isValid( ) --> True si estamos en posición válida del árbol
7 // AnyType retrieve( ) --> Devolver el elemento de la posición actual
8 // void first( ) --> Hacer que la pos. actual sea la del primer nodo
9 // void advance( ) --> Avanzar (prefijo)
10 // *****ERRORES*****
11 // Generar excepciones para operaciones ilegales de acceso o avance
12
13 class PreOrder<AnyType> extends TreeIterator<AnyType>
14 {
15     /**
16      * Construct the iterator. The current position is set to null.
17      */
18     public PreOrder( BinaryTree<AnyType> theTree )
19     {
20         super( theTree );
21         s = new ArrayStack<BinaryNode<AnyType>>();
22         s.push( t.getRoot() );
23     }
24
25     /**
26      * Establecer la posición actual en el primer elemento, de acuerdo con
27      * el esquema de recorrido en preorden.
28      */
29     public void first( )
30     {
31         s.makeEmpty( );
32         if( t.getRoot( ) != null )
33         {
34             s.push( t.getRoot( ) );
35             advance( );
36         }
37     }
38
39     public void advance( )
40     { /* Figura 18.30 */ }
41
42     private Stack<BinaryNode<AnyType>> s; // Pila de objetos BinaryNode
43 }
```

---

Figura 18.29 Esqueleto de la clase PreOrder y todos sus miembros excepto advance.

```

1  /**
2   * Avanzar la posición actual hasta el siguiente nodo del árbol,
3   * de acuerdo con el esquema de recorrido en preorden.
4   * @throws NoSuchElementException si la iteración se ha
5   * terminado antes de la llamada.
6   */
7  public void advance( )
8  {
9      if( s.isEmpty( ) )
10     {
11         if( current == null )
12             throw new NoSuchElementException( );
13         current = null;
14         return;
15     }
16
17     current = s.topAndPop( );
18
19     if( current.getRight( ) != null )
20         s.push( current.getRight( ) );
21     if( current.getLeft( ) != null )
22         s.push( current.getLeft( ) );
23 }

```

Figura 18.30 Rutina advance de la clase Iteradora PreOrder.

El recorrido por niveles implementa una técnica más general conocida como *búsqueda en anchura*. Hemos proporcionado un ejemplo para ilustrar esta técnica en un contexto más general en la Sección 14.2.

```

1 // Clase LevelOrder; mantiene la "posición actual"
2 // de acuerdo con el esquema de recorrido por niveles.
3 //
4 // CONSTRUCCIÓN: con el árbol al que está asociado el iterador
5 //
6 // *****OPERACIONES PÚBLICAS*****
7 // boolean isValid( ) --> True si estamos en posición válida del árbol
8 // AnyType retrieve( ) --> Devolver el elemento de la posición actual
9 // void first( ) --> Hacer que la pos. actual sea la del primer nodo
10 // void advance( ) --> Avanzar (prefijo)
11 // *****ERRORES*****
12 // Generar excepciones para operaciones ilegales de acceso o avance

```

Continúa

Figura 18.31 Esqueleto de la clase Iteradora LevelOrder.

```

13
14 class LevelOrder<AnyType> extends TreeIterator<AnyType>
15 {
16     /**
17      * Construir el iterador.
18      */
19     public LevelOrder( BinaryTree<AnyType> theTree )
20     {
21         super( theTree );
22         q = new ArrayQueue<BinaryNode<AnyType>>();
23         q.enqueue( t.getRoot( ) );
24     }
25
26     public void first( )
27     { /* Figura 18.32 */ }
28
29     public void advance( )
30     { /* Figura 18.32 */ }
31
32     private Queue<BinaryNode<AnyType>> q; // Cola de objetos BinaryNode
33 }

```

Figura 18.31 (Continuación).

```

1  /**
2  * Establecer la posición actual en el primer elemento, de acuerdo con
3  * el esquema de recorrido por niveles.
4  */
5  public void first( )
6  {
7      q.makeEmpty( );
8      if( t.getRoot( ) != null )
9      {
10          q.enqueue( t.getRoot( ) );
11          advance( );
12      }
13  }
14
15 /**
16 * Avanzar la posición actual hasta el siguiente nodo del árbol,
17 * de acuerdo con el esquema de recorrido por niveles.
18 * @throws NoSuchElementException si la iteración ha
19 * terminado antes de la llamada.
20 */

```

Continúa

Figura 18.32 Rulinas `first` y `advance` para la clase iteradora `LevelOrder`.

```

21 public void advance( )
22 {
23     if( q.isEmpty( ) )
24     {
25         if( current == null )
26             throw new NoSuchElementException( );
27         current = null;
28         return;
29     }
30
31     current = q.dequeue( );
32
33     if( current.getLeft( ) != null )
34         q.enqueue( current.getLeft( ) );
35     if( current.getRight( ) != null )
36         q.enqueue( current.getRight( ) );
37 }

```

Figura 18.32 (Continuación).

## Resumen

En este capítulo hemos hablado del árbol y, en particular, del árbol binario. Hemos ilustrado la utilización de árboles para implementar sistemas de archivos en muchas computadoras, así como algunas otras aplicaciones, como árboles de expresión y codificación, que ya hemos explorado de manera más completa en la Parte Tres. Los algoritmos utilizados para árboles emplean ampliamente la recursión. Hemos examinado tres algoritmos recursivos de recorrido (en preorden, en postorden y en orden) y hemos mostrado 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, conocida con el nombre de búsqueda en anchura. En el Capítulo 19 examinaremos otro tipo fundamental de árbol: el *árbol de búsqueda binaria*.



## Conceptos clave

- altura de un nodo** La longitud del camino que va desde un nodo hasta la hoja más profunda en un árbol. (642)
- ancestro y descendiente** Si existe un camino desde el nodo  $u$  hasta el nodo  $v$ , entonces  $u$  es un ancestro de  $v$  y  $v$  es un descendiente de  $u$ . (642)
- ancestro propio y descendiente propio** En un camino desde el nodo  $u$  hasta el nodo  $v$ , si  $u \neq v$ , entonces  $u$  es un ancestro propio de  $v$  y  $v$  es un descendiente propio de  $u$ . (642)
- árbol** Definido no recursivamente es un conjunto de nodos, junto con las aristas dirigidas que los conectan. Definido recursivamente, un árbol o bien está vacío o está compuesto por una raíz y cero o más subárboles. (641)

**árbol binario** Un árbol en el que ningún nodo puede tener más de dos hijos. Una definición conveniente de este tipo de árbol es la que tiene carácter recursivo. (649)

**hermanos** Nodos que tienen el mismo padre. (642)

**hoja** Un nodo de un árbol que no tiene hijos. (642)

**método del primer hijo/siguiente hermano** Una implementación de árbol general en la que cada nodo almacena dos enlaces por elemento: uno al hijo situado más a la izquierda (si es que el nodo no es una hoja) y uno a su hermano situado a su derecha (si es que el nodo no es ya el hermano situado más a la derecha). (643)

**padre e hijo** Los padres y los hijos se definen de forma natural. Una arista dirigida conecta el parente con el hijo. (642)

**profundidad de un nodo** La longitud del camino que va desde la raíz hasta un nodo de un árbol. (642)

**recorrido del árbol en postorden** El trabajo en un nodo se realiza después de haber evaluado a sus hijos. El recorrido requiere un tiempo constante por nodo. (646)

**recorrido del árbol en preorden** El trabajo en un nodo se realiza antes de procesar a sus hijos. El recorrido requiere un tiempo constante por nodo. (646)

**recorrido en orden** El nodo actual se procesa entre las dos llamadas recursivas. (657)

**recorrido por niveles** Los nodos se visitan de arriba a abajo y de izquierda a derecha. El recorrido por niveles se implementa utilizando una cola. Este recorrido es una búsqueda en anchura. (667)

**tamaño de un nodo** El número de descendientes que tiene un nodo (incluyendo el propio nodo). (642)



## Errores comunes

1. Permitir que un nodo pertenezca a dos árboles simultáneamente suele ser una mala idea, porque los cambios en un subárbol pueden provocar inadvertidamente cambios en múltiples subárboles.
2. Un error común es no comprobar la existencia de árboles vacíos. Si este fallo forma parte de un algoritmo recursivo, lo más probable es que el programa se cuelgue.
3. Un error común a la hora de trabajar con árboles es pensar iterativamente en lugar de recursivamente. Primero, diseñe los algoritmos recursivamente. Después, conviértalos a algoritmos iterativos, si es que resulta apropiado.



## Internet

Muchos de los ejemplos explicados en este capítulo se analizan en el Capítulo 19, en el que hablaremos de los árboles de búsqueda binaria. En consecuencia, el único código disponible en línea es el correspondiente a las clases iteradoras.

**BinaryNode.java**

Contiene la clase `BinaryNode`.

**BinaryTree.java**

Contiene la implementación de `BinaryTree`.

**TestTreeIterator.java**

Contiene la implementación de la jerarquía `TreeIterator`.



## Ejercicios

### EN RESUMEN

**18.1** Para cada nodo del árbol mostrado en la Figura 18.33

- Indique el nodo padre.
- Enumere los hijos.
- Enumere los hermanos.
- Calcule la altura.

**18.2** Para el árbol mostrado en la Figura 18.33, determine

- Qué nodo es la raíz.
- Qué nodos son hojas.
- La profundidad del árbol.
- El resultado de los recorridos en preorden, en postorden, en orden y por niveles.

**18.3** Muestre las operaciones de la pila cuando se aplica un recorrido en orden y en preorden al árbol mostrado en la Figura 18.25.

**18.4** ¿Cuál es la salida del método presentado en la Figura 18.34, en la página siguiente, para el árbol mostrado en la Figura 18.25?

### EN TEORÍA

**18.5** Suponga que un árbol binario tiene hojas  $I_1, I_2, \dots, I_M$  con profundidades  $d_1, d_2, \dots, d_M$ , respectivamente. Demuestre que  $\sum_{i=1}^M 2^{-d_i} \leq 1$  y determine cuándo es cierta la igualdad (esto se conoce con el nombre de *desigualdad de Kraft*).

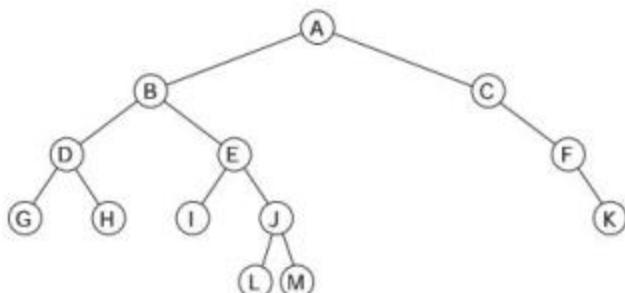


Figura 18.33 Árbol para los Ejercicios 18.1 y 18.2.

```

1  public static <AnyType> void mysteryPrint( BinaryNode<AnyType> t )
2  {
3      if( t != null )
4      {
5          System.out.println( t.getElement( ) );
6          mysteryPrint( t.getLeft( ) );
7          System.out.println( t.getElement( ) );
8          mysteryPrint( t.getRight( ) );
9          System.out.println( t.getElement( ) );
10     }
11 }

```

Figura 18.34 Programa misterioso para el Ejercicio 18.4.

- 18.6** ¿Cuántos enlaces `null` hay en un árbol binario de  $N$  nodos? ¿Cuántos hay en un árbol  $M$ -ario de  $N$  nodos?
- 18.7** Demuestre que el número máximo de nodos en un árbol binario de altura  $H$  es  $2^{H+1} - 1$ .
- 18.8** Un *nodo completo* es un nodo que tiene dos hijos. Demuestre que un árbol binario el número de nodos completos más 1 es igual al número de hojas.

## EN LA PRÁCTICA

- 18.9** Escriba de nuevo la clase iteradora para que genere una excepción cuando se aplica `first` a un árbol vacío. ¿Por qué esto podría ser una mala idea?
- 18.10** Implemente algunas de las rutinas recursivas con comprobaciones que garanticen que no se efectúe una llamada recursiva para un subárbol `null`. Compare el tiempo de ejecución con otras rutinas idénticas que retrasen esa comprobación hasta la primera línea de la rutina recursiva.
- 18.11** Escriba métodos eficientes (y proporcione sus tiempos de ejecución O mayúscula) que tomen una referencia a la raíz de un árbol binario  $T$  y calculen
- El número de hojas en  $T$ .
  - El número de nodos en  $T$  que contienen un hijo no `null`.
  - El número de nodos en  $T$  que contienen dos hijos no `null`.
- 18.12** Suponga que un árbol binario almacena enteros. Escriba métodos eficientes (y proporcione sus tiempos de ejecución O mayúscula) que tomen una referencia a la raíz de un árbol binario  $T$  y calculen
- El número de elementos de datos impares.
  - La suma de todos los elementos del árbol.
  - El número de nodos con dos hijos que contienen el mismo valor.

## PROYECTOS DE PROGRAMACIÓN

**18.13** Implemente el comando *du*.

**18.14** Escriba un método que enumere todos los directorios vacíos contenidos en un directorio especificado (incluyendo sus subdirectorios).

**18.15** Escriba un método que enumere todos los archivos contenidos en un directorio (incluyendo sus subdirectorios) que sean mayores que un tamaño especificado.

**18.16** Escriba un método que devuelva el nombre completo del archivo más grande contenido en un directorio especificado (incluyendo sus subdirectorios).

**18.17** Escriba un programa que enumere todos los archivos de un directorio (y sus subdirectorios), de forma similar al comando *ls* de Unix o al comando *dir* de Windows. Observe que al encontrar un directorio, no debe imprimir de forma inmediata su contenido recursivamente. En lugar de ello, al explorar cada directorio, coloque todos los subdirectorios en una *List*. **Después** de haber impreso las entradas del directorio, procese entonces cada subdirectorio recursivamente. Por cada archivo que se enumere, incluya la hora de modificación, el tamaño del archivo y, si se trata de un directorio, indíquelo en la salida. Para cada directorio, imprima el nombre completo del directorio antes de imprimir su contenido.

**18.18** Escriba un método que enumere todos los archivos contenidos en un directorio (incluyendo sus subdirectorios) que hayan sido modificados hoy.

**18.19** Se puede utilizar un programa para generar automáticamente un árbol binario para autoedición. Puede escribir dicho programa asignando una coordenada *x-y* a cada nodo del árbol, dibujando un círculo alrededor de cada coordenada y conectando cada nodo no raíz con su padre. Suponga que tiene un árbol binario almacenado en memoria y que cada nodo tiene dos miembros de datos adicionales para almacenar las coordenadas. Suponga que (0, 0) es la esquina superior izquierda.

- a. La coordenada *x* se puede calcular asignando el número del recorrido en orden. Escriba una rutina para hacer eso para cada nodo del árbol.
- b. La coordenada *y* se puede calcular utilizando el inverso de la profundidad del nodo. Escriba una rutina para hacer eso para cada nodo del árbol.
- c. En términos de una cierta unidad imaginaria, ¿cuáles serían las dimensiones de la imagen? Determine también cómo se pueden ajustar las unidades para que el árbol tenga siempre una altura aproximadamente igual a dos tercios de su anchura.



# Árboles de búsqueda binaria

Para grandes cantidades de entrada, el tiempo de acceso lineal de las listas enlazadas resulta prohibitivo. En este capítulo vamos a analizar una alternativa a la lista enlazada: el *árbol de búsqueda binaria*, una estructura de datos simple que se puede contemplar como una extensión del algoritmo de búsqueda binaria para permitir inserciones y borrados. El tiempo de ejecución para la mayoría de las operaciones es  $O(\log N)$  como promedio. Lamentablemente, el tiempo del caso peor es  $O(N)$  por cada operación.

En este capítulo veremos

- El árbol de búsqueda binaria básico.
- Un método para añadir estadísticas de orden (es decir, la operación `findKth`).
- Tres formas distintas de eliminar el caso peor  $O(N)$ : en concreto el *árbol AVL*, el *árbol rojo-negro* y el *árbol-AA*.
- La implementación del `TreeSet` y el `TreeMap` de la API de Colecciones.
- La utilización del *árbol-B* para buscar de forma rápida en una base de datos de gran tamaño.

## 19.1 Ideas básicas

En el caso general, buscamos un elemento utilizando su *clave*. Por ejemplo, podríamos buscar el examen correspondiente a un estudiante utilizando su identificador ID de estudiante. En este caso, el número ID se denomina clave del elemento.

El *árbol de búsqueda binaria* satisface la propiedad de búsqueda ordenada; es decir, para todo nodo  $X$  del árbol, los valores de todas las claves contenidas en el subárbol izquierdo son menores que la clave de  $X$  y los valores de todas las claves contenidas en el subárbol derecho son mayores que la clave de  $X$ . El valor que se muestra en la Figura 19.1(a) es un árbol de búsqueda binaria, pero el árbol mostrado en la Figura 19.1(b) no lo es, porque la clave 8 no corresponde al subárbol izquierdo de la clave 7. Esta propiedad del árbol de búsqueda binaria implica que todos los elementos del árbol se pueden ordenar de forma coherente (de hecho, un recorrido en orden nos proporciona los elementos ordenados). Asimismo, esta propiedad implica que no se permiten

Para cualquier nodo del *árbol de búsqueda binaria*, todos los nodos con claves más pequeñas se encuentran en el subárbol izquierdo y todos los nodos con claves mayores se encuentran en el subárbol derecho. No se permiten duplicados.

elementos duplicados. Podríamos permitir esos elementos duplicados, pero suele ser mejor, generalmente, almacenar los diferentes elementos que tienen claves idénticas en una estructura secundaria. Si estos elementos fueran duplicados exactos, lo mejor es mantener un elemento y llevar la cuenta del número de duplicados.

#### Propiedad de orden en un árbol de búsqueda binaria

En un árbol de búsqueda binaria, para todo nodo  $X$ , todas las claves contenidas en el árbol izquierdo de  $X$  tienen valores inferiores al de la clave de  $X$ , y todas las claves contenidas en el árbol derecho de  $X$  tienen valores superiores al de la clave de  $X$ .

### 19.1.1 Las operaciones

Una operación `find` se realiza bifurcándose repetidamente a la izquierda o a la derecha dependiendo del resultado de una comparación.

En 5, iríamos hacia la derecha y nos encontraríamos con un enlace `null`, por lo que no podríamos al final encontrar 6, como se muestra en la Figura 19.2(a). La Figura 19.2(b) muestra que podemos insertar 6 en el punto en que ha terminado esa búsqueda que ha concluido sin éxito.

La operación `findMin` se realiza siguiendo los nodos izquierdos hasta que deje de haber un hijo izquierdo. La operación `findMax` es similar.

El árbol de búsqueda binaria soporta de manera eficiente las operaciones `findMin` y `findMax`. Para realizar una operación `findMin`, comenzamos en la raíz y vamos bifurcándonos repetidamente a la izquierda, hasta que deje de haber un hijo izquierdo. El punto en el que nos detengamos será el elemento más pequeño. La operación `findMax` es similar, salvo porque la bifurcación se realiza hacia la derecha. Observe que el coste de todas las operaciones es proporcional al número de nodos contenidos en el camino de búsqueda. El coste tiende a ser logarítmico, aunque puede ser lineal en el caso peor. Estableceremos formalmente este resultado posteriormente en el capítulo.

La operación más costosa es la de eliminación, `remove`. Una vez que hemos encontrado el nodo que hay que eliminar, tenemos que considerar varias posibilidades. El problema es que la elimina-

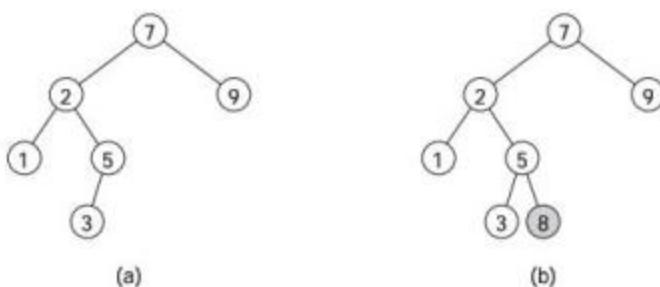


Figura 19.1 Dos árboles binarios: (a) un árbol de búsqueda; (b) no es un árbol de búsqueda.

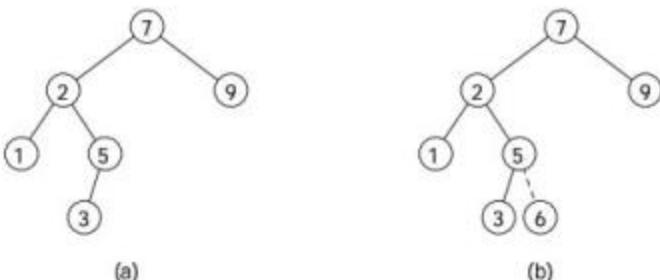


Figura 19.2 Árboles de búsqueda binaria: (a) antes de la inserción y (b) después de la inserción de 6.

ción de un nodo puede dejar desconectadas partes del árbol. Si eso sucede, deberemos reasociar con cuidado el árbol manteniendo la propiedad de orden que hace de él un árbol de búsqueda binaria. También queremos evitar tener que hacer el árbol innecesariamente profundo, porque la profundidad del árbol afecta al tiempo de ejecución de los algoritmos del árbol.

Cuando estamos diseñando un algoritmo complejo, a menudo lo más fácil es resolver primero el caso más simple, dejando el caso más complicado para el final. Por tanto, al examinar los diversos casos, vamos a comenzar por el más fácil. Si el nodo es una hoja, su eliminación no desconecta el árbol, por lo que lo podemos borrar de forma inmediata. Si el nodo tiene solo un hijo, podemos eliminarlo después de ajustar el enlace que su padre tiene para apuntar al nodo con el fin de sostener este. Esto se ilustra en la Figura 19.3, con la eliminación del nodo 5. Observe que `removeMin` y `removeMax` no son operaciones complejas, porque los nodos afectados o son hojas o solo tienen un hijo. Observe que la raíz también es un caso especial, porque carece de padre. Sin embargo, al implementar el método `remove`, el caso especial se gestiona de manera automática.

El caso más complicado es aquel en el que un nodo tiene dos hijos. La estrategia general consiste en reemplazar el elemento de este nodo por el elemento más pequeño del subárbol derecho (que puede ser localizado fácilmente, como ya hemos mencionado antes) y luego eliminar ese nodo (que ahora estará lógicamente vacío). Esa segunda operación `remove` es fácil de hacer porque, como acabamos de indicar, el nodo mínimo de un árbol no

La operación `remove` es difícil porque los nodos que no son hojas mantienen el árbol conectado y no queremos que quede desconectado después del borrado.

Si un nodo tiene un hijo, se le puede eliminar haciendo que su padre lo sostenga. La raíz es un caso especial, porque no tiene padre.

Un nodo con dos hijos se sustituye utilizando el elemento más pequeño del subárbol derecho. A continuación, se elimina ese otro nodo.

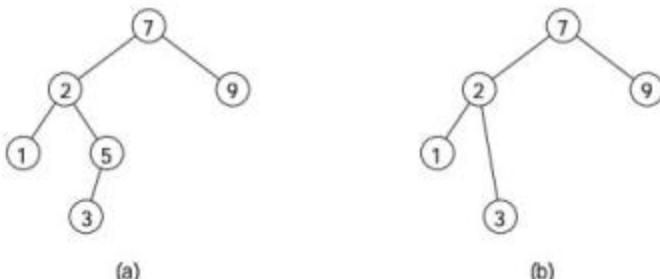
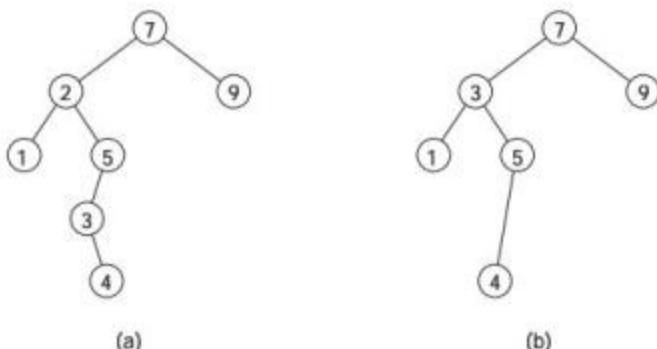


Figura 19.3 Borrado del nodo 5 con un hijo: (a) antes y (b) después.



**Figura 19.4** Borrado del nodo 2 con dos hijos: (a) antes y (b) después.

tiene un hijo izquierdo. La Figura 19.4 muestra un árbol inicial y el resultado de eliminar el nodo 2. Sustituimos el nodo por el nodo más pequeño (3) de su subárbol derecho y luego eliminamos 3 del subárbol derecho. Observe que en todos los casos, eliminar un nodo no hace que el árbol sea más profundo.<sup>1</sup> Muchas alternativas sí que hacen que el árbol sea más profundo; por tanto, esas alternativas son opciones menos recomendables.

### 19.1.2 Implementación java

En principio, el árbol de búsqueda binaria es fácil de implementar. Para impedir que las características de Java oscurezcan el código, vamos a introducir unas cuantas simplificaciones. En primer lugar, la Figura 19.5 muestra la clase `BinaryNode`. En la nueva clase `BinaryNode`, hacemos que todo tenga visibilidad de paquete. En la práctica, lo más normal es que `BinaryNode` fuera una clase anidada. La clase `BinaryNode` contiene la lista usual de miembros de datos (el elemento y dos enlaces).

`root` hace referencia a la raíz del árbol, y tendrá un valor `null` si el árbol está vacío.

Las funciones públicas de la clase invocan a rutinas privadas ocultas.

El esqueleto de la clase `BinarySearchTree` se muestra en la Figura 19.6. El único miembro de datos es la referencia a la raíz del árbol, `root`. Si el árbol está vacío, `root` es `null`.

Los métodos públicos de la clase `BinarySearchTree` tienen implementaciones que invocan a los métodos ocultos. El constructor, declarado en la línea 21, simplemente configura `root` como `null`. Los métodos públicamente visibles se enumeran en las líneas 24 a 39.

A continuación, tenemos varios métodos que operan sobre un nodo que se pasa como parámetro, una técnica general que ya hemos utilizado en el Capítulo 18. La idea es que las rutinas de la clase con visibilidad pública invocan a estas rutinas ocultas, pasándolas `root` como parámetro. Estas rutinas ocultas se encargan de realizar todo el trabajo. En unos cuantos casos, empleamos `protected` en lugar de `private` porque derivamos otra clase de `BinarySearchTree` en la Sección 19.2.

<sup>1</sup> Sin embargo, el borrado puede incrementar la profundidad media de nodo, si se elimina un nodo poco profundo.

```

1 package weiss.nonstandard;
2
3 // Nodo básico almacenado en árboles de búsqueda binaria no equilibrados.
4 // Observe que esta clase no es accesible fuera
5 // de este paquete.
6
7 class BinaryNode<AnyType>
8 {
9     // Constructor
10    BinaryNode( AnyType theElement )
11    {
12        element = theElement;
13        left = right = null;
14    }
15
16    // Datos; accesibles por parte de otras rutinas del paquete
17    AnyType element;           // Los datos del nodo
18    BinaryNode<AnyType> left; // Hijo izquierdo
19    BinaryNode<AnyType> right; // Hijo derecho
20 }

```

Figura 19.5 La clase `BinaryNode` para el árbol de búsqueda binaria.

El método `insert` añade `x` al árbol actual, invocando la rutina oculta `insert` con `root` como parámetro adicional. Esta acción falla si `x` ya se encuentra en el árbol; en ese caso, se genera una excepción `DuplicateItemException`. Las operaciones `findMin`, `findMax` y `find` devuelven el elemento mínimo, el elemento máximo o el elemento indicado (respectivamente) del árbol. Si no se encuentra el elemento, porque el árbol está vacío o porque el elemento indicado no está presente, entonces se devuelve `null`. La Figura 19.7 muestra el método privado `elementAt` que implementa la lógica `elementAt`.

La operación `removeMin` elimina el elemento mínimo del árbol; genera una excepción si el árbol está vacío. La operación `remove` elimina un elemento especificado `x` del árbol; genera una excepción en caso necesario. Los métodos `makeEmpty` e `isEmpty` utilizan la lógica habitual.

Como suele ser típico en la mayoría de las estructuras de datos, la operación `find` es más fácil que `insert` e `insert` es más fácil que `remove`. La Figura 19.8 ilustra la rutina `find`. Mientras que no se alcance un enlace `null`, tendremos una correspondencia o necesitaremos bifurcarnos a izquierda o a derecha. El código implementa este algoritmo de forma bastante sencilla. Observe el orden de las comprobaciones. La comparación con `null` debe realizarse en primer lugar; en caso contrario, el acceso a `t.element` sería ilegal. Las restantes comprobaciones se ordenan poniendo en último lugar el caso menos probable. Es posible hacer una implementación recursiva, pero nosotros empleamos en su lugar un bucle; utilizaremos la recursión en los métodos `insert` y `remove`. En el Ejercicio 19.16 le pediremos que escriba los algoritmos de búsqueda de forma recursiva.

```

1 package weiss.nonstandard;
2
3 // Clase BinarySearchTree
4 //
5 // CONSTRUCCIÓN: sin ningún inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void insert( x )      --> Insertar x
9 // void remove( x )      --> Eliminar x
10 // void removeMin( )     --> Eliminar elemento mínimo
11 // Comparable find( x )  --> Devolver elemento que se corresponde con x
12 // Comparable findMin( ) --> Devolver el menor elemento
13 // Comparable findMax( ) --> Devolver el mayor elemento
14 // boolean isEmpty( )    --> Devolver true si está vacío; false, en otro caso
15 // void makeEmpty( )     --> Eliminar todos los elementos
16 // *****ERRORES*****
17 // insert, remove y removeMin generan excepciones en caso necesario
18
19 public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
20 {
21     public BinarySearchTree( )
22         { root = null; }
23
24     public void insert( AnyType x )
25         { root = insert( x, root ); }
26     public void remove( AnyType x )
27         { root = remove( x, root ); }
28     public void removeMin( )
29         { root = removeMin( root ); }
30     public AnyType findMin( )
31         { return elementAt( findMin( root ) ); }
32     public AnyType findMax( )
33         { return elementAt( findMax( root ) ); }
34     public AnyType find( AnyType x )
35         { return elementAt( find( x, root ) ); }
36     public void makeEmpty( )
37         { root = null; }
38     public boolean isEmpty( )
39         { return root == null; }
40
41     private AnyType elementAt( BinaryNode<AnyType> t )
42         { /* Figura 19.7 */ }
43     private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
44         { /* Figura 19.8 */ }
45     protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
46         { /* Figura 19.9 */ }

```

Continúa

Figura 19.6 El esqueleto de la clase BinarySearchTree.

```

47     private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
48         /* Figura 19.9 */
49     protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
50         /* Figura 19.10 */
51     protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
52         /* Figura 19.11 */
53     protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
54         /* Figura 19.12 */
55
56     protected BinaryNode<AnyType> root;
57 }

```

**Figura 19.6 (Continuación).**

```

1 /**
2  * Método interno para obtener el campo element.
3  * @param t el nodo.
4  * @return el campo element o null si t es null.
5  */
6 private AnyType elementAt( BinaryNode<AnyType> t )
7 {
8     return t == null ? null : t.element;
9 }

```

**Figura 19.7** El método `elementAt`.

```

1 /**
2  * Método interno para encontrar un elemento en un subárbol.
3  * @param x es el elemento que hay que buscar.
4  * @param t el nodo que actúa como raíz del árbol.
5  * @return el nodo que contiene el elemento encontrado.
6  */
7 private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
8 {
9     while( t != null )
10    {
11        if( x.compareTo( t.element ) < 0 )
12            t = t.left;
13        else if( x.compareTo( t.element ) > 0 )
14            t = t.right;
15        else
16            return t; // Encontrada correspondencia
17    }
18
19    return null; // No encontrada
20 }

```

**Figura 19.8** La operación `find` para árboles de búsqueda binaria.

A primera vista, instrucciones como `t=t.left` parecen cambiar la raíz del árbol. Sin embargo, no es así, porque `t` se pasa por valor. En la llamada inicial, `t` es simplemente una *copia* de `root`.

Debido a la llamada por valor, el argumento real (`root`) no varía.

Para `insert`, debemos devolver la nueva raíz del árbol y reconectar el árbol.

Aunque `t` cambia, `root` no lo hace. Las llamadas a `findMin` y `findMax` son aun más simples, porque la bifurcación se realiza siempre incondicionalmente en una única dirección. Estas rutinas se muestran en la Figura 19.9. Observe cómo se maneja el caso de un árbol vacío.

La rutina `insert` se muestra en la Figura 19.10. Aquí utilizamos la recursión para simplificar el código. También es posible una implementación no recursiva; aplicaremos esta técnica cuando hablaremos de los árboles rojo-negro posteriormente en el capítulo. El algoritmo básico es simple. Si el árbol está vacío, podemos crear un árbol de un solo nodo. La comprobación se lleva a cabo en la línea 10, y el nuevo nodo se crea en la línea 11. Observe que, como antes, los cambios locales a `t` se pierden. Así, devolvemos la nueva raíz, `t`, en la línea 18.

```

1  /**
2   * Método interno para encontrar el elemento mínimo en un subárbol.
3   * @param t el nodo que actúa como raíz del árbol.
4   * @return el nodo que contiene el elemento mínimo.
5   */
6  protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
7  {
8      if( t != null )
9          while( t.left != null )
10             t = t.left;
11
12     return t;
13 }
14
15 /**
16 * Método interno para encontrar el elemento máximo en un subárbol.
17 * @param t el nodo que actúa como raíz del árbol.
18 * @return el nodo que contiene el elemento máximo.
19 */
20 private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
21 {
22     if( t != null )
23         while( t.right != null )
24             t = t.right;
25
26     return t;
27 }
```

Figura 19.9 Los métodos `findMin` y `findMax` para árboles de búsqueda binaria.

```

1  /**
2  * Método interno para insertar en un subárbol.
3  * @param x el elemento que hay que insertar.
4  * @param t el nodo que actúa como raíz del árbol.
5  * @return la nueva raíz.
6  * @throws DuplicateItemException si x ya está presente.
7  */
8  protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
9  {
10     if( t == null )
11         t = new BinaryNode<AnyType>( x );
12     else if( x.compareTo( t.element ) < 0 )
13         t.left = insert( x, t.left );
14     else if( x.compareTo( t.element ) > 0 )
15         t.right = insert( x, t.right );
16     else
17         throw new DuplicateItemException( x.toString() ); // Duplicado
18     return t;
19 }

```

Figura 19.10 La operación recursiva `insert` para la clase `BinarySearchTree`.

Si el árbol no está ya vacío, tenemos tres posibilidades. En primer lugar, si el elemento que tenemos que insertar es más pequeño que el elemento contenido en el nodo `t`, podemos invocar `insert` recursivamente sobre el subárbol izquierdo. En segundo lugar, si el elemento es más grande que el elemento contenido en el nodo `t`, podemos invocar `insert` recursivamente sobre el subárbol derecho (estos dos casos están codificados en las líneas 12 a 15). En tercer lugar, si el elemento que hay que insertar se corresponde con el elemento contenido en `t`, generamos una excepción.

Las restantes rutinas se ocupan del borrado. Como hemos descrito anteriormente, la operación `removeMin` es simple porque el nodo mínimo no tiene ningún hijo izquierdo. Por tanto, lo único que hay que hacer es soslayar el nodo eliminado, lo que parece exigirnos llevar la cuenta de quién es el padre del nodo actual a medida que vamos descendiendo por el árbol. Pero, de nuevo, podemos evitar el uso explícito de un enlace al padre, utilizando la recursión. El código se muestra en la Figura 19.11.

Si el árbol `t` está vacío, `removeMin` falla. En caso contrario, si `t` tiene un hijo izquierdo, eliminamos recursivamente el elemento mínimo del subárbol izquierdo a través de la llamada recursiva de la línea 13. Si alcanzamos la línea 17, sabemos que estamos actualmente en el nodo mínimo, y por tanto `t` será la raíz de un subárbol que carece de hijo izquierdo. Si asignamos a `t` el valor `t.right`, `t` será ahora la raíz de un subárbol en el que ahora ya no estará su anterior elemento mínimo. Como antes, devolvemos la raíz del subárbol resultante. Eso es lo que hacemos en la línea 17. ¿Pero eso no hace que quede desconectado el árbol? La respuesta de nuevo es que no. Si `t` era `root`, se devuelve el nuevo `t` y se asigna a `root` en el método público. Si `t` no era `root`, entonces `p.left`, donde `p` es el parente de `t` en el momento de la

La raíz del nuevo subárbol debe ser devuelta en las rutinas `remove`. Lo que hacemos en la práctica es mantener al parente en la pila de recursión.

```

1  /**
2   * Método interno para eliminar el elemento mínimo de un subárbol.
3   * @param t el nodo que actúa como raíz del árbol.
4   * @return la nueva raíz.
5   * @throws ItemNotFoundException si t está vacío.
6   */
7  protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
8  {
9      if( t == null )
10         throw new ItemNotFoundException();
11     else if( t.left != null )
12     {
13         t.left = removeMin( t.left );
14         return t;
15     }
16     else
17         return t.right;
18 }

```

Figura 19.11 El método `removeMin` para la clase `BinarySearchTree`.

llamada recursiva. El método que tiene `p` como parámetro (en otras palabras, el método que invocó al método actual) cambia `p.left` al nuevo `t`. Así, el enlace `left` del padre hace referencia a `t` y el árbol queda conectado. En conjunto, es una maniobra bastante ingeniosa –hemos mantenido el padre en la pila de recursión, en lugar de mantenerlo explícitamente en un bucle iterativo.

Habiendo utilizado este truco para el caso simple, podemos a continuación adaptarlo para la rutina general `remove` mostrada en la Figura 19.12. Si el árbol está vacío, la operación `remove` no tendrá éxito y podemos generar una excepción en la línea 11. Si no encontramos una correspondencia, podemos invocar `remove` recursivamente para el subárbol izquierdo o derecho según sea apropiado. En caso contrario, llegaremos a la línea 16, lo que indica que hemos encontrado el nodo que hay que eliminar.

La rutina `remove` incluye algunos trucos de codificación, pero no es demasiado compleja si se utiliza la recursión. Los casos correspondientes a un único hijo, a la raíz con un hijo y a cero hijos se tratan todos ellos de forma conjunta en la línea 22.

Recuerde (como se ilustra en la Figura 19.4) que, si hay dos hijos, sustituimos el nodo por el elemento mínimo del subárbol derecho y luego eliminamos ese mínimo del subárbol derecho (lo que está codificado en las líneas 18 y 19). En caso contrario, tendremos uno o cero hijos. Si hay un hijo izquierdo, hacemos `t` igual a su hijo izquierdo, como haríamos en `removeMax`. En caso contrario, sabemos que no hay hijo izquierdo y que podemos hacer `t` igual a su hijo derecho. Este procedimiento está codificado de forma sucinta en la línea 22, que también cubre el caso de un nodo hoja.

Hay dos observaciones que hacer con respecto a esta implementación. En primer lugar, durante las operaciones básicas `insert`, `find` o `remove`, utilizamos dos comparaciones de tres vías por cada nodo al que accedemos, para distinguir entre los casos `<`, `=` y `>`. Obviamente, podemos calcular `x.compareTo(t.element)` una vez por cada iteración de bucle y reducir el coste a una comparación de tres vías por nodo. Sin embargo, en realidad

```

1  /**
2  * Método interno para eliminar de un subárbol.
3  * @param x el elemento que hay que eliminar.
4  * @param t el nodo que actúa como raíz del árbol.
5  * @return la nueva raíz.
6  * @throws ItemNotFoundException si no se encuentra x.
7  */
8  protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
9  {
10    if( t == null )
11      throw new ItemNotFoundException( x.toString() );
12    if( x.compareTo( t.element ) < 0 )
13      t.left = remove( x, t.left );
14    else if( x.compareTo( t.element ) > 0 )
15      t.right = remove( x, t.right );
16    else if( t.left != null && t.right != null ) // Dos hijos
17    {
18      t.element = findMin( t.right ).element;
19      t.right = removeMin( t.right );
20    }
21    else
22      t = ( t.left != null ) ? t.left : t.right;
23    return t;
24  }

```

Figura 19.12 El método `remove` para la clase `BinarySearchTree`.

podemos apañarnos con una sola comparación de dos vías por nodo. La estrategia es similar a la que aplicamos en el algoritmo de búsqueda binaria de la Sección 5.6. Hablaremos de la aplicación de esta técnica a los árboles de búsqueda binaria en la Sección 19.6.2, cuando ilustremos el algoritmo de borrado para árboles AA.

En segundo lugar, no necesitamos utilizar recursión para llevar a cabo la inserción. De hecho, una implementación recursiva es probablemente más lenta que una implementación no recursiva. Expondremos una implementación iterativa de `insert` en la Sección 19.5.3 en el contexto de los árboles rojo-negro.

## 19.2 Estadísticas de orden

El árbol de búsqueda binaria nos permite encontrar el elemento mínimo o el máximo en un tiempo que es equivalente a una operación `find` para un elemento arbitrariamente especificado. En ocasiones, necesitamos poder acceder también al  $K$ -ésimo elemento más pequeño, para un valor  $K$  arbitrario proporcionado como parámetro. Podemos hacer esto si llevamos la cuenta de cuál es el tamaño de cada nodo del árbol.

Podemos implementar `findKth` manteniendo el tamaño de cada nodo a medida que actualizamos el árbol.

Recuerde de la Sección 18.1 que el tamaño de un nodo es igual al número de sus descendientes (incluyendo el mismo). Suponga que queremos encontrar el  $K$ -ésimo elemento más pequeño y que  $K$  es un valor comprendido entre 1 y el número de nodos que componen el árbol. La Figura 19.13 muestra tres casos posibles, dependiendo de la relación entre  $K$  y el tamaño del subárbol izquierdo, designado por  $S_L$ . Si  $K$  es igual a  $S_L + 1$ , la raíz será el  $K$ -ésimo elemento más pequeño y podemos paramos. Si  $K$  es más pequeño que  $S_L + 1$  (es decir, es menor o igual que  $S_L$ ), el  $K$ -ésimo elemento más pequeño deberá estar en el subárbol izquierdo y lo podemos encontrar de forma recursiva. (La recursión se puede evitar; la utilizamos para simplificar la descripción del algoritmo.) En caso contrario, el  $K - S_L - 1$ -ésimo elemento más pequeño del subárbol derecho y se puede encontrar de manera recursiva.

El trabajo principal consiste en mantener el tamaño de los nodos durante las modificaciones de árbol. Estos cambios se producen durante las operaciones `insert`, `remove` y `removeMin`. En principio, esta tarea de mantenimiento es bastante simple. Durante una operación `insert`, cada nodo recorrido en el camino hasta el punto de inserción ve incrementado el tamaño de su subárbol en un nodo. Por tanto, el tamaño de cada nodo se incrementa en 1, y el nodo insertado tendrá tamaño 1. En `removeMin`, cada nodo del camino que va hasta el mínimo pierde un nodo de su subárbol; por tanto, el tamaño de cada nodo se reducirá en 1. Durante una operación `remove`, todos los nodos contenidos en el camino que va hasta el nodo físicamente eliminado pierden también un nodo de su subárbol. En consecuencia, podemos mantener esos tamaños, con solo un pequeño coste adicional de procesamiento.

### 19.2.1 Implementación Java

Derivamos una nueva clase que soporta las estadísticas de orden.

Lógicamente, los únicos cambios requeridos son la adición de `findKth` y el mantenimiento de un miembro de datos `size` en `insert`, `remove` y `removeMin`. Derivamos una nueva clase a partir de `BinarySearchTree`, el esqueleto de la cual se muestra en la Figura 19.14. Proporcionamos una clase anidada que amplía `BinaryNode` y añade un miembro de datos `size`.

`BinarySearchWithRank` añade un único método público, denominado `findKth`, mostrado en las líneas 31 y 32. Los restantes métodos públicos se heredan sin modificaciones. Debemos sustituir algunas de las rutinas recursivas `protected` (líneas 36–41).

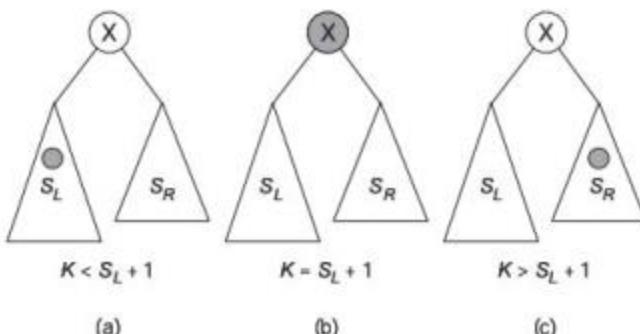


Figura 19.13 Utilización del miembro de datos `size` para implementar `findKth`.

```

1 package weiss.nonstandard;
2
3 // Clase BinarySearchTreeWithRank
4 //
5 // CONSTRUCCIÓN: sin ningún inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // findKth( k ) Comparable--> Devolver k-ésimo elemento más pequeño
9 // Todas las demás operaciones se heredan
10 // *****ERRORES*****
11 // Se genera IllegalArgumentException si k está fuera de límites
12
13 public class BinarySearchTreeWithRank<AnyType extends Comparable<? super AnyType>>
14     extends BinarySearchTree<AnyType>
15 {
16     private static class BinaryNodeWithSize<AnyType> extends BinaryNode<AnyType>
17     {
18         BinaryNodeWithSize( AnyType x )
19             { super( x ); size = 0; }
20
21         int size;
22     }
23
24     /**
25      * Encontrar el k-ésimo elemento más pequeño del árbol.
26      * @param k el rango deseado (1 es el elemento más pequeño).
27      * @return el k-ésimo elemento más pequeño del árbol.
28      * @throws IllegalArgumentException si k es menor que 1
29      * o mayor que el tamaño del subárbol.
30     */
31     public AnyType findKth( int k )
32         { return findKth( k, root ).element; }
33
34     protected BinaryNode<AnyType> findKth( int k, BinaryNode<AnyType> t )
35         { /* Figura 19.15 */ }
36     protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> tt )
37         { /* Figura 19.16 */ }
38     protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> tt )
39         { /* Figura 19.18 */ }
40     protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> tt )
41         { /* Figura 19.17 */ }
42 }

```

Figura 19.14 Esqueleto de la clase BinarySearchTreeWithRank.

La operación `findKth` se puede implementar fácilmente una vez que se conocen los valores de los miembros de datos que indican el tamaño.

Las operaciones `insert` y `remove` son potencialmente complejas, porque no tenemos que actualizar la información de tamaño si la operación no tiene éxito.

La operación `findKth` mostrada en la Figura 19.15 está escrita recursivamente, aunque es obvio que no tendría por qué ser así. Sigue la descripción del algoritmo de forma textual. La comprobación con respecto a `null` en la línea 10 es necesaria, porque `k` podría tener un valor no legal. Las líneas 12 y 13 calculan el tamaño del subárbol izquierdo. Si el subárbol izquierdo existe, obtenemos la respuesta necesaria accediendo a su miembro de datos `size`. Si el subárbol izquierdo no existe, podemos asumir que su tamaño es igual a 0. Observe que esta comprobación se realiza después de asegurarnos de que `t` no es `null`.

La operación `insert` se muestra en la Figura 19.16. La parte potencialmente compleja es que, si la llamada para la inserción tiene éxito, necesitamos incrementar el miembro de datos `size` de `t`. Si la llamada recursiva falla, el miembro de datos `size` de `t` no se ve modificado y debe generarse una excepción. En una inserción que no tenga éxito, ¿pueden verse modificados algunos tamaños? La respuesta es no; `size` solo se actualiza si la llamada recursiva tiene éxito sin que se genere una excepción. Observe que cuando se asigna a un nuevo nodo mediante una llamada a `new`, el miembro de datos `size` se configura con el valor 0 en el constructor `BinaryNodeWithSize`, y luego se incrementa en la línea 20.

La Figura 19.17 muestra que hemos utilizado el mismo truco para `removeMin`. Si la llamada recursiva tiene éxito, se reduce el valor del miembro de datos `size`; si la llamada recursiva falla, `size` no se ve modificado. La operación `remove` es similar y se muestra en la Figura 19.18.

```

1  /**
2   * Método interno para encontrar el k-ésimo elemento mínimo de un subárbol.
3   * @param k el rango deseado (1 es el elemento mínimo).
4   * @return nodo que contiene el k-ésimo elemento más pequeño de un subárbol.
5   * @throws IllegalArgumentException si k es menor que 1
6   * o mayor que el tamaño del subárbol.
7   */
8  protected BinaryNode<AnyType> findKth( int k, BinaryNode<AnyType> t )
9  {
10    if( t == null )
11      throw new IllegalArgumentException( );
12    int leftSize = ( t.left != null ) ?
13        ((BinaryNodeWithSize<AnyType>) t.left).size : 0;
14
15    if( k <= leftSize )
16      return findKth( k, t.left );
17    if( k == leftSize + 1 )
18      return t;
19    return findKth( k - leftSize - 1, t.right );
20  }

```

Figura 19.15 La operación `findKth` para un árbol de búsqueda con estadísticas de orden.

```

1 /**
2 * Método interno para insertar en un subárbol.
3 * @param x el elemento que hay que insertar.
4 * @param tt el nodo que actúa como raíz del árbol.
5 * @return la nueva raíz.
6 * @throws DuplicateItemException si x ya está presente.
7 */
8 protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> tt )
9 {
10 BinaryNodeWithSize<AnyType> t = (BinaryNodeWithSize<AnyType>) tt;
11
12 if( t == null )
13 t = new BinaryNodeWithSize<AnyType>( x );
14 else if( x.compareTo( t.element ) < 0 )
15 t.left = insert( x, t.left );
16 else if( x.compareTo( t.element ) > 0 )
17 t.right = insert( x, t.right );
18 else
19 throw new DuplicateItemException( x.toString() );
20 t.size++;
21 return t;
22 }

```

Figura 19.16 La operación `insert` para un árbol de búsqueda con estadísticas de orden.

```

1 /**
2 * Método interno para encontrar elemento más pequeño de un subárbol,
3 * ajustando los campos de tamaño según sea apropiado.
4 * @param t el nodo que actúa como raíz del árbol.
5 * @return la nueva raíz.
6 * @throws ItemNotFoundException si el subárbol está vacío.
7 */
8 protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> tt )
9 {
10 BinaryNodeWithSize<AnyType> t = (BinaryNodeWithSize<AnyType>) tt;
11
12 if( t == null )
13 throw new ItemNotFoundException( );
14 if( t.left == null )
15 return t.right;
16
17 t.left = removeMin( t.left );
18 t.size--;
19 return t;
20 }

```

Figura 19.17 La operación `removeMin` para un árbol de búsqueda con estadísticas de orden.

```

1  /**
2  * Método interno para eliminar de un subárbol.
3  * @param x el elemento que hay que eliminar.
4  * @param t el nodo que actúa como raíz del árbol.
5  * @return la nueva raíz.
6  * @throws ItemNotFoundException si no se encuentra x.
7  */
8  protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> tt )
9  {
10    BinaryNodeWithSize<AnyType> t = (BinaryNodeWithSize<AnyType>) tt;
11
12    if( t == null )
13      throw new ItemNotFoundException( x.toString() );
14    if( x.compareTo( t.element ) < 0 )
15      t.left = remove( x, t.left );
16    else if( x.compareTo( t.element ) > 0 )
17      t.right = remove( x, t.right );
18    else if( t.left != null && t.right != null ) // Dos hijos
19    {
20      t.element = findMin( t.right ).element;
21      t.right = removeMin( t.right );
22    }
23    else
24      return ( t.left != null ) ? t.left : t.right;
25
26    t.size--;
27    return t;
28 }

```

Figura 19.18 La operación `remove` para un árbol de búsqueda con estadísticas de orden.

### 19.3 Análisis de las operaciones con árboles de búsqueda binaria

El coste de cada operación con un árbol de búsqueda binaria (`insert`, `find` y `remove`) es proporcional al número de nodos a los que se accede durante la operación. Podemos por tanto cargarle al acceso a cualquier nodo del árbol un coste igual a 1 más su profundidad (recuerde que la profundidad mide el número de aristas de un camino, en lugar del número de nodos), lo que nos da el coste de una búsqueda que tenga éxito.

El coste de una operación es proporcional a la profundidad del último nodo al que se accede. El coste es logarítmico para un árbol bien equilibrado, pero podría llegar a ser lineal para un árbol degenerado.

La Figura 19.19 muestra dos árboles. La Figura 19.19(a) ilustra un árbol equilibrado de 15 nodos. El coste para acceder a cualquier nodo es como máximo 4 unidades, y algunos nodos requieren un menor número de accesos. Esta situación es análoga a la que se presenta en el algoritmo de búsqueda binaria. Si el árbol está perfectamente equilibrado, el coste del acceso es logarítmico.

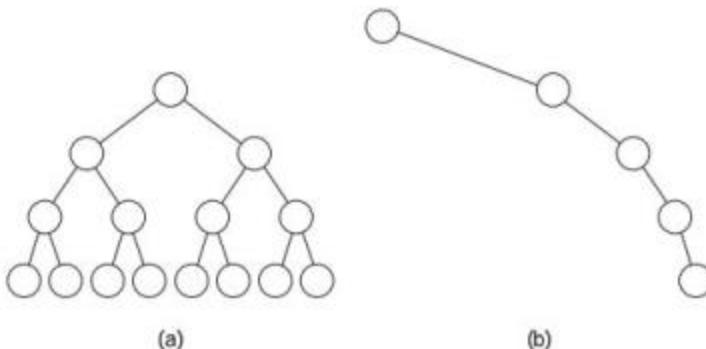


Figura 19.19 (a) El árbol equilibrado tiene una profundidad de  $\log N$ . (b) El árbol no equilibrado tiene una profundidad de  $N - 1$ .

Lamentablemente, no tenemos ninguna garantía de que el árbol esté perfectamente equilibrado. El árbol mostrado en la Figura 19.19(b) es el ejemplo clásico de un árbol no equilibrado. En él, los  $N$  nodos están contenidos en el camino que va hasta el nodo más profundo, por lo que el tiempo de búsqueda de caso peor será  $O(N)$ . Puesto que el árbol de búsqueda ha degenerado en una lista enlazada, el tiempo promedio requerido para buscar *en este caso concreto* equivale a la mitad del coste de caso peor y es también  $O(N)$ . Por tanto, tenemos dos extremos: en el caso mejor, el coste de acceso es logarítmico y en el caso peor el coste de acceso es lineal. ¿Cuál será entonces el coste promedio? ¿La mayoría de los árboles de búsqueda binaria tiende al caso equilibrado o al no equilibrado? ¿O existe algún punto intermedio, como  $\sqrt{N}$ ? La respuesta es idéntica al caso del algoritmo de ordenación rápida: el promedio es un 38 por ciento peor que el mejor de los casos.

Como promedio, la profundidad es un 38 por ciento superior a la del caso mejor. Este resultado es idéntico al obtenido utilizando el algoritmo de ordenación rápida.

Vamos a demostrar en esta sección que la profundidad media de todos los nodos en un árbol de búsqueda binaria es logarítmica bajo la suposición de que cada árbol se cree como resultado de secuencias aleatorias de inserción (sin operaciones *remove*). Para ver lo que eso significa, considere el resultado de insertar tres elementos en un árbol de búsqueda binaria vacío. Lo único que nos importa es su ordenación relativa, por lo que podemos asumir sin pérdida de generalidad que los tres elementos son 1, 2 y 3. Entonces hay seis posibles órdenes de inserción: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) y (3, 2, 1). Vamos a asumir en nuestra demostración que todos los órdenes de inserción son igualmente probables. Los árboles de búsqueda binaria que

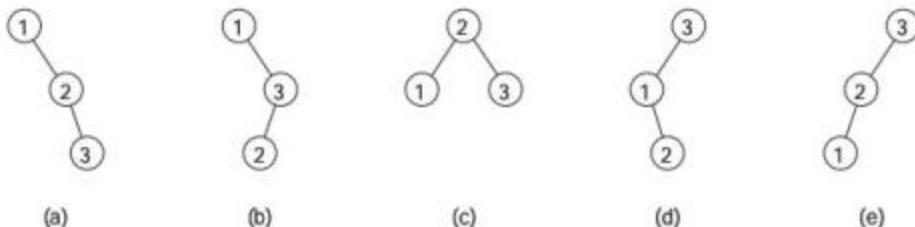


Figura 19.20 Árboles de búsqueda binaria que se pueden obtener al insertar una permutación de 1, 2 y 3; el árbol equilibrado mostrado en (c) se obtiene con una probabilidad dos veces mayor que cualquiera de los otros.

pueden obtenerse como resultado de estas inserciones se muestran en la Figura 19.20. Observe que el árbol con raíz 2, mostrado en la Figura 19.20(c), se forma a partir de la secuencia de inserción (2, 3, 1) o de la secuencia (2, 1, 3). Por tanto, algunos árboles son más probables de obtener que otros y, como veremos, los árboles equilibrados tienen una mayor probabilidad de obtenerse que los árboles no equilibrados (aunque este resultado no es evidente a partir del caso de solo tres elementos).

Comenzamos con la siguiente definición.

**Definición:** La longitud interna de camino de un árbol binario es la suma de las profundidades de sus nodos.

La longitud interna de camino se utiliza para medir el coste de una búsqueda que tenga éxito.

Al dividir la longitud interna de camino de un árbol entre el número de nodos del árbol obtenemos la profundidad media de nodo. Sumando 1 a este promedio, obtenemos el coste promedio de una búsqueda en el árbol que tenga éxito. Por tanto, lo que queremos es calcular la longitud promedio interna de camino para un árbol de búsqueda binario, donde el promedio se calcula para todas las permutaciones de entrada (igualmente probables). Podemos hacer esto fácilmente contemplando el árbol de forma recursiva y utilizando técnicas extraídas del análisis del algoritmo de ordenación rápida estudiado en la Sección 8.6. La longitud promedio interna de camino se establece en el Teorema 19.1.

### Teorema 19.1

La longitud interna de camino de un árbol de búsqueda binaria es aproximadamente  $1,38 N \log N$  en promedio, bajo la suposición de que todas las permutaciones sean equiprobables.

### Demostración

Sea  $D(N)$  la longitud interna promedio de camino para árboles de  $N$  nodos, de modo que  $D(1) = 0$ . Un árbol  $T$  de  $N$  nodos estará compuesto por un subárbol izquierdo de  $i$  nodos y un subárbol derecho de  $(N - i - 1)$  nodos más una raíz con profundidad 0, para  $0 \leq i < N$ . Por hipótesis, cada valor de  $i$  es igualmente probable. Para un valor de  $i$  dado,  $D(i)$  es la longitud promedio interna de camino del subárbol izquierdo con respecto a su raíz. En  $T$ , todos estos nodos tienen una profundidad un nivel mayor. Por tanto, la contribución promedio de los nodos del subárbol izquierdo a la longitud interna de camino de  $T$  será  $(1/N) \sum_{i=0}^{N-1} D(i)$ , más 1 por cada nodo del subárbol izquierdo. Lo mismo cabe decir del subárbol derecho. Obtenemos así la fórmula de recurrencia  $D(N) = (2/N) \left( \sum_{i=0}^{N-1} D(i) + N - 1 \right)$ , que es idéntica a la recurrencia para el algoritmo de ordenación rápida resuelta en la Sección 8.6. El resultado es una longitud promedio interna de camino de  $\mathcal{O}(N \log N)$ .

La longitud externa de camino se utiliza para medir el coste de una búsqueda que no tenga éxito.

El algoritmo de inserción implica que el coste de una inserción es igual al coste de una búsqueda que no tenga éxito, la cual se mide utilizando la longitud externa de camino. En una inserción o en una búsqueda que no tenga éxito, terminaremos por encontrarnos con la comprobación  $t == null$ . Recuerde que en un árbol de  $N$  nodos hay  $N + 1$  enlaces  $null$ . La longitud externa de camino mide el número total de nodos a los que se accede, incluyendo el nodo  $null$  para cada uno de estos  $N + 1$  enlaces  $null$ . El nodo  $null$  se denomina en ocasiones *nodo externo del árbol*, lo que explica el término *longitud externa de camino*. Como veremos más adelante en el capítulo, puede ser conveniente sustituir el nodo  $null$  por un nodo centinela.

**Definición:** La longitud externa de camino de un árbol de búsqueda binaria es la suma de las profundidades de los  $N + 1$  enlaces nulos. El nodo nulo terminal se considera un nodo a este respecto.

Si sumamos uno al resultado de dividir la longitud promedio externa de camino entre  $N + 1$ , obtenemos el coste promedio de una búsqueda que no tenga éxito o de una inserción. Al igual que sucede con el algoritmo de búsqueda binaria, el coste promedio de una búsqueda que no tenga éxito solo es ligeramente superior que el coste de una búsqueda que sí lo tenga, lo que se deduce del Teorema 19.2.

### Teorema 19.2

Para cualquier árbol  $T$ , sea  $IPL(T)$  la longitud interna de camino de  $T$  y sea  $EPL(T)$  su longitud externa de camino. Entonces, si  $T$  tiene  $N$  nodos,  $EPL(T) = IPL(T) + 2N$ .

### Demostración

Este teorema se demuestra por inducción y se deja como Ejercicio 19.8 para el lector.

Resulta tentador decir inmediatamente que estos resultados implican que el tiempo medio de ejecución de todas las operaciones es  $O(\log N)$ . Esta implicación es cierta en la práctica, pero no la hemos demostrado analíticamente, porque la suposición empleada para demostrar los resultados anteriores no tiene en cuenta el algoritmo de borrado. De hecho, un examen atento sugiere que podríamos tener problemas con nuestro algoritmo de borrado, porque la operación `remove` siempre sustituye los nodos borrados de dos hijos por un nodo del subárbol derecho. Este resultado parecería tener el efecto de llegar a desequilibrar el árbol, que tendería a estar escorado hacia la izquierda. Se ha demostrado que si construimos un árbol de búsqueda binaria aleatorio y luego realizamos aproximadamente  $N^2$  parejas de combinaciones `insert`/`remove` aleatorias, los árboles de búsqueda binaria tendrán una profundidad esperada de  $O(\sqrt{N})$ . Sin embargo, un número razonable de operaciones `insert` y `remove` aleatorias (en las que el orden de `insert` y `remove` es también aleatorio) no desequilibra el árbol de una manera observable. De hecho, para árboles de búsqueda pequeños, el algoritmo de `remove` parece equilibrar el árbol. En consecuencia, podemos razonablemente asumir que, para entradas aleatorias, todas las operaciones se efectúan en un tiempo promedio logarítmico, aunque este resultado no lo hemos demostrado matemáticamente. En el Ejercicio 19.17 describiremos algunas estrategias de borrado alternativas.

Las operaciones `remove` aleatorias no preservan la aleatoriedad de un árbol. Los efectos no se comprenden del todo desde el punto de vista teórico, pero aparentemente son despreciables en la práctica.

El problema más importante no es el potencial desequilibrio provocado por el algoritmo `remove`, más bien, el problema es que si la secuencia de entrada está ordenada, nos encontraremos con el árbol de caso peor. Cuando esto suceda, tendremos una grave situación. Nos encontraremos con un tiempo lineal por operación (para una serie de  $N$  operaciones) en lugar de con un coste logarítmico por operación. Este caso es análogo al caso de pasar elementos a un algoritmo de ordenación rápida, pero haciendo que en su lugar se ejecute una ordenación por inserción. El tiempo resultante de ejecución es completamente inaceptable. Además, no son solo las entradas ordenadas las que resultan problemáticas, sino también cualquier entrada que contenga largas secuencias no aleatorias. Una solución a este problema consiste en insistir en una condición estructural adicional denominada *equilibrio*: no se permite que ningún nodo tenga una profundidad excesiva.

Se pueden utilizar diversos algoritmos para implementar un *árbol de búsqueda binaria equilibrado*, que tiene una propiedad estructural adicional que garantiza una profundidad logarítmica

Un árbol de búsqueda binaria equilibrado tiene una propiedad estructural añadida que garantiza una profundidad logarítmica en caso peor. Las actualizaciones son más lentas, pero los accesos son más rápidos.

en caso peor. La mayoría de estos algoritmos son mucho más complicados que los correspondientes a los árboles de búsqueda binaria estándar, y todos ellos requieren un tiempo medio mayor para la inserción y el borrado. Sin embargo, proporcionan protección contra esos embarazosos casos simples que conducen a un rendimiento muy pobre de los árboles de búsqueda binaria desequilibrados. Asimismo, puesto que son equilibrados, tienden a proporcionar un tiempo de acceso más rápido que el correspondiente a los árboles estándar. Normalmente, sus longitudes internas de camino están muy próximas al valor óptimo  $N \log N$ , en lugar de a  $1,38N \log N$ , por lo que el tiempo de búsqueda es aproximadamente un 25 por ciento más rápido.

## 19.4 Árboles AVL

El árbol AVL fue el primer árbol de búsqueda binaria equilibrado que se desarrolló. Tiene una importancia de carácter histórico y también ilustra la mayoría de las ideas utilizadas en los restantes esquemas similares.

El primer árbol de búsqueda binaria equilibrado que se desarrolló fue el *árbol AVL* (llamado así en honor a sus descubridores, Adelson-Velskii y Landis), que ilustra las ideas características de una amplia clase de árboles de búsqueda binaria equilibrados. Se trata de un árbol de búsqueda binaria que tiene una condición adicional de equilibrio. Cualquier condición de equilibrio debe ser fácil de mantener y garantiza que la profundidad del árbol sea  $O(\log N)$ . La idea más simple consiste en exigir que los subárboles izquierdo y derecho tengan la misma altura. La recursión dicta que esta idea se aplica a todos los nodos del árbol, porque cada uno es, en sí mismo, la raíz de algún subárbol.

Esta condición de equilibrio garantiza que la profundidad del árbol sea logarítmica. Sin embargo, es demasiado restrictiva, porque es difícil insertar nuevos elementos al mismo tiempo que se mantiene el equilibrio. Por tanto, la definición de un árbol AVL utiliza una noción de equilibrio algo más débil, pero que sigue siendo lo suficientemente fuerte como para garantizar una profundidad logarítmica.

**Definición:** Un *árbol AVL* es un árbol de búsqueda binaria con la condición adicional de equilibrio de que, para cualquier nodo del árbol, las alturas de los subárboles izquierdo y derecho pueden diferir como máximo en 1. Como es usual, la altura de un subárbol vacío es igual a -1.

### 19.4.1 Propiedades

Todo nodo de un árbol AVL tiene subárboles cuyas alturas difieren como máximo en 1. Un subárbol vacío tiene una altura -1.

La Figura 19.21 muestra dos árboles de búsqueda binaria. El árbol ilustrado en la Figura 19.21(a) satisface la condición de equilibrio AVL y es, por tanto, un árbol AVL. El árbol de la Figura 19.21(b), que es el resultado de insertar 1, utilizando el algoritmo habitual, no es un árbol AVL, porque los nodos marcados con sombreado oscuro tienen subárboles izquierdos cuyas alturas son 2 unidades mayores que las de sus subárboles derechos. Si se insertara el valor 13 empleando el algoritmo de inserción habitual de los árboles de búsqueda binaria, el nodo 16 también violaría la regla de equilibrio. La razón es que el subárbol izquierdo tendría altura 1, mientras que el subárbol derecho tendría altura -1.

valores 13 empleando el algoritmo de inserción habitual de los árboles de búsqueda binaria, el nodo 16 también violaría la regla de equilibrio. La razón es que el subárbol izquierdo tendría altura 1, mientras que el subárbol derecho tendría altura -1.

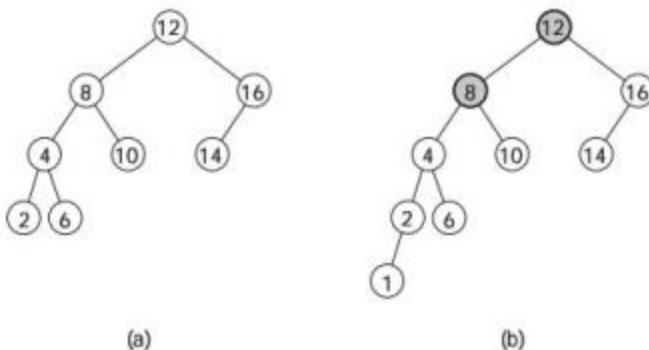


Figura 19.21 Dos árboles de búsqueda binaria: (a) un árbol AVL; (b) un árbol no AVL (los nodos no equilibrados se muestran más oscuros).

La condición de equilibrio AVL implica que el árbol tiene solo una profundidad logarítmica. Para demostrar esta afirmación, necesitamos demostrar que un árbol de altura  $H$  debe tener al menos  $C^H$  nodos para alguna constante  $C > 1$ . En otras palabras, el número mínimo de nodos en un árbol es exponencial con respecto a su altura. Entonces, la profundidad máxima de un árbol de  $N$  elementos estará dada por  $\log_C N$ . El Teorema 19.3 muestra que todo árbol AVL de altura  $H$  tiene un cierto número mínimo de nodos.

Un árbol AVL tiene una altura que es como máximo un 44 por ciento mayor que la altura mínima.

**Teorema 19.3** Un árbol AVL de altura  $H$  tiene al menos  $F_{H+3} - 1$  nodos, donde  $F_i$  es el  $i$ -ésimo número de Fibonacci (véase la Sección 7.3.4).

#### Demostración

Sea  $S_H$  el tamaño del árbol AVL más pequeño de altura  $H$ . Claramente,  $S_0 = 1$  y  $S_1 = 2$ . La Figura 19.22 muestra que el árbol más pequeño de altura  $H$  debe tener subárboles de alturas  $H - 1$  y  $H - 2$ . La razón es que al menos uno de los subárboles tiene altura  $H - 1$  y la condición de equilibrio implica que las alturas de los subárboles pueden diferir como máximo en 1. Estos subárboles deben, ellos mismos, tener el menor número posible de nodos para sus correspondientes alturas, por lo que  $S_H = S_{H-1} + S_{H-2} + 1$ . La demostración puede completarse utilizando un argumento de inducción.

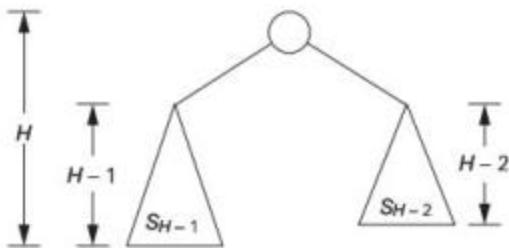


Figura 19.22 Árbol mínimo de altura  $H$ .

A partir del Ejercicio 7.8,  $F_i = \phi^i / \sqrt{5}$ , donde  $\phi = (1 + \sqrt{5}) / 2 \approx 1,618$ . En consecuencia, un árbol AVL de altura  $H$  tiene al menos (aproximadamente)  $\phi^{H+3} / \sqrt{5}$  nodos. Por tanto, su profundidad es como máximo logarítmica. La altura de un árbol AVL satisface la relación

$$H < 1,44 \log(N+2) - 1,328 \quad (19.1)$$

La profundidad de un nodo típico de un árbol AVL es muy próximo al valor óptimo  $\log N$ .

Una actualización en un árbol AVL podría destruir el equilibrio. Habrá entonces que re-equilibrar el árbol antes de poder considerar la operación completada.

Solo podrá verse afectado el equilibrio de aquellos nodos que estén situados en el camino que va de la raíz hasta el punto de inserción.

Si corregimos el equilibrio en el nodo desequilibrado más profundo, volvemos a re-equilibrar el árbol completo. Hay cuatro casos que tendríamos potencialmente que corregir; dos de ellos son simétricos con respecto a los otros dos.

El equilibrio se restaura mediante rotaciones del árbol. Una rotación simple intercambia los papeles del padre y del hijo, al mismo tiempo que mantiene el orden de búsqueda.

por lo que la altura de caso peor es, como máximo, aproximadamente un 44 por ciento superior al mínimo posible para los árboles binarios.

La profundidad media de un nodo en un árbol AVL construido aleatoriamente tiende a ser muy próxima a  $\log N$ . La respuesta exacta no se ha establecido todavía de manera analítica. Ni siquiera sabemos si la forma es  $\log N + C_0(1+\epsilon)\log N + C$ , para algún  $\epsilon$  que sería aproximadamente 0,01. Las simulaciones han sido incapaces de demostrar de manera convincente si una de las dos formas es más plausible que la otra.

Una consecuencia de estos argumentos es que todas las operaciones de búsqueda de un árbol AVL tienen cotas logarítmicas de caso peor. La dificultad estriba en que las operaciones que modifican el árbol, como `insert` y `remove`, ya no son tan simples como antes. La razón es que una inserción (o un borrado) puede destruir el equilibrio de varios nodos del árbol, como se muestra en la Figura 19.21. El equilibrio deberá ser entonces restaurado antes de poder considerar completada la operación. Describiremos aquí el algoritmo de inserción y dejaremos el algoritmo de borrado como Ejercicio 19.10 para el lector.

Una observación crucial es que, después de una inserción, solo los nodos que se encuentran en el camino que va desde el punto de inserción a la raíz podrán haber visto modificado su equilibrio, ya que solo se modifican los subárboles de esos nodos. Este resultado se aplica a todos los algoritmos de árboles de búsqueda equilibrados. A medida que seguimos el camino hacia arriba hasta la raíz y actualizamos la información de equilibrado, podemos toparnos con un nodo cuyo nuevo equilibrio viola la condición AVL. En esta sección, mostraremos cómo re-equilibrar el árbol en el primero (es decir, en el más profundo) de esos nodos y demostraremos que este re-equilibrado garantiza que todo el árbol satisface la propiedad AVL.

El nodo que hay que re-equilibrar es  $X$ . Puesto que cualquier nodo tiene como máximo dos hijos y un desequilibrio de alturas requiere que las alturas de los dos subárboles de  $X$  difieran en 2, pueden producirse cuatro casos distintos de violación del equilibrio:

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 casos simétricos con respecto a  $X$ , al igual que lo son los casos 2 y 3. En consecuencia, solo existen dos casos básicos desde el punto de vista teórico. Desde el punto de vista de la programación seguirán existiendo, por supuesto, cuatro casos y numerosos casos especiales.

El primer caso, en el que la inserción tiene lugar en el exterior (es decir, izquierda-izquierda o derecha-derecha), se puede corregir mediante una única rotación del árbol. Una *rotación simple* intercambia los papeles del padre y el

hijo, al mismo tiempo que mantiene el orden de búsqueda. El segundo caso, en el que la inserción tiene lugar en el interior (es decir, izquierda-derecha o derecha-izquierda) se puede tratar mediante una *rotación doble*, que es ligeramente más compleja. Estas operaciones fundamentales con el árbol se utilizan en diversas ocasiones en los algoritmos para árboles equilibrados. En el resto de esta sección vamos a describir estas rotaciones y a demostrar que son suficientes para mantener la condición de equilibrio.

#### 19.4.2 Rotación simple

La Figura 19.23 muestra la rotación simple que permite corregir el caso 1. En la Figura 19.23(a), el nodo  $k_2$  viola la propiedad de equilibrio AVL porque su subárbol izquierdo es dos niveles más profundo que su subárbol derecho (en esta sección marcamos los niveles mediante líneas punteadas). La situación mostrada es el único escenario posible del caso 1 que permite a  $k_2$  satisfacer la propiedad AVL antes de la inserción, pero violarla después de ella. El subárbol  $A$  ha crecido hasta tener un nivel adicional, lo que hace que sea dos niveles más profundo que  $C$ . El subárbol  $B$  no puede estar al mismo nivel que el nuevo  $C$ , porque entonces  $k_2$  habría estado desequilibrado *antes* de la inserción. El subárbol  $B$  no puede estar al mismo nivel que  $C$  porque entonces  $k_1$  habría sido el primer nodo del camino que violara la condición de equilibrio AVL (y estamos afirmando que ese primer nodo es  $k_2$ ).

Una rotación simple sirve para tratar los casos exteriores (1 y 4). Efectuamos una rotación entre un nodo y su hijo. El resultado es un árbol de búsqueda binaria que satisface la propiedad AVL.

Idealmente, para re-equilibrar el árbol, lo que queríramos es mover  $A$  hacia arriba un nivel y  $C$  hacia abajo un nivel. Observe que estas acciones son más que lo que la propiedad AVL exige. Para llevarlas a cabo, reordenamos los nodos en un árbol de búsqueda equivalente, como se muestra en la Figura 19.23(b). He aquí un escenario abstracto: piense en el árbol como si fuera flexible, sujetelo por el nodo  $k_1$ , cierre los ojos y agite el árbol dejando que la gravedad actúe. El resultado será que  $k_1$  será la nueva raíz. La propiedad del árbol de búsqueda binaria nos dice que en el árbol original,  $k_2 > k_1$ , por lo que  $k_2$  se convierte en el hijo derecho de  $k_1$  en el nuevo árbol. Los subárboles  $A$  y  $C$  continúan siendo el hijo izquierdo de  $k_1$  y el hijo derecho de  $k_2$ , respectivamente. El subárbol  $B$ , que alberga los elementos entre  $k_1$  y  $k_2$  en el árbol original, puede colocarse como hijo izquierdo de  $k_2$  en el nuevo árbol, con lo que se satisfacen todos los requisitos de ordenación.

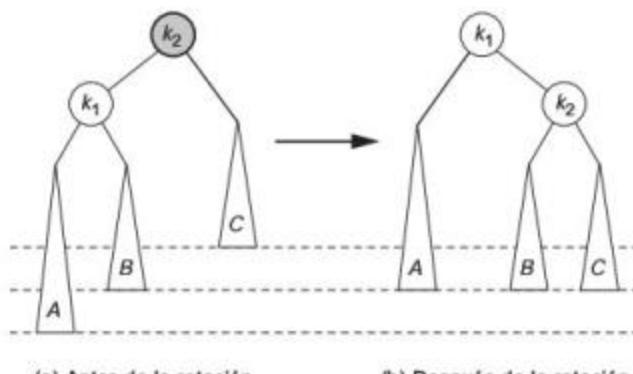


Figura 19.23 Rotación simple para corregir el caso 1.

Una rotación basta para corregir los casos 1 y 4 en un árbol AVL.

Este trabajo solo requiere cambiar los pocos enlaces a hijos mostrados en el pseudocódigo de la Figura 19.24, y nos da como resultado otro árbol binario que es un árbol AVL. Este resultado se produce porque  $A$  se mueve hacia arriba un nivel,  $B$  permanece en el mismo nivel y  $C$  se mueve hacia abajo un nivel. Por tanto,  $k_1$  y  $k_2$  no solo satisfacen los requisitos AVL, sino que también tienen subárboles que son de la misma altura. Además, la nueva altura de todo el subárbol es *exactamente la misma* que la altura del subárbol original antes de la inserción que hizo que  $A$  creciera. Por tanto, no hace falta ninguna actualización adicional de las alturas en el camino que va hasta la raíz y, en consecuencia, *no hacen faltan rotaciones adicionales*. En este capítulo emplearemos esta rotación simple en otros algoritmos para árboles equilibrados.

La Figura 19.25(a) muestra que después de la inserción de 1 en un árbol AVL, el nodo 8 pasa a estar desequilibrado. Se trata, claramente, de un problema de caso 1, porque el valor 1 se encuentra en el subárbol izquierdo-izquierdo de 8. Por tanto, haremos una rotación simple de 8 y 4, obteniendo así el árbol mostrado en la Figura 19.25(b). Como hemos dicho anteriormente en esta sección, el caso 4 representa un caso simétrico. La rotación requerida se muestra en la Figura 19.26 y el pseudocódigo que la implementa se proporciona en la Figura 19.27. Esta rutina, junto con otras rotaciones presentadas en esta sección, está replicada en varios árboles de búsqueda más adelante en el texto. Estas rutinas de rotación aparecen en el código en línea para diversas implementaciones de árboles de búsqueda equilibrados.

```

1  /**
2   * Rotar un nodo del árbol binario con un hijo izquierdo.
3   * Para árboles AVL, esta es una rotación simple para el caso 1.
4   */
5  static BinaryNode rotateWithLeftChild( BinaryNode k2 )
6  {
7      BinaryNode k1 = k2.left;
8      k2.left = k1.right;
9      k1.right = k2;
10     return k1;
11 }
```

Figura 19.24 Pseudocódigo para una rotación simple (caso 1).

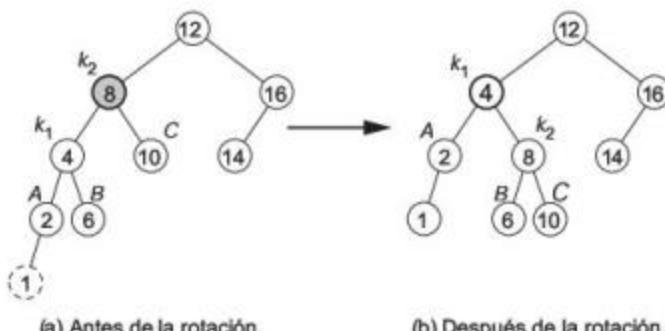


Figura 19.25 Una rotación simple permite corregir el árbol AVL después de insertar el valor 1.

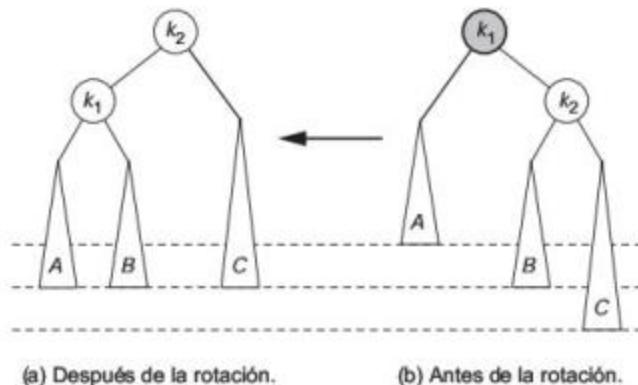


Figura 19.26 Rotación simple simétrica para corregir el caso 4.

```

1  /**
2   * Rotar un nodo del árbol binario con un hijo derecho.
3   * Para árboles AVL, esta es una rotación simple para el caso 4.
4   */
5  static BinaryNode rotateWithRightChild( BinaryNode k1 )
6  {
7      BinaryNode k2 = k1.right;
8      k1.right = k2.left;
9      k2.left = k1;
10     return k2;
11 }

```

Figura 19.27 Pesudocódigo para una rotación simple (caso 4).

### 19.4.3 Rotación doble

La rotación simple tiene un problema: como muestra la Figura 19.28, no funciona para el caso 2 (o, por simetría, para el caso 3). El problema es que el subárbol  $Q$  es demasiado profundo y una rotación simple no hace que su profundidad disminuya. La rotación doble que resuelve el problema se muestra en la Figura 19.29.

El hecho de que el subárbol  $Q$  en la Figura 19.28 sea el subárbol en el que se acaba de insertar un elemento garantiza que no está vacío. Podemos asumir que tiene una raíz y dos subárboles (posiblemente vacíos), por lo que podemos contemplar el árbol completo como cuatro subárboles conectados por tres nodos. Vamos a renombrar por tanto los cuatro árboles como  $A$ ,  $B$ ,  $C$  y  $D$ . Como sugiere la Figura 19.29, o bien el subárbol  $B$  o bien el subárbol  $C$  estarán a un nivel de profundidad dos unidades mayor que el subárbol  $D$  (a menos que ambos estén vacíos, en cuyo caso lo estarán los dos) pero no estamos seguros cuál de los subárboles es. En realidad, no importa en absoluto; aquí dibujamos tanto  $B$  como  $C$  a 1,5 niveles por debajo de  $D$ .

La rotación simple no corrige los casos interiores (2 y 3). Estos casos requieren una rotación doble, en la que están involucrados tres nodos y cuatro subárboles.

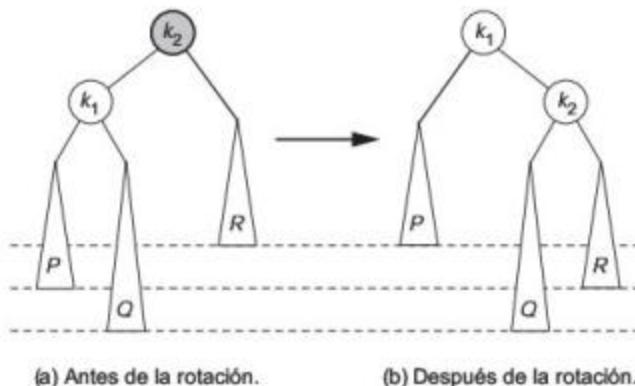


Figura 19.28 La rotación simple no resuelve el caso 2.

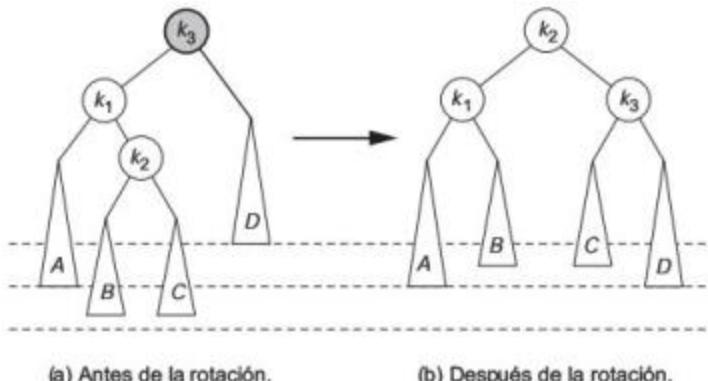


Figura 19.29 La rotación doble izquierda-derecha para corregir el caso 2.

Para re-equilibrar, no podemos dejar  $k_3$  como raíz. En la Figura 19.28 hemos mostrado que una rotación entre  $k_3$  y  $k_1$  no funciona, por lo que la única alternativa es colocar  $k_2$  como raíz. Hacer esto obliga a  $k_1$  a ser el hijo de izquierdo de  $k_2$  y a  $k_3$  a ser el hijo derecho de  $k_2$ . También determina las ubicaciones resultantes de los cuatro subárboles, y el árbol resultante satisface la propiedad AVL. Asimismo, como sucedía con la rotación simple, restaura la altura al valor que tenía antes de la inserción, garantizando así que todo el re-equilibrado y actualización de alturas está completo.

Por ejemplo, la Figura 19.30(a) muestra el resultado de insertar 5 en un árbol AVL. Se provoca un desequilibrio de altura en el nodo 8, lo que nos da como resultado un problema de caso 2.

Realizamos una rotación doble en ese nodo, obteniendo así el árbol mostrado en la Figura 19.30(b).

La Figura 19.31 muestra que el caso 3 simétrico también se puede corregir mediante una rotación doble. Finalmente, observe que, aunque una rotación doble parece compleja, es equivalente a la siguiente secuencia:

- Una rotación entre el hijo y el nieto de  $X$ .
- Una rotación entre  $X$  y su nuevo hijo.

Una rotación doble es equivalente a dos rotaciones simples.

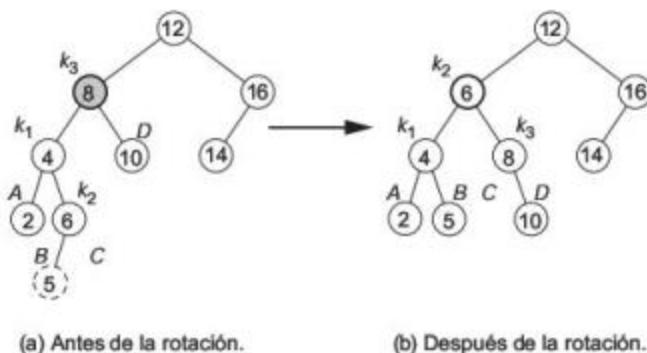


Figura 19.30 Una rotación doble corrige el árbol AVL después de la inserción de 5.

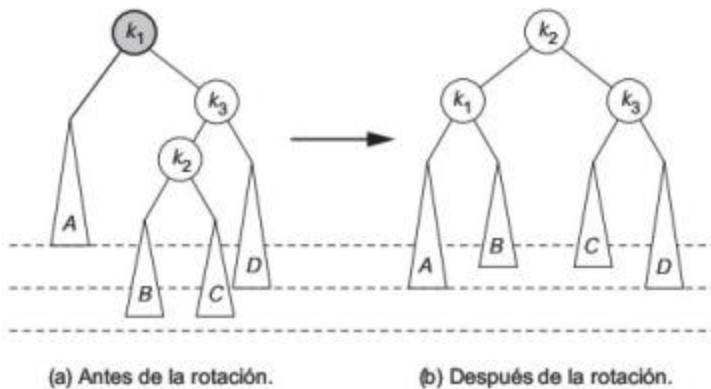


Figura 19.31 Rotación doble derecha-izquierda para corregir el caso 3.

El pseudocódigo para implementar la rotación doble para el caso 2 es compacto y se muestra en la Figura 19.32. El pseudocódigo simétrico para el caso 3 se muestra en la Figura 19.33.

```

1  /**
2   * Rotación doble de un nodo de un árbol binario: primero hijo izquierdo
3   * con su hijo derecho; después el nodo k3 son su nuevo hijo izquierdo.
4   * Para árboles AVL, esta es una rotación doble para el caso 2.
5   */
6  static BinaryNode doubleRotateWithLeftChild( BinaryNode k3 )
7  {
8      k3.left = rotateWithRightChild( k3.left );
9      return rotateWithLeftChild( k3 );
10 }

```

Figura 19.32 Pseudocódigo para una rotación doble (caso 2).

```

1  /**
2   * Rotación doble de un nodo de un árbol binario: primero hijo derecho
3   * con su hijo izquierdo; después el nodo k1 son su nuevo hijo derecho.
4   * Para árboles AVL, esta es una rotación doble para el caso 3.
5   */
6  static BinaryNode doubleRotateWithRightChild( BinaryNode k1 )
7  {
8      k1.right = rotateWithLeftChild( k1.right );
9      return rotateWithRightChild( k1 );
10 }

```

Figura 19.33 Pesudocódigo para una rotación doble (caso 3).

#### 19.4.4 Resumen de la inserción AVL

Una implementación AVL descuidada no resulta excesivamente compleja, pero no es eficiente. Desde entonces se han descubierto otros árboles de búsqueda equilibrados más convenientes, así que no merece la pena implementar un árbol AVL.

A continuación proporcionamos un breve resumen de cómo se implementa una inserción AVL. El método más simple de implementar una inserción AVL es un algoritmo recursivo. Para insertar un nuevo nodo con clave  $X$  en un árbol AVL  $T$ , lo insertamos recursivamente en el subárbol apropiado de  $T$  (que designaremos como  $T_{LR}$ ). Si la altura de  $T_{LR}$  no cambia, habremos terminado. En caso contrario, si aparece un desequilibrio de alturas en  $T$ , realizamos la rotación simple o doble que sea apropiada (con raíz en  $T$ ), dependiendo de  $X$  y de las claves en  $T$  y  $T_{LR}$ , con lo que habremos terminado (porque la altura anterior es igual a la altura después de la rotación). Podríamos describir este algoritmo recursivo como una *implementación descuidada*.

Por ejemplo, en cada nodo comparamos las alturas de los subárboles. Sin embargo, en general, almacenar el resultado de la comparación en el nodo es más eficiente que mantener la información de altura. Esta técnica permite evitar el cálculo repetitivo de los factores de equilibrio. Además, la recursión tiene un coste adicional sustancialmente mayor que una versión iterativa. La razón es que, en la práctica, bajamos por el árbol y luego deshacemos completamente el camino, en lugar de detenernos en cuanto se ha ejecutado una rotación. En consecuencia, en la práctica, se utilizan otros esquemas para árboles de búsqueda equilibrados.

## 19.5 Árboles rojo-negro

Un árbol rojo-negro es una buena alternativa al árbol AVL. Los detalles de codificación tienden a proporcionar una implementación más rápida, porque se puede utilizar una única pasada arriba-abajo durante las rutinas de inserción y borrado.

Una alternativa históricamente popular al árbol AVL es el *árbol rojo-negro*, en el que se puede utilizar una única pasada arriba-abajo durante las rutinas de inserción y borrado. Este enfoque contrasta con el del árbol AVL, en el que se utiliza una pasada hacia abajo del árbol para determinar el punto de inserción y una pasada hacia arriba para actualizar las alturas y posiblemente re-equilibrar el árbol. Como resultado, una cuidadosa implementación no recursiva del árbol rojo-negro es más simple y rápida que una implementación de un árbol AVL. Como en los árboles AVL, las operaciones con los árboles rojo-negro requieren un tiempo logarítmico de caso peor.

Un árbol rojo-negro es un árbol de búsqueda binaria que tiene las siguientes propiedades de ordenación:

1. Todo nodo está coloreado de rojo o de negro.
2. La raíz es negra.
3. Si un nodo es rojo, sus hijos deben ser negros.
4. Todo camino desde un nodo hasta un enlace  $\text{null}$  debe contener el mismo número de nodos negros.

En este análisis de los árboles rojo-negro, los nodos sombreados representan nodos rojos. La Figura 19.34 muestra un árbol rojo-negro. Todo camino desde la raíz hasta un nodo  $\text{null}$  contiene tres nodos negros.

Podemos demostrar por inducción que, si todo camino desde la raíz hasta un nodo  $\text{null}$  contiene  $B$  nodos negros, el árbol debe contener al menos  $2^B - 1$  nodos negros. Además, como la raíz es negra y no puede haber dos nodos consecutivos rojos en un camino, la altura de un árbol rojo-negro es como máximo  $2 \log(N + 1)$ . En consecuencia, se garantiza que la operación de búsqueda sea una operación logarítmica.

La dificultad, como de costumbre, es que las operaciones pueden modificar el árbol y posiblemente destruir sus propiedades de coloreado. Esta posibilidad hace que la inserción sea difícil y el borrado todavía más. En primer lugar, vamos a implementar la inserción y luego examinaremos el algoritmo de borrado.

Está prohibido que haya nodos rojos consecutivos, y todos los caminos tienen el mismo número de nodos negros.

Los nodos sombreados son rojos a lo largo de todo este análisis.

Está garantizado que la profundidad de un árbol rojo-negro sea logarítmica. Tipicamente, la profundidad es la misma que para un árbol AVL.

### 19.5.1 Inserción abajo-arriba

Recuerde que todo nuevo elemento se inserta siempre como una hoja del árbol. Si coloreáramos un nuevo elemento de negro, violaríamos la propiedad 4, porque crearíamos un camino más largo de nodos negros. Por tanto, el nuevo elemento deberá colorearse en rojo. Si el padre es negro, habremos terminado; por tanto, la inserción del valor 25 en el árbol mostrado en la Figura 19.34 es trivial. Si el padre ya es rojo, violaríamos la propiedad 3 al tener nodos rojos consecutivos. En este caso, tenemos que ajustar el árbol para garantizar el cumplimiento de la propiedad 3, y además tenemos que hacerlo sin que se viole la propiedad 4. Las operaciones básicas utilizadas son los cambios de color y las rotaciones de árbol.

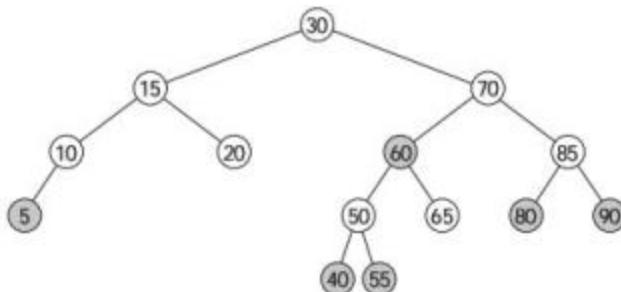


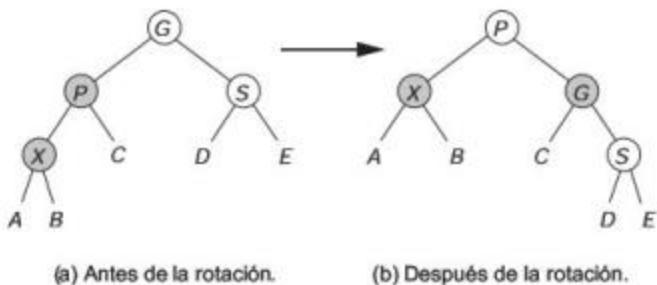
Figura 19.34 Árbol rojo-negro: la secuencia de inserción es 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5 y 55 (los nodos sombreados son rojos).

Los nuevos elementos se deben colorear de rojo. Si el padre fuera rojo deberemos volver a colorear y/o rotar para eliminar la existencia de nodos rojos consecutivos.

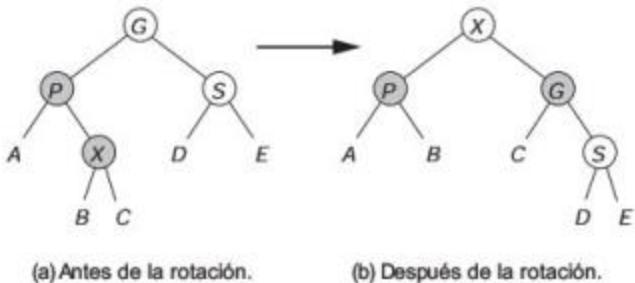
Si el hermano del padre es negro, una rotación simple o doble permite corregir las cosas, como en un árbol AVL.

Tenemos que considerar varios casos (cada uno con sus correspondientes simetrías) si el padre es rojo. En primer lugar, suponga que el hermano del padre es negro (adoptamos el convenio de que los nodos  $\text{n} \geq 11$  son negros), lo que se aplicaría a la inserción de los valores 3 u 8, pero no a la inserción de 99. Sea  $X$  la nueva hoja añadida, sea  $P$  su padre, sea  $S$  el hermano del padre (si existe) y sea  $G$  el abuelo. Solo  $X$  y  $P$  son rojos en este caso;  $G$  es negro porque en caso contrario habría habido dos nodos rojos consecutivos *antes* de la inserción –una violación de la propiedad 3. Adoptando la terminología de los árboles AVL, diremos que  $X$  puede ser, en relación con  $G$ , un nodo exterior o interior.<sup>2</sup> Si  $X$  es un nieto exterior, una rotación simple de su padre y de su abuelo y una serie de cambios de color permitirán restaurar la propiedad 3. Si  $X$  es un nieto interior, será necesaria una doble rotación junto con algunos cambios de color. La rotación simple se muestra en la Figura 19.35 y la rotación doble se ilustra en la Figura 19.36. Aun cuando  $X$  sea una hoja, hemos dibujado un caso más general que permite que  $X$  se encuentre en mitad del árbol. Utilizaremos esta rotación más general posteriormente en el algoritmo.

Antes de continuar, veamos por qué estas rotaciones son correctas. Necesitamos asegurarnos de que nunca haya dos nodos rojos consecutivos. Por ejemplo, como se muestra en la Figura 19.36, los



**Figura 19.35** Si  $S$  es negro, una rotación simple entre el padre y el abuelo, junto con los cambios apropiados de color permitirá restaurar la propiedad 3, en caso de que  $X$  sea un nieto exterior.



**Figura 19.36** Si  $S$  es negro, una rotación doble que afecte a  $X$ , al padre y al abuelo, junto con los apropiados cambios de color permitirá restaurar la propiedad 3, en caso de que  $X$  sea un nieto interior.

<sup>2</sup> Véase la Sección 19.4.1 en la página 696.

únicos casos posibles de nodos rojos consecutivos serían entre  $P$  y uno de sus hijos o entre  $G$  y  $C$ . Pero las raíces de  $A$ ,  $B$  y  $C$  deben ser negras; en caso contrario, habría habido violaciones adicionales de la propiedad 3 en el árbol original. En el árbol original, hay un nodo negro en el camino que va desde la raíz del subárbol hasta  $A$ ,  $B$  y  $C$ , y dos nodos negros en los caminos que van hasta  $D$  y  $E$ . Podemos verificar que este patrón se sigue manteniendo después de la rotación y el recolorado.

Hasta ahora, todo bien. ¿Pero qué sucede si  $S$  es rojo cuando intentamos insertar el valor 79 en el árbol de la Figura 19.34? Entonces, ni la rotación simple ni la doble funcionan, porque ambas nos dan como resultado nodos rojos consecutivos. De hecho, en este caso, debe haber tres nodos en el camino que va hasta  $D$  y  $E$ , y solo uno de ellos puede ser negro. Por consiguiente, tanto  $S$  como la nueva raíz del subárbol deberán colorearse de rojo. Por ejemplo, el caso de rotación simple que se produce cuando  $X$  es un nieto exterior se muestra en la Figura 19.37. Aunque esta rotación parece funcionar, hay un problema: ¿qué pasa si el padre de la raíz del subárbol (es decir, el bisabuelo original de  $X$ ) es también rojo? Podríamos ir replicando este procedimiento hacia arriba hasta la raíz, hasta que ya no hubiera dos nodos rojos consecutivos o alcanzáramos la raíz (que se cambiara al color negro). Pero entonces estaríamos volviendo a hacer una pasada hacia arriba en el árbol, como en el árbol AVL.

Si el hermano del padre es rojo entonces, después de corregir las cosas, incluimos nodos rojos consecutivos a un nivel superior. Necesitamos iterar hacia arriba en el árbol para corregir la situación.

### 19.5.2 Árboles rojo-negro arriba-abajo

Para evitar la posibilidad de tener que ir rotando hacia arriba en el árbol, aplicamos un procedimiento arriba-abajo a medida que buscamos el punto de inserción. Específicamente, se garantiza que, cuando lleguemos a una hoja e insertemos un nodo,  $S$  no sea rojo. Entonces, podemos limitarnos a añadir una hoja roja y, en caso necesario, a utilizar una rotación (simple o doble). El procedimiento es conceptualmente sencillo.

En el camino de bajada, cuando nos encontramos con un nodo  $X$  que tiene dos hijos rojos, hacemos que  $X$  sea rojo y sus dos hijos negros. La Figura 19.38 muestra este cambio de color. (Si  $X$  es la raíz, será transformada en roja por este proceso. Después, podemos volver a colorearla de negro, sin violar ninguna de las propiedades de los árboles rojo-negro.) El número de nodos negros en los caminos situados por debajo de  $X$  no se verá

Para evitar tener que iterar de nuevo hacia arriba en el árbol, nos aseguramos, a medida que descendemos por el árbol, de que el padre del hermano no sea rojo. Podemos hacer esto mediante cambios de color y/o rotaciones.

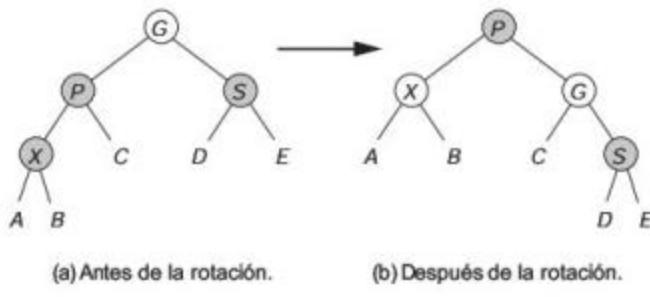


Figura 19.37 Si  $S$  es rojo, una rotación simple entre el padre y el abuelo, junto con los cambios apropiados de color permitirá restaurar la propiedad 3 entre  $X$  y  $P$ .

modificado. Sin embargo, si el padre de  $X$  es rojo, introduciríamos dos nodos rojos consecutivos. Pero en este caso, podemos aplicar la rotación simple de la Figura 19.35 o la rotación doble de la Figura 19.36. ¿Pero qué sucede si el hermano del padre de  $X$  es también rojo? *Esta situación no puede suceder.* Si en el camino de bajada por el árbol vemos un nodo  $Y$  que tiene dos hijos rojos, sabemos que el nieto de  $Y$  tiene que ser negro. Y como los hijos de  $Y$  también se transforman en negros debido al cambio de color (incluso después de la rotación que pueda tener lugar) no nos encontraremos con otro nodo rojo durante dos niveles. Por tanto, cuando nos encontramos con  $X$ , si el padre de  $X$  es rojo, el hermano del padre de  $X$  no puede ser también rojo.

Por ejemplo, suponga que queremos insertar el valor 45 en el árbol mostrado en la Figura 19.34. En el camino de bajada por el árbol, nos encontramos con el nodo 50, que tiene dos hijos rojos. Por tanto, realizamos un cambio de color, haciendo que 50 pase a ser rojo y que 40 y 55 pasen a ser negros. El resultado se muestra en la Figura 19.39. Sin embargo, ahora tanto 50 como 60 son rojos. Realizamos una rotación simple (porque 50 es un nodo exterior) entre 60 y 70, haciendo así que 60 sea la raíz negra del subárbol derecho de 30 y haciendo que 70 sea rojo, como se muestra en la Figura 19.40. Después continuamos realizando una acción idéntica si vemos en el camino otros nodos que tengan dos hijos rojos. En este caso sucede que no hay ninguno más.

Al llegar a la hoja, insertamos 45 como nodo rojo, y como el padre es negro, habremos terminado. El árbol resultante se muestra en la Figura 19.41. Si el padre hubiera sido rojo, habríamos tenido que efectuar una rotación.

Como muestra la Figura 19.41, el árbol rojo-negro resultante suele estar bien equilibrado. Los experimentos sugieren que el número de nodos recorridos durante una búsqueda típica en un árbol rojo-negro es casi idéntico al promedio correspondiente a los árboles AVL, aunque las propiedades de equilibrio del árbol rojo-negro son ligeramente más débiles. La ventaja de un árbol rojo-negro es el gasto adicional relativamente pequeño que se requiere para realizar una inserción y el hecho de que, en la práctica, las rotaciones se producen de forma relativamente infrecuente.

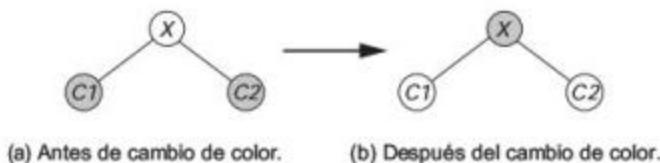


Figura 19.38 Cambio de color: solo si el padre de  $X$  es rojo continuamos con una rotación.

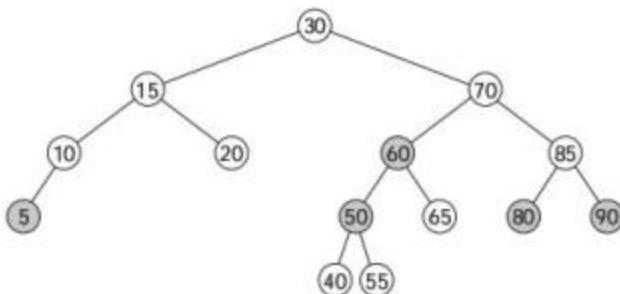


Figura 19.39 Un cambio de color en 50 provocaría una violación; puesto que la violación es exterior, una rotación simple permite corregirla.

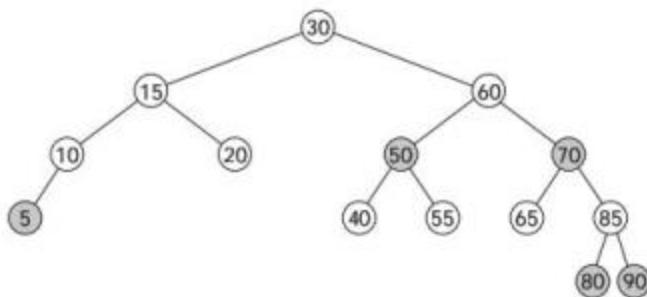


Figura 19.40 Resultado de una rotación simple que corrige la violación en el nodo 50.

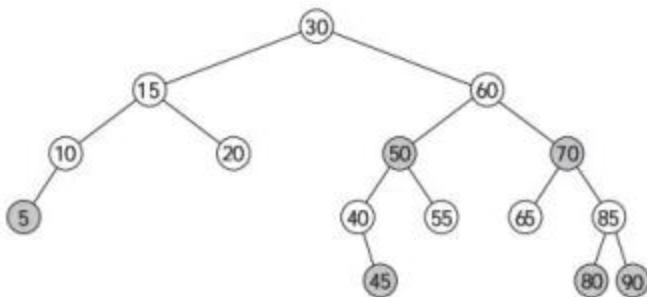


Figura 19.41 Inserción de 45 como un nodo rojo.

### 19.5.3 Implementación Java

Una implementación real es complicada, no solo por las muchas rotaciones posibles, sino también por la posibilidad de que algunos subárboles (como el subárbol derecho del nodo que contiene el valor 10 en la Figura 19.41) pueden estar vacíos y por el caso especial que representa la raíz (que, entre otras cosas, no tiene padre). Para eliminar los casos especiales, empleamos dos tipos de nodos centinela.

- Utilizamos `nullNode` en lugar de un enlace `null`; `nullNode` estará siempre coloreado en negro.
- Utilizamos `header` como pseudoráiz; tiene un valor de clave de  $-\infty$  y un enlace derecho a la raíz real.

Eliminaremos los casos especiales utilizando un centinela para el nodo `null` y una pseudoráiz. Hacer esto requiere pequeñas modificaciones en casi todas las rutinas.

Por tanto, incluso las rutinas básicas como `isEmpty` tendrán que ser modificadas. En consecuencia, heredar de `BinarySearchTree` no tiene sentido, así que lo que hacemos es escribir la clase partiendo de cero. La clase `RedBlackNode`, que está anidada en `RedBlackTree`, se muestra en la Figura 19.42 y es sencilla. El esqueleto de la clase `RedBlackTree` se muestra en la Figura 19.43. Las líneas 55 y 56 declaran los centinelas de los que hemos hablado anteriormente. En la rutina `insert` se emplean cuatro referencias –`current`, `parent`, `grand` y `great`– para hacer referencia al nodo actual, al padre, al abuelo y al bisabuelo. Su situación en las

En el camino hacia abajo, mantenemos referencias al nodo actual, al padre, al abuelo y al bisabuelo.

```

1  private static class RedBlackNode<AnyType>
2  {
3      // Constructores
4      RedBlackNode( AnyType theElement )
5      {
6          this( theElement, null, null );
7      }
8
9      RedBlackNode( AnyType theElement, RedBlackNode<AnyType> lt,
10                  RedBlackNode<AnyType> rt )
11     {
12         element = theElement;
13         left = lt;
14         right = rt;
15         color = RedBlackTree.BLACK;
16     }
17
18     AnyType element;           // Los datos del nodo
19     RedBlackNode<AnyType> left;    // Hijo izquierdo
20     RedBlackNode<AnyType> right;   // Hijo derecho
21     int color;                // Color
22 }

```

Figura 19.42 La clase RedBlackNode.

líneas 62 a 65 permite que sean compartidas por `insert` y la rutina `handleReorient`. El método `remove` no está implementado.

Las rutinas restantes son similares a las rutinas correspondientes de `BinarySearchTree`, salvo porque tienen diferentes implementaciones, debido a los nodos centinela. Podríamos proporcionar al constructor el valor  $-\infty$ , para inicializar el nodo de cabecera, pero no hacemos eso. La alternativa es utilizar el método `compare`, definido en las líneas 38 y 39, cuando sea apropiado. En la Figura 19.44 se muestra un constructor. El constructor asigna `nullNode` y luego la cabecera y configura los enlaces `left` y `right` de la cabecera con el valor `nullNode`.

La Figura 19.45 muestra el cambio más simple que resulta del uso de los nodos centinela. La comprobación con respecto al valor `null` necesita ser sustituida por una comprobación con respecto a `nullNode`.

Para la rutina `find` mostrada en la Figura 19.46, utilizamos un truco común. Antes de iniciar la búsqueda, colocamos `x` en el centinela `nullNode`. De esa forma se garantiza que terminaremos por encontrar una correspondencia con `x`, incluso si `x` no está en el árbol. Si la correspondencia se produce en `nullNode`, podemos decir que el elemento no se ha encontrado. Empleamos este truco en el procedimiento `insert`.

El método `insert` se deduce directamente de nuestra descripción y se muestra en la Figura 19.47. El bucle `while` que abarca las líneas 11 a 20

Las comparaciones con `null` se sustituyen por comparaciones con `nullNode`.

Al realizar una operación `find`, copiamos `x` en el centinela `nullNode` para evitar comprobaciones adicionales.

```
1 package weiss.nonstandard;
2
3 // Clase RedBlackTree
4 //
5 // CONSTRUCCIÓN: sin parámetros
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // Igual que BinarySearchTree; omitido por brevedad
9 // *****ERRORES*****
10 // remove e insert generan excepciones en caso necesario.
11
12 public class RedBlackTree<AnyType extends Comparable<? super AnyType>>
13 {
14     public RedBlackTree( )
15         { /* Figura 19.44 */ }
16
17     public void insert( AnyType item )
18         { /* Figura 19.47 */ }
19     public void remove( AnyType x )
20         { /* No implementado */ }
21
22     public AnyType findMin( )
23         { /* Véase el código en línea */ }
24     public AnyType findMax( )
25         { /* Similar a findMin */ }
26     public AnyType find( AnyType x )
27         { /* Figura 19.46 */ }
28
29     public void makeEmpty( )
30         { header.right = nullNode; }
31     public boolean isEmpty( )
32         { return header.right == nullNode; }
33     public void printTree( )
34         { printTree( header.right ); }
35
36     private void printTree( RedBlackNode<AnyType> t )
37         { /* Figura 19.45 */ }
38     private final int compare( AnyType item, RedBlackNode<AnyType> t )
39         { /* Figura 19.47 */ }
40     private void handleReorient( AnyType item )
41         { /* Figura 19.48 */ }
```

*Continúa*

Figura 19.43 El esqueleto de la clase RedBlackTree.

```

42     private RedBlackNode<AnyType>
43         rotate( AnyType item, RedBlackNode<AnyType> parent )
44     { /* Figura 19.49 */ }
45
46     private static <AnyType>
47     RedBlackNode<AnyType> rotateWithLeftChild( RedBlackNode<AnyType> k2 )
48         { /* Implementación de la forma usual; ver código en línea */ }
49     private static <AnyType>
50     RedBlackNode<AnyType> rotateWithRightChild( RedBlackNode<AnyType> k1 )
51         { /* Implementación de la forma usual; ver código en línea */ }
52     private static class RedBlackNode<AnyType>
53         { /* Figura 19.42 */ }
54
55     private RedBlackNode<AnyType> header;
56     private RedBlackNode<AnyType> nullNode;
57
58     private static final int BLACK = 1; // BLACK debe ser 1
59     private static final int RED = 0;
60
61     // Usados en la rutina insert y sus rutinas auxiliares
62     private RedBlackNode<AnyType> current;
63     private RedBlackNode<AnyType> parent;
64     private RedBlackNode<AnyType> grand;
65     private RedBlackNode<AnyType> great;
66 }

```

---

**Figura 19.43 (Continuación).**

```

1  /**
2  * Construir el árbol.
3  */
4  public RedBlackTree( )
5  {
6      nullNode = new RedBlackNode<AnyType>( null );
7      nullNode.left = nullNode.right = nullNode;
8      header = new RedBlackNode<AnyType>( null );
9      header.left = header.right = nullNode;
10 }

```

---

**Figura 19.44** El constructor de RedBlackTree.

va recorriendo el árbol hacia abajo y corrige los nodos que tengan dos hijos rojos, invocando `handleReorient`, como se muestra en la Figura 19.48. Para hacer esto, no solo mantiene el nodo

```
1  /**
2   * Método interno para imprimir un subárbol en orden.
3   * @param t el nodo que actúa como raíz del árbol.
4   */
5  private void printTree( RedBlackNode<AnyType> t )
6  {
7      if( t != nullNode )
8      {
9          printTree( t.left );
10         System.out.println( t.element );
11         printTree( t.right );
12     }
13 }
```

Figura 19.45 El método printTree para la clase RedBlackTree.

```
1  /**
2   * Encontrar un elemento en el árbol.
3   * @param x el elemento que hay que buscar.
4   * @return el elemento correspondiente o null si no se encuentra.
5   */
6  public AnyType find( AnyType x )
7  {
8      nullNode.element = x;
9      current = header.right;
10
11     for( ; ; )
12     {
13         if( x.compareTo( current.element ) < 0 )
14             current = current.left;
15         else if( x.compareTo( current.element ) > 0 )
16             current = current.right;
17         else if( current != nullNode )
18             return current.element;
19         else
20             return null;
21     }
22 }
```

Figura 19.46 La rutina find de RedBlackTree. Observe el uso de header y nullNode.

actual, sino también el padre, el abuelo y el bisabuelo. Observe que, después de una rotación, los valores almacenados en el abuelo y el bisabuelo ya no son correctos. Sin embargo, serán restaurados

```
1  /**
2   * Insertar en el árbol.
3   * @param item el elemento que hay que insertar.
4   * @throws DuplicateItemException si el elemento ya está presente.
5   */
6  public void insert( AnyType item )
7  {
8      current = parent = grand = header;
9      nullNode.element = item;
10
11     while( compare( item, current ) != 0 )
12     {
13         great = grand; grand = parent; parent = current;
14         current = compare( item, current ) < 0 ?
15             current.left : current.right;
16
17         // Comprobar si hay dos hijos rojos, corregir en caso afirmativo
18         if( current.left.color == RED && current.right.color == RED )
19             handleReorient( item );
20     }
21
22     // La inserción falla si ya está presente
23     if( current != nullNode )
24         throw new DuplicateItemException( item.toString() );
25     current = new RedBlackNode<AnyType>( item, nullNode, nullNode );
26
27     // Asociar con el parent
28     if( compare( item, parent ) < 0 )
29         parent.left = current;
30     else
31         parent.right = current;
32     handleReorient( item );
33 }
34
35 /**
36  * Comparar item y t.element, utilizando compareTo, teniendo en cuenta
37  * que si t es el nodo cabecera, entonces item es siempre mayor.
38  * Esta rutina se invoca si es posible que t sea la cabecera.
39  * Si no es posible que t sea la cabecera, usar compareTo directamente.
40  */
41 private final int compare( AnyType item, RedBlackNode<AnyType> t )
42 {
43     if( t == header )
44         return 1;
45     else
46         return item.compareTo( t.element );
47 }
```

---

Figura 19.47 Las rutinas insert y compare para la clase RedBlackTree.

```

1  /**
2  * Rutina interna que se invoca durante una inserción si un nodo
3  * tiene dos hijos rojos. Realiza cambios de color y rotaciones.
4  * @param item el elemento que se está insertando.
5  */
6  private void handleReorient( AnyType item )
7  {
8      // Hacer el cambio de color
9      current.color = RED;
10     current.left.color = BLACK;
11     current.right.color = BLACK;
12
13     if( parent.color == RED )           // Hay que rotar
14     {
15         grand.color = RED;
16         if( ( compare( item, grand ) < 0 ) != ( compare( item, parent ) < 0 ) )
17             parent = rotate( item, grand ); // Iniciar rotación doble
18             current = rotate( item, great );
19
20         current.color = BLACK;
21     }
22     header.right.color = BLACK;        // Hacer que la raíz sea negra
23 }

```

Figura 19.48 La rutina `handleReorient`, que se invoca si un nodo tiene dos hijos rojos o cuando se inserta un nuevo nodo.

antes del momento en que se los vuelva a necesitar. Cuando el bucle termina, o bien se ha encontrado `x` (como indica `current!=nullNode`) o bien no se ha encontrado `x` (como indica `current==nullNode`). Si no se ha encontrado `x`, generamos una excepción en la línea 24. En caso contrario, `x` no se encuentra en el árbol y habrá que hacer que sea un hijo de `parent`. Asignamos un nuevo nodo (como nuevo nodo `current`), lo asociamos al padre y llamamos a `handleReorient` en las líneas 25 a 32.

En las líneas 11 y 14 vemos la llamada a `compare`, que se utiliza porque la cabecera podía ser uno de los nodos implicados en la comparación. El valor en la cabecera es  $-\infty$  desde el punto de vista lógico, pero en la práctica es `null`. La implementación de `compare` garantiza que el valor de la cabecera se considere siempre menor que cualquier otro valor. `compare` también se muestra en la Figura 19.47.

El código utilizado para llevar a cabo una rotación simple se muestra en el método `rotate` de la Figura 19.49. Puesto que el árbol resultante debe estar conectado a un parent, `rotate` toma el nodo parent como parámetro. En lugar de llevar la cuenta del tipo de rotación (izquierda o derecha) a medida que descendemos por el árbol, lo que hacemos es pasar `item` como parámetro. Esperamos que haya muy pocas rotaciones durante la inserción, por lo que hacer las cosas de esta forma no solo es más simple sino también más rápido.

El código es relativamente compacto para la serie de casos que existen, y más si tenemos en cuenta el hecho de que la implementación no es recursiva. Por estas razones, el árbol rojo-negro ofrece un buen rendimiento.

El método `rotate` tiene cuatro posibilidades. El operador ?: hace el código más compacto, pero es lógicamente equivalente a una comprobación `if / else`.

```

1  /**
2   * Rutina interna que realiza una rotación simple o doble.
3   * Puesto que el resultado se conecta al padre, hay 4 casos.
4   * Invocado por handleReorient.
5   * @param item el elemento en handleReorient.
6   * @param parent el parent de la raíz del subárbol rotado.
7   * @return la raíz del subárbol rotado.
8   */
9  private RedBlackNode<AnyType>
10 rotate( AnyType item, RedBlackNode<AnyType> parent )
11 {
12     if( compare( item, parent ) < 0 )
13         return parent.left = compare( item, parent.left ) < 0 ?
14             rotateWithLeftChild( parent.left ) : // LL
15             rotateWithRightChild( parent.left ) : // LR
16     else
17         return parent.right = compare( item, parent.right ) < 0 ?
18             rotateWithLeftChild( parent.right ) : // RL
19             rotateWithRightChild( parent.right ); // RR
20 }

```

Figura 19.49 Una rutina para realizar una rotación apropiada.

La rutina `handleReorient` invoca a `rotate` de la manera necesaria para llevar a cabo una rotación simple o doble. Puesto que una rotación doble no es otra cosa que dos rotaciones simples, podemos comprobar si estamos en un caso interior y, en caso afirmativo, realizar una rotación extra entre el nodo actual y su parent (pasándole el abuelo a `rotate`). En cualquiera de los casos efectuamos una rotación entre el parent y el abuelo (pasando el bisabuelo a `rotate`). Esta acción está codificada de forma sucinta en las líneas 16 y 17 de la Figura 19.48.

#### 19.5.4 Borrado arriba-abajo

el borrado en los árboles rojo-negro también se puede realizar con la técnica arriba-abajo. No hace falta decir que una implementación real es bastante complicada porque, para empezar, el algoritmo `remove` para árboles de búsqueda desequilibrados no es nada trivial. El algoritmo normal de borrado en árboles de búsqueda binaria elimina nodos que sean hojas o que tengan un hijo. Recuerde que los nodos con dos hijos nunca se eliminan, lo que se hace es sustituir su contenido.

Si el nodo que hay que borrar es rojo, no hay ningún problema. Sin embargo, si el nodo que hay que borrar es negro, su eliminación violaría la propiedad 4. La solución al problema consiste en garantizar que ningún nodo que vayamos a borrar sea rojo.

El borrado es bastante complejo. La idea básica consiste en garantizar que el nodo borrado sea rojo.

A lo largo de esta explicación, llamaremos  $X$  al nodo actual,  $T$  a su hermano y  $P$  a su parent. Comenzamos coloreando la raíz centinela como roja. A medida que recorremos el árbol, tratamos de garantizar que  $X$  sea rojo. Cuando lleguemos a un nodo nuevo, estaremos seguros de que  $P$  es

rojo (inductivamente, por el invariante que estamos tratando de mantener) y que  $X$  y  $T$  son negros (porque no podemos tener dos nodos rojos consecutivos). Hay dos casos principales, junto con las variantes simétricas usuales (que aquí omitiremos).

En primer lugar, suponga que  $X$  tiene dos hijos negros. Hay tres subcasos, que dependen de los hijos de  $T$ :

1.  $T$  tiene dos hijos negros: cambiamos los colores (Figura 19.50).
2.  $T$  tiene un hijo rojo exterior: realizamos una rotación simple (Figura 19.51).
3.  $T$  tiene un hijo rojo interior: realizamos una rotación doble (Figura 19.52).

El examen de las rotaciones muestra que si  $T$  tiene dos hijos rojos, una rotación simple o una doble permitirá conseguir nuestro objetivo (por lo que tiene sentido realizar solo una rotación simple). Observe que, si  $X$  es una hoja, sus dos hijos son negros, por lo que siempre podemos aplicar uno de estos tres mecanismos para hacer que  $X$  sea rojo.

En segundo lugar, suponga que uno de los hijos de  $X$  es rojo. Puesto que las rotaciones en el primer caso principal siempre colorean  $X$  de rojo, si  $X$  tiene un hijo rojo, se introducirían nodos rojos consecutivos. Por tanto, necesitamos una solución alternativa. En este caso, pasamos al siguiente

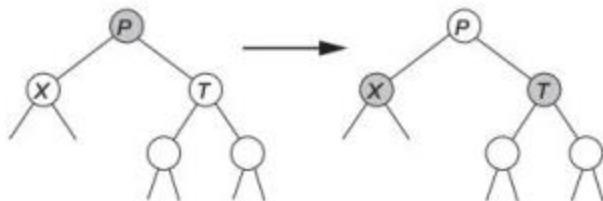


Figura 19.50  $X$  tiene dos hijos negros y los dos hijos de su hermano son también negros.

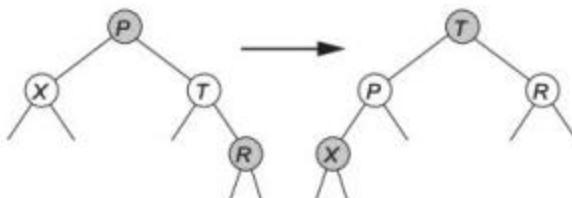


Figura 19.51  $X$  tiene dos hijos negros y el hijo exterior de su hermano es rojo; hacemos una rotación simple.

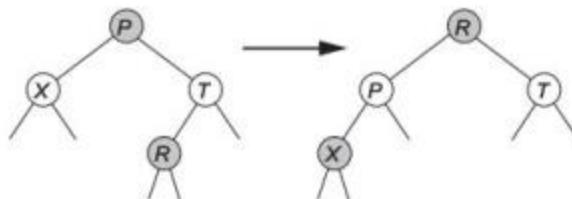


Figura 19.52  $X$  tiene dos hijos negros y el hijo interior de su hermano es rojo; hacemos una rotación doble.

nivel, obteniendo unos nuevos  $X$ ,  $T$  y  $P$ . Si tenemos suerte, nos toparemos con un nodo rojo (tenemos al menos un 50 por ciento de posibilidades de que esto suceda), haciendo así que el nodo actual sea rojo. En caso contrario, tenemos la situación mostrada en la Figura 19.53; es decir, que el  $X$ actual es negro, el  $T$ actual es rojo y el  $P$ actual es negro. Podemos entonces rotar  $T$  y  $P$ , haciendo así que el nuevo padre de  $X$  sea rojo;  $X$  y su nuevo abuelo serán negros. Ahora  $X$  todavía no es rojo, pero ya estamos de nuevo en el punto de partida (aunque a un nivel más de profundidad). Este resultado es lo suficientemente bueno, porque demuestra que podemos descender iterativamente por el árbol. Así, mientras que terminemos por alcanzar un nodo que tenga dos hijos negros o nos topemos con un nodo rojo, no habrá problema. Este resultado está garantizado para el algoritmo de borrado, porque los dos posibles estados finales son

- $X$  es una hoja, lo cual siempre será tratado por el caso principal, ya que  $X$  tiene dos hijos negros.
- $X$  tiene solo un hijo, en cuyo caso se aplica el caso principal si el hijo es negro y, si el hijo es rojo, podemos borrar  $X$ , en caso necesario, y hacer que el hijo sea negro.

*El borrado perezoso*  
consiste en marcar los  
elementos como borrados.

En ocasiones se utiliza la técnica del *borrado perezoso*, en el que los elementos se marcan como borrados pero sin llegar a borrarlos realmente. Sin embargo, el borrado perezoso desperdicia espacio y complica otras rutinas (véase el Ejercicio 19.18).

## 19.6 Árboles AA

El árbol AA es el método preferido cuando hace falta un árbol equilibrado, cuando una implementación descuidada es aceptable y cuando hacen falta borrados.

Debido a las muchas posibles rotaciones, el árbol rojo-negro es bastante complicado de codificar. En particular, la operación *remove* plantea un desafío considerable. En esta sección vamos a describir un árbol de búsqueda equilibrado simple, pero bastante útil, conocido con el nombre de *árbol AA*. El árbol AA es el método preferido cuando hace falta un árbol equilibrado, cuando resulta aceptable una implementación descuidada y cuando son necesarios borrados. El árbol AA añade una condición adicional al árbol rojo-negro: los hijos izquierdos no pueden ser rojos.

Esta sencilla restricción simplifica enormemente los algoritmos de los árboles rojo-negro por dos razones: en primer lugar, elimina aproximadamente la mitad de los casos de reestructuración del árbol; en segundo lugar, simplifica el algoritmo *remove* al eliminar un caso bastante molesto; es decir, si un nodo interno tiene solo un hijo, el hijo debe ser un hijo derecho rojo, porque los hijos

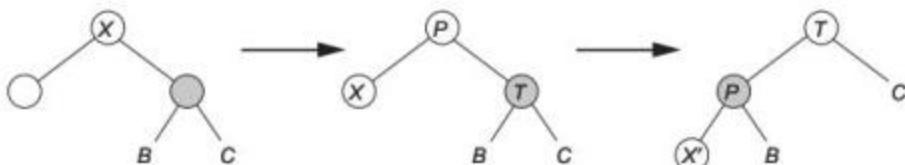


Figura 19.53  $X$  es negro y al menos uno de sus hijos es rojo; si pasamos al siguiente nivel y aterrizamos en un hijo rojo, no hay problema; si no es así, rotamos el hermano y el padre.

izquierdos rojos son ahora ilegales, mientras que un único hijo negro violaría la propiedad 4 de los árboles rojo-negro. Por tanto, siempre podemos sustituir un nodo interno por el nodo más pequeño de su subárbol derecho. Ese nodo más pequeño, o bien es una hoja, o tiene un hijo rojo y puede ser fácilmente soslayado y eliminado.

Para simplificar la implementación todavía más, representamos la información de equilibrio de una forma más directa. En lugar de almacenar un color con cada nodo, almacenamos el nivel del nodo. El *nivel de un nodo* representa el número de enlaces izquierdos en el camino que va hasta el nodo centinela `nullNode` y es

- Nivel 1, si el nodo es una hoja.
- El nivel de su padre, si el nodo es rojo.
- Uno menos que el nivel de su padre, si el nodo es negro.

El nivel de un nodo en un árbol AA representa el número de enlaces izquierdos en el camino que va hasta el nodo centinela `nullNode`.

El resultado es un árbol AA. Si traducimos el requisito estructural de colores a niveles, sabemos que el hijo izquierdo debe ser un nivel inferior que su padre y que el hijo derecho puede ser cero o un nivel inferior a su padre (pero no más). Un *enlace horizontal* es una conexión entre un nodo y un hijo que tenga el mismo nivel que él. Las propiedades de coloreado implican

1. Los enlaces horizontales son enlaces derechos (porque solo los hijos derechos pueden ser rojos).
2. No puede haber dos enlaces horizontales consecutivos (porque no puede haber nodos rojos consecutivos).
3. Los nodos de nivel 2 o superior deben tener dos hijos.
4. Si un nodo no tiene un enlace horizontal derecho, sus dos hijos tienen el mismo nivel.

Un enlace horizontal en un árbol AA es una conexión entre un nodo y un hijo suyo que tenga el mismo nivel. Un enlace horizontal solo puede ir hacia la derecha y no debería haber dos enlaces horizontales consecutivos.

La Figura 19.54 muestra un árbol AA de ejemplo. La raíz de este árbol es el nodo con clave 30. La búsqueda se realiza con el algoritmo usual. Y, como suele ser habitual, las rutinas `insert` y `remove` son más complicadas debido a que los algoritmos naturales para el árbol de búsqueda binaria pueden inducir una violación de las propiedades de enlace horizontal. No le resultará sorprendente que digamos que las rotaciones del árbol pueden corregir todos los problemas con los que nos encontramos.

### 19.6.1 Inserción

La inserción de un nuevo elemento siempre se realiza en el nivel inferior. Como es habitual, eso puede crear problemas. En el árbol mostrado en la Figura 19.54, la inserción de 2 crearía un enlace horizontal izquierdo, mientras que la inserción de 45 generaría enlaces derechos consecutivos. En consecuencia, después de haber añadido un nodo en el nivel inferior, puede que necesitemos llevar a cabo algunas rotaciones para restaurar las propiedades de los enlaces horizontales.

Las inserciones se realizan utilizando el algoritmo recursivo usual y dos llamadas a método.

En ambos casos, una rotación simple basta para corregir el problema. Eliminamos los enlaces horizontales izquierdos efectuando una rotación entre el nodo y su hijo izquierdo, un procedimiento

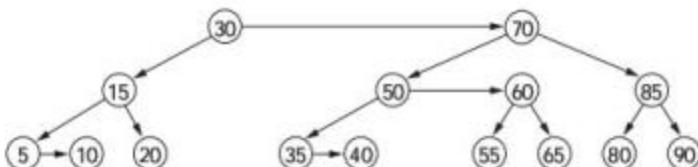


Figura 19.54 Un árbol AA resultado de la inserción de 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55 y 35

Los enlaces horizontales izquierdos se eliminan mediante una operación *skew* (rotación entre un nodo y su hijo izquierdo). Los enlaces horizontales derechos consecutivos se corrigen mediante una operación *split* (rotación entre un nodo y su hijo derecho). Una operación *skew* precede a una operación *split*.

denominado *skew*. Corregimos los enlaces horizontales derechos consecutivos, efectuando una rotación entre el primero y el segundo (de los tres) nodos unidos por los dos enlaces, un procedimiento denominado *split*.

El procedimiento *skew* se ilustra en la Figura 19.55 y el procedimiento *split* en la Figura 19.56. Aunque una operación *skew* elimina un enlace horizontal izquierdo, puede crear enlaces derechos horizontales consecutivos, porque el hijo derecho de *X* podría ser también horizontal. Por tanto, realizaríamos primero una operación *skew* y luego una *split*. Después de una operación *split*, el nodo intermedio aumenta de nivel. Eso puede provocar problemas para el padre original de *X*, al crear un enlace horizontal izquierdo o enlaces derechos horizontales consecutivos: ambos problemas pueden corregirse aplicando la estrategia *skew/split* en el camino de ascenso hacia la raíz. Puede realizarse automáticamente si utilizamos recursión, y una implementación recursiva de *insert* solo es dos llamadas a método más larga que la rutina correspondiente para un árbol de búsqueda no equilibrado.

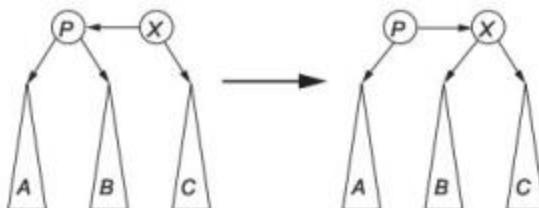


Figura 19.55 El procedimiento *skew* es una rotación simple entre *X* y *P*.

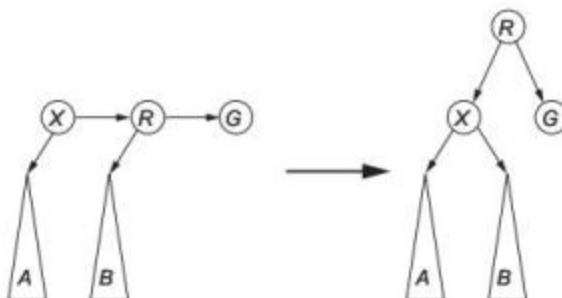


Figura 19.56 El procedimiento *split* es una rotación simple entre *X* y *R*; observe que el nivel de *R* se incrementa.

Para ilustrar cómo funciona el algoritmo, insertamos 45 en el árbol AA mostrado en la Figura 19.54. En la Figura 19.57, cuando se añade 45 en el nivel inferior, se forman enlaces horizontales consecutivos. Entonces se aplican parejas *skew/split* según sea necesario, desde el nivel inferior hacia la raíz. Así, en el nodo 35 hace falta una operación *split* debido a la existencia de los enlaces horizontales derechos consecutivos. El resultado de la operación *split* se muestra en la Figura 19.58. Cuando la recursión vuelve al nodo 50, nos encontramos con un enlace horizontal izquierdo. Por ello, realizamos una operación *skew* en 50 para eliminar el enlace horizontal izquierdo (el resultado se muestra en la Figura 19.59) y luego una operación *split* en 40 para eliminar los enlaces horizontales derechos consecutivos. El resultado después de la operación *split* se ilustra en la Figura 19.60. El resultado de la operación *split* es

Este es un raro algoritmo en el sentido de que es más difícil de simular en papel que de implementar en computadora.

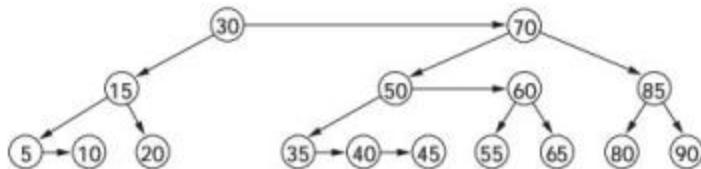


Figura 19.57 Despues de la insercción de 45 en el árbol de ejemplo; se introducen enlaces horizontales consecutivos comenzando en 35.

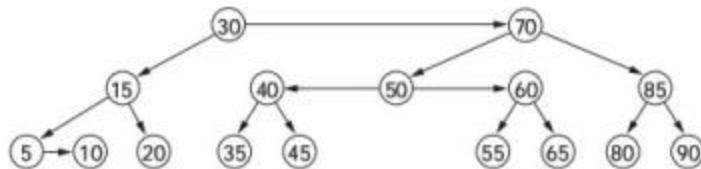


Figura 19.58 Despues de la operación *split* en 35; se introduce un enlace horizontal izquierdo en 50.

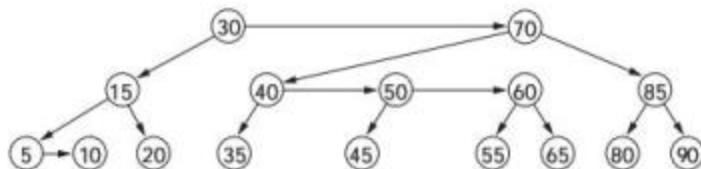


Figura 19.59 Despues de la operación *skew* en 50; se introducen nodos horizontales consecutivos comenzando en 40.

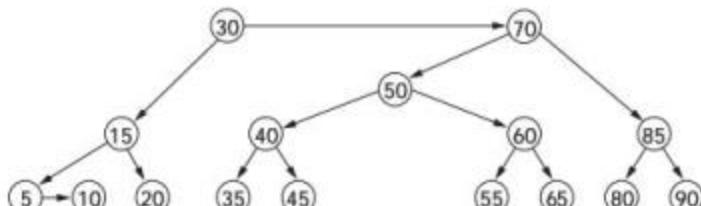


Figura 19.60 Despues de la operación *split* en 40; el nodo 50 está en el mismo nivel que 70, induciendo un enlace horizontal izquierdo ilegal.

que 50 estará en el nivel 3 y será un hijo horizontal izquierdo de 70. Por tanto, necesitaremos realizar otra pareja de operaciones *skew/split*. La operación *skew* en 70 elimina el enlace horizontal izquierdo del nivel superior, pero crea nodos horizontales derechos consecutivos, como se muestra en la Figura 19.61. Después de aplicar la operación *split* final, los nodos horizontales consecutivos se eliminan y 50 pasa a ser la nueva raíz del árbol. El resultado se muestra en la Figura 19.62.

## 19.6.2 Borrado

El borrado se simplifica, porque el caso de un único hijo solo se puede presentar en el nivel 1 y estamos dispuestos a utilizar la recursión.

Para árboles de búsqueda binaria en general, el algoritmo *remove* se descompone en tres casos: el elemento que hay que eliminar es una hoja, tiene un único hijo o tiene dos hijos. Para los árboles AA, tratamos el caso de un hijo de la misma forma que el caso de dos hijos, porque el caso de un solo hijo solo se puede presentar en el nivel 1. Además, el caso de dos hijos también es sencillo, porque el nodo utilizado como valor de sustitución está garantizado que se encuentre en el nivel 1, y en el caso peor solo tendrá un enlace horizontal derecho. Por tanto, todo se reduce a ser capaz de aplicar la operación *remove* para eliminar un nodo de nivel 1. Claramente, esta acción podría afectar al equilibrio (considere, por ejemplo, la eliminación del valor 20 en la Figura 19.62).

Sea  $T$  el nodo actual y asumamos que vamos a emplear recursión. Si el borrado ha modificado uno de los hijos de  $T$  a un valor que sea dos menos que el nivel de  $T$ , también habrá que reducir el nivel de  $T$  (solo el hijo en el que haya entrado la llamada recursiva podría verse afectado en realidad, pero por simplicidad no vamos a preocuparnos de cuál es). Además, si  $T$  tiene un enlace horizontal derecho, el nivel de su hijo derecho también tendrá que ser reducido. En este punto, podríamos tener

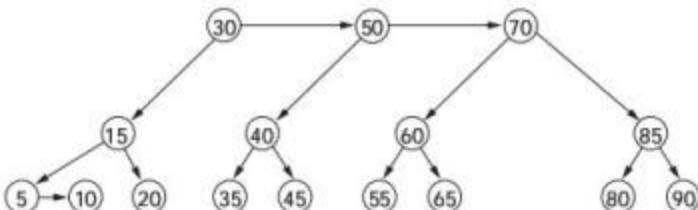


Figura 19.61 Despues de la operación *skew* en 70; se introducen enlaces horizontales consecutivos comenzando en 30.

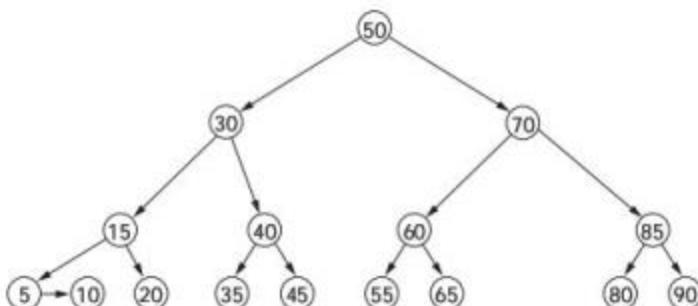


Figura 19.62 Despues de la operación *split* en 30; la inserción está completa.

seis nodos en el mismo nivel:  $T$ , el hijo derecho horizontal  $R$  de  $T$ , los dos hijos de  $R$  y los dos hijos derechos horizontales de esos hijos. La Figura 19.63 muestra el escenario más simple posible.

Después de eliminar el nodo 1, el nodo 2 y por tanto el nodo 5 pasarán a ser nodos de nivel 1. En primer lugar, deberemos corregir el enlace horizontal izquierdo que se ha introducido ahora entre los nodos 5 y 3. Hacer esto requiere esencialmente dos rotaciones: una entre los nodos 5 y 3 y luego otra entre los nodos 5 y 4. En este caso, el nodo actual  $T$  no se ve involucrado. Sin embargo, si un borrado viniera del lado derecho, el nodo izquierdo de  $T$  podría de repente pasar a ser horizontal; eso requeriría una rotación doble similar (comenzando en  $T$ ). Para evitar comprobar todos los casos, nos limitamos a invocar `skew` tres veces. Una vez que hemos hecho eso, dos llamadas a `split` bastan para reordenar los enlaces horizontales.

Después de un borrado recursivo, tres operaciones `skew` y dos operaciones `split` garantizan que se consiga re-equilibrar el árbol.

### 19.6.3 Implementación Java

El esqueleto de la clase para árboles AA se muestra en la Figura 19.64 e incluye una clase anidada para los nodos. Buena parte de la clase mostrada en la figura duplica el código anterior que hemos proporcionado para los árboles. De nuevo, utilizamos un nodo centinela `nullNode`; sin embargo, no necesitamos una pseudorafz. El constructor asigna `nullNode`, como para los árboles rojo-negro y hace que `root` lo refiera. El `nullNode` se encuentran en el nivel 0. Las rutinas utilizan rutinas auxiliares privadas.

La implementación es relativamente simple comparada con la del árbol rojo-negro.

En la Figura 19.65 se muestra el método `insert`. Como hemos mencionado anteriormente en esta sección, es casi idéntico al `insert` recursivo del árbol de búsqueda binaria. La única diferencia es que añade una llamada a `skew` seguida de una llamada a `split`. En la Figura 19.66, `skew` y `split` se implementan fácilmente, utilizando las rotaciones de árbol ya existentes. Finalmente, `remove` se muestra en la Figura 19.67.

Como ayuda, mantenemos dos variables de instancia, `deletedNode` y `lastNode`. Cuando recorremos un hijo derecho, ajustamos `deletedNode`. Puesto que invocamos `remove` recursivamente hasta alcanzar la parte inferior (no comprobamos la igualdad en el recorrido hacia abajo), tenemos garantizado que, si el elemento que hay que eliminar se encuentra en el árbol, `deletedNode` hará referencia al nodo que lo contenga. Observe que esta técnica se puede utilizar en el procedimiento `find` para sustituir las comparaciones de tres vías realizadas en cada nodo por comparaciones de dos vías en cada nodo más una comprobación adicional de igualdad en la parte inferior. `lastNode` apunta al nodo de nivel 1 en el que termina esta búsqueda. Puesto que no nos detenemos hasta alcanzar la parte inferior, si el elemento se

La variable `deletedNode` hace referencia al nodo que contiene  $x$  si se ha encontrado  $x$  o a `nullNode` si  $x$  no se ha encontrado. La variable `lastNode` hace referencia al nodo de sustitución. Utilizamos comparaciones de dos vías en lugar de comparaciones de tres vías.

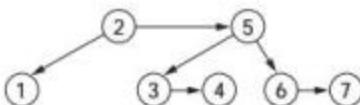


Figura 19.63 Cuando se borra 1, todos los nodos pasan a ser de nivel 1, introduciendo así enlaces horizontales izquierdos.

```

1 package weiss.nonstandard;
2
3 // Clase AATree
4 //
5 // CONSTRUCCIÓN: sin ningún inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // Igual que BinarySearchTree; omitidos por brevedad
9 // *****ERRORES*****
10 // insert y remove generan excepciones en caso necesario
11
12 public class AATree<AnyType extends Comparable<? super AnyType>>
13 {
14     public AATree( )
15     {
16         nullNode = new AANode<AnyType>( null, null, null );
17         nullNode.left = nullNode.right = nullNode;
18         nullNode.level = 0;
19         root = nullNode;
20     }
21
22     public void insert( AnyType x )
23     { root = insert( x, root ); }
24
25     public void remove( AnyType x )
26     { deletedNode = nullNode; root = remove( x, root ); }
27     public AnyType findMin( )
28     { /* Implementación como es usual; véase código en línea */ }
29     public AnyType findMax( )
30     { /* Implementación como es usual; véase código en línea */ }
31     public AnyType find( AnyType x )
32     { /* Implementación como es usual; véase código en línea */ }
33     public void makeEmpty( )
34     { root = nullNode; }
35     public boolean isEmpty( )
36     { return root == nullNode; }
37
38     private AANode<AnyType> insert( AnyType x, AANode<AnyType> t )
39     { /* Figura 19.65 */ }
40     private AANode<AnyType> remove( AnyType x, AANode<AnyType> t )
41     { /* Figura 19.67 */ }

```

Continúa

Figura 19.64 Esqueleto de la clase para los árboles AA.

```
42     private AANode<AnyType> skew( AANode<AnyType> t )
43         { /* Figura 19.66 */ }
44     private AANode<AnyType> split( AANode<AnyType> t )
45         { /* Figura 19.66 */ }
46
47     private static <AnyType>
48     AANode<AnyType> rotateWithLeftChild( AANode<AnyType> k2 )
49         { /* Implementación como es usual; véase código en línea */ }
50     private static <AnyType>
51     AANode<AnyType> rotateWithRightChild( AANode<AnyType> k1 )
52         { /* Implementación como es usual; véase código en línea */ }
53
54     private static class AANode<AnyType>
55     {
56         // Constructores
57         AANode( AnyType theElement )
58         {
59             element = theElement;
60             left = right = nullNode;
61             level = 1;
62         }
63
64         AnyType element;      // Los datos del nodo
65         AANode<AnyType> left; // Hijo izquierdo
66         AANode<AnyType> right; // Hijo derecho
67         int level; // Level
68     }
69
70     private AANode<AnyType> root;
71     private AANode<AnyType> nullNode;
72
73     private AANode<AnyType> deletedNode;
74     private AANode<AnyType> lastNode;
75 }
```

Figura 19.64 (Continuación).

encuentra en el árbol, `lastNode` hará referencia al nodo de nivel 1 que contenga el valor sustituto y debe ser eliminado del árbol.

Después de que termine una llamada recursiva dada, nos encontraremos en el nivel 1 o no. Si estamos en el nivel 1 podemos copiar el valor del nodo en el nodo interno que hay que sustituir; a continuación, podemos soltar el nodo de nivel 1. En caso contrario, estaremos en un nivel superior y tendremos que determinar si se ha violado la condición de equilibrio. En caso afirmativo, restauraremos el equilibrio y luego hacemos tres llamadas a `skew` y dos a `split`. Como hemos explicado anteriormente, estas acciones garantizan que se restauren las propiedades del árbol AA.

```

1  /**
2   * Método interno para insertar en un subárbol.
3   * @param x el elemento que hay que insertar.
4   * @param t el nodo que actúa como raíz del árbol.
5   * @return la nueva raíz.
6   * @throws DuplicateItemException si x ya está presente.
7   */
8  private AANode<AnyType> insert( AnyType x, AANode<AnyType> t )
9  {
10    if( t == nullNode )
11      t = new AANode<AnyType>( x, nullNode, nullNode );
12    else if( x.compareTo( t.element ) < 0 )
13      t.left = insert( x, t.left );
14    else if( x.compareTo( t.element ) > 0 )
15      t.right = insert( x, t.right );
16    else
17      throw new DuplicateItemException( x.toString() );
18
19    t = skew( t );
20    t = split( t );
21
22  }

```

Figura 19.65 La rutina `insert` para la clase `AATree`.

## 19.7 Implementación de las clases `TreeSet` y `TreeMap` de la API de Colecciones

En esta sección proporcionamos una implementación razonablemente eficiente de las clases `TreeSet` y `TreeMap` de la API de Colecciones. El código es una mezcla de la implementación de lista enlazada de la API de Colecciones presentada en la Sección 17.5 y de la implementación del árbol AA de la Sección 19.6. Algunos detalles del árbol AA no se reproducen aquí, porque tanto las rutinas privadas fundamentales, como las rotaciones del árbol prácticamente no varían. Esas rutinas están contenidas en el código en línea. Otras rutinas, como las rutinas privadas `insert` y `remove`, son solo ligeramente diferentes que las de la Sección 19.6, pero las reescribimos aquí para mostrar las similitudes y también por razones de exhaustividad.

La implementación básica recuerda la de la clase `LinkedList` estándar con sus clases de nodo, de conjunto y de iterador. Sin embargo, hay dos diferencias principales entre las clases.

1. La clase `TreeSet` se puede construir con un `Comparator` y el `Comparator` se guarda como un miembro de datos.
2. Las rutinas de iteración de `TreeSet` son más complejas que las de la clase `LinkedList`.

```
1  /**
2   * Primitiva skew para árboles AA.
3   * @param t el nodo que actúa como raíz del árbol.
4   * @return la nueva raíz después de la rotación.
5   */
6  private static <AnyType> AANode<AnyType> skew( AANode<AnyType> t )
7  {
8      if( t.left.level == t.level )
9          t = rotateWithLeftChild( t );
10     return t;
11 }
12
13 /**
14  * Primitiva split para árboles AA.
15  * @param t el nodo que actúa como raíz del árbol.
16  * @return la nueva raíz después de la rotación.
17  */
18 private static <AnyType> AANode<AnyType> split( AANode<AnyType> t )
19 {
20     if( t.right.right.level == t.level )
21     [
22         t = rotateWithRightChild( t );
23         t.level++;
24     }
25     return t;
26 }
```

Figura 19.66 Los procedimientos `skew` y `split` para la clase `AATree`.

La iteración es la parte más complicada. Debemos decidir cómo queremos llevar a cabo el recorrido. Tenemos varias alternativas a nuestra disposición:

1. Utilizar enlaces a los padres.
2. Hacer que el iterador mantenga una pila que represente los nodos que componen el camino hasta la posición actual.
3. Hacer que cada nodo mantenga un enlace a su sucesor según un recorrido en orden, una técnica conocida con el nombre de *árbol enhebrado*.

Para hacer que el código sea lo más parecido posible al código del árbol AA de la Sección 19.6, utilizaremos la opción de hacer que el iterador mantenga una pila. Dejamos la solución basada en los enlaces a los nodos padre como Ejercicio 19.21 para el lector.

La Figura 19.68 muestra el esqueleto de la clase `TreeSet`. La declaración de nodo se muestra en las líneas 12 y 13; el cuerpo de la declaración es idéntico al de `AANode` de la Sección 19.6. En la línea 18 se especifica el miembro de datos que almacena el objeto función para comparación. Las

```

1  /**
2  * Método interno para borrar de un subárbol.
3  * @param x el elemento que hay que borrar.
4  * @param t el nodo que actúa como raíz del árbol.
5  * @return la nueva raíz.
6  * @throws ItemNotFoundException si no se encuentra x.
7  */
8  private AANode<AnyType> remove( AnyType x, AANode<AnyType> t )
9  {
10     if( t != nullNode )
11     {
12         // Paso 1: buscar hacia abajo en el árbol y
13         // configurar lastNode y deletedNode
14         lastNode = t;
15         if( x.compareTo( t.element ) < 0 )
16             t.left = remove( x, t.left );
17         else
18         {
19             deletedNode = t;
20             t.right = remove( x, t.right );
21         }
22
23         // Paso 2: si estamos al final del árbol y
24         // x está presente, eliminarlo
25         if( t == lastNode )
26         {
27             if( deletedNode == nullNode ||
28                 x.compareTo( deletedNode.element ) != 0 )
29                 throw new ItemNotFoundException( x.toString() );
30             deletedNode.element = t.element;
31             t = t.right;
32         }
33
34         // Paso 3: en caso contrario, no estamos al final; re-equilibrar
35     } else
36         if( t.left.level < t.level - 1 || t.right.level < t.level - 1 )
37     {
38         if( t.right.level > --t.level )
39             t.right.level = t.level;
40         t = skew( t );
41         t.right = skew( t.right );
42         t.right.right = skew( t.right.right );
43         t = split( t );
44         t.right = split( t.right );
45     }
46 }
47 return t;
48 }

```

Figura 19.67 El método remove para árboles AA.

rutinas y campos de las líneas 54–55, 57–58, 62–63 y 70–77 son esencialmente idénticas a las correspondientes del árbol AA. Por ejemplo, las diferencias entre el método `insert` de las líneas 54 y 55 y el de la clase `AATree` es que la versión de `AATree` genera una excepción si se inserta un duplicado, mientras que esta rutina `insert` vuelve inmediatamente; asimismo, esta versión de `insert` mantiene los miembros de datos `size` y `modCount` y esta utiliza un comparador.

```
1 package weiss.util;
2
3 import java.io.Serializable;
4 import java.io.IOException;
5
6 public class TreeSet<AnyType> extends AbstractCollection<AnyType>
7 implements SortedSet<AnyType>
8 {
9     private class TreeSetIterator implements Iterator<AnyType>
10    { /* Figura 19.74 */ }
11
12    private static class AANode<AnyType> implements Serializable
13    { /* Igual que en la Figura 19.64 */ }
14
15    private int modCount = 0;
16    private int theSize = 0;
17    private AANode<AnyType> root = null;
18    private Comparator<? super AnyType> cmp;
19    private AANode<AnyType> nullNode;
20
21    public TreeSet( )
22    { /* Figura 19.69 */ }
23    public TreeSet( Comparator<? super AnyType> c )
24    { /* Figura 19.69 */ }
25    public TreeSet( SortedSet<AnyType> other )
26    { /* Figura 19.69 */ }
27    public TreeSet( Collection<? extends AnyType> other )
28    { /* Figura 19.69 */ }
29
30    public Comparator<? super AnyType> comparator( )
31    { /* Figura 19.69 */ }
32    private void copyFrom( Collection<? extends AnyType> other )
33    { /* Figura 19.69 */ }
34
35    public int size( )
36    { return theSize; }
```

Continúa

Figura 19.68 Esqueleto de la clase TreeSet.

```
37
38     public AnyType first( )
39         { /* Similar a findMin; véase el código en línea. */ }
40     public AnyType last( )
41         { /* Similar a findMax; véase el código en línea. */ }
42
43     public AnyType getMatch( AnyType x )
44         { /* Figura 19.70 */ }
45
46     private AANode<AnyType> find( AnyType x )
47         { /* Figura 19.69 */ }
48     private int compare( AnyType lhs, AnyType rhs )
49         { /* Figura 19.69 */ }
50     public boolean contains( Object x )
51         { /* Figura 19.69 */ }
52     public boolean add( AnyType x )
53         { /* Figura 19.71 */ }
54     private AANode<AnyType> insert( AnyType x, AANode<AnyType> t )
55         { /* Figura 19.71 */ }
56
57     private AANode<AnyType> deletedNode;
58     private AANode<AnyType> lastNode;
59
60     public boolean remove( Object x )
61         { /* Figura 19.72 */ }
62     private AANode<AnyType> remove( AnyType x, AANode<AnyType> t )
63         { /* Figura 19.73 */ }
64     public void clear( )
65         { /* Figura 19.72 */ }
66
67     public Iterator<AnyType> iterator( )
68         { return new TreeSetIterator( ); }
69
70     private static <AnyType> AANode<AnyType> skew( AANode<AnyType> t )
71         { /* Igual que en la Figura 19.66 */ }
72     private static <AnyType> AANode<AnyType> split( AANode<AnyType> t )
73         { /* Igual que en la Figura 19.66 */ }
74     private static <AnyType> AANode<AnyType> rotateWithLeftChild( AANode<AnyType> k2 )
75         { /* Igual que lo habitual */ }
76     private static <AnyType> AANode<AnyType> rotateWithRightChild( AANode<AnyType> k1 )
77         { /* Igual que lo habitual */ }
78 }
```

---

Figura 19.68 (Continuación)

Los constructores y el accesor `comparator` para la clase `TreeSet` se muestran en la Figura 19.69. También se incluye la rutina auxiliar privada `copyFrom`. La Figura 19.70 implementa la rutina pública `getMatch`, que es un método no estándar (que se utiliza para servir de ayuda con el `TreeMap` posteriormente). El método `find` privado es idéntico al de la Sección 19.6. El método `compare` utiliza el comparador si se proporciona uno; en caso contrario, asume que los parámetros son `Comparable` y utiliza su método `compareTo`. Si no se proporciona ningún comparador y los parámetros no son `Comparable`, entonces se generará una excepción `ClassCastException`, la cual es una acción bastante razonable para este tipo de situación.

En la Figura 19.71 se muestra el método público `add`. Este método simplemente invoca al método privado `insert`, que es similar al código que hemos visto anteriormente en la Sección 19.6. Observe que `add` tiene éxito si y solo si el tamaño del conjunto cambia.

La Figura 19.72 muestra los métodos públicos `remove` y `clear`. El método público de borrado llama a una rutina privada `remove`, mostrada en la Figura 19.73, que es muy similar al código presentado en la Sección 19.6. Los cambios principales son el uso de un comparador (a través del método `compare`) y el código adicional de las líneas 31 y 32.

La clase iteradora se muestra en la Figura 19.74; `current` está posicionado en el nodo que contiene el siguiente elemento no visto. La parte complicada está en el mantenimiento de la pila, `path`, que incluye todos los nodos que componen el camino desde el nodo actual, pero no el propio nodo actual. El constructor simplemente sigue todos los enlaces izquierdos, insertando en la pila todos los nodos del camino salvo el último. También mantenemos el número de elementos visitados, facilitando así la comprobación `hasNext`.

La rutina fundamental es el método privado `next`, mostrado en la Figura 19.75. Después de registrar el valor contenido en el nodo actual y configurar `lastVisited` (para `remove`), necesitamos hacer avanzar `current`. Si el nodo actual tiene un hijo derecho, vamos a la derecha y luego vamos hacia la izquierda cuando sea posible (líneas 11 a 17). En caso contrario, como ilustran las líneas 21 a 32, tenemos que retroceder en el camino que va hacia la raíz, hasta encontrar el nodo en el que hemos girado a la izquierda. Dicho nodo, que debe existir, porque en caso contrario se habría generado una excepción en la línea 4, es el siguiente nodo de la iteración.

La Figura 19.76 muestra la rutina `remove`, que es notablemente complicada. La parte relativamente sencilla se muestra en las líneas 3–15, en las que, después de algunas comprobaciones de error, eliminamos el elemento del árbol en la línea 11. En la línea 13, corregimos el valor de `expectedModCount`, para no obtener una posterior excepción `ConcurrentModificationException` para este iterador (únicamente). En la línea 14, decrementamos `visited` (para que `hasNext` funcione), y en la línea 15, configuramos `lastVisited` como `null`, para no permitir que se realice una operación `remove` consecutiva.

Si no hemos eliminado el último elemento de la iteración, entonces tenemos que reinicializar la pila, porque las rotaciones pueden haber reordenado el árbol. Esto se hace en las líneas 20 a 36. La línea 35 es necesaria porque no queremos que `current` esté en la pila.

Terminamos proporcionando una implementación de la clase `TreeMap`. Un `TreeMap` es simplemente un `TreeSet` en el que almacenamos parejas clave/valor. De hecho, podemos realizar una observación similar para `HashMap` en relación con `HashSet`. Así, implementaremos la clase abstracta `MapImpl` con visibilidad de paquete, que puede construirse a partir de cualquier `Set` (o `Map`). `TreeMap` y `HashMap` amplían `MapImpl`, proporcionando implementaciones de los métodos abstractos. El esqueleto de la clase para `MapImpl` se muestra en las Figuras 19.77 y 19.78.

```
1  /**
2  * Construir un TreeSet vacío.
3  */
4  public TreeSet( )
5  {
6      nullNode = new AANode<AnyType>( null, null, null );
7      nullNode.left = nullNode.right = nullNode;
8      nullNode.level = 0;
9      root = nullNode;
10     cmp = null;
11 }
12 /**
13 * Construir un TreeSet vacío con un comparador especificado.
14 */
15 public TreeSet( Comparator<? super AnyType> c )
16     { this( ); cmp = c; }
17
18 /**
19 * Construir un TreeSet a partir de otro SortedSet.
20 */
21 public TreeSet( SortedSet<AnyType> other )
22     { this( other.comparator( ) ); copyFrom( other ); }
23
24 /**
25 * Construir un TreeSet a partir de cualquier colección.
26 * Usa un algoritmo O( N log N ), pero podría mejorarse.
27 */
28 public TreeSet( Collection<? extends AnyType> other )
29     { this( ); copyFrom( other ); }
30
31 /**
32 * Devuelve el comparador utilizado por este TreeSet.
33 * @return el comparador o null si se usa el comparador predeterminado.
34 */
35 public Comparator<? super AnyType> comparator( )
36     { return cmp; }
37
38 /**
39 * Copia cualquier colección en un nuevo TreeSet.
40 */
41 private void copyFrom( Collection<? extends AnyType> other )
42 {
43     clear( );
44     for( AnyType x : other )
45         add( x );
46 }
```

---

Figura 19.69 Constructores y método comparador para TreeSet.

```
1  /**
2  * Este método no forma parte del estándar Java.
3  * Como contains, comprueba si x está en el conjunto.
4  * Si está, devuelve la referencia al objeto correspondiente;
5  * en caso contrario, devuelve null.
6  * @param x el objeto que hay que buscar.
7  * @return si contains(x) es false, el valor de retorno es null;
8  * en caso contrario, el valor de retorno es el objeto que hace que
9  * contains(x) devuelva true.
10 */
11 public AnyType getMatch( AnyType x )
12 {
13     AANode<AnyType> p = find( x );
14     if( p == null )
15         return null;
16     else
17         return p.element;
18 }
19
20 /**
21 * Encontrar un elemento en el árbol.
22 * @param x el elemento que hay que buscar.
23 * @return el elemento correspondiente o null si no se encuentra.
24 */
25 private AANode<AnyType> find( AnyType x )
26 {
27     AANode<AnyType> current = root;
28     nullNode.element = x;
29
30     for( ; ; )
31     {
32         int result = compare( x, current.element );
33
34         if( result < 0 )
35             current = current.left;
36         else if( result > 0 )
37             current = current.right;
38         else if( current != nullNode )
39             return current;
40         else
41             return null;
```

Continúa

Figura 19.70 Métodos de búsqueda para TreeSet.

```

42     }
43 }
44
45 private int compare( AnyType lhs, AnyType rhs )
46 {
47     if( cmp == null )
48         return ((Comparable) lhs).compareTo( rhs );
49     else
50         return cmp.compare( lhs, rhs );
51 }

```

Figura 19.70 (Continuación)

```

1 /**
2 * Añade un elemento a esta colección.
3 * @param x cualquier objeto.
4 * @return true si este elemento se ha añadido a la colección.
5 */
6 public boolean add( AnyType x )
7 {
8     int oldSize = size( );
9
10    root = insert( x, root );
11    return size( ) != oldSize;
12 }
13
14 /**
15 * Método interno para insertar en un subárbol.
16 * @param x el elemento que hay que insertar.
17 * @param t el nodo que actúa como raíz del árbol.
18 * @return la nueva raíz.
19 */
20 private AANode<AnyType> insert( AnyType x, AANode<AnyType> t )
21 {
22     if( t == nullNode )
23     {
24         t = new AANode<AnyType>( x, nullNode, nullNode );
25         modCount++;
26         theSize++;
27     }
28     else
29     {

```

Continúa

Figura 19.71 Métodos de inserción para TreeSet.

```
30     int result = compare( x, t.element );
31
32     if( result < 0 )
33         t.left = insert( x, t.left );
34     else if( result > 0 )
35         t.right = insert( x, t.right );
36     else
37         return t;
38 }
39
40 t = skew( t );
41 t = split( t );
42 return t;
43 }
```

Figura 19.71 (Continuación)

```
1 /**
2  * Elimina un elemento de esta colección.
3  * @param x cualquier objeto.
4  * @return true si este elemento se ha eliminado de la colección.
5  */
6 public boolean remove( Object x )
7 {
8     int oldSize = size( );
9
10    deletedNode = nullNode;
11    root = remove( AnyType ) x, root );
12
13    return size( ) != oldSize;
14 }
15
16 /**
17  * Cambiar el tamaño de esta colección a cero.
18  */
19 public void clear( )
20 {
21     theSize = 0;
22     modCount++;
23     root = nullNode;
24 }
```

Figura 19.72 Métodos públicos de borrado para TreeSet.

```

1  /**
2  * Método interno para borrar de un subárbol.
3  * @param x el elemento que hay que borrar.
4  * @param t el nodo que actúa como raíz del árbol.
5  * @return la nueva raíz.
6  */
7  private AANode<AnyType> remove( AnyType x, AANode<AnyType> t )
8  {
9      if( t != nullNode )
10     {
11         // Paso 1: buscar hacia abajo en el árbol y
12         // configurar lastNode y deletedNode
13         lastNode = t;
14         if( compare( x, t.element ) < 0 )
15             t.left = remove( x, t.left );
16         else
17         {
18             deletedNode = t;
19             t.right = remove( x, t.right );
20         }
21
22         // Paso 2: si estamos al final del árbol y
23         // x no está presente, eliminarlo
24         if( t == lastNode )
25         {
26             if( deletedNode == nullNode ||
27                 compare( x, deletedNode.element ) != 0 )
28                 return t; // Elemento no encontrado; no hacer nada
29             deletedNode.element = t.element;
30             t = t.right;
31             theSize--;
32             modCount++;
33         }
34
35         // Paso 3: en caso contrario, no estamos al final; re-equilibrar
36     else
37         if( t.left.level < t.level - 1 || t.right.level < t.level - 1 )
38         {
39             if( t.right.level > --t.level )
40                 t.right.level = t.level;
41             t = skew( t );
42             t.right = skew( t.right );
43             t.right.right = skew( t.right.right );
44             t = split( t );
45             t.right = split( t.right );
46         }
47     }
48 }
49

```

---

Figura 19.73 Método remove privado para TreeSet.

```
1  /**
2   * Esta es la implementación de TreeSetIterator.
3   * Mantiene una noción de una posición actual y, por supuesto
4   * la referencia implícita al TreeSet.
5   */
6  private class TreeSetIterator implements Iterator<AnyType>
7  {
8      private int expectedModCount = modCount;
9      private int visited = 0;
10     private Stack<AANode<AnyType>> path = new Stack<AANode<AnyType>>();
11     private AANode<AnyType> current = null;
12     private AANode<AnyType> lastVisited = null;
13
14     public TreeSetIterator( )
15     {
16         if( isEmpty( ) )
17             return;
18
19         AANode<AnyType> p = null;
20         for( p = root; p.left != nullNode; p = p.left )
21             path.push( p );
22
23         current = p;
24     }
25
26     public boolean hasNext( )
27     {
28         if( expectedModCount != modCount )
29             throw new ConcurrentModificationException( );
30
31         return visited < size( );
32     }
33
34     public AnyType next( )
35     { /* Figura 19.75 */ }
36
37     public void remove( )
38     { /* Figura 19.76 */ }
39 }
```

Figura 19.74 Esqueleto de la clase interna TreeSetIterator.

```
1  public AnyType next( )
2  {
3      if( !hasNext( ) )
4          throw new NoSuchElementException( );
5
6      AnyType value = current.element;
7      lastVisited = current;
8
9      if( current.right != nullNode )
10     {
11         path.push( current );
12         current = current.right;
13         while( current.left != nullNode )
14         {
15             path.push( current );
16             current = current.left;
17         }
18     }
19     else
20     {
21         AANode<AnyType> parent;
22
23         for( ; !path.isEmpty( ); current = parent )
24         {
25             parent = path.pop( );
26
27             if( parent.left == current )
28             {
29                 current = parent;
30                 break;
31             }
32         }
33     }
34
35     visited++;
36     return value;
37 }
```

---

Figura 19.75 El método next para TreeSetIterator.

En la línea 10 se declara un miembro de datos, el conjunto subyacente `theSet`. Las parejas clave/valor están representadas por una implementación concreta de la clase `Map.Entry`; esta implementación es parcialmente suministrada por la clase abstracta `Pair` que amplía `MapImpl` (en las líneas 52 a 72). En `TreeMap`, esta clase `Pair` es ampliada aun más proporcionando `compareTo`, mientras que en `HashMap` se la amplía proporcionando `equals` y `hashCode`.

```
1  public void remove( )
2  {
3      if( expectedModCount != modCount )
4          throw new ConcurrentModificationException( );
5
6      if( lastVisited == null )
7          throw new IllegalStateException( );
8
9      AnyType valueToRemove = lastVisited.element;
10
11     TreeSet.this.remove( valueToRemove );
12
13     expectedModCount++;
14     visited--;
15     lastVisited = null;
16
17     if( !hasNext( ) )
18         return;
19
20     // El código restante reinicializa la pila en caso de rotaciones
21     AnyType nextValue = current.element;
22     path.clear( );
23     AANode<AnyType> p = root;
24     for( ; ; )
25     {
26         path.push( p );
27         int result = compare( nextValue, p.element );
28         if( result < 0 )
29             p = p.left;
30         else if( result > 0 )
31             p = p.right;
32         else
33             break;
34     }
35     path.pop( );
36     current = p;
37 }
```

Figura 19.76 El método remove para TreeSetIterator.

Las líneas 17 a 21 declaran los tres métodos abstractos. Se trata de factorías que crean el objeto concreto apropiado y lo devuelven a través del tipo interfaz. Por ejemplo, en TreeMap, makeEmptyKeySet devuelve un TreeSet recién construido, mientras que en HashMap, makeEmptyKeySet devuelve un HashSet recién construido. Lo más importante es que makePair crea un objeto de tipo Map.Entry que representa la pareja clave/valor. Para un TreeSet, el objeto

```

1 package weiss.util;
2
3 /**
4 * MapImpl implementa el Map sobre un conjunto.
5 * Debe ser ampliada por TreeMap y HashMap, con
6 * llamadas encadenadas al constructor.
7 */
8 abstract class MapImpl<KeyType,ValueType> implements Map<KeyType,ValueType>
9 {
10     private Set<Map.Entry<KeyType,ValueType>> theSet;
11
12     protected MapImpl( Set<Map.Entry<KeyType,ValueType>> s )
13         { theSet = s; }
14     protected MapImpl( Map<KeyType,ValueType> m )
15         { theSet = clonePairSet( m.entrySet( ) ); }
16
17     protected abstract Map.Entry<KeyType,ValueType>
18             makePair( KeyType key, ValueType value );
19     protected abstract Set<KeyType> makeEmptyKeySet( );
20     protected abstract Set<Map.Entry<KeyType,ValueType>>
21             clonePairSet( Set<Map.Entry<KeyType,ValueType>> pairSet );
22
23     private Map.Entry<KeyType,ValueType> makePair( KeyType key )
24         { return makePair( (KeyType) key, null ); }
25     protected Set<Map.Entry<KeyType,ValueType>> getSet( )
26         { return theSet; }
27
28     public int size( )
29         { return theSet.size( ); }
30     public boolean isEmpty( )
31         { return theSet.isEmpty( ); }
32     public boolean containsKey( KeyType key )
33         { return theSet.contains( makePair( key ) ); }
34     public void clear( )
35         { theSet.clear( ); }
36     public String toString( )
37     {
38         StringBuilder result = new StringBuilder( "{" );
39         for( Map.Entry<KeyType,ValueType> e : entrySet( ) )
40             result.append( e + ", " );
41         result.replace( result.length() - 2, result.length(), "}" );
42         return result.toString( );
43     }
44
45     public ValueType get( KeyType key )
46         { /* Figura 19.79 */ }

```

Figura 19.77 Esqueleto de la clase auxiliar MapImpl (parte 1).

```
47    public ValueType put( KeyType key, ValueType value )
48        { /* Figura 19.79 */ }
49    public ValueType remove( KeyType key )
50        { /* Figura 19.79 */ }
51    // Clase Pair
52    protected static abstract class Pair<KeyType,ValueType>
53        implements Map.Entry<KeyType,ValueType>
54    {
55        public Pair( KeyType k, ValueType v )
56            { key = k; value = v; }
57
58        final public KeyType getKey( )
59            { return key; }
60
61        final public ValueType getValue( )
62            { return value; }
63
64        final public ValueType setValue( ValueType newValue )
65            { ValueType oldValue = value; value = newValue; return oldValue; }
66
67        final public String toString( )
68            { return key + "=" + value; }
69
70        private KeyType key;
71        private ValueType value;
72    }
73
74    // Vistas
75    public Set<KeyType> keySet( )
76        { return new KeySetClass( ); }
77    public Collection<ValueType> values( )
78        { return new ValueCollectionClass( ); }
79    public Set<Map.Entry<KeyType,ValueType>> entrySet( )
80        { return getSet( ); }
81
82    private abstract class ViewClass<AnyType> extends AbstractCollection<AnyType>
83        { /* Figura 19.80 */ }
84    private class KeySetClass extends ViewClass<KeyType> implements Set<KeyType>
85        { /* Figura 19.80 */ }
86    private class ValueCollectionClass extends ViewClass<ValueType>
87        { /* Figura 19.80 */ }
88
89    private class ValueCollectionIterator implements Iterator<ValueType>
90        { /* Figura 19.81 */ }
91    private class KeySetIterator implements Iterator<KeyType>
92        { /* Figura 19.81 */ }
93    }
```

Figura 19.78 Esqueleto de la clase auxiliar MapImpl (parte 2).

resulta ser `Comparable` y aplica el comparador de `TreeSet` a la clave. Más adelante analizaremos los detalles de esto.

Muchas de las rutinas del mapa se traducen a operaciones sobre el conjunto subyacente, como se muestra en las líneas 28 a 35. Las rutinas básicas `get`, `put` y `remove` se muestran en la Figura 19.79. Estas rutinas simplemente efectúan la traducción a operaciones sobre el conjunto. Todas requieren una llamada a `makePair` para crear un objeto del mismo tipo de los contenidos en `theSet`; `put` es una rutina representativa de esta estrategia.

La parte complicada de la clase `MapImpl` consiste en proporcionar la capacidad de obtener las vistas de las claves y valores. En la declaración de la clase `MapImpl` en la Figura 19.78, vemos que `keySet`, implementada en las líneas 75 y 76, devuelve una referencia a una instancia de una clase interna llamada `KeySetClass`, y `values`, implementada en las líneas 77 y 78, devuelve una referencia a una instancia de una clase interna denominada `ValueCollectionClass`. Las clases `KeySetClass` y `ValueCollectionClass` tienen algunos aspectos comunes, así que amplían la clase interna genérica llamada `ViewClass`. Estas tres clases aparecen en las líneas 82 a 87 de la declaración de la clase y su implementación se muestra en la Figura 19.80.

En la Figura 19.80, vemos que en la clase `ViewClass` genérica, las llamadas a `clear` y `size` se delegan en el mapa subyacente. Esta clase es abstracta, porque `AbstractCollection` no proporciona el método `iterator` especificado en `Collection`, como tampoco lo hace `ViewClass`. La clase `ValueCollectionClass` amplía `ViewClass<ValueType>` y proporciona un método `iterator`; este método devuelve una instancia recién construida de la clase interna `ValueCollectionIterator` (que por supuesto implementa la interfaz `Iterator`). `ValueCollectionIterator` delega las llamadas a `next` y `hasNext` y se muestra en la Figura 19.81; la analizaremos dentro de unos momentos. `KeySetClass` amplía `ViewClass<KeyType>`, pero como es un `Set`, debe proporcionar el método (no estándar) `getMatch` además del método `iterator`. Puesto que la propia clase `KeySet` no será utilizada para representar un `Map`, este método no será necesario, por lo que la implementación simplemente genera una excepción. También proporcionamos un método `remove` para eliminar el par asociado clave/valor del mapa subyacente. Si este método no se proporciona, el predeterminado que se hereda de `AbstractCollection` utiliza una búsqueda secuencial, que es enormemente inefficiente.

La Figura 19.81 completa la clase `MapImpl` proporcionando implementaciones de `KeySetIterator` y `ValueCollectionIterator`. Ambas mantienen un iterador que tiene una vista sobre el mapa subyacente, y ambas delegan las llamadas a `next`, `hasNext` y `remove` en el mapa subyacente. En el caso de `next`, se devuelve la parte apropiada del objeto `Map.Entry` que está siendo vista por el iterador del mapa.

```

1 /**
2 * Devuelve el valor almacenado en el mapa asociado con la clave.
3 * @param key la clave que hay que buscar.
4 * @return el valor correspondiente a la clave o null si no
5 * se encuentra la clave. Dado que los valores null están permitidos,
6 * comprobar si el valor de retorno es null puede no ser una forma
7 * segura de determinar si la clave está en el mapa.
8 */

```

Continúa

Figura 19.79 Implementaciones de los métodos básicos de `MapImpl`.

```
9 public ValueType get( KeyType key )
10 {
11     Map.Entry<KeyType,ValueType> match = theSet.getMatch( makePair( key ) );
12
13     if( match == null )
14         return null;
15     else
16         return match.getValue();
17 }
18
19 /**
20 * Añade al mapa la pareja clave/valor, sustituyendo el
21 * valor original si la clave ya estaba presente.
22 * @param key la clave que hay que insertar.
23 * @param value el valor que hay que insertar.
24 * @return el valor antiguo asociado con la clave, o
25 * null si la clave no estaba presente antes de esta llamada.
26 */
27 public ValueType put( KeyType key, ValueType value )
28 {
29     Map.Entry<KeyType,ValueType> match = theSet.getMatch( makePair( key ) );
30
31     if( match != null )
32         return match.setValue( value );
33
34     theSet.add( makePair( key, value ) );
35     return null;
36 }
37
38 /**
39 * Elimina la clave y su valor del mapa.
40 * @param key la clave que hay que eliminar.
41 * @return el valor anterior asociado con la clave,
42 * o null si la clave no estaba presente antes de esta llamada.
43 */
44 public ValueType remove( KeyType key )
45 {
46     ValueType oldValue = get( key );
47     if( oldValue != null )
48         theSet.remove( makePair( key ) );
49
50     return oldValue;
51 }
```

Figura 19.79 (Continuación).

```
1 /**
2 * Clase abstracta para modelar una vista (vista de clave o de valor).
3 * Implementa los métodos size y clear, pero no el método iterator.
4 * La vista delega en el mapa subyacente.
5 */
6 private abstract class ViewClass<AnyType> extends AbstractCollection<AnyType>
7 {
8     public int size( )
9         { return MapImpl.this.size( ); }
10
11    public void clear( )
12        { MapImpl.this.clear( ); }
13 }
14
15 /**
16 * Clase para modelar la vista del conjunto de claves.
17 * remove se sustituye (en caso contrario se usa una búsqueda secuencial).
18 * iterator proporciona un KeySetIterator (véase la Figura 19.81).
19 * getMatch, la parte no estándar de weiss.util.Set no es necesaria.
20 */
21 private class KeySetClass extends ViewClass<KeyType> implements Set<KeyType>
22 {
23     public boolean remove( Object key )
24         { return MapImpl.this.remove( (KeyType) key ) != null; }
25
26     public Iterator<KeyType> iterator( )
27         { return new KeySetIterator( ); }
28
29     public KeyType getMatch( KeyType key )
30         { throw new UnsupportedOperationException( ); }
31 }
32
33 /**
34 * Clase para modelar la vista de la colección de valores.
35 * Se utiliza remove predeterminado que es una búsqueda secuencial.
36 * iterator proporciona un ValueCollectionIterator (véase la Figura 19.81).
37 */
38 private class ValueCollectionClass extends ViewClass<ValueType>
39 {
40     public Iterator<ValueType> iterator( )
41         { return new ValueCollectionIterator( ); }
42 }
```

---

Figura 19.80 Clases de vistas para MapImpl.

```
1  /**
2   * Clase utilizada para iterar a través de la vista de claves.
3   * Delega en un iterador del conjunto subyacente de entradas.
4   */
5  private class KeySetIterator implements Iterator<KeyType>
6  {
7      private Iterator<Map.Entry<KeyType,ValueType>> itr = theSet.iterator();
8
9      public boolean hasNext( )
10     { return itr.hasNext( ); }
11
12    public void remove( )
13     { itr.remove( ); }
14
15    public KeyType next( )
16     { return itr.next( ).getKey( ); }
17 }
18
19 /**
20  * Clase para iterar a través de la vista de la colección de valores.
21  * Delega en un iterador del conjunto subyacente de entradas.
22  */
23 private class ValueCollectionIterator implements Iterator<ValueType>
24 {
25     private Iterator<Map.Entry<KeyType,ValueType>> itr = theSet.iterator();
26
27     public boolean hasNext( )
28     { return itr.hasNext( ); }
29
30     public void remove( )
31     { itr.remove( ); }
32
33     public ValueType next( )
34     { return itr.next( ).getValue( ); }
35 }
```

Figura 19.81 Clases de iterador de vista.

Habiendo escrito `MapImpl`, `TreeMap` resulta ser simple, como se muestra en la Figura 19.82. La mayor parte del código se centra en torno a la definición de la clase privada interna `Pair`, que implementa la interfaz `Map.Entry` ampliando `MapImpl.Pair`. La clase `Pair` implementa `Comparable`, utilizando el comparador sobre la clave, si es que se suministra uno, o efectuando una conversión de tipo a `Comparable`.

```
1 package weiss.util;
2
3 public class TreeMap<KeyType,ValueType> extends MapImpl<KeyType,ValueType>
4 {
5     public TreeMap( )
6         { super( new TreeSet<Map.Entry<KeyType,ValueType>>() ); }
7     public TreeMap( Map<KeyType,ValueType> other )
8         { super( other ); }
9     public TreeMap( Comparator<? super KeyType> comparator )
10    {
11        super( new TreeSet<Map.Entry<KeyType,ValueType>>() );
12        keyCmp = comparator;
13    }
14
15    public Comparator<? super KeyType> comparator( )
16        { return keyCmp; }
17
18    protected Map.Entry<KeyType,ValueType> makePair( KeyType key, ValueType value)
19        { return new Pair( key, value ); }
20
21    protected Set<KeyType> makeEmptyKeySet( )
22        { return new TreeSet<KeyType>( keyCmp ); }
23
24    protected Set<Map.Entry<KeyType,ValueType>>
25        clonePairSet( Set<Map.Entry<KeyType,ValueType>> pairSet )
26        { return new TreeSet<Map.Entry<KeyType,ValueType>>( pairSet ); }
27
28    private final class Pair extends MapImpl.Pair<KeyType,ValueType>
29                    implements Comparable<Map.Entry<KeyType,ValueType>>
30    {
31        public Pair( KeyType k, ValueType v )
32            { super( k, v ); }
33
34        public int compareTo( Map.Entry<KeyType,ValueType> other )
35        {
36            if( keyCmp != null )
37                return keyCmp.compare( getKey( ), other.getKey( ) );
38            else
39                return (( Comparable ) getKey( ) ).compareTo( other.getKey( ) );
40        }
41    }
42
43    private Comparator<? super KeyType> keyCmp;
44 }
```

---

Figura 19.82 Implementación de TreeMap.

## 19.8 Árboles-B

Hasta ahora, hemos asumido que podemos almacenar una estructura de datos completa en la memoria principal de una computadora. Sin embargo, suponga que tenemos más datos de los que caben en la memoria principal y que, como resultado, debemos hacer que la estructura de datos resida en disco. Cuando esto sucede, las reglas del juego cambian, porque el modelo O mayúscula ya no tiene sentido.

El problema es que un análisis O mayúscula supone que todas las operaciones son iguales. Sin embargo, esto no es cierto, especialmente cuando está involucrada la E/S de disco. Por un lado, una máquina de 500-MIPS supuestamente ejecuta 500 millones de instrucciones por segundo. Eso es bastante rápido, principalmente porque la velocidad depende básicamente de las propiedades eléctricas. Por otro lado, un disco es un dispositivo mecánico. Su velocidad depende en buena medida del tiempo requerido para hacer girar el disco y desplazar un cabezal del disco. Muchos discos rotan a 7.200 RPM. Por tanto, en 1 minuto, hacen 7.200 revoluciones; esto quiere decir que una revolución tarda  $1/120$  segundos, es decir, 8,3 ms. En promedio, cabría esperar que tengamos que hacer rotar un disco la mitad de una vuelta para encontrar lo que estamos buscando, pero esto se ve compensado por el tiempo necesario para mover el cabezal del disco, por lo que obtenemos un tiempo de acceso de 8,3 ms. (Esta estimación es bastante benévolas; los tiempos de acceso comprendidos entre 9 y 11 ms son más comunes.) En consecuencia, podemos hacer aproximadamente 120 accesos a disco por segundo. Este número de accesos parece adecuado, hasta que lo comparamos con la velocidad del procesador: tenemos 500 millones de instrucciones frente a 120 accesos a disco. Dicho de otra forma, cada acceso a disco equivale a 4.000.000 de instrucciones. Por supuesto, todos estos números son cálculos burdos, pero las velocidades relativas están bastante claras: los accesos a disco son increíblemente costosos. Además, las velocidades de procesador se están incrementando a una tasa mucho mayor que las velocidades de disco (son los *tamaños* de disco los que se están incrementando muy rápidamente). Por tanto, estaríamos dispuestos a realizar un montón de cálculos simplemente para ahorrarnos un acceso a disco. En casi todos los casos, el número de accesos a disco domina el tiempo de ejecución. Reduciendo a la mitad el número de accesos a disco, podemos dividir por dos el tiempo de ejecución.

He aquí cómo se comportaría el árbol de búsqueda típico con un disco. Suponga que queremos acceder a los registros que contienen los permisos de conducir de los ciudadanos del estado de Florida. Vamos a suponer que tenemos 10.000.000 de elementos, que cada clave tiene 32 bytes (que representan un nombre) y que cada registro es de 256 bytes. Suponemos que este conjunto de datos no cabe en memoria principal y que nosotros somos uno de los 20 usuarios del sistema (por lo que tenemos 1/20 de los recursos). En esta situación, podemos ejecutar 25 millones de instrucciones en 1 segundo o realizar seis accesos a disco.

El árbol de búsqueda binaria no equilibrado es un desastre. En el caso peor, tiene una profundidad lineal y por tanto podría requerir 10 millones de accesos a disco. En promedio, una búsqueda con éxito requeriría  $1,38 \log N$  accesos a disco y como  $\log 10.000.000$  es aproximadamente igual a 24, una búsqueda media necesitaría 32 accesos a disco, lo que equivale a 5 segundos. En un árbol típico construido aleatoriamente, cabría esperar que unos cuantos nodos estuvieran almacenados a una profundidad tres veces mayor;

cuando los datos son demasiados para caber en memoria, el número de accesos a disco pasa a ser importante. Un acceso a disco es increíblemente costoso comparado con una instrucción típica de computadora.

Incluso el rendimiento logarítmico es inaceptable. Necesitamos realizar las búsquedas en tres o cuatro accesos. Las actualizaciones pueden tardar un poco más.

requerirían unos 100 accesos a disco o 16 segundos. Un árbol rojo-negro sería algo mejor: el caso peor de  $1,44 \log N$  es bastante improbable que se produzca y el caso típico es muy próximo a  $\log N$ . Por tanto, un árbol rojo-negro emplearía unos 25 accesos a disco como promedio, lo que requiere 4 segundos.

**Un árbol de búsqueda  $M$ -ario** permite ramificaciones de  $M$  vías. A medida que se incrementa el grado de ramificación, la profundidad disminuye.

Queremos reducir los accesos a disco a un número constante que sea muy pequeño, como por ejemplo tres o cuatro accesos. Para ello, estamos dispuestos a escribir código complicado porque las instrucciones de la máquina prácticamente no tienen coste, siempre y cuando no lleguemos a soluciones ridículamente poco razonables. Un árbol de búsqueda binaria no funciona, porque el árbol rojo-negro típico está próximo a la altura óptima y no podemos bajar de  $\log N$  con un árbol de búsqueda binaria. La solución es intuitivamente simple: si tenemos un mayor grado de ramificación, la altura disminuirá. Así, mientras que un árbol binario perfecto de 31 nodos tiene 5 niveles, un árbol 5-ario de 31 nodos solo tiene tres niveles, como se muestra en la Figura 19.83. Un *árbol de búsqueda  $M$ -ario* permite bifurcaciones de  $M$  vías, y a medida que se incrementa el grado de bifurcación, la profundidad se reduce. Mientras que un árbol binario completo tiene una altura que es aproximadamente  $\log_2 N$ , un árbol  $M$ -ario completo tiene una altura de aproximadamente  $\log_M N$ .

Podemos crear un árbol de búsqueda  $M$ -ario de forma bastante similar a como hemos creado un árbol de búsqueda binaria. En un árbol de búsqueda binaria, necesitamos una clave para decidir cuál de las dos ramas seguir. En un árbol de búsqueda  $M$ -ario necesitamos  $M - 1$  claves para decidir qué rama tomar. Para hacer que este esquema sea eficiente en el caso peor, necesitamos garantizar que el árbol de búsqueda  $M$ -ario sea equilibrado de alguna forma. En caso contrario, como en un árbol de búsqueda binaria, podría degenerar en una lista enlazada. De hecho, queremos una condición de equilibrio todavía más restrictiva; es decir, no queremos que un árbol de búsqueda  $M$ -ario degenera ni siquiera en un árbol de búsqueda binaria, porque entonces estaríamos limitados a un número de acceso del orden de  $\log N$ .

**Un árbol-B** es la estructura de datos más popular para búsquedas limitadas por disco.

Una forma de implementar esto consiste en utilizar un *árbol-B* que es la estructura de datos más popular para búsquedas limitadas por disco. Aquí, vamos a describir el árbol-B básico<sup>3</sup>; existen muchas variantes y mejoras, y las implementaciones son un tanto complejas, porque es necesario tener en cuenta unos cuantos casos distintos. Sin embargo, en principio esta técnica garantiza que solo se realice un pequeño número de accesos a disco.

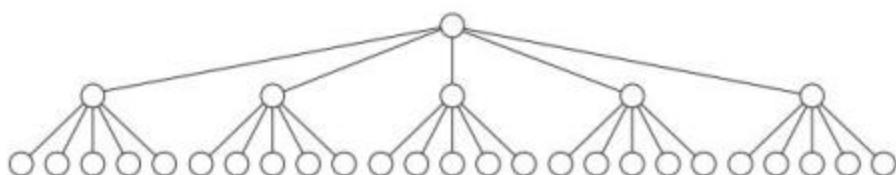


Figura 19.83 Un árbol 5-ario de 31 nodos solo tiene tres niveles.

<sup>3</sup> Lo que vamos a describir se conoce popularmente como árbol-B\*.

Un árbol-B de orden  $M$  es un árbol  $M$ -ario con las siguientes propiedades.<sup>4</sup>

- Los elementos de datos se almacenan en las hojas.
- Los nodos que no son hojas almacenan hasta  $N - 1$  claves con el fin de guiar la búsqueda; la clave  $i$  representa la clave más pequeña en el subárbol  $i + 1$ .
- La raíz es una hoja o tiene entre 2 y  $M$  hijos.
- Todos los nodos que no son hojas (excepto la raíz) tienen entre  $\lceil M/2 \rceil$  y  $M$  hijos.
- Todas las hojas están a la misma profundidad y tienen entre  $\lceil L/2 \rceil$  y  $L$  elementos de datos, para un cierto valor de  $L$  (enseñaremos explicaremos cómo se determina  $L$ ).

En la Figura 19.84 se muestra un ejemplo de un árbol-B de orden 5. Observe que todos los nodos que no son hojas tienen entre tres y cinco hijos (y por tanto entre dos y cuatro claves); la raíz podría posiblemente tener solo dos hijos. Aquí,  $L = 5$ , lo que quiere decir que  $L$  y  $M$  coinciden en este ejemplo, aunque esta condición no es necesaria. Puesto que  $L$  es 5, cada hoja tiene entre tres y cinco elementos de datos. Exigir que los nodos estén llenos a la mitad garantiza que el árbol-B no degenera en un árbol binario simple. Diversas definiciones de los árboles-B modifican esta estructura, fundamentalmente en algunos aspectos menores, pero la definición presentada aquí es una de las más comúnmente utilizadas.

Cada nodo representa un bloque de disco, por lo que elegimos  $M$  y  $L$  teniendo en cuenta el tamaño de los elementos que se estén almacenando. Suponga que cada bloque almacena 8.192 bytes. En nuestro ejemplo de Florida, cada clave utiliza 32 bytes, así que en un árbol-B de orden  $M$ , tendríamos  $M - 1$  claves, lo que nos da un total de  $32M - 32$  bytes más  $M$  ramas. Puesto que cada rama es esencialmente el número de otro bloque de disco, podemos asumir que una rama ocupa 4 bytes. Por tanto, las ramas utilizan  $4M$  bytes y la memoria total requerida para un nodo que no sea una hoja es  $36M - 32$ . El valor máximo de  $M$  para el que  $36M - 32$  no supera 8.192 es 228, así que elegiríamos  $M = 228$ . Como cada registro de datos ocupa 256 bytes, podríamos

El árbol-B tiene toda una serie de propiedades estructurales.

Los nodos deben estar llenos al menos a la mitad, para garantizar que el árbol no degenera en un árbol binario simple.

Elegimos los máximos valores de  $M$  y  $L$  que permitan encajar un nodo en un solo bloque de disco.

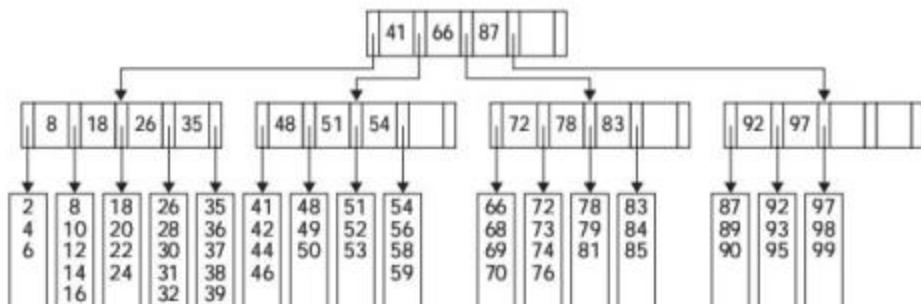


Figura 19.84 Un árbol-B de orden 5.

<sup>4</sup> Las propiedades 3 y 5 deben relajarse para las primeras  $L$  inserciones ( $L$  es un parámetro utilizado en la propiedad 5).

almacenar 32 registros en cada bloque. Por tanto, seleccionaríamos  $L = 32$ . Cada hoja tendrá entre 16 y 32 registros de datos y cada nodo interno (excepto la raíz) puede ramificarse en al menos 114 vías. Para los 10.000.000 de registros, habrá como máximo 625.000 hojas. En consecuencia, en el caso peor, las hojas se encontrarían en el nivel 4. En términos más concretos, el número de accesos de caso peor está dado por aproximadamente  $\log_{M^2} N$ , más o menos 1.

El problema que nos resta es cómo añadir y eliminar elementos del árbol-B. En las ideas que hemos esbozado, observe que vuelven a aparecer muchos de los temas que hemos comentado anteriormente.

Si la hoja tiene espacio para un nuevo elemento, lo insertamos y habremos terminado.

Si la hoja está llena, podemos insertar un nuevo elemento dividiendo la hoja y formando dos nodos medio vacíos.

Comenzaremos examinando la operación de inserción. Suponga que queremos insertar el valor 57 en el árbol-B de la Figura 19.84. Una búsqueda hacia abajo del árbol revela que 57 no se encuentra ya incluido en el árbol. Podemos añadir 57 a la hoja como un quinto elemento, pero podríamos tener que reorganizar todos los datos de la hoja para hacerlo. Sin embargo, el coste es despreciable comparado con el del acceso a disco, que en este caso incluye también una escritura en disco.

Este procedimiento parece relativamente sencillo, porque la hoja no se encontraba ya llena. Suponga que ahora queremos insertar el valor 55. La Figura 19.85 muestra un problema: la hoja en la que debería ir 55 ya está llena. La solución es simple: ahora tenemos  $L + 1$  elementos, por lo que lo dividimos en dos hojas, que está garantizado que tendrán el número mínimo de registros de datos necesario. Aquí, formaríamos dos hojas con tres elementos cada una. Hacen falta dos accesos a disco para escribir estas hojas y un tercer acceso a disco para actualizar el padre. Observe que en el padre, se modifican tanto las claves como las ramas, pero lo hace de una forma controlada que se puede calcular fácilmente. El árbol-B resultante se muestra en la Figura 19.86. Aunque dividir los nodos consume tiempo, porque requiere al menos dos escrituras en disco adicionales, es un suceso relativamente raro. Por ejemplo, si  $L$  es 32, cuando se divide 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 se produzca otra división. Dicho de otra forma, por cada división que se realiza habrá aproximadamente  $L/2$  no divisiones.

La división de un nodo en el ejemplo anterior ha funcionado porque el padre no tenía el máximo número de hijos. ¿Pero qué sucedería si lo tuviera? Suponga que insertamos 40 en el árbol-B mostrado en la Figura 19.86. Debemos dividir la hoja que contiene las claves 35 a 39 (y que ahora

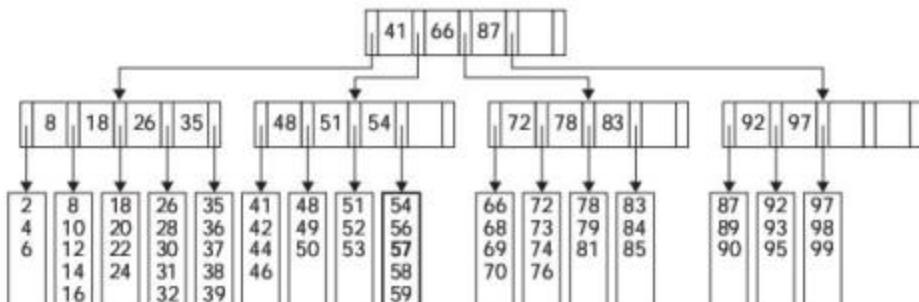


Figura 19.85 El árbol-B después de la inserción de 57 en el árbol mostrado en la Figura 19.84.

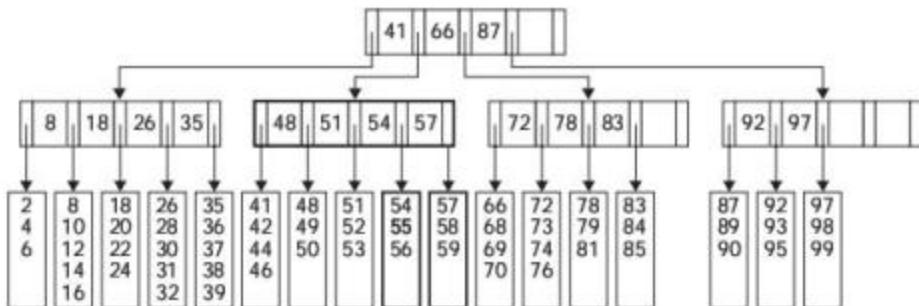


Figura 19.86 La inserción de 55 en el árbol-B mostrado en la Figura 19.85 provoca una división en dos hojas.

contiene también 40 en dos hojas. Pero hacer esto haría que el padre tuviera seis hijos, y solo está permitido que tenga cinco. La solución consiste en dividir el padre, mostrándose el resultado de esta operación en la Figura 19.87. Al dividir el padre, tenemos que actualizar los valores de las claves y también el padre del padre, lo que exige dos escrituras en disco adicionales (así que esta inserción nos cuesta cinco escrituras en disco). De nuevo, sin embargo, las claves cambian de una forma muy controlada, aunque es verdad que el código no resulta simple debido al gran número de casos posibles.

Cuando se divide un nodo que no es una hoja, como aquí, su padre pasa a tener un hijo más, ¿qué sucede si el padre ya ha alcanzado su límite de hijos? Entonces continuamos dividiendo nodos hacia arriba del árbol hasta encontrar un padre que no necesite ser dividido o hasta alcanzar la raíz. Observe que ya hemos presentado esta idea en los árboles rojo-negro de actualización abajo-arriba y en los árboles AA. Si dividimos la raíz, tendremos dos raíces, pero obviamente este resultado es inaceptable. Sin embargo, podemos crear una nueva raíz que tenga esas raíces divididas como hijos, lo cual es la razón de que a la raíz se le conceda la excepción de tener un mínimo de dos hijos. También es la única manera de que un árbol-B pueda crecer en altura. No hace falta decir que tener que dividir nodos a lo largo de todo el camino que va hasta la raíz es un suceso extremadamente raro, porque un árbol con cuatro niveles indica que la raíz ha sido dividida dos veces a lo largo de toda la secuencia de inserciones (suponiendo que no se hayan producido borrados). De hecho, la división de cualquier nodo que no sea una hoja es también bastante rara.

La división de un nodo crea un hijo adicional para el padre del nodo hoja. Si el padre ya tiene el número máximo de hijos, será necesario dividirlo también.

Puede que tengamos que continuar dividiendo nodos durante todo el camino de subida por el árbol (aunque esta posibilidad es improbable). En el caso peor, dividiríamos la raíz creando una nueva raíz con dos hijos.

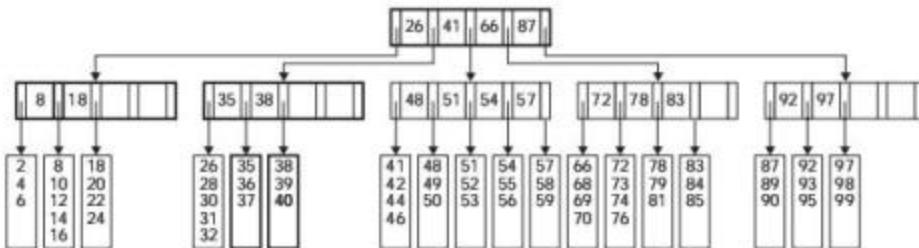


Figura 19.87 La inserción de 40 en el árbol-B mostrado en la Figura 19.86 provoca una división en dos hojas y luego una división del nodo padre.

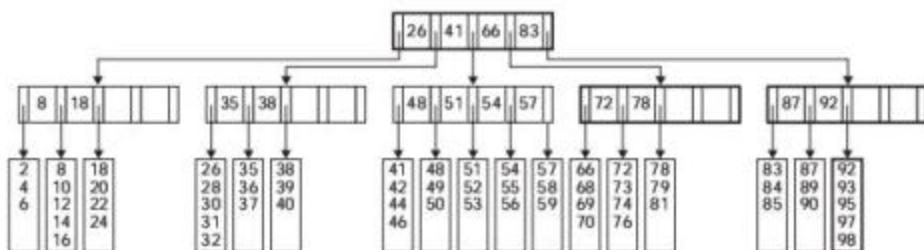


Figura 19.88 El árbol-B después de la eliminación de 99 en el árbol mostrado en la Figura 19.87.

Existen otras formas de tratar el desbordamiento de hijos. Una técnica consiste en enviar un hijo hacia arriba para su adopción, por si acaso un vecino tiene espacio para él. Por ejemplo, para insertar 29 en el árbol-B mostrado en la Figura 19.87, podríamos hacer sitio para él desplazando 32 a la siguiente hoja. Esta técnica requiere una modificación del padre, porque las claves se ven afectadas. Sin embargo, tiende a hacer que los nodos estén más llenos y ahorra espacio a largo plazo.

El borrado funciona a la inversa: si una hoja pierde un hijo, puede que sea necesario combinarla con otra hoja. La combinación de nodos puede continuar durante todo el camino hacia arriba del árbol, aunque esta posibilidad es improbable. En el caso peor, la raíz pierde uno de sus dos hijos. Entonces podemos borrar la raíz y emplear el otro hijo como la nueva raíz.

Podemos realizar un borrado encontrando el elemento que haya que eliminar y borrándolo. El problema es que si la hoja en la que estaba contenido tuviera un número mínimo de elementos de datos, ahora pasaría a estar por debajo del mínimo. Podemos rectificar la situación adoptando un elemento de una hoja vecina si es que el nodo vecino no tiene él mismo el número mínimo de elementos. Si lo tuviera, podemos combinar nuestro nodo hoja con el vecino para formar una hoja completa. Lamentablemente, en este caso el padre habrá perdido un hijo. Si eso hace que el padre caiga por debajo de su mínimo, seguiremos la misma estrategia. Este proceso podría repetirse durante todo el camino de ascenso hacia la raíz. La raíz no puede tener un solo hijo (e incluso si eso se permitiera, sería algo bastante estúpido). Si una raíz se quedara con un solo hijo como resultado del proceso de adopción, lo que hacemos es eliminar la raíz, haciendo que su hijo sea la nueva raíz del árbol –esta es la única forma de que un árbol-B pierda altura. Suponga que queremos eliminar el elemento 99 del árbol-B mostrado en la Figura 19.87. La hoja tiene solo dos elementos y su vecino está ya en su valor mínimo de tres, así que combinamos los elementos en una nueva hoja de cinco elementos. Como resultado el padre solo tiene dos hijos. Sin embargo, puede adoptar de un vecino, porque el vecino tiene cuatro hijos. Como resultado de la adopción, ambos terminan teniendo tres hijos, como se muestra en la Figura 19.88.

## Resumen

Los árboles de búsqueda binaria soportan casi todas las operaciones útiles en el diseño de algoritmos, y el coste promedio logarítmico es muy pequeño. Las implementaciones no recursivas de los árboles de búsqueda son algo más rápidas que las versiones recursivas, pero estas últimas son más compactas, más elegantes y más fáciles de comprender y de depurar. El problema con los árboles de búsqueda es que su rendimiento depende fuertemente de que la entrada sea aleatoria. Si no lo es, el tiempo de ejecución se incrementa significativamente, incluso hasta el punto en que los árboles de búsqueda se transforman en costosas listas enlazadas.

Las formas de tratar con este problema implican todas ellas reestructurar el árbol para garantizar que haya un cierto equilibrio en cada nodo. La reestructuración se consigue mediante rotaciones del árbol que preservan la propiedad del árbol de búsqueda binaria. El coste de una búsqueda es típicamente menor que para un árbol de búsqueda binaria no equilibrado, porque el nodo promedio tiende a estar más próximo a la raíz. Sin embargo, los costes de inserción y borrado suelen ser mayores. Las variantes equilibradas difieren en la cantidad de esfuerzo de codificación requerido para implementar las operaciones que modifican el árbol.

El esquema clásico es el árbol AVL en el que, para todo nodo, las alturas de sus subárboles izquierdo y derecho pueden diferir en como máximo 1. El problema práctico con los árboles AVL es que implican un gran número de casos distintos, haciendo que el coste adicional de cada inserción y borrado sea relativamente alto. Hemos examinado dos alternativas en este capítulo. La primera era el árbol rojo-negro de procesamiento arriba-abajo. Su ventaja principal es que el re-equilibrado se puede implementar en una única pasada descendente del árbol, en lugar de tener que realizar la tradicional pasada hacia abajo y luego otra hacia arriba. Esta técnica proporciona un código más simple y una mayor velocidad que los árboles AVL. La segunda variante es el árbol AA, que es similar al árbol rojo-negro de procesamiento abajo-arriba. Su ventaja principal es una implementación recursiva relativamente simple, tanto para la inserción como para el borrado. Ambas estructuras utilizan nodos centinela para eliminar los molestos casos especiales.

Solo debería utilizar un árbol de búsqueda binaria no equilibrado si está seguro de que los datos son razonablemente aleatorios o que la cantidad de datos es relativamente pequeña. Utilice el árbol rojo-negro si le preocupa la velocidad (y no le preocupa demasiado el borrado). Use el árbol AA si desea obtener una implementación sencilla con una velocidad más que aceptable. Emplee el árbol-B cuando la cantidad de datos sea demasiado grande como para almacenarlos en la memoria principal.

En el Capítulo 22 examinaremos otra alternativa: el árbol *splay*. Se trata de una alternativa interesante al árbol de búsqueda equilibrado: es simple de codificar y bastante competitivo en la práctica. En el Capítulo 20 examinaremos la tabla hash, un método completamente distinto que se utiliza para implementar operaciones de búsqueda.



## Conceptos clave

**árbol AA** Un árbol de búsqueda equilibrado que es el preferido cuando hace falta un caso peor  $\mathcal{O}(\log N)$ , cuando una implementación descuidada es aceptable y cuando hacen falta borrados. (718)

**árbol AVL** Un árbol de búsqueda binaria con la propiedad adicional de equilibrio de que, para cualquier nodo del árbol, la altura de los subárboles izquierdo y derecho puede diferir en como máximo 1. Tiene una importancia histórica, ya que fue el primer árbol de búsqueda equilibrado. También ilustra la mayoría de las ideas utilizadas en otros esquemas de árbol de búsqueda. (696)

**árbol-B** La estructura de datos más popular para búsquedas limitadas por disco. Existen muchas variantes de la misma idea. (748)

**árbol de búsqueda binaria** Una estructura de datos que soporta la inserción, la búsqueda y el borrado en un tiempo promedio  $\mathcal{O}(\log N)$ . Para cualquier nodo del árbol de búsqueda binaria, todos los nodos con claves más pequeñas se encuentran en el subár-

bol izquierdo y todos los nodos con claves más grandes se encuentran en el subárbol derecho. No se permiten duplicados. (677)

**árbol de búsqueda binaria equilibrado** Un árbol que tiene una propiedad estructural añadida para garantizar una profundidad logarítmica en caso peor. Las actualizaciones son más lentas que con el árbol de búsqueda binaria, pero los accesos son más rápidos. (696)

**árbol Mario** Un árbol que permite bifurcaciones de  $M$  vías; a medida que se incrementa la bifurcación, la profundidad se reduce. (748)

**árbol rojo-negro** Un árbol de búsqueda equilibrado que constituye una buena alternativa al árbol AVL porque se puede utilizar una sola pasada arriba-abajo durante las rutinas de inserción y borrado. Los nodos se marcan con color rojo y negro en una forma restringida que garantiza una profundidad logarítmica. Los detalles de codificación tienden a proporcionar una implementación más rápida. (704)

**borrado perezoso** Un método que marca los elementos como borrados, pero que no los borra en realidad. (718)

**enlace horizontal** En un árbol AA, una conexión entre un nodo y un hijo que está situado al mismo nivel. Un enlace horizontal solo debería ir hacia la derecha y no deben existir dos enlaces horizontales consecutivos. (719)

**longitud externa de camino** La suma del coste de acceder a todos los nodos externos del árbol en un árbol binario, lo que mide el coste de una búsqueda que no tenga éxito. (694)

**longitud interna de camino** La suma de las profundidades de los nodos en un árbol binario, lo que mide el coste de una búsqueda que tenga éxito. (694)

**nivel de un nodo** En un árbol AA, el número de enlaces izquierdos en el camino que va hasta el nodo centinela `nullNode`. (719)

**nodo externo del árbol** El nodo `null`. (694)

**rotación doble** Equivalente a dos rotaciones simples. (701)

**rotación simple** Intercambio de los papeles del padre y del hijo al tiempo que se mantiene el orden de búsqueda. El equilibrio se restaura mediante rotaciones del árbol. (698)

**skew** Eliminación de enlaces horizontales izquierdos, realizando una rotación entre un nodo y su hijo izquierdo. (720)

**split** Eliminación de enlaces horizontales derechos consecutivos, realizando una rotación entre un nodo y su hijo derecho. (720)



## Errores comunes

1. Utilizar un árbol de búsqueda no equilibrado cuando la secuencia de entrada no sea aleatoria dará un rendimiento muy pobre.
2. La operación `remove` es bastante complicada de codificar correctamente, especialmente para un árbol de búsqueda equilibrado.

- 3 El borrado perezoso es una buena alternativa a la operación `remove` estándar, pero entonces es necesario modificar otras rutinas, como `findMin`.
- 4 El código para árboles de búsqueda equilibrados es siempre muy proclive a errores.
- 5 Olvidarse de devolver una referencia a la nueva raíz del subárbol es un error para los métodos auxiliares privados `insert` y `remove`. El valor de retorno se debe asignar a `root`.
- 6 Si se utilizan nodos centinela y luego se escribe código que se olvida de esos nodos centinela, pueden aparecer bucles infinitos. Un error bastante común es comprobar si un nodo es `null` cuando lo que se está empleando es un nodo centinela `nullNode`.



## Internet

Todo el código de este capítulo está disponible en línea.

### **BinarySearchTree.java**

Contiene la implementación de `BinarySearchTree`; `BinaryNode.java` contiene la declaración de nodo.

### **BinarySearchTreeWithRank.java**

Añade estadísticas de orden.

### **Rotations.java**

Contiene las rotaciones básicas, en forma de métodos estáticos.

### **RedBlackTree.java**

Contiene la implementación de la clase `RedBlackTree`.

### **AATree.java**

Contiene la implementación de la clase `AATree`.

### **TreeSet.java**

Contiene la implementación de la clase `TreeSet`.

### **MapImpl.java**

Contiene la clase abstracta `MapImpl`.

### **TreeMap.java**

Contiene la implementación de la clase `TreeMap`.



## Ejercicios

### EN RESUMEN

- 19.1 Dibuje todos los árboles AVL que pueden resultar de la inserción de permutaciones de 1, 2 y 3. ¿Cuántos árboles habrá? ¿Cuáles son las probabilidades de obtener cada árbol, si todas las permutaciones son equiprobables?
- 19.2 Repita el Ejercicio 19.1 para cuatro elementos.
- 19.3 Repita los Ejercicios 19.1 y 19.2 para un árbol rojo-negro.
- 19.4 Muestre el resultado de insertar 3, 1, 4, 6, 9, 2, 5 y 7 en un árbol de búsqueda binaria inicialmente vacío. A continuación muestre el resultado de borrar la raíz.
- 19.5 Dibuje todos los árboles de búsqueda binaria que pueden resultar de la inserción de permutaciones de 1, 2, 3 y 4. ¿Cuántos árboles habrá? ¿Cuáles son las probabilidades de obtener cada árbol, si todas las permutaciones son equiprobables?

- 19.6** Muestre que el resultado de insertar 2, 5, 4, 1, 9, 3, 6 y 7 en un árbol AVL inicialmente vacío. Después muestre el resultado para un árbol rojo-negro de procesamiento arriba-abajo.

### EN TEORÍA

- 19.7** Muestre el resultado de insertar los elementos 1 a 15 en orden en un árbol AVL inicialmente vacío. Generalice este resultado (con la consiguiente demostración) para demostrar qué sucede con los elementos 1 a  $2^k - 1$  cuando se insertan en un árbol AVL inicialmente vacío.
- 19.8** Demuestre el Teorema 19.2.
- 19.9** Demuestre que todo árbol AVL se puede colorear como un árbol rojo-negro. ¿Satisfacen todos los árboles rojo-negro la propiedad del árbol AVL?
- 19.10** Proporcione un algoritmo para realizar la operación `remove` en un árbol AVL.
- 19.11** Demuestre que la altura de un árbol rojo-negro es como máximo aproximadamente igual a  $2 \log N$  y proporcione una secuencia de inserción que permita alcanzar dicha cota.

### EN LA PRÁCTICA

- 19.12** Escriba un método para un árbol de búsqueda binaria que admita dos claves, `low` y `high`, e imprima todos los elementos  $X$  que se encuentran en el rango especificado por `low` y `high`. Su programa debería ejecutarse en un tiempo promedio  $O(K + \log N)$ , donde  $K$  es el número de claves impresas. Por tanto, si  $K$  es pequeña, solo se debería examinar una pequeña parte del árbol. Utilice un método recursivo oculto y no emplee un iterador en orden. Acote el tiempo de ejecución de su algoritmo.
- 19.13** Implemente de forma no recursiva `findKth`, utilizando la misma técnica empleada para una operación `find` no recursiva.
- 19.14** Una representación alternativa que permite la operación `findKth` consiste en almacenar en cada nodo el valor del tamaño del subárbol izquierdo más 1. ¿Por qué sería ventajosa esta técnica? Escriba de nuevo la clase de árbol de búsqueda para emplear esta representación.
- 19.15** Escriba un método para un árbol de búsqueda binaria que tome dos enteros, `low` y `high`, y construya un árbol de búsqueda binaria `BinarySearchTreeWithRank` óptimamente equilibrado que contenga todos los enteros comprendidos entre `low` y `high`, ambos incluidos. Todas las hojas deben estar al mismo nivel (si el tamaño del árbol es 1 menos que una potencia de 2) o en dos niveles consecutivos. *Su rutina debe ejecutarse en un tiempo lineal.* Pruebe su rutina utilizándola para resolver el problema de Josefo presentado en la Sección 13.1.
- 19.16** Implemente recursivamente `find`, `findMin` y `findMax`.

### PROYECTOS DE PROGRAMACIÓN

- 19.17** Escriba un programa para evaluar empíricamente las siguientes estrategias para eliminar nodos con dos hijos. Recuerde que una estrategia implica sustituir el

- valor de un nodo borrado por algún otro valor. ¿Qué estrategia proporciona el mejor equilibrado? ¿Cuál requiere menos tiempo de procesador para procesar una secuencia completa de operaciones?
- Sustituya por el valor del nodo mayor,  $X$ , en  $T_L$  y elimine recursivamente  $X$ .
  - Alternativamente, sustituya por el valor en el nodo mayor de  $T_L$  o el valor en el nodo más pequeño de  $T_R$  y elimine de forma recursiva el nodo apropiado.
- 19.18** Rehaga la clase `BinarySearchTree` para implementar el borrado perezoso. Observe que el hacer esto afecta a todas las rutinas. Especialmente complicadas son `findMin` y `findMax`, que ahora deben implementarse recursivamente.
- 19.19** Implemente el árbol de búsqueda binaria para utilizar una sola comparación de dos vías por nivel para `find`, `insert` y `remove`.
- 19.20** Implemente las operaciones del árbol de búsqueda con estadísticas de orden para el árbol de búsqueda equilibrado que prefiera.
- 19.21** Implemente de nuevo la clase `TreeSet` utilizando enlaces a los padres.
- 19.22** Implemente el método `higher` de `TreeSet`, que devuelve el menor elemento del conjunto que sea estrictamente superior al elemento especificado. A continuación, implemente el método `ceiling`, que devuelve el menor elemento del conjunto que sea mayor o igual que el elemento especificado. Ambas rutinas devuelven `null` si no existe tal elemento.
- 19.23** Implemente el método `lower` de `TreeSet`, que devuelve el mayor elemento del conjunto que sea estrictamente menor que el elemento especificado. A continuación, implemente el método `floor`, que devuelve el mayor elemento del conjunto que sea menor o igual que el elemento especificado. Ambas rutinas devuelven `null` si no existe tal elemento.
- 19.24** Modifique las clases `TreeSet` y `TreeMap` de modo que sus iteradores sean bidireccionales.
- 19.25** Implemente de nuevo la clase `TreeSet` añadiendo a cada nodo dos enlaces: `next` y `previous`, que representen el elemento previo y el elemento siguiente que se obtendrían en un recorrido del árbol en orden. Añada también nodos de cabecera y de cola para evitar los casos especiales relativos a los elementos máximo y mínimo. Esto simplifica considerablemente la implementación del iterador, pero requiere revisar los métodos mutadores.
- 19.26** Implemente el método `descendingSet` de `TreeSet` que devuelve una vista del conjunto cuyos métodos iteradores y `toString` ven los elementos en orden decreciente. Los cambios en el conjunto subyacente deberían reflejarse inmediatamente en la vista, y viceversa.
- 19.27** El método `subList(from,to)` de `List` devuelve una vista de la lista que va desde la posición `from`, inclusive, a la posición `to`, exclusive. Añada `subList` a la implementación de `ArrayList` en el Capítulo 15 (puede modificar `weiss.util.ArrayList` o ampliarla en otro paquete, o ampliar `java.util.ArrayList`). En la librería Java, `subList` está escrita en términos de la interfaz `List`, pero escriba la suya en términos de `ArrayList`. Necesitará definir una clase anidada

`SubList` que amplíe `ArrayList`, y hacer que `subList` devuelva una referencia a una nueva instancia de `SubList`. Para que las cosas sean más sencillas, haga que `SubList` mantenga una referencia al `ArrayList` principal, que almacene el tamaño de la sublista y que almacene el desplazamiento dentro del `ArrayList` principal. También puede hacer que todos los mutadores de la `SubList` generen una excepción, de modo que la sublista devuelta sea en la práctica inmutable. Su clase `SubList` debería proporcionar una clase interna `SubListIterator` que implemente `weiss.util.ListIterator`. Pruebe su código con el método de la Figura 19.89. Observe que las sublistas pueden crear sublistas, todas las cuales hacen referencia a porciones más pequeñas del mismo `ArrayList` principal. ¿Cuál es el tiempo de ejecución de su código?

- 19.28** Utilice el método `subList` que ha escrito para implementar el algoritmo recursivo de suma máxima de subsecuencia contigua de la Figura 7.20 para listas `ArrayList`. Utilice iteradores en las dos sublistas para gestionar los dos bucles. Observe que hasta Java 6, la implementación será bastante lenta si utiliza `java.util`, pero si el Ejercicio 19.27 se ha implementado razonablemente, su código debería tener un rendimiento comparable con la implementación de la Figura 7.20.
- 19.29** Amplíe el Ejercicio 19.27 para añadir código que compruebe la existencia de modificaciones estructurales en el `ArrayList` principal. Una modificación estructural en el `ArrayList` principal invalida todas las sublistas. Después, añada comprobaciones de la existencia de modificaciones estructurales en una sublista, lo que invalidaría todas las sublistas descendientes.
- 19.30** El método `headSet` se puede utilizar para obtener el rango de un valor `x` en el `TreeSet` `t: t.headSet(x, true).size()`. Sin embargo, no hay ninguna garantía de que esto sea eficiente. Implemente el código de la Figura 19.90 y compruebe el tiempo de

```

1  public static Random r = new Random( );
2
3  public static long sum( ArrayList<Integer> arr )
4  {
5      if( arr.size( ) == 0 )
6          return 0;
7      else
8      {
9          int idx = r.nextInt( arr.size( ) );
10
11         return arr.get( idx ) +
12             sum( arr.subList( 0, idx ) ) +
13             sum( arr.subList( idx + 1, arr.size( ) ) );
14     }
15 }
```

Figura 19.89 Recursión y sublistas de matrices. Un ejemplo para el Ejercicio 19.27.

```
1  public static void testRank( int N )
2  {
3      TreeSet<Integer> t = new TreeSet<Integer>();
4
5      for( int i = 1; i <= N; i++ )
6          t.add( i );
7
8      for( int i = 1; i <= N; i++ )
9          if( t.headSet( i, true ).size( ) != i )
10             throw new IllegalStateException();
11 }
```

Figura 19.90 Comprueba la velocidad de un cálculo de rango (Ejercicio 19.30).

ejecución para diversos valores de  $N$ . ¿Qué puede decir acerca del coste de obtener el rango de un valor  $x$ ? Describa cómo mantener el `TreeSet` y las vistas de modo que el coste de `size` y por tanto el coste de calcular un rango sea  $O(\log N)$ .



## Referencias

Puede encontrar más información sobre árboles de búsqueda binaria, y en particular sobre las propiedades matemáticas de los árboles en [18] y [19].

Diversos artículos tratan con la falta teórica de equilibrio provocada por los algoritmos de borrado sesgados en los árboles de búsqueda binaria. Hibbard [16] propuso el algoritmo de borrado original y estableció que un borrado preserva la aleatoriedad de los árboles. Solo se ha realizado un análisis completo para árboles con tres nodos [17] y cuatro nodos [3]. Eppinger [10] proporcionó evidencias empíricas tempranas de no aleatoriedad y Culberson y Munro [7] y [8] proporcionaron algunas evidencias analíticas, pero no una demostración completa para el caso general de inserciones y borrados entremezclados. La afirmación de que el nodo más profundo en un árbol de búsqueda binaria aleatorio es tres veces más profundo que el nodo típico está demostrada en [11]; el resultado no es en absoluto sencillo.

Adelson-Velskii y Landis propusieron los árboles AVL [1]. En [19] se presenta un algoritmo de borrado. El análisis de los costes medios de búsqueda en un árbol AVL está incompleto, pero en [20] se proporcionan algunos resultados. El algoritmo para árboles rojo-negro con procesamiento arriba-abajo está tomado de [15]; en [21] se proporciona una descripción más accesible. En [12] se incluye una implementación de árboles rojo-negro con procesamiento arriba-abajo sin nodos centinela; proporciona una demostración convincente de la utilidad de `nullNode`. El árbol AA está basado en el árbol-B binario simétrico presentado en [4]. La implementación mostrada en el texto está adaptada a partir de la descripción de [2]. En [13] se describen muchos otros árboles de búsqueda equilibrados.

Los árboles-B aparecieron por primera vez en [5]. La implementación descrita en el artículo original permite almacenar datos en los nodos internos, además de en las hojas. La estructura de datos descrita aquí se denomina en ocasiones árbol-B<sup>+</sup>. Puede encontrar en [9] la información sobre el árbol-B\*, descrito en el Ejercicio 19.14. En [6] se proporciona un repaso de los diferentes tipos de árboles-B. En [14] se incluyen resultados empíricos para los diversos esquemas. En [12] puede encontrar una implementación C++.

1. G. M. Adelson-Velskii y E. M. Landis, "An Algorithm for the Organization of Information", *Soviet Math. Doklady* 3 (1962), 1259–1263.
2. A. Andersson, "Balanced Search Trees Made Simple", *Proceedings of the Third Workshop on Algorithms and Data Structures* (1993), 61–71.
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. Weiss\_4e\_19. fm Pág. 770 viernes, 4 de septiembre de 2009 9:58 AM references 771
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. Flammig, *Practical Data Structures in C++*, John Wiley & Sons, New York, NY, 1994.
13. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 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, "Some 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: Vol. 1: Fundamental Algorithms*, 3<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1997.
19. D. E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching*, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1998.
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++*, Partes 1–4 (Fundamental Algorithms, Data Structures, Sorting, Searching) 3<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1998.



## Tablas hash

En el Capítulo 19 hemos hablado del árbol de búsqueda binaria que nos permite realizar diversas operaciones sobre un conjunto de elementos. En este capítulo vamos a ver la tabla hash, que solo soporta un subconjunto de las operaciones permitidas por los árboles de búsqueda binaria. La implementación de las tablas hash se denomina frecuentemente *hashing* e incluye inserciones, borrados y búsquedas en un tiempo medio constante.

A diferencia del árbol de búsqueda binaria, el tiempo de ejecución promedio de las operaciones con una tabla hash se basa en propiedades estadísticas, más que en las expectativas de disponer de entradas con apariencia aleatoria. Esta mejora se obtiene a expensas de un pérdida de la información de ordenación correspondiente a los elementos: no se soportan en un tiempo lineal operaciones tales como `findMin` y `findMax` o la impresión de una tabla completa en orden. En consecuencia, la tabla hash y el árbol de búsqueda binaria tienen aplicaciones y propiedades de rendimiento algo diferentes.

En este capítulo veremos

- Varios métodos de implementar la tabla hash.
- Comparaciones analíticas entre estos métodos.
- Algunas aplicaciones de las tablas hash.
- Comparaciones entre las tablas hash y los árboles de búsqueda binaria.

### 20.1 Ideas básicas

La *tabla hash* soporta la extracción o borrado de cualquier elemento nombrado. Queremos podemos soportar las operaciones básicas en un tiempo constante, al igual que para la pila y la cola. Puesto que los accesos están mucho menos restringidos, este soporte parece casi un objetivo imposible de alcanzar. Es decir, con toda seguridad, cuando el tamaño del conjunto se incremente, las búsquedas en el mismo deberán requerir más tiempo. Sin embargo, veremos que esto no es necesariamente así.

Suponga que todos los elementos con los que estamos tratando son enteros pequeños no negativos, comprendidos entre 0 y 65.535. Podemos utilizar una matriz simple para implementar

La tabla hash se utiliza para implementar un conjunto en tiempo constante por cada operación.

cada operación de la forma siguiente. En primer lugar, inicializamos una matriz `a` indexada entre 0 y 65.535 con todo 0s. Para realizar `insert(i)`, ejecutamos `a[i]++`. Observe que `a[i]` representa el número de veces que `i` ha sido insertado. Para realizar `find(i)`, verificamos que `a[i]` no sea 0. Para realizar `remove(i)`, nos aseguramos de que `a[i]` sea positivo y luego ejecutamos `a[i]--`. El tiempo requerido por cada operación es obviamente constante; incluso el coste adicional de inicialización de la matriz requiere una cantidad constante de trabajo (65.536 asignaciones).

Hay dos problemas con esta solución. En primer lugar, suponga que tuviéramos enteros de 32 bits, en lugar de enteros de 16 bits. Entonces, la matriz `a` debería contener 4.000 millones de elementos, lo que resulta complicado desde el punto de vista práctico. En segundo lugar, si los elementos no son enteros, sino cadenas de caracteres (o incluso algo más genérico), no pueden utilizarse para indexar una matriz. El segundo problema no es, en realidad, un problema en absoluto. Al igual que un número 1234 es una colección de dígitos 1, 2, 3 y 4, la cadena de caracteres "junk" es una colección de caracteres 'j', 'u', 'n' y 'k'. Observe que el número 1234 es simplemente  $1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$ . Recuerde de la Sección 12.1 que un carácter ASCII se puede representar típicamente con 7 bits como un número comprendido entre 0 y 127. Puesto que un carácter es básicamente un entero de pequeño tamaño, podemos interpretar una cadena de caracteres como un entero. Una posible representación sería '`j`' ·  $128^3$  + '`u`' ·  $128^2$  + '`n`' ·  $128^1$  + '`k`' ·  $128^0$ . Esta técnica nos permitiría emplear la implementación basada en una matriz simple que hemos analizado anteriormente.

Una función hash convierte el elemento en un entero que sea adecuado para indexar una matriz en la que se almacene el elemento. Si la función hash fuera biunívoca podríamos acceder al elemento empleando su índice matricial.

El problema con esta estrategia es que la representación en forma de entero que acabamos de describir nos proporciona números enteros de un tamaño enorme. La representación para "junk" nos da 224.229.227, y las cadenas de mayor longitud generan representaciones todavía mayores. Este resultado nos devuelve al primero de los problemas mencionados: ¿cómo evitamos utilizar una matriz de un tamaño absurdamente grande?

Para hacerlo, empleamos una función que asigna los números de gran tamaño (o las cadenas de caracteres interpretadas como números) a números más pequeños y manejables. Una función que establezcan una correspondencia entre un elemento y un índice de pequeño tamaño se conoce con el nombre de *función hash*. Si `x` es un entero arbitrario (no negativo), entonces `x%tableSize` genera un número comprendido entre 0 y `tableSize-1` que es adecuado para indexar en una matriz de tamaño `tableSize`. Si `s` es una cadena de caracteres, podemos convertir `s` en un entero de gran tamaño `x` utilizando el método sugerido anteriormente y luego aplicar el operador mod (%) para obtener un índice adecuado. Así, si `tableSize` es 10.000, "junk" se indexaría como 9.227. En la Sección 20.2 hablaremos detalladamente de la implementación de la función hash para cadenas de caracteres.

El uso de la función hash introduce una complicación: puede que haya dos o más elementos distintos a los que les corresponda la misma posición, provocando una *colisión*. Esta situación no se puede evitar nunca, porque hay muchos más elementos que posiciones disponibles. Sin embargo, existen muchos métodos para resolver rápidamente una colisión. Investigaremos tres de los métodos más sencillos: sondeo lineal, sondeo cuadrático y encadenamiento separado. Cada uno de estos métodos es simple de implementar, pero sus respectivos rendimientos son distintos, dependiendo de lo llena que esté la matriz.

Puesto que la función hash no es biunívoca, varios elementos se corresponderán con un mismo índice y provocarán una colisión.

## 20.2 Función hash

Calcular la función hash para cadenas de caracteres presenta una sutil complicación: la conversión de la `String` `s` a `x` genera un entero que casi con seguridad será mayor que los valores que la máquina permite almacenar, porque  $128^4 = 2^{28}$ . Este tamaño de entero es solo 8 veces menor que el valor `int` más grande permitido. En consecuencia, no podemos esperar obtener el valor de la función hash calculando una potencia de 128. En lugar de ello, utilizamos la siguiente observación. Un polinomio genérico

$$A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0 \quad (20.1)$$

puede evaluarse como

$$((A_3X + A_2)X + A_1)X + A_0 \quad (20.2)$$

Observe que en la Ecuación 20.2 evitamos calcular el polinomio directamente, lo cual resulta conveniente por tres razones distintas. En primer lugar, evita tener un resultado intermedio de gran tamaño, el cual, como ya hemos visto, provocaría un desbordamiento. En segundo lugar, el cálculo de esa ecuación implica únicamente tres multiplicaciones y tres sumas; un polinomio de grado  $N$  se calcula en  $N$  multiplicaciones y sumas. Estas operaciones son menos costosas que el cálculo de la Ecuación 20.1. En tercer lugar, el cálculo se hace de izquierda a derecha ( $A_3$  se corresponde con ‘j’,  $A_2$  se corresponde con ‘u’, etc. y  $X$  es 128).

Utilizando un truco podemos evaluar la función hash de manera eficiente y sin desbordamiento.

Sin embargo, sigue existiendo un problema de desbordamiento. El resultado del cálculo sigue siendo el mismo y es probable que sea demasiado grande. Pero aquí hay que recordar que solo necesitamos el resultado `mod tableSize`. Aplicando el operador `%` después de cada multiplicación (o suma), podemos cerciorarnos de que los resultados intermedios sigan siendo pequeños.<sup>1</sup> La función hash resultante se muestra en la Figura 20.1. Una desventaja de esta función hash es que el cálculo del operador `mod` es costosa. Pero, como el desbordamiento está permitido (y sus resultados son coherentes en una plataforma determinada), podemos acelerar un poco más la función hash realizando una única operación `mod` inmediatamente antes de volver. Lamentablemente, la multiplicación repetida por 128 tendería a desplazar los primeros caracteres hacia la izquierda, fuera de la respuesta. Para aliviar este problema, multiplicamos por 37 en lugar de por 128, lo que ralentiza el desplazamiento de los primeros caracteres.

El resultado se muestra en la Figura 20.2. no se trata necesariamente de la mejor de las funciones posibles. Asimismo, en algunas aplicaciones (por ejemplo, si estamos utilizando cadenas de caracteres largas), nos gustaría mejorarla. Sin embargo, hablando en términos generales, la función es bastante buena. Observe que el desbordamiento podría introducir números negativos. Por ello, si la operación `mod` genera un valor negativo, lo hacemos positivo (líneas 15 y 16). Observe también que el resultado obtenido al permitir el desbordamiento y realizar una operación `mod` al final no coincide

La función hash debe ser simple de calcular, pero también debe distribuir las claves de manera equitativa. Si hay demasiadas colisiones, el rendimiento de la tabla hash se verá enormemente afectado.

<sup>1</sup> La Sección 7.4 habla de las propiedades de la operación `mod`.

```

1 // Función hash aceptable
2 public static int hash( String key, int tableSize )
3 {
4     int hashVal = 0;
5
6     for( int i = 0; i < key.length( ); i++ )
7         hashVal = ( hashVal * 128 + key.charAt( i ) )
8                                         % tableSize;
9     return hashVal;
10 }

```

Figura 20.1 Un primer intento de implementar una función hash.

con el que se obtiene al realizar la operación mod después de cada paso. Por tanto, hemos modificado ligeramente la función hash, lo que no es un problema.

Aunque la velocidad es un aspecto importante a la hora de diseñar una función hash, también queremos asegurarnos de que distribuye las claves de manera equitativa. En consecuencia, no debemos llevar demasiado lejos nuestras optimizaciones. Un ejemplo sería la función hash mostrada en la Figura 2.3, que simplemente suma los caracteres que forman la clave y devuelve el resultado mod tableSize. Difícilmente encontraríamos una función más simple. La función es fácil de implementar y calcula un valor hash muy rápidamente. Sin embargo, si tableSize es grande, la función no distribuye las claves demasiado bien. Por ejemplo, suponga que tableSize es 10.000.

```

1 /**
2  * Una rutina hash para objetos String.
3  * @param key el String al que hay que aplicar el hash.
4  * @param tableSize el tamaño de la tabla hash.
5  * @return el valor hash.
6 */
7 public static int hash( String key, int tableSize )
8 {
9     int hashVal = 0;
10
11    for( int i = 0; i < key.length( ); i++ )
12        hashVal = 37 * hashVal + key.charAt( i );
13
14    hashVal %= tableSize;
15    if( hashVal < 0 )
16        hashVal += tableSize;
17
18    return hashVal;
19 }

```

Figura 20.2 Una función hash más rápida que aprovecha el mecanismo de desbordamiento.

```

1 // Una función hash inadecuada si tableSize es grande.
2 public static int hash( String key, int tableSize )
3 {
4     int hashVal = 0;
5
6     for( int i = 0; i < key.length(); i++ )
7         hashVal += key.charAt( i );
8
9     return hashVal % tableSize;
10 }

```

Figura 20.3 Una función hash inadecuada si tableSize es grande.

Suponga también que todas las claves tienen una longitud de 8 caracteres o menos. Puesto que un char ASCII es un entero entre 0 y 127, la función hash solo puede asumir valores comprendidos entre 0 y 1.016 ( $127 \times 8$ ). Esta restricción no permite, obviamente, una distribución equitativa. Cualquier velocidad que hayamos ganado al acelerar el cálculo de la función hash se verá más que compensado por el esfuerzo necesario para resolver un número de colisiones mayor que el esperado. Sin embargo, en el Ejercicio 20.15 se describe una alternativa razonable.

Finalmente, observe que 0 es un posible resultado de la función hash, por lo que las tablas hash se indexan comenzando en 0.

La tabla va de 0 a  
tableSize - 1.

### 20.2.1 hashCode en java.lang.String

En Java, los tipos de la librería que se pueden insertar razonablemente en un HashSet o que se pueden utilizar como clave en un HashMap ya tienen definidos los métodos equals y hashCode. En particular, la clase String tiene un hashCode cuya implementación resulta crítica para el rendimiento de los HashSet y HashMap donde están involucrados objetos String.

La historia del método hashCode de String es en sí misma bastante instructiva. Las primeras versiones de Java utilizaban esencialmente la misma implementación de la Figura 20.2, incluyendo el multiplicador constante 37, pero sin las líneas 14 a 16. Pero posteriormente se “optimizó” la implementación de modo que si la cadena String tenía una longitud mayor de 15 caracteres, solo se utilizaban para calcular el hashCode 8 o 9 caracteres, espaciados de forma más o menos equitativa dentro del objeto String. Esta versión se utilizó en Java 1.0.2 y Java 1.1, pero resultó ser una mala idea, porque había muchas aplicaciones que contenían grandes grupos de objetos String de gran longitud que eran en cierta forma similares. Dos de esos ejemplos eran los mapas en los que se utilizaban como claves direcciones URL como <http://www.cnn.com/> y los mapas en los que se empleaban nombres completos de archivo como

[/a/file.cs.fiu.edu./disk/storage137/user/weiss/public\\_html/dsj4/code.](http://a/file.cs.fiu.edu./disk/storage137/user/weiss/public_html/dsj4/code.)

El rendimiento en estos mapas se degradaba considerablemente, porque las claves tendían a generar un número relativamente bajo de códigos hash distintos.

En Java 1.2, se volvió al método `hashCode` de la versión más simple, empleando 31 como multiplicador constante. No hace falta decir que los programadores que diseñaron la librería Java se cuentan entre los de más talento del mundo, de modo que es fácil comprender que diseñar una función hash de primera categoría es un proceso lleno de trampas y no tan simple como puede parecer.

En Java 1.3, se probó una nueva idea con algo más de éxito. Puesto que la parte más costosa de las operaciones con tablas hash es el cálculo del código `hashCode`, el método `hashCode` de la clase `String` contiene una optimización importante: cada objeto `String` almacena internamente el valor de su `hashCode`. Inicialmente es 0, pero si se invoca `hashCode`, se recuerda posteriormente el valor obtenido. De ese modo, si tratamos de calcular una segunda vez `hashCode` con el mismo objeto `String`, podemos evitar volver a realizar los costosos cálculos. Esta técnica se denomina *almacenamiento en caché del código hash* y representa otro de esos clásicos compromisos entre tiempo y espacio. La Figura 20.4 muestra una implementación de la clase `String` en la que se almacena en caché el código hash.

El almacenamiento en caché del código hash funciona únicamente porque los objetos `String` son inmutables: si permitiéramos que el objeto `String` cambiara, se invalidaría el valor de `hashCode` y tendríamos que volver a asignar a este un valor 0. Aunque está claro que para dos objetos `String` con el mismo estado hay que calcular de manera independiente su código hash, también hay muchas situaciones en las que se consulta una y otra vez el código hash de un mismo objeto `String`. Una situación en la que sirve de ayuda el almacenamiento en caché del código hash es durante el recálculo de la distribución hash de los elementos, porque todos los objetos `String` implicados en ese recálculo ya tendrán almacenados en caché sus códigos hash.

## 20.3 Sondeo lineal

En el *sondeo lineal*, las colisiones se resuelven explorando secuencialmente una matriz (con encadenamiento circular entre el final y el principio), hasta encontrar una celda vacía.

Ahora que disponemos de una función hash, tenemos que decidir qué hacer cuando se produce una colisión. Específicamente, si al aplicar la función hash a  $X$  obtenemos una posición que ya está ocupada, ¿dónde colocamos ese elemento? La estrategia más simple posible es la del *sondeo lineal*, que consiste en buscar secuencialmente en la matriz hasta encontrar una celda vacía. La búsqueda salta circularmente desde la última posición hasta la primera en caso necesario. La Figura 20.5 muestra el resultado de insertar las claves 89, 18, 49, 58 y 9 en una tabla hash, cuando se utiliza un sondeo lineal.

En el ejemplo estamos asumiendo que disponemos de una función hash que devuelve la clave  $X \bmod$  el tamaño de la tabla. La Figura 20.5 incluye el resultado de la función hash.

La primera colisión se produce al insertar 49; el 49 se coloca en la siguiente celda disponible –en concreto, en la celda 0, que está vacía. Después, 58 colisiona con 18, 89 y 49 antes de encontrar una celda vacía, tres posiciones más allá, en la posición 1. La colisión para el elemento 9 se resuelve de forma similar. Mientras que la tabla sea lo suficientemente grande, siempre se podrá encontrar una celda vacía. Sin embargo, el tiempo necesario para encontrar una celda vacía puede llegar a ser muy largo. Por ejemplo, si solo queda una celda vacía en la tabla, puede que tengamos que realizar una búsqueda de tabla completa para encontrarla. En promedio, cabría esperar tener que explorar la mitad de la tabla para encontrarla, lo cual está bastante lejos del tiempo constante por acceso que estábamos intentando obtener. Pero si la tabla está relativamente vacía, las inserciones no deberían ser demasiado costosas. Enseguida analizaremos esta técnica.

```

1  public final class String
2  {
3      public int hashCode( )
4      {
5          if( hash != 0 )
6              return hash;
7
8          for( int i = 0; i < length( ); i++ )
9              hash = hash * 31 + (int) charAt( i );
10         return hash;
11     }
12
13     private int hash = 0;
14 }

```

Figura 20.4 Extracto del método hashCode de la clase String.

hash ( 89, 10 ) = 9	Después de Insertar 89	Después de Insertar 18	Después de Insertar 49	Después de Insertar 58	Después de Insertar 9
	0		49	49	49
	1			58	58
	2				9
	3				
	4				
	5				
	6				
	7				
	8	18	18	18	18
	9	89	89	89	89

Figura 20.5 Tabla hash con sondeo lineal después de cada inserción.

El algoritmo `find` simplemente sigue la misma técnica que el algoritmo `insert`. Si llega a una celda vacía, querrá decir que el elemento que estamos buscando no se ha encontrado; en caso contrario, terminará por encontrar la correspondencia deseada. Por ejemplo, para encontrar 58, comenzamos por la celda 8 (tal como nos ha indicado la función hash). Vemos allí un elemento, pero no es el que estamos buscando, así que probamos con la celda 9. De nuevo, tenemos

El algoritmo `find` sigue la misma secuencia de sondeo que el algoritmo `insert`.

un elemento, pero no es el deseado, así que buscamos en la celda 0 y luego en la celda 1, hasta encontrar una correspondencia. Una operación `find` para el valor 19 implicaría probar las celdas 9, 0, 1 y 2 antes de encontrar una posición vacía en la celda 3. Eso querrá decir que no se ha encontrado el valor 19.

No se puede realizar un borrado estándar porque, como sucede con los árboles de búsqueda binaria, cada elemento de la tabla hash no solo se representa a sí mismo, sino que también conecta entre sí otros elementos, sirviendo como posible celda de almacenamiento durante la resolución de colisiones. Por tanto, si eliminamos 89 de la tabla hash, casi todas las restantes operaciones `find`

**Se debe utilizar un borrado perezoso.**

fallarán. En consecuencia, lo que hacemos es implementar un mecanismo de *borrado perezoso*, que consiste en marcar los elementos como borrados en lugar de eliminarlos físicamente de la tabla. Esta información se almacena en un miembro de datos adicional. Cada uno de los elementos de la tabla podrá estar *activo* o *borrado*.

### 20.3.1 Un análisis simplista del sondeo lineal

El análisis simplista del sondeo lineal se basa en la suposición de que los sondeos sucesivos son independientes. Esta suposición no es cierta y, por tanto, el análisis subestima el coste de las búsquedas y de las inserciones.

El factor de carga de una tabla hash con sondeo es la fracción de la tabla que está llena. Varía entre 0 (vacía) y 1 (llena).

Para estimar el rendimiento del sondeo lineal, hacemos dos suposiciones:

1. La tabla hash es de gran tamaño.
2. Cada sondeo de la tabla hash es independiente del sondeo anterior.

La primera suposición es razonable; en caso contrario, estaríamos perdiendo el tiempo con una tabla hash. La segunda suposición dice que si la fracción de la tabla que está llena es  $\lambda$ , cada vez que examinemos una celda la probabilidad de que esté ocupada también será  $\lambda$ , independientemente de las operaciones de sondeo anteriores. La independencia es una importante propiedad estadística, que simplifica enormemente el análisis de sucesos aleatorios. Lamentablemente, como se explica en la Sección 20.3.2, la suposición de independencia no solo no está justificada, sino que es totalmente errónea. Por tanto, el análisis simplista que vamos a realizar es incorrecto. Aun así es útil realizarlo porque nos dice lo que cabe esperar conseguir si tenemos más cuidado con respecto al modo de resolver las colisiones. Como hemos mencionado anteriormente en el capítulo, el rendimiento de la tabla hash dependerá de lo llena que esté. El grado en el que la tabla esté llena vendrá dado por el denominado factor de carga.

**Definición:** El factor de carga,  $\lambda$ , de una tabla hash con sondeo es la fracción de la tabla que está llena. El factor de carga varía entre 0 (vacía) y 1 (completamente llena).

Con esto podemos proporcionar un análisis simple, pero incorrecto, del sondeo lineal en el Teorema 20.1.

#### Teorema 20.1

Si asumimos que los sondeos son independientes, el número medio de celdas examinadas durante una inserción utilizando sondeo lineal es  $1/(1 - \lambda)$ .

#### Demostración

Para un tabla con un factor de carga igual a  $\lambda$ , la probabilidad de que cualquier celda esté vacía es  $1 - \lambda$ . En consecuencia, el número esperado de intentos independientes requeridos para encontrar una celda vacía será  $1/(1 - \lambda)$ .

En la demostración del Teorema 20.1 utilizamos el hecho de que, si la probabilidad de que un cierto suceso ocurra es  $p$ , entonces se requiere en promedio  $1/p$  intentos hasta que se produce el suceso, suponiendo que los intentos sean independientes entre sí. Por ejemplo, el número esperado de veces que tendremos que arrojar una moneda hasta obtener cara es dos, y el número esperado de veces que tendremos que arrojar un dado de seis caras hasta obtener un 4 es seis, suponiendo que los intentos sean independientes.

### 20.3.2 Lo que realmente sucede: agrupamiento primario

Lamentablemente, la suposición de independencia no se cumple como se muestra en la Figura 20.6. La parte (a) de la figura muestra el resultado de llenar una tabla hash hasta un 70 por ciento de capacidad, si todos los sondeos sucesivos son independientes. La parte (b) muestra el resultado de la técnica de sondeo lineal. Observe la serie de agrupamientos. Este fenómeno se conoce con el nombre de *agrupamiento primario* o *clustering primario*.

En el fenómeno del agrupamiento primario se forman grandes bloques de celdas ocupadas. Cualquier clave cuya función hash la asigne a uno de estos agrupamientos requerirá un número excesivo de intentos para resolver la colisión, después de lo cual contribuirá a aumentar el tamaño del agrupamiento. No solo los elementos que colisionan debido a que los resultados de aplicarles las funciones hash coinciden provocan un rendimiento degenerado, sino que también un elemento que colisione con una posición alternativa para otro elemento hará que el rendimiento se reduzca. El análisis matemático requerido para tener en cuenta este fenómeno es complejo, pero ha sido resuelto, proporcionándonos el Teorema 20.2.

El efecto del agrupamiento primario es la formación de grandes agrupamientos de celdas ocupadas que hacen que las inserciones en el agrupamiento sean costosas (y además la inserción hace que el agrupamiento sea aun mayor).

#### Teorema 20.2

El número medio de celdas examinadas en una inserción utilizando el sondeo lineal es aproximadamente  $(1 + 1/(1 - \lambda)^2)/2$ .

#### Demostración

La demostración queda fuera del alcance de este texto. Consulte la referencia [6].



Figura 20.6 Ilustración del fenómeno del agrupamiento primario en el sondeo lineal (b), comparado con la ausencia de agrupamiento (a) y el agrupamiento secundario, menos significativo en el sondeo cuadrático (c). Las líneas representan celdas ocupadas y el factor de carga es 0,7.

La agrupación primaria es un problema para factores de carga altos. Para tablas medio llenas el efecto no es desastroso.

Para una tabla semillena, obtenemos 2,5 como número medio de celdas examinadas durante una inserción. Este resultado es casi el mismo que lo que indicaba el análisis simplista. La principal diferencia se manifiesta a medida que  $\lambda$  se aproxima a 1. Por ejemplo, si la tabla está llena al 90 por ciento,  $\lambda = 0,9$ . El análisis simplista sugiere que habría que examinar 10 celdas –lo que es bastante, pero tampoco excesivo. Sin embargo, según el Teorema 20.2, la respuesta real es que hará falta examinar unas 50 celdas. Eso es excesivo (especialmente porque este número es solo un promedio, con lo que algunas inserciones deben ser aun peores).

### 20.3.3 Análisis de la operación `find`

Una operación `find` que no tenga éxito cuesta lo mismo que una inserción.

El coste de una operación `find` que tenga éxito es un promedio de los costes de inserción para todos los factores de carga más pequeños.

El coste de una inserción se puede utilizar para obtener una cota del coste de una operación `find`. Hay dos tipos de operación `find`: las que tienen éxito y las que no. Una operación `find` que no tenga éxito es fácil de analizar. La secuencia de celdas examinadas para una búsqueda sin éxito del elemento  $X$  es la misma que la secuencia examinada para insertar  $X$  con la operación `insert`. Por tanto, tenemos una respuesta inmediata en lo que se refiere al coste de una operación `find` que no tenga éxito.

Para las operaciones `find` que sí tengan éxito, las cosas son ligeramente más complicadas. La Figura 20.5 muestra una tabla con  $\lambda = 0,5$ . Así, el coste medio de una inserción es 2,5. El coste medio para encontrar con `find` el elemento recién insertado sería entonces 2,5, independientemente de cuántas inserciones se produzcan después. El coste medio para encontrar el primer elemento insertado en la tabla será siempre 1,0. Por tanto, en una tabla con  $\lambda = 0,5$ , algunas búsquedas serán sencillas y otras más difíciles. En particular, el coste de una búsqueda del elemento  $X$  que tenga éxito es igual al coste de insertar  $X$  en el momento en que  $X$  fue insertado. Para calcular el tiempo medio requerido para realizar una búsqueda con éxito en una tabla con un factor de carga  $\lambda$ , debemos calcular el coste medio de inserción, promediando entre todos los factores de carga que han terminado dándonos un factor  $\lambda$ . Con este trabajo preliminar, podemos calcular los tiempos medios de búsqueda para el sondeo lineal, como se enuncia y se demuestra en el Teorema 20.3.

#### Teorema 20.3

El número medio de celdas examinadas en una búsqueda que no tenga éxito utilizando un sondeo lineal es aproximadamente  $(1 + 1/(1 - \lambda))^2/2$ . El número medio de celdas examinadas en una búsqueda que tenga éxito es aproximadamente  $(1 + 1/(1 - \lambda))/2$ .

#### Demostración

El coste de una búsqueda sin éxito coincide con el coste de una inserción. Para una búsqueda con éxito, calculamos el coste medio de inserción para toda la secuencia de inserciones. Puesto que la tabla es grande, podemos calcular este promedio evaluando la integral

$$S(\lambda) = \frac{1}{\lambda} \int_{x=0}^{\lambda} I(x) dx$$

Continúa

Demostración  
(cont.)

En otras palabras, el coste medio de una búsqueda que tenga éxito para una tabla con un factor de carga  $\lambda$  es igual al coste de una inserción en una tabla con factor de carga  $x$ , promediado entre los factores de carga 0 a  $\lambda$ . A partir del Teorema 20.2, podemos obtener la siguiente ecuación:

$$\begin{aligned} S(\lambda) &= \frac{1}{\lambda} \int_{x=0}^{\lambda} \frac{1}{2} \left( 1 + \frac{1}{(1-x)^2} \right) dx \\ &= \frac{1}{2\lambda} \left( x + \frac{1}{(1-x)} \right) \Big|_{x=0}^{\lambda} \\ &= \frac{1}{2\lambda} \left( \left( \lambda + \frac{1}{(1-\lambda)} \right) - 1 \right) \\ &= \frac{1}{2} \left( \frac{2-\lambda}{1-\lambda} \right) \\ &= \frac{1}{2} \left( 1 + \frac{1}{\lambda-1} \right) \end{aligned}$$

Podemos aplicar la misma técnica para obtener el coste de una operación `find` con éxito utilizando la suposición de independencia entre sucesos (utilizando  $I(x) = 1/(1-x)$  en el Teorema 20.3). Si no se produce agrupamiento, el coste medio de una operación `find` con éxito para el caso de sondeo lineal es  $-\ln(1-\lambda)/\lambda$ . Si el factor de carga es 0,5, el número promedio de sondeos para una búsqueda con éxito utilizando sondeo lineal es 1,5, mientras que el análisis sin agrupamiento sugiere 1,4 sondeos. Observe que este promedio no depende de la ordenación de las clases de entrada; depende únicamente de la equitativa que sea la función hash. Observe también que, aun cuando dispongamos de buenas funciones hash, habrá secuencias de sondeo tanto cortas como largas que contribuyan al promedio. Por ejemplo, es seguro que existirán algunas secuencias de longitud 4, 5 y 6, incluso en una tabla hash que esté medio llena. (Determinar la secuencia de sondeo más larga esperada requiere cálculos enormemente complicados.) El fenómeno del agrupamiento primario no solo hace que la secuencia promedio de sondeos sea más larga, sino que también hace que una secuencia larga de sondeos sea más probable. El principal problema con el agrupamiento primario es, por tanto, que el rendimiento se degrada enormemente cuando se realizan inserciones con factores de carga altos. Asimismo, algunas de las secuencias de sondeo más largas con las que típicamente nos toparemos (las que se encuentran por encima de la media) también tienen una probabilidad mayor de presentarse.

Para reducir el número de sondeos, necesitamos un esquema de resolución de colisiones que evite la aparición del fenómeno del agrupamiento primario. Sin embargo, observe que si la tabla está medio llena, eliminar los efectos del agrupamiento primario solo permitiría ahorrar la mitad de un sondeo como promedio para una operación de inserción o una búsqueda que no tenga éxito, y que solo permitiría ahorrar un décimo de sondeo como promedio en una búsqueda que tenga éxito. Aunque cabría esperar que se reduzca la probabilidad de obtener una secuencia de sondeo algo más larga, *el sondeo lineal no es una estrategia tan mala*. Dado que es tan fácil de implementar, cualquier método que empleemos para eliminar el fenómeno del agrupamiento primario debe tener una complejidad comparable. En caso contrario, estaríamos invirtiendo demasiado tiempo en

ahorrar únicamente una fracción de sondeo. Uno de esos métodos alternativos sencillos es el *sondeo cuadrático*.

## 20.4 Sondeo cuadrático

**El sondeo cuadrático**  
examina las celdas que  
están situadas a una  
distancia de 1, 4, 9, etc. del  
punto de sondeo original.

El *sondeo cuadrático* es un método de resolución de colisiones que elimina el problema del agrupamiento primario que presenta el sondeo lineal. Lo consigue examinando determinadas celdas situadas a una cierta distancia del punto de sondeo original. Su nombre se deriva del uso de la fórmula  $F(j) = j^2$  para resolver las colisiones. Específicamente, si la función hash se evalúa como  $H$  y una búsqueda en la celda  $H$  no permite finalizar la operación, probamos sucesivamente con las celdas  $H + 1^2, H + 2^2, H + 3^2, \dots, H + i^2$  (con encadenamiento circular entre el final y el principio de la tabla). Esta estrategia difiere de la estrategia de sondeo lineal consistente en buscar en  $H + 1, H + 2, H + 3, \dots, H + i$ .

La Figura 20.7 muestra la tabla que resulta cuando se emplea el sondeo cuadrático en lugar del sondeo lineal para la secuencia de inserción mostrada en la Figura 20.5. Cuando 49 colisiona con 89, la primera alternativa que se intenta está situada a una celda de distancia. Esta celda está vacía, por lo que se coloca allí el 49. A continuación, el 58 colisiona en la posición 8. Se prueba con la celda situada en la posición 9 (que está a una celda de distancia), pero se produce otra colisión. Se encuentra una celda vacante en la siguiente celda que se prueba, que está a  $2^2 = 4$  posiciones de distancia, con respecto a la posición hash original. De ese modo, el 58 se coloca en la celda 2. Lo mismo sucede con el 9. Observe que las posiciones alternativas para los elementos a los que les

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash ( 9, 10 ) = 9
```

	Después de insertar 89	Después de insertar 18	Después de insertar 49	Después de insertar 58	Después de insertar 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figura 20.7 Una tabla hash con sondeo cuadrático después de cada inserción (observe que el tamaño de la tabla se ha elegido descuidadamente, porque no es un número primo).

corresponde un resultado hash de 8 y las posiciones alternativas para los elementos a los que les corresponde un resultado hash de 9 no coincide. La larga secuencia de sondeo para insertar 58 no tuvo ningún efecto sobre la inserción subsiguiente de 9, lo que contrasta con lo que sucedía en el caso del sondeo lineal.

Tenemos que tomar en consideración algunos cuantos detalles antes de escribir el código.

Recuerde que los puntos de sondeo posteriores están a un número cuadrático de posiciones de distancia con respecto al punto de sondeo original.

- En el sondeo lineal, cada sondeo prueba con una celda diferente. ¿Está garantizado en el sondeo cuadrático que, cuando se pruebe celda, no la hayamos probado ya durante el curso del acceso actual? ¿Está garantizado en el sondeo cuadrático que, cuando estemos insertando  $X$  y la tabla no esté llena, terminaremos insertando  $X$ ?
- El sondeo lineal se puede implementar fácilmente. El sondeo cuadrático parece requerir operaciones de multiplicación y de módulo. ¿Esta aparente complejidad añadida hace que el sondeo cuadrático resulte poco práctico?
- ¿Qué sucede (tanto en el sondeo lineal como en el cuadrático) si el factor de carga se hace demasiado alto? ¿Podemos expandir dinámicamente la tabla, como suele hacerse con otras estructuras de datos basadas en matrices?

Afortunadamente, las noticias son relativamente buenas en todos los casos citados. Si el tamaño de la tabla es primo y el factor de carga nunca es mayor de 0,5, siempre podremos insertar un nuevo elemento  $X$  y ninguna celda será sondeada dos veces durante un mismo acceso. Sin embargo, para que estas garantías se cumplan, tenemos que asegurarnos de que el tamaño de la tabla sea un número primo. Podemos demostrar este caso en el Teorema 20.4. Para ser exhaustivos, la Figura 20.8 muestra una rutina que genera números primos, utilizando el algoritmo mostrado en la Figura 9.8 (no merece la pena utilizar un algoritmo más complejo).

Si el tamaño de la tabla es primo y el factor de carga no es mayor que 0,5, todos los sondeos se realizarán en posiciones diferentes y siempre se podrá insertar un elemento.

#### Teorema 20.4

Si se utiliza el sondeo cuadrático y el tamaño de la tabla es un número primo, entonces siempre se podrá insertar un nuevo elemento si la tabla está como mínimo medio vacía. Además, en el curso de la inserción ninguna celda será sondeada dos veces.

#### Demostración

Sea  $M$  el tamaño de la tabla. Supongamos que  $M$  es un primo impar mayor que 3. Vamos a demostrar que las primeras  $\lceil M/2 \rceil$  posiciones alternativas (incluyendo la original) son distintas. Dos de estas posiciones son  $H + i^2 \pmod{M}$  y  $H + j^2 \pmod{M}$ , donde  $0 \leq i, j \leq \lfloor M/2 \rfloor$ . Vamos a suponer que el teorema no es cierto y que esas dos ubicaciones son la misma y que  $i \neq j$ . Entonces

$$\begin{aligned} H + i^2 &\equiv H + j^2 \pmod{M} \\ i^2 &\equiv j^2 \pmod{M} \\ i^2 - j^2 &\equiv 0 \pmod{M} \\ (i - j)(i + j) &\equiv 0 \pmod{M} \end{aligned}$$

Continúa

### Demostración (cont.)

Puesto que  $M$  es primo, querrá decir que  $i - j$  o  $i + j$  es divisible por  $M$ . Puesto que  $i$  y  $j$  son distintos y su suma es menor que  $M$ , ninguna de esas dos posibilidades puede producirse. Con esto llegamos a una contradicción. De aquí se deduce que las primeras  $\lceil M/2 \rceil$  alternativas (incluyendo la ubicación original) son todas ellas distintas, lo que garantiza que una inserción tenga éxito si la tabla está al menos medio vacía.

Si la tabla está más que medio llena, aunque solo sea por 1 posición, la inserción podría fallar (aunque el fallo es extremadamente improbable). Si hacemos que el tamaño de la tabla sea un número primo y mantenemos el factor de carga por debajo de 0,5, tenemos garantizado el éxito de la inserción. Si el tamaño de la tabla no es primo, el número de ubicaciones alternativas puede verse enormemente reducido. Por ejemplo, si el tamaño de la tabla fuera 16, las únicas ubicaciones alternativas estarían situadas a las distancias 1, 4, o 9 del punto de sondeo original. De nuevo, el tamaño no es realmente un problema: aunque no dispondríamos de  $\lceil M/2 \rceil$  alternativas garantizadas, usualmente tendríamos más de las necesarias. Sin embargo, es mejor jugar sobre seguro y utilizar la teoría como guía durante el proceso de selección de parámetros. Además, se ha demostrado empíricamente que los números primos tienden a ser muy adecuados para las tablas hash, porque tienden a eliminar parte de la no aleatoriedad que en ocasiones introducen las funciones hash.

El sondeo cuadrático se puede implementar sin multiplicaciones ni operaciones módulo. Puesto que no se ve afectado por el fenómeno del agrupamiento primario, resulta mejor que el sondeo lineal en la práctica.

La segunda consideración importante tiene relación con la eficiencia. Recuerde que, para un factor de carga de 0,5, la eliminación del fenómeno del agrupamiento primario solo permite ahorrar 0,5 sondeos para una inserción típica y un 0,1 sondeos para una búsqueda con éxito típica. Es verdad que obtenemos algunas ventajas adicionales: encontrarnos con una secuencia de sondeo larga es significativamente menos probable. Sin embargo, si realizar un sondeo mediante la técnica del sondeo cuadrático requiere el doble tiempo, estará claro que no merece la pena el esfuerzo. El sondeo lineal se implementa mediante una suma simple (sumando 1), mediante una

```

1  /**
2   * Método para encontrar un número primo mayor o igual que n.
3   * @param n el número inicial (tiene que ser positivo).
4   * @return un número primo mayor o igual que n.
5   */
6  private static int nextPrime( int n )
7  {
8      if( n % 2 == 0 )
9          n++;
10
11     for( ; !isPrime( n ); n += 2 )
12         ;
13
14     return n;
15 }
```

Figura 20.8 Una rutina utilizada en el sondeo cuadrático para encontrar un número primo mayor o igual que  $N$ .

comprobación para determinar si hace falta volver al principio de la tabla por haber alcanzado el final y, muy raramente, mediante una resta (si tenemos que volver al principio de la tabla). La fórmula del sondeo cuadrático sugiere que tenemos que sumar 1 (para pasar de  $i - 1$  a  $i$ ), realizar una multiplicación (para calcular  $\lambda^i$ ), hacer otra suma y luego una operación módulo. Ciertamente, este cálculo parece ser demasiado costoso como para resultar práctico. Sin embargo, podemos emplear el siguiente truco, como se explica en el Teorema 20.5.

### Teorema 20.5

El sondeo cuadrático se puede implementar sin costosas multiplicaciones y divisiones.

#### Demostración

Sea  $H_{i-1}$  el sondeo calculado más recientemente ( $H_0$  es la posición hash original) y sea  $H_i$  el sondeo que estamos intentando calcular. Entonces tenemos

$$\begin{aligned} H_i &= H_0 + i^2 \pmod{M} \\ H_{i-1} &= H_0 + (i-1)^2 \pmod{M} \end{aligned} \tag{20.3}$$

Si restamos estas dos ecuaciones, obtenemos

$$H_i = H_{i-1} + 2i - 1 \pmod{M} \tag{20.4}$$

La Ecuación 20.4 nos dice que podemos calcular el nuevo valor  $H_i$  a partir del valor anterior  $H_{i-1}$  sin necesidad de elevar  $i$  al cuadrado. Aunque seguimos teniendo una multiplicación, esa multiplicación es por 2, que se puede implementar de manera trivial mediante un desplazamiento de bits en la mayoría de las computadoras. ¿Y qué pasa con la operación módulo? Esta operación, asimismo, no es realmente necesaria, porque la expresión  $2i - 1$  tiene que ser menor que  $M$ . Por tanto, si la sumamos a  $H_{i-1}$ , el resultado seguirá siendo o bien menor que  $M$  (en cuyo caso no necesitamos calcular el módulo) o solo un poco mayor que  $M$  (en cuyo caso podemos calcular el módulo simplemente restando  $M$ ).

El Teorema 20.5 muestra que podemos calcular la siguiente posición que hay que sondear utilizando una suma (para incrementar  $\lambda$ ), un desplazamiento de bits (para evaluar  $2\lambda$ ), una resta de 1 unidad (para calcular  $2i - 1$ ), otra suma (para incrementar la posición anterior en  $2i - 1$ ), una comprobación para determinar si hay que volver al principio de la tabla y, muy raramente, una resta para implementar la operación módulo. La diferencia es por tanto, un desplazamiento de bits, una resta de 1 unidad y una suma por cada sondeo. El coste de esta operación es probablemente menor que el coste de realizar un sondeo adicional, en el caso de que estén involucradas claves complejas (como cadenas de caracteres).

El detalle final que tenemos que analizar es el de la expansión dinámica. Si el factor de carga es mayor que 0,5, lo que queríamos es duplicar el tamaño de la tabla hash. Pero esta solución plantea unas cuantas cuestiones. En primer lugar, ¿qué dificultad tendrá el encontrar otro número primo? La respuesta es que los números primos son fáciles de encontrar. Esperamos tener que probar solamente  $O(\log N)$  números hasta encontrar un número que sea primo. En consecuencia, la rutina mostrada en la Figura 20.8 es muy rápida. La comprobación de primalidad tarda como mucho un tiempo  $O(N^{1/2})$ .

Expanda la tabla en cuanto el factor de carga alcance el valor 0,5; a dicho proceso se le denomina *rehashing*. Efectúe siempre la duplicación seleccionando un número primo. Los números primos son fáciles de encontrar.

por lo que la búsqueda de un número primo tarda como máximo un tiempo  $O(M^2 \log N)$ .<sup>2</sup> Este coste es mucho menor que el coste  $O(N)$  de transferir el contenido de la tabla antigua a la nueva.

Una vez que hemos asignado una matriz de mayor tamaño, ¿nos limitamos a copiar todos los elementos? La respuesta es por supuesto que no. La nueva matriz implica una nueva función hash, por lo que no podemos emplear las posiciones de la matriz antigua. Por tanto, tenemos que examinar cada elemento de la tabla anterior, calcular su nuevo valor hash e insertarlo en la nueva tabla hash. Este proceso se denomina *rehashing* y se implementa fácilmente en Java.

#### 20.4.1 Implementación Java

El usuario debe proporcionar un método hashCode adecuado para los objetos.

Ahora estamos listos para proporcionar una implementación Java completa de una tabla hash con sondeo cuadrático. Lo haremos implementando la mayor parte de las estructuras HashSet y HashMap de la API de Colecciones. Recuerde que tanto HashSet como HashMap requieren un método hashCode. El método hashCode no tiene parámetro tableSize; los algoritmos para tablas hash realizan una operación módulo al final internamente, después de utilizar la función hash suministrada por el usuario. El esqueleto de clase para HashSet se muestra en la Figura 20.9. Para que los algoritmos funcionen correctamente, equals y hashCode deben ser coherentes. Es decir, si dos objetos son iguales, sus valores hash también deben ser iguales.

La tabla hash está compuesta por una matriz de referencias HashEntry. Cada referencia HashEntry puede ser o null o un objeto que almacene un elemento y un miembro de datos que nos diga si la entrada está activa o borrada. Puesto que las matrices de tipo genérico son ilegales, HashEntry no es genérica. La clase anidada HashEntry se muestra en la Figura 20.10. La matriz se declara en la línea 49. Necesitamos llevar la cuenta tanto del tamaño lógico del HashSet como del número de elementos de la tabla hash (incluyendo los elementos como borrados); estos valores se almacenan en currentSize y occupied, respectivamente, que se declaran en las líneas 46 y 47.

```

1 package weiss.util;
2
3 public class HashSet<AnyType> extends AbstractCollection<AnyType>
4     implements Set<AnyType>
5 {
6     private class HashSetIterator implements Iterator<AnyType>
7         { /* Figura 20.18 */ }
8     private static class HashEntry implements java.io.Serializable
9         { /* Figura 20.10 */ }
10

```

Continúa

Figura 20.9 El esqueleto de clase para una tabla hash con sondeo cuadrático.

<sup>2</sup> Esta rutina también es necesaria si añadimos un constructor que permita al usuario especificar un tamaño inicial aproximado para la tabla hash. La implementación de la tabla hash debe garantizar que se utilice un número primo.

```
11 public HashSet( )
12 { /* Figura 20.11 */ }
13 public HashSet( Collection<? extends AnyType> other )
14 { /* Figura 20.11 */ }
15
16 public int size( )
17 { return currentSize; }
18 public Iterator<AnyType> iterator( )
19 { return new HashSetIterator( ); }
20
21 public boolean contains( Object x )
22 { /* Figura 20.12 */ }
23 private static boolean isActive( HashEntry [ ] arr, int pos )
24 { /* Figura 20.13 */ }
25 public AnyType getMatch( AnyType x )
26 { /* Figura 20.12 */ }
27
28 public boolean remove( Object x )
29 { /* Figura 20.14 */ }
30 public void clear( )
31 { /* Figura 20.14 */ }
32 public boolean add( AnyType x )
33 { /* Figura 20.15 */ }
34 private void rehash( )
35 { /* Figura 20.16 */ }
36 private int findPos( Object x )
37 { /* Figura 20.17 */ }
38
39 private void allocateArray( int arraySize )
40 { array = new HashEntry[ arraySize ]; }
41 private static int nextPrime( int n )
42 { /* Figura 20.8 */ }
43 private static boolean isPrime( int n )
44 { /* Véase el código en linea */ }
45
46 private int currentSize = 0;
47 private int occupied = 0;
48 private int modCount = 0;
49 private HashEntry [ ] array;
50 }
```

Figura 20.9 (Continuación).

```

1  private static class HashEntry implements java.io.Serializable
2  {
3      public Object element; // el elemento
4      public boolean isActive; // false si está marcado como borrado
5
6      public HashEntry( Object e )
7      {
8          this( e, true );
9      }
10
11     public HashEntry( Object e, boolean i )
12     {
13         element = e;
14         isActive = i;
15     }
16 }

```

Figura 20.10 La clase anidada HashEntry.

La disposición general es similar a la de TreeSet.

La mayoría de las rutinas tienen solo unas cuantas líneas de código porque llaman a `findPos` para realizar el sondeo cuadrático.

El resto de la clase contiene declaraciones para las rutinas y el iterador de la tabla hash. La disposición general es similar a la de TreeSet.

Se declaran tres métodos privados; los describiremos cuando se utilicen en la implementación de la clase. Ahora podemos analizar la implementación de la clase HashSet.

Los constructores de la tabla hash se muestran en la Figura 20.11; no tienen nada de especial. La rutina de búsqueda, `contains`, y el método no estándar `getMatch` se muestran en la Figura 20.12. `contains` utiliza el método privado `isActive`, mostrado en la Figura 20.13. Tanto `contains` como `getMatch` llaman también a `findPos`, que se muestra posteriormente, para implementar el sondeo cuadrático. El método `findPos` es el único

lugar de todo el código que depende del mecanismo de sondeo cuadrático. Por tanto, `contains` y `getMatch` son fáciles de implementar: un elemento se habrá encontrado si el resultado de `findPos` es una celda activa (si `findPos` se detiene en una celda activa, debe existir una correspondencia). De forma similar, la rutina `remove` mostrada en la Figura 20.14 es corta. Comprobamos si `findPos` nos lleva hasta una celda activa; en caso afirmativo, la celda se marca como borrada. En caso contrario, se devuelve `false` inmediatamente. Observe que esto hace que se reduzca `currentSize`, pero no `occupied`. Asimismo, si hay muchos elementos borrados, se modifica el tamaño de la tabla hash, en las líneas 16 y 17. El mantenimiento de `modCount` es idéntico al de otros componentes de la API de Colecciones que hemos implementado previamente. `clear` elimina todos los elementos del HashSet.

La rutina `add` se muestra en la Figura 20.15. En la línea 8 llamamos a `findPos`. Si se consigue encontrar `x`, devolvemos `false` en la línea 10 porque no están permitidos los elementos duplicados. En caso contrario, `findPos` proporciona el lugar en el que insertar `x`. La inserción se realiza en la línea 12. Ajustamos `currentSize`, `occupied` y `modCount` en las líneas 13 a 15 y volvemos, a menos que haga falta un rehash; si es así, invocamos el método privado `rehash`.

```

1  private static final int DEFAULT_TABLE_SIZE = 101;
2
3  /**
4   * Construir un HashSet vacío.
5   */
6  public HashSet( )
7  {
8      allocateArray( DEFAULT_TABLE_SIZE );
9      clear( );
10 }
11
12 /**
13 * Construir un HashSet a partir de cualquier colección.
14 */
15 public HashSet( Collection<? extends AnyType> other )
16 {
17     allocateArray( nextPrime( other.size( ) * 2 ) );
18     clear( );
19
20     for( AnyType val : other )
21         add( val );
22 }

```

Figura 20.11 Inicialización de la tabla hash.

El código que implementa el *rehashing* se muestra en la Figura 20.16. La línea 7 guarda una referencia a la tabla original. En las líneas 10 a 12 creamos una nueva tabla hash vacía, que tendrá un factor de carga de 0,25 cuando *rehash* termine. Después recorremos la matriz original y añadimos con *add* los elementos activos a la nueva tabla. La rutina *add* utiliza la nueva función *hash* (ya que está basada lógicamente en el tamaño de *array*, que ha variado) y resuelve automáticamente todas las colisiones. Podemos estar seguros de que la llamada recursiva a *add* (en la línea 17) no provoca otro *rehash*. Alternativamente, podríamos sustituir la línea 17 por dos líneas de código encerradas entre llaves (véase el Ejercicio 20.14).

La rutina *add* realiza  
rehashing si la tabla está  
(medio) llena.

Hasta ahora, nada de lo que hemos hecho depende del sondeo cuadrático. La Figura 20.17 implementa *findPos*, que se encarga finalmente de tratar con el algoritmo de sondeo cuadrático. Continuaremos explorando la tabla hasta encontrar una celda vacía o una correspondencia. Las líneas 22 a 25 implementan directamente la metodología descrita en el Teorema 20.5, utilizando dos sumas. Existen complicaciones adicionales, porque *null* es un elemento válido en el *HashSet*; el código ilustra por qué es preferible asumir que *null* no es válido.

La Figura 20.18 proporciona la implementación de la clase interna iteradora. Es código relativamente estándar, aunque contiene algunas complicaciones. *visited* representa el número de llamadas a *next*, mientras que *currentPos* representa el índice del último objeto devuelto por *next*.

```

1  /**
2   * Este método no es estándar Java.
3   * Al igual que contains, comprueba si x se encuentra en el conjunto.
4   * Si está, devuelve la referencia al objeto correspondiente;
5   * en caso contrario, devuelve null.
6   * @param x el objeto que hay que buscar.
7   * @return si contains(x) es false, el valor de retorno es null;
8   * en caso contrario, el valor de retorno es el objeto que hace que
9   * contains(x) devuelva true.
10  */
11 public AnyType getMatch( AnyType x )
12 {
13     int currentPos = findPos( x );
14
15     if( isActive( array, currentPos ) )
16         return (AnyType) array[ currentPos ].element;
17     return null;
18 }
19
20 /**
21 * Comprueba si algún elemento se encuentra en esta colección.
22 * @param x cualquier objeto.
23 * @return true si esta colección contiene un elemento igual a x.
24 */
25 public boolean contains( Object x )
26 {
27     return isActive( array, findPos( x ) );
28 }

```

Figura 20.12 Las rutinas de búsqueda para una tabla hash con sondeo cuadrático.

```

1 /**
2  * Comprueba si el elemento de pos está activo.
3  * @param pos una posición de la tabla hash.
4  * @param arr matriz de obj. HashEntry (puede ser oldArray durante el rehash).
5  * @return true si esta posición está activa.
6  */
7 private static boolean isActive( HashEntry [ ] arr, int pos )
8 {
9     return arr[ pos ] != null && arr[ pos ].isActive;
10 }

```

Figura 20.13 El método isActive para una tabla hash con sondeo cuadrático.

```

1  /**
2   * Elimina un elemento de esta colección.
3   * @param x cualquier objeto.
4   * @return true si este elemento ha sido eliminado de la colección.
5   */
6  public boolean remove( Object x )
7  {
8      int currentPos = findPos( x );
9      if( !isActive( array, currentPos ) )
10         return false;
11
12     array[ currentPos ].isActive = false;
13     currentSize--;
14     modCount++;
15
16     if( currentSize < array.length / 8 )
17         rehash( );
18
19     return true;
20 }
21
22 /**
23 * Cambia el tamaño de esta colección a cero.
24 */
25 public void clear( )
26 {
27     currentSize = occupied = 0;
28     modCount++;
29     for( int i = 0; i < array.length; i++ )
30         array[ i ] = null;
31 }

```

Figura 20.14 Las rutinas `remove` y `clear` para una tabla hash con sondeo cuadrático.

Finalmente, la Figura 20.19 implementa `HashMap`. Se parece bastante a `TreeMap`, salvo porque `Pair` es una clase anidada en lugar de una clase interna (no necesita acceder a un objeto externo), y porque implementa tanto `equals` como `hashCode` en lugar de la interfaz `Comparable`.

## 20.4.2 Análisis del sondeo cuadrático

El sondeo cuadrático no ha sido todavía analizado matemáticamente, aunque sabemos que elimina el fenómeno del agrupamiento primario. En el sondeo cuadrático, los elementos cuyo valor hash corresponde a una misma posición provocan un sondeo de las mismas celdas alternativas, fenómeno

```

1  /**
2   * Añade un elemento a esta colección.
3   * @param x cualquier objeto.
4   * @return true si este elemento ha sido añadido a la colección.
5   */
6  public boolean add( AnyType x )
7  {
8      int currentPos = findPos( x );
9      if( isActive( array, currentPos ) )
10         return false;
11
12      if( array[ currentPos ] == null )
13          occupied++;
14      array[ currentPos ] = new HashEntry( x, true );
15      currentSize++;
16      modCount++;
17
18      if( occupied > array.length / 2 )
19          rehash( );
20
21      return true;
22 }

```

---

Figura 20.15 La rutina add para una tabla hash con sondeo cuadrático.

```

1  /**
2   * Rutina privada para realizar el rehashing.
3   * Puede ser invocada tanto por add como por remove.
4   */
5  private void rehash( )
6  {
7      HashEntry [ ] oldArray = array;
8
9      // Crear una nueva tabla vacía
10     allocateArray( nextPrime( 4 * size( ) ) );
11     currentSize = 0;
12     occupied = 0;
13
14     // Copiar la tabla
15     for( int i = 0; i < oldArray.length; i++ )
16         if( isActive( oldArray, i ) )
17             add( (AnyType) oldArray[ i ].element );
18 }

```

---

Figura 20.16 El método rehash para una tabla hash con sondeo cuadrático.

```

1  /**
2   * Método que realiza la resolución del sondeo cuadrático.
3   * @param x el elemento que hay que buscar.
4   * @return la posición donde termina la búsqueda.
5  */
6  private int findPos( Object x )
7  {
8      int offset = 1;
9      int currentPos = ( x == null ) ?
10                     0 : Math.abs( x.hashCode() % array.length );
11
12     while( array[ currentPos ] != null )
13     {
14         if( x == null )
15         {
16             if( array[ currentPos ].element == null )
17                 break;
18         }
19         else if( x.equals( array[ currentPos ].element ) )
20             break;
21
22         currentPos += offset;           // Calcular i-ésimo sondeo
23         offset += 2;
24         if( currentPos >= array.length ) // Implementar el módulo
25             currentPos -= array.length;
26     }
27
28     return currentPos;
29 }

```

**Figura 20.17** La rutina que se encarga finalmente de tratar con el sondeo cuadrático.

que se conoce con el nombre de *agrupamiento secundario*. De nuevo, no podemos asumir que los sondeos sucesivos sean independientes entre sí. El agrupamiento secundario es difícil de analizar desde el punto de vista teórico. Los resultados de simulación sugieren que, con carácter general, este fenómeno provoca menos de medio sondeo adicional por cada búsqueda, y que este incremento solo se produce para factores de carga altos. La Figura 20.6 ilustra la diferencia entre el sondeo lineal y el sondeo cuadrático y en ella podemos ver que el sondeo cuadrático no se ve tan afectado por el fenómeno del agrupamiento como el sondeo lineal.

El sondeo cuadrático se implementa en `findPos`. Utiliza el truco que hemos mencionado antes para evitar las multiplicaciones y las operaciones módulo.

```
1  /**
2  * Esta es la implementación del HashSetIterator.
3  * Mantiene una noción de una posición actual y, por supuesto,
4  * la referencia implícita al HashSet.
5  */
6  private class HashSetIterator implements Iterator<AnyType>
7  {
8      private int expectedModCount = modCount;
9      private int currentPos = -1;
10     private int visited = 0;
11
12     public boolean hasNext( )
13     {
14         if( expectedModCount != modCount )
15             throw new ConcurrentModificationException( );
16
17         return visited != size( );
18     }
19
20     public AnyType next( )
21     {
22         if( !hasNext( ) )
23             throw new NoSuchElementException( );
24
25         do
26         {
27             currentPos++;
28         } while( currentPos < array.length &&
29                  !isActive( array, currentPos ) );
30
31         visited++;
32         return (AnyType) array[ currentPos ].element;
33     }
34
35     public void remove( )
36     {
37         if( expectedModCount != modCount )
38             throw new ConcurrentModificationException( );
39         if( currentPos == -1 || !isActive( array, currentPos ) )
40             throw new IllegalStateException( );
41
42         array[ currentPos ].isActive = false;
43         currentSize--;
44         visited--;
45         modCount++;
46         expectedModCount++;
47     }
48 }
```

---

Figura 20.18 La clase interna HashSetIterator.

```
1 package weiss.util;
2
3 public class HashMap<KeyType,ValueType> extends MapImpl<KeyType,ValueType>
4 {
5     public HashMap( )
6         { super( new HashSet<Map.Entry<KeyType,ValueType>>() ); }
7
8     public HashMap( Map<KeyType,ValueType> other )
9         { super( other ); }
10
11    protected Map.Entry<KeyType,ValueType> makePair( KeyType key, ValueType value)
12        { return new Pair<KeyType,ValueType>( key, value ); }
13
14    protected Set<KeyType> makeEmptyKeySet( )
15        { return new HashSet<KeyType>(); }
16
17    protected Set<Map.Entry<KeyType,ValueType>>
18    clonePairSet( Set<Map.Entry<KeyType,ValueType>> pairSet )
19    {
20        return new HashSet<Map.Entry<KeyType,ValueType>>( pairSet );
21    }
22
23    private static final class Pair<KeyType,ValueType>
24    extends MapImpl.Pair<KeyType,ValueType>
25    {
26        public Pair( KeyType k, ValueType v )
27            { super( k, v ); }
28
29        public int hashCode( )
30        {
31            KeyType k = getKey( );
32            return k == null ? 0 : k.hashCode( );
33        }
34
35        public boolean equals( Object other )
36        {
37            if( other instanceof Map.Entry )
38            {
39                KeyType thisKey = getKey( );
40                KeyType otherKey = ((Map.Entry<KeyType,ValueType>) other).getKey( );
41
```

*Continua***Figura 20.19** La clase `HashMap`.

```

42     if( thisKey == null )
43         return thisKey == otherKey;
44     return thisKey.equals( otherKey );
45 }
46 else
47     return false;
48 }
49 }
50 }

```

Figura 20.19 (Continuación).

En el fenómeno del agrupamiento secundario, los elementos cuyo valor hash se corresponde con una misma posición hacen que se sondeen las mismas celdas alternativas. El agrupamiento secundario constituye todavía una incógnita menor, desde el punto de vista del análisis teórico.

El doble hash es una técnica de hashing que no sufre el fenómeno del agrupamiento secundario. Utiliza una segunda función hash para dirigir la resolución de colisiones.

Hay disponibles técnicas que eliminan el fenómeno del agrupamiento secundario. La más popular es el *doble hash*, en el que se utiliza una segunda función hash para dirigir la resolución de colisiones. Específicamente, sondeamos a una distancia  $\text{Hash}_2(X)$ ,  $2\text{Hash}_2(X)$ , etc. La segunda función hash debe ser elegida con cuidado (por ejemplo, *never* puede ser 0 el resultado de su evaluación) y todas las celdas deben poder ser sondeadas. Una función como  $\text{Hash}_2(X) = R - (X \bmod R)$ , donde  $R$  es un número primo menor que  $M$ , suele funcionar bien. El doble hash es interesante desde el punto de vista teórico, porque se puede demostrar que utiliza esencialmente el mismo número de sondeos que el análisis puramente aleatorio del sondeo lineal parece sugerir. Sin embargo, es algo más complicado de implementar que el sondeo cuadrático y requiere que se preste una atención cuidadosa a algunos detalles.

No parece que exista ninguna buena razón para no utilizar la estrategia del sondeo cuadrático, a menos que el gasto adicional derivado de la necesidad de mantener una tabla medio vacía sea inaceptable. Eso podría suceder en otros lenguajes de programación, por ejemplo, si los elementos que se estuvieran almacenando fueran de muy gran tamaño.

## 20.5 Hash con encadenamiento separado

El *hash con encadenamiento separado* es una alternativa muy eficiente en términos de espacio al sondeo cuadrático; en esta técnica se mantiene una matriz de listas enlazadas. Es menos sensible a los factores de carga altos.

Una alternativa bastante popular y muy eficiente en términos de espacio al sondeo cuadrático es el *hash con encadenamiento separado*, en el que se mantiene una matriz de listas enlazadas. Para una matriz de listas enlazadas,  $L_0, L_1, \dots, L_{M-1}$ , la función hash nos dice en qué lista hay que insertar un elemento  $X$  y luego, durante una operación *find*, qué lista contiene a  $X$ . La idea es que, aunque la búsqueda en una lista enlazada es una operación lineal, si las listas son suficientemente cortas, el tiempo de búsqueda será muy rápido. En particular, suponga que el factor de carga,  $N/M$ , es  $\lambda$  y que no está acotado por 1,0. En ese caso, la lista típica tendrá una longitud  $\lambda$ , haciendo que el número esperado de sondeos para una inserción o una búsqueda que no tenga éxito sea  $\lambda$ , y que el número esperado de sondeos para una búsqueda que tenga éxito sea  $1 + \lambda/2$ . La razón

es que una búsqueda con éxito deberá producirse en una lista no vacía y que, en dicha lista, cabe esperar que tengamos que recorrer la mitad de la misma. El coste relativo de una búsqueda con éxito frente a una búsqueda que no lo tenga es inusual, en el sentido de que, si  $\lambda < 2$ , la búsqueda con éxito es más costosa que la que no lo tiene. Sin embargo, esta condición tiene bastante sentido, porque muchas búsquedas sin éxito van a encontrarse con una lista enlazada vacía.

Un factor de carga típico es 1,0; los factores de carga menores no mejoran el rendimiento de manera significativa, y además requieren malgastar espacio adicional. El atractivo del mecanismo de encadenamiento separado es que el rendimiento no se ve afectado por un incremento moderado del factor de carga; por tanto, puede evitarse el rehashing. Para aquellos lenguajes que no permiten la expansión dinámica de matrices, este aspecto resulta importante. Además, el número esperado de sondeos en una búsqueda es inferior que en el sondeo cuadrático, particularmente para las búsquedas que no tenga éxito.

Podemos implementar el mecanismo de encadenamiento separado utilizando nuestras clases existentes de listas enlazadas. Sin embargo, como el nodo de cabecera representa un gasto de espacio y no es realmente necesario, si el espacio fuera un problema podríamos elegir no reutilizar componentes, implementado en su lugar una lista simple de tipo pila. El esfuerzo de codificación es bastante pequeño. Además, el coste en términos de espacio es esencialmente de una referencia por nodo, más una referencia adicional por cada lista; por ejemplo, cuando el factor de carga es 1,0, tendremos dos referencias por cada elemento. Esta característica podría tener su importancia en otros lenguajes de programación si el tamaño de un elemento fuera grande. En ese caso, tendríamos los mismos compromisos que en el caso de la implementación de las pilas con matrices o con listas enlazadas. La API de Colecciones de Java utiliza el mecanismo de encadenamiento separado para tablas hash con un factor de carga predeterminado de 0,75.

Para las tablas hash con encadenamiento separado, un factor de carga razonable es 1,0. Un factor de carga más pequeño no mejora significativamente el rendimiento; un factor de carga moderadamente más grande resulta aceptable y permite ahorrar espacio.

Para ilustrar la complejidad (o más bien, la relativa falta de complejidad) de la tabla hash con encadenamiento separado, la Figura 20.20 proporciona un pequeño esbozo de la implementación básica de este mecanismo. En esa implementación se evitan temas como el rehashing, no se implementa el método `remove` y ni siquiera se lleva la cuenta del tamaño actual. De todos modos, muestra cuál es la lógica básica de los métodos `add` y `contains`, ambos de los cuales utilizan el código hash para seleccionar la apropiada lista simplemente enlazada.

## 20.6 Comparación entre las tablas hash y los árboles de búsqueda binaria

También podemos utilizar árboles de búsqueda binaria para implementar las operaciones `insert` y `find`. Aunque las cotas correspondientes de tiempo promedio son  $O(\log N)$ , los árboles de búsqueda binaria también soportan rutinas que exigen un orden y que son por tanto más potentes. Utilizando una tabla hash no podemos encontrar de manera eficiente el elemento mínimo o ampliar la tabla para permitir el cálculo de una estadística de orden. No podemos buscar de manera eficiente una cadena a menos que conozcamos la cadena de caracteres exacta. Un árbol de búsqueda binaria podría encontrar

Utilice la tabla hash en lugar de un árbol de búsqueda binaria si no necesita estadísticas de orden y le preocupa la posible existencia de entradas no aleatorias.

```

1 class MyHashSet<AnyType>
2 {
3     public MyHashSet( )
4         { this( 101 ); }
5
6     public MyHashSet( int numLists )
7         { lists = new Node[ numLists ]; }
8
9     public boolean contains( AnyType x )
10    {
11        for( Node<AnyType> p = lists[ myHashCode( x ) ]; p != null; p = p.next )
12            if( p.data.equals( x ) )
13                return true;
14
15        return false;
16    }
17
18    public boolean add( AnyType x )
19    {
20        int whichList = myHashCode( x );
21
22        for( Node<AnyType> p = lists[ whichList ]; p != null; p = p.next )
23            if( p.data.equals( x ) )
24                return false;
25
26        lists[ whichList ] = new Node<AnyType>( x, lists[ whichList ] );
27        return true;
28    }
29
30    private int myHashCode( AnyType x )
31        { return Math.abs( x.hashCode( ) % lists.length ); }
32
33    private Node<AnyType> [ ] lists;
34
35    private static class Node<AnyType>
36    {
37        Node( AnyType d, Node<AnyType> n )
38        {
39            data = d;
40            next = n;
41        }
42
43        AnyType data;
44        Node<AnyType> next;
45    }
46 }

```

Figura 20.20 Implementación simplificada de una tabla hash con encadenamiento separado.

rápidamente todos los elementos en un cierto rango, mientras que esta capacidad no está soportada por una tabla hash. Además, la cota  $O(\log N)$  no es necesariamente mucho peor que  $O(1)$ , especialmente porque en los árboles de búsqueda no hacen falta multiplicaciones o divisiones.

El caso peor para las tablas hash suele ser el resultado de un error de implementación, mientras que las entradas ordenadas pueden hacer que los árboles de búsqueda binaria tengan un rendimiento bastante inadecuado. Los árboles de búsqueda equilibrados son bastante costosos de implementar. Por tanto, si no se requiere información de orden y existe la más mínima sospecha de que la entrada pudiera estar ordenada, es preferible emplear como estructura de datos una tabla hash.

## 20.7 Aplicaciones de las tablas hash

Las aplicaciones de las tablas hash son muy numerosas. Los compiladores utilizan tablas hash para llevar la cuenta de las variables declaradas en el código fuente. La estructura de datos correspondiente se denomina *tabla de símbolos*. Las tablas hash son la aplicación ideal de este problema específico, porque solo se realizan operaciones `insert` y `find`. Los identificadores son normalmente cortos, por lo que la función hash se puede calcular rápidamente. En esta aplicación, la mayoría de las búsquedas tienen éxito.

Las aplicaciones de las tablas hash son muy numerosas.

Otro uso común de las tablas hash es en los programas de juegos. A medida que el programa explora a través de las diferentes líneas de juego, lleva la cuenta de las posiciones con las que ya se ha encontrado, calculando una función hash basada en la posición (y almacenando su movimiento a partir de esa posición). Si vuelve a aparecer la misma posición, usualmente debido a una simple transposición de movimientos, el programa puede ahorrarse una costosa serie de recálculos. Esta característica general de todos los programas de juegos se denomina *tabla de transposición*. Hemos hablado de esta característica en la Sección 10.2, a la hora de implementar el algoritmo del juego de las tres en raya.

Un tercer uso de las tablas hash es en los comprobadores ortográficos en línea. Si es importante la detección de errores ortográficos (en lugar de la corrección de esos errores), se puede calcular el valor hash de cada una de las palabras de un diccionario completo y comprobar las palabras en un tiempo constante. Las tablas hash están muy adaptadas a esta aplicación, porque las palabras no tienen por qué estar alfabetizadas. El imprimir los errores ortográficos en el orden en el que aparecen en el documento es perfectamente aceptable.

## Resumen

Las tablas hash se pueden utilizar para implementar las operaciones `insert` y `find` en un tiempo promedio constante. Es especialmente importante prestar atención a detalles tales como el factor de carga a la hora de utilizar las tablas hash; en caso contrario, las cotas de tiempo constante dejan de tener sentido. También es importante seleccionar la función hash con cuidado cuando la clave no sea un valor entero o una cadena de caracteres corta. Hay que elegir una función fácilmente calculable y que distribuya equitativamente los valores.

Para la técnica de encadenamiento separado, el factor de carga es normalmente próximo a 1, aunque el rendimiento no se degrada significativamente a menos que el factor de carga sea muy

grande. En el caso del sondeo cuadrático, el tamaño de la tabla debe ser un número primo y el factor de carga no debe sobrepasar el valor 0,5. Debe utilizarse el *rehashing* para el sondeo cuadrático con el fin de permitir que la tabla crezca y mantener así el factor de carga correcto. Esta técnica cobra su importancia en caso de que el espacio escasee y no sea posible declarar simplemente una tabla hash de tamaño enorme.

Con esto completamos nuestro análisis de los algoritmos básicos de búsqueda. En el Capítulo 21 examinaremos el montículo binario, que implementa la cola con prioridad y soporta, por tanto, un acceso eficiente al elemento mínimo de una colección de elementos.



## Conceptos clave

**agrupamiento primario** Durante el sondeo lineal se forman grandes agrupamientos de celdas ocupadas, haciendo que las inserciones en esos agrupamientos sean muy costosas (además de que cada inserción hace que crezca aún más el agrupamiento) y afectando así al rendimiento. (771)

**agrupamiento secundario** Agrupamiento que se produce cuando los elementos a los que la función hash hace corresponder con una misma posición de la tabla, sondean las mismas celdas alternativas. Constituye todavía un problema menor que no se ha resuelto teóricamente. (788)

**borrado perezoso** La técnica consistente en marcar los componentes como borrados en lugar de eliminarlos físicamente de una tabla hash. Esta técnica es necesaria a la hora de sondear tablas hash. (718, 770)

**colisión** El resultado cuando dos o más elementos de una tabla hash se corresponden con una misma posición. Este problema es inevitable, porque hay más elementos que posiciones. (764)

**doble hash** Una técnica de hash que no sufre el problema del agrupamiento secundario. Se emplea una segunda función hash para dirigir el mecanismo de resolución de colisiones. (788)

**encadenamiento separado** Una alternativa al sondeo cuadrático que es bastante eficiente en términos de espacio y en la que se mantiene una matriz de listas enlazadas. Es menos sensible a los factores de carga altos y exhibe algunos de los compromisos que ya hemos tomado en consideración al analizar las implementaciones de pilas basadas en matriz y en lista enlazada. (788)

**factor de carga** El número de elementos de una tabla hash dividido entre el tamaño de la matriz de la tabla. Representa la fracción de la tabla que está llena. En una tabla hash con sondeo lineal, el factor de carga va de 0 (vacía) a 1 (llena). Con la técnica de encadenamiento separado, puede ser mayor que 1. (770)

**función hash** Una función que convierte el elemento en un entero adecuado para indexar la matriz en la que el elemento se va a almacenar. Si la función hash fuera biunívoca, podríamos acceder al elemento utilizando su índice matricial pero, como la función hash no es biunívoca, varios elementos colisionarán en un mismo índice. (788)

**hashing** La implementación de tablas hash para realizar inserciones, borrados y búsquedas. (788)

**sondeo cuadrático** Un método de resolución de colisiones que examina las celdas situadas a una distancia de 1, 4, 9 etc. del punto de sondeo original. (774)

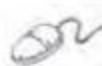
**sondeo lineal** Una forma de evitar colisiones explorando secuencialmente una matriz hasta encontrar una celda vacía. (768)

**tabla hash** Una tabla utilizada para implementar un diccionario en un tiempo constante por cada operación. (763)



## Errores comunes

1. La función hash devuelve un valor `int`. Puesto que los cálculos intermedios permiten el desbordamiento, la variable lógica debería comprobar que el resultado de la operación módulo no sea negativo, para evitar correr el riesgo de que el valor de retorno esté fuera de límites.
2. El rendimiento de una tabla de sondeo se degrada enormemente a medida que el factor de carga se aproxima a 1,0. No deje que esto suceda. Efectúe un rehash cuando el factor de carga alcance 0,5.
3. El rendimiento de todos los métodos hash depende de la utilización de una buena función de hash. Un error bastante común es el de proporcionar una función inadecuada.



## Internet

Tiene a su disposición la tabla hash con sondeo cuadrático.

**HashSet.java**

Contiene la implementación de la clase `HashSet`.

**HashMap.java**

Contiene la implementación de la clase `HashMap`.



## Ejercicios

### EN RESUMEN

- 20.1** Dada la entrada {437, 123, 617, 419, 456, 674, 199}, un tamaño de tabla fijo de 10 y una función hash  $H(X) = X \bmod 10$ , muestre las tablas hash resultantes con
- Sondeo lineal.
  - Sondeo cuadrático.
- 20.2** Muestre el resultado de aplicar un rehash a las tablas de sondeo del Ejercicio 20.1. Efectúe el rehash con un tamaño de tabla que sea un número primo.
- 20.3** ¿Cuál es el tamaño de tabla de sondeo apropiado si el número de elementos de la tabla hash es 8?

- 20.4** Explique cómo se realiza el borrado en las tablas hash tanto de sondeo como con encadenamiento separado.
- 20.5** ¿Cuáles son los índices de la matriz para una tabla hash de tamaño 5?
- 20.6** ¿Cuál es el número esperado de sondeos tanto para búsquedas con éxito como sin éxito, en una tabla de sondeo lineal con factor de carga igual a 0,25?

### EN TEORÍA

- 20.7** Bajo ciertas suposiciones, el coste esperado de una inserción en una tabla hash con agrupamiento secundario está dado por  $1/(1 - \lambda) - \lambda - \ln(1 - \lambda)$ . Lamentablemente, esta fórmula no es precisa para el sondeo cuadrático. Sin embargo, suponiendo que lo fuera,
- ¿Cuál sería el coste esperado de una búsqueda sin éxito?
  - ¿Cuál sería el coste esperado de una búsqueda con éxito?
- 20.8** Una estrategia de resolución de colisiones alternativa consiste en definir una secuencia  $F(j) = R_j$ , donde  $R_0 = 0$  y  $R_1, R_2, \dots, R_{M-1}$  es una permutación aleatoria de los primeros  $M - 1$  enteros (recuerde que el tamaño de la tabla es  $M$ ).
- Demuestre que, con esta estrategia, si la tabla no está llena, la colisión siempre se puede resolver.
  - ¿Cabría esperar que esta estrategia eliminara el fenómeno del agrupamiento primario?
  - ¿Cabría esperar que esta estrategia eliminara el fenómeno del agrupamiento secundario?
- 20.9** Si se implementa el rehashing en cuanto el factor de carga alcanza el valor 0,5, cuando se inserte el último elemento el factor de carga será como mínimo 0,25 y como máximo 0,5. ¿Cuál será el factor de carga esperado? En otras palabras, ¿es cierto o falso que el factor de carga será 0,375 como media?
- 20.10** Cuando se implementa el paso de rehashing hay que utilizar  $O(N)$  sondeos para reinserir los  $N$  elementos. Proporcione una estimación del número de sondeos (es decir,  $No$   $2N$  o alguna otra cosa). *Pista:* calcule el coste medio de cada operación de inserción en la nueva tabla. Estas inserciones varían entre el factor de carga 0 y el factor de carga 0,25.
- 20.11** Se utiliza una tabla hash con sondeo cuadrático para almacenar 7.000 objetos `String`. Suponga que el factor de carga es 0,4 y que la longitud media de las cadenas de caracteres es 6. Determine
- El tamaño de la tabla hash.
  - La cantidad de memoria utilizada para almacenar los 7.000 objetos `String`.
  - La cantidad de memoria adicional utilizada por la tabla hash.

### EN LA PRÁCTICA

- 20.12** Proporcione una implementación completa de `HashSet` empleando el mecanismo de encadenamiento separado.

**20.13** Implemente el sondeo lineal.

**20.14** Para la tabla hash con sondeo, implemente el código de rehashing sin realizar una llamada recursiva a `add`.

**20.15** Experimente con una función hash que examine cada uno de los caracteres de una cadena. ¿Es esta una mejor elección que la del texto? Explique su respuesta.

## PROYECTOS DE PROGRAMACIÓN

**20.16** Busque en Internet un buen diccionario en línea. Seleccione un tamaño de tabla que sea al menos el doble que el del diccionario. Aplique la función hash descrita en el texto y almacene un recuento del número de palabras que han correspondido a cada posición. Obtendrá con ello una distribución: a un cierto porcentaje de las posiciones no le habrá correspondido ninguna palabra, a otras les habrá correspondido solo una, a otras dos, etc. Compare esta distribución con lo que ocurriría en el caso de utilizar números teóricamente aleatorios (que hemos visto en la Sección 9.3).

**20.17** Realice simulaciones para comparar el rendimiento observado del mecanismo de hash con los resultados teóricos. Declare una tabla hash con sondeo, inserte en la tabla 10.000 enteros generados aleatoriamente y cuente el número medio de sondeos utilizados. Este número es el coste medio de una búsqueda que tenga éxito. Repita la prueba varias veces para obtener un buen promedio. Ejecútela tanto para sondeo lineal como para sondeo cuadrático, y hágalo para factores de carga finales iguales a 0,1, 0,2, ..., 0,9. Declare siempre la tabla de modo que no haga falta rehashing. Es decir, la prueba para un factor de carga de 0,4 declararía una tabla de tamaño aproximadamente igual a 25.000 (ajustado para que sea un número primo).



## Referencias

A pesar de la aparente simplicidad del algoritmo de hash, buena parte del análisis es bastante difícil y hay muchas cuestiones que todavía no han sido resueltas. Asimismo, existen muchas ideas interesantes que intentan, en general, que sea poco probable que se den las posibilidades de caso peor durante el manejo de las tablas hash.

Uno de los primeros artículos sobre hashing es [1]. En [6] se proporciona una gran cantidad de información sobre el tema, incluyendo un análisis de la técnica de hash con sondeo lineal. El doble hash se analiza en [5] y [7]. En [12] se describe otro esquema más de resolución de colisiones, denominado *hash con coalescencia*. [8] proporciona una excelente panorámica de la materia y [9] contiene sugerencias y consejos a la hora de elegir funciones hash. En [4] podrá encontrar resultados de simulación y analíticos precisos para todos los métodos descritos en este capítulo. El hash uniforme, en el que no existe agrupamiento, es óptimo en lo que respecta al coste de una búsqueda con éxito [13].

Si las claves de entrada se conocen de antemano, se pueden definir funciones hash perfectas que no permiten colisiones [1]. En [2] y [3] se proporcionan algunos esquemas hash más complicados, para los que el caso peor no depende de la entrada concreta, sino de los números aleatorios seleccionados por el algoritmo. Estos esquemas garantizan que solo

se produzca un número constante de colisiones en el caso peor (aunque la construcción de una función hash puede requerir un tiempo muy largo en el caso, improbable, de que los números aleatorios obtenidos sean inadecuados). Son útiles para implementar tablas en hardware.

En [10] se describe un método para implementar el Ejercicio 20.8.

- 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 der 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<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1991.
- 5.** L. J. Guibas y E. Szemerédi, "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<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1998.
- 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.

# Una cola con prioridad: el montículo binario

La cola con prioridad es una estructura de datos fundamental que solo permite acceder al elemento mínimo. En este capítulo vamos a analizar una implementación de la cola con prioridad; en concreto, analizaremos una elegante estructura de datos llamada *montículo binario*. El montículo binario soporta la inserción de nuevos elementos y el borrado del elemento mínimo en un tiempo logarítmico de caso peor. Utiliza únicamente una matriz y es fácil de implementar.

En este capítulo veremos

- Las propiedades básicas del montículo binario.
- Cómo realizar las operaciones `insert` y `deleteMin` en un tiempo logarítmico.
- Un algoritmo de tiempo lineal para la construcción del montículo.
- Una implementación en Java 5 de la clase `PriorityQueue`.
- Un algoritmo de ordenación fácilmente implementado, *heapsort*, que se ejecuta en un tiempo  $O(N \log N)$ , pero sin utilizar ninguna memoria adicional.
- La utilización de montículos para implementar una ordenación externa.

## 21.1 Ideas básicas

Como hemos explicado en la Sección 6.9, la cola con prioridad soporta el acceso y el borrado del elemento mínimo mediante las operaciones `findMin` y `deleteMin`, respectivamente. Podríamos emplear una lista enlazada simple, realizando las inserciones en un tiempo constante en la parte delantera, pero entonces el encontrar y/o borrar el mínimo requeriría una exploración lineal de la lista. Alternativamente, podemos obligar a que la lista se mantenga siempre ordenada. Esta condición hace que el acceso al elemento mínimo y el borrado del mismo sean muy poco costosos, pero entonces las inserciones serían lineales.

Otra forma de implementar las colas con prioridad consiste en utilizar un árbol de búsqueda binaria, que nos da un tiempo medio de ejecución  $O(\log N)$  para ambas operaciones. Sin embargo,

Una lista enlazada o una matriz requieren que alguna operación utilice un tiempo lineal.

Un árbol de búsqueda no equilibrado no tiene una buena cota de caso peor. Un árbol de búsqueda equilibrado requiere una gran cantidad de trabajo.

La cola con prioridad tiene propiedades que representan un compromiso entre una cola y un árbol de búsqueda binaria.

el árbol de búsqueda binaria no es una elección conveniente, porque la entrada no es normalmente lo suficientemente aleatoria. Podríamos utilizar un árbol de búsqueda equilibrado, pero las estructuras mostradas en el Capítulo 19 son engorrosas de implementar y en la práctica proporcionan un rendimiento inadecuado. (En el Capítulo 22, sin embargo, hablaremos de una estructura de datos, el *árbol splay*, que se ha demostrado empíricamente que es una buena alternativa en algunas ocasiones.)

Por un lado, como la cola con prioridad solo soporta algunas de las operaciones de los árboles de búsqueda, no debería ser más costosa de implementar que un árbol de búsqueda. Por otro lado, la cola con prioridad es más potente que una cola simple, porque podemos emplear la cola con prioridad para implementar una cola de la forma siguiente. En primer lugar, insertamos cada elemento con una indicación de su momento de inserción; después, una operación `deleteMin` que tome como base el tiempo mínimo de inserción permitirá implementar una operación `dequeue`. En consecuencia, cabe esperar que obtengamos una implementación cuyas propiedades presenten un compromiso entre una cola y un árbol de búsqueda.

Este compromiso se puede alcanzar mediante el montículo binario, que

- Puede implementarse utilizando una matriz simple (como la cola).
- Soporta las operaciones `insert` y `deleteMin` en un tiempo de caso peor  $O(\log N)$ , lo que constituye un compromiso entre el árbol de búsqueda binaria y la cola.
- Soporta la operación `insert` en un tiempo medio constante y la operación `findMin` en un tiempo constante de caso peor (como la cola).

El montículo binario es el método clásico para implementar las colas con prioridad.

las propiedades, por lo que cada operación con un montículo binario no debe terminar hasta que las dos propiedades hayan sido restauradas. Este resultado es fácil de conseguir. (En este capítulo, utilizaremos la palabra *montículo* para referirnos al montículo binario.)

### 21.1.1 Propiedad estructural

La única estructura que proporciona cotas dinámicas logarítmicas es el árbol, así que parece natural organizar los datos del montículo en forma de árbol. Puesto que queremos que la cota logarítmica constituya una garantía de caso peor, el árbol debe estar equilibrado.

El montículo es un *árbol binario completo*, que puede ser representado mediante una matriz simple y que garantiza una profundidad logarítmica.

Un *árbol binario completo* es un árbol completamente lleno, con la posible excepción del nivel inferior, que se llenará de izquierda a derecha y en el que no puede faltar ningún nodo. En la Figura 21.1 se muestra un ejemplo de un árbol binario completo de 10 elementos. Si el nodo *J* fuera un hijo derecho de *E*, el árbol no sería completo porque faltaría un nodo.

El árbol completo tiene una serie de propiedades útiles. En primer lugar, la altura (la longitud máxima de camino) de un árbol binario completo de

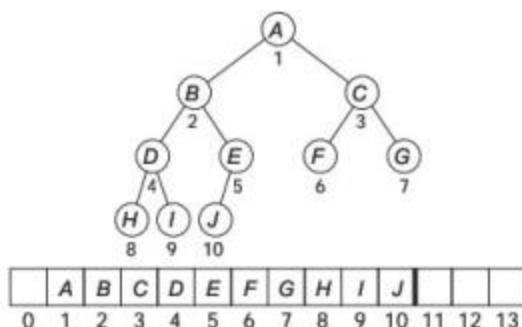


Figura 21.1 Un árbol binario completo y su representación matricial.

$N$  nodos es como máximo  $\lfloor \log N \rfloor$ . La razón es que un árbol completo de altura  $H$  tiene entre  $2^H$  y  $2^{H+1}-1$  nodos. Esta característica implica que podemos esperar un comportamiento logarítmico de caso peor si restringimos los cambios en la estructura al camino que va desde la raíz hasta una hoja determinada.

En segundo lugar (y esta característica es tan importante como la anterior), en un árbol binario completo no hacen falta enlaces `left` y `right`. Como se muestra en la Figura 21.1, podemos representar un árbol binario completo almacenando en una matriz la sucesión de nodos resultante de un recorrido en orden. Colocamos la raíz en la posición 1 (a menudo la posición 0 se deja vacía, por una razón que expondremos en breve). También necesitamos mantener un entero que nos diga el número de nodos que componen actualmente el árbol. Entonces, para cualquier elemento en la posición  $i$  de la matriz, su hijo izquierdo estará en la posición  $2i$ . Si esta posición es superior al número de nodos del árbol, sabremos que el hijo izquierdo no existe. De forma similar, el hijo derecho estará ubicado inmediatamente después del hijo izquierdo; por tanto, se encontrará en la posición  $2i + 1$ . De nuevo, compararemos ese valor con el tamaño real del árbol para asegurarnos de que el hijo existe. Finalmente, el padre se encontrará en la posición  $\lfloor i/2 \rfloor$ .

El padre se encuentra en la posición  $\lfloor i/2 \rfloor$ , el hijo izquierdo en la posición  $2i$  y el hijo derecho en la posición  $2i + 1$ .

Observe que todo nodo excepto la raíz tiene un parente. Si la raíz tuviera que tener un parente, el cálculo lo colocaría en la posición 0. Por eso reservamos la posición 0 para un elemento ficticio que pudiera servir como parente de la raíz. Hacer esto permite simplificar una de las operaciones. Si en lugar de ello eligiéramos colocar la raíz en la posición 0, las ubicaciones de los hijos y del parente del nodo situado en la posición  $i$  variarían ligeramente (en el Ejercicio 21.10 le pediremos que determine las nuevas ubicaciones).

La utilización de una matriz para representar un árbol se denomina *representación implícita*. Como resultado de esta representación no solo no hacen falta enlaces a los hijos, sino que las operaciones que necesitan recorrer el árbol son extremadamente simples y muy probablemente sean muy rápidas en la mayoría de las computadoras. La entidad que implementa el montículo está compuesta por una matriz de objetos y un entero que indica el tamaño actual del montículo.

La utilización de una matriz para representar un árbol se denomina *representación implícita*.

En este capítulo, los montículos se dibujan como árboles para hacer más fácil la visualización de los algoritmos. Sin embargo, en la implementación de estos árboles utilizaremos una matriz. No vamos a emplear la representación implícita para todos los árboles de búsqueda; en el Ejercicio 21.9 se indican algunos de los problemas que surgen cuando se intenta hacer esto.

### 21.1.2 Propiedad de ordenación del montículo

*La propiedad de ordenación del montículo afirma que en un montículo el elemento de un nodo siempre es mayor o igual que el elemento de su padre.*

La propiedad que permite realizar las operaciones rápidamente es la denominada *propiedad de ordenación del montículo*. Queremos poder encontrar el mínimo rápidamente, así que tendría sentido que el elemento menor se encontrara en la raíz. Si tenemos en cuenta que cualquier subárbol debería ser también (recursivamente) un montículo, cada nodo debería ser menor que todos sus descendientes. Aplicando esta lógica, llegamos a la propiedad de ordenación del montículo.

#### Propiedad de ordenación del montículo

En un montículo, para todo nodo  $X$  con padre  $P$ , la clave de  $P$  es menor o igual que la clave de  $X$ .

El padre de la raíz puede almacenarse en la posición 0 y se le puede asignar un infinito negativo como valor.

La propiedad de ordenación del montículo se ilustra en la Figura 21.2. En la Figura 21.3(a), el árbol es un montículo, pero en la Figura 21.3(b) no lo es (la línea de puntos muestra la violación de la propiedad de ordenación del montículo). Observe que la raíz no tiene un parente. En la representación implícita, podríamos colocar el valor  $-\infty$  en la posición 0 para eliminar este caso especial a la hora de implementar el montículo. Mediante la propiedad de ordenación del montículo, vemos que el elemento mínimo siempre estará situado en la raíz. Por tanto, `findMin` es una operación de tiempo constante. Un *montículo maximal* soporta el acceso al elemento máximo *en lugar de* al mínimo. Para implementar el montículo maximal basta con realizar una serie de cambios menores en la implementación que aquí proporcionamos.

### 21.1.3 Operaciones permitidas

Ahora que hemos acordado cuál será la representación, podemos empezar a escribir el código para nuestra implementación de `java.util.PriorityQueue`. Ya sabemos que nuestro montículo soporta las operaciones básicas `insert`, `findMin` y `deleteMin`, así como las rutinas usuales `isEmpty` y `makeEmpty`. La Figura 21.4 muestra el esqueleto de la clase, utilizando los convenios de denominación de `java.util.PriorityQueue`. Haremos referencia a estas operaciones utilizando tanto los nombres históricos como sus equivalentes en `java.util`.

Comenzamos examinando los métodos públicos. En las líneas 9 a 14 se declaran tres constructores. El tercer constructor acepta una colección de elementos que deberían encontrarse inicialmente en la cola con prioridad. ¿Por qué no insertar los elementos de uno en uno?

La razón es que, en numerosas aplicaciones, podemos añadir múltiples elementos antes de que tenga lugar la operación `deleteMin`. En esos casos, no necesitamos que la ordenación del

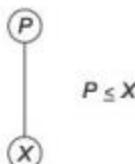


Figura 21.2 Propiedad de ordenación del montículo.

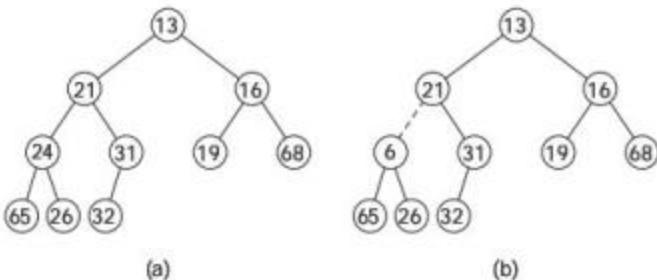


Figura 21.3 Dos árboles completos: (a) un montículo; (b) no es un montículo.

montículo se respete hasta que tenga lugar la operación `deleteMin`. La operación `buildHeap`, declarada en la línea 32, restaura el orden del montículo (independientemente de lo liado que esté el montículo), y posteriormente veremos que esta operación funciona en un tiempo lineal. Por tanto, si necesitamos insertar  $N$  elementos en el montículo antes de realizar la primera operación `deleteMin`, colocarlos de cualquier manera en la raíz y luego realizar una única operación `buildHeap` resulta más eficiente que realizar  $N$  inserciones.

El método `add` se declara en la línea 25. Este método añade un nuevo elemento  $x$  al montículo, realizando las operaciones necesarias para mantener la propiedad de ordenación del montículo.

El resto de las operaciones son como cabría esperar. La rutina `element` se declara en la línea 23 y devuelve el elemento mínimo del montículo. `remove` se declara en la línea 27 y elimina y luego devuelve el elemento mínimo. Las rutinas usuales `size`, `clear` e `iterator` se declaran en las líneas 16 a 21.

Los constructores se muestran en la Figura 21.5. Todos ellos inicializan la matriz, el tamaño y el comparador; el tercer constructor además copia la colección que se le pasa como parámetro y luego invoca a `buildHeap`. La Figura 21.6 muestra `element`.

Proporcionamos un constructor que acepta una colección que contenga un conjunto inicial de elementos y que luego invoca `buildHeap`.

## 21.2 Implementación de las operaciones básicas

La propiedad de ordenación del montículo parece hasta ahora bastante prometedora, porque permite un fácil acceso al elemento mínimo. Ahora vamos a demostrar que podemos soportar de manera eficiente las operaciones de inserción y `deleteMin` en un tiempo logarítmico. Realizar las dos operaciones requeridas es sencillo (tanto desde el punto de vista conceptual como práctico): el trabajo necesario implica meramente asegurarse de que se mantenga la propiedad de ordenación del montículo.

### 21.2.1 Inserción

Para insertar un elemento  $X$  en el montículo, primero debemos añadir un nodo al árbol. La única opción existente es crear un hueco en la siguiente ubicación disponible; en caso contrario, el árbol no estaría completo y violaríamos la propiedad estructural. Si  $X$  se puede situar en el hueco sin violar la propiedad de ordenación del montículo, lo hacemos así y habremos terminado. En caso

```
1 package weiss.util;
2
3 /**
4 * Clase PriorityQueue implementada mediante un montículo binario.
5 */
6 public class PriorityQueue<AnyType> extends AbstractCollection<AnyType>
7 implements Queue<AnyType>
8 {
9     public PriorityQueue( )
10    { /* Figura 21.5 */ }
11    public PriorityQueue( Comparator<? super AnyType> c )
12    { /* Figura 21.5 */ }
13    public PriorityQueue( Collection<? extends AnyType> coll )
14    { /* Figura 21.5 */ }
15
16    public int size( )
17    { return currentSize; }
18    public void clear( )
19    { currentSize = 0; }
20    public Iterator<AnyType> iterator( )
21    { /* Véase el código en línea */ }
22
23    public AnyType element( )
24    { /* Figura 21.6 */ }
25    public boolean add( AnyType x )
26    { /* Figura 21.9 */ }
27    public AnyType remove( )
28    { /* Figura 21.13 */ }
29
30    private void percolateDown( int hole )
31    { /* Figura 21.14 */ }
32    private void buildHeap( )
33    { /* Figura 21.16 */ }
34
35    private int currentSize; // Número de elementos del montículo
36    private AnyType [ ] array; // La matriz del montículo
37    private Comparator<? super AnyType> cmp;
38
39    private void doubleArray( )
40    { /* Véase el código en línea */ }
41    private int compare( AnyType lhs, AnyType rhs )
42    { /* Mismo código que en TreeSet; véase la Figura 19.70 */ }
43 }
```

---

Figura 21.4 El esqueleto de la clase PriorityQueue.

```
1  private static final int DEFAULT_CAPACITY = 100;
2
3  /**
4   * Construir una PriorityQueue vacía.
5   */
6  public PriorityQueue( )
7  {
8      currentSize = 0;
9      cmp = null;
10     array = (AnyType[]) new Object[ DEFAULT_CAPACITY + 1 ];
11 }
12
13 /**
14  * Construir una PriorityQueue vacía con un comparador especificado.
15  */
16 public PriorityQueue( Comparator<? super AnyType> c )
17 {
18     currentSize = 0;
19     cmp = c;
20     array = (AnyType[]) new Object[ DEFAULT_CAPACITY + 1 ];
21 }
22
23
24 /**
25  * Construir una PriorityQueue a partir de otra Collection.
26  */
27 public PriorityQueue( Collection<? extends AnyType> coll )
28 {
29     cmp = null;
30     currentSize = coll.size( );
31     array = (AnyType[]) new Object[ ( currentSize + 2 ) * 11 / 10 ];
32
33     int i = 1;
34     for( AnyType item : coll )
35         array[ i++ ] = item;
36     buildHeap( );
37 }
```

Figura 21.5 Constructores para la clase PriorityQueue.

contrario, desplazamos hasta el hueco el elemento que se encuentre en su nodo padre, haciendo que el hueco ascienda como una burbuja hasta la raíz. Continuamos este proceso hasta que  $X$  pueda insertarse en el hueco sin problemas. La Figura 21.7 muestra que para insertar el valor 14, creamos

```

1  /**
2   * Devuelve el elemento más pequeño de la cola con prioridad.
3   * @return el elemento más pequeño.
4   * @throws NoSuchElementException si la cola está vacía.
5   */
6  public AnyType element( )
7  {
8      if( isEmpty( ) )
9          throw new NoSuchElementException( );
10     return array[ 1 ];
11 }

```

Figura 21.6 La rutina element.

La inserción se implementa creando un hueco en la siguiente ubicación disponible y luego propagandolo hacia arriba, hasta poder colocar en el el nuevo elemento sin que se produzca una violación del principio de ordenación del montículo con respecto al nodo padre del hueco.

en el nodo recién añadido. Iteramos el bucle en la línea 15 mientras el elemento en el nodo padre sea mayor que x. La línea 16 mueve el elemento del parente hacia abajo para ocupar el hueco y luego la tercera expresión del bucle for mueve el hueco hacia arriba, para ocupar el lugar del parente. Cuando el bucle termina, la línea 17 coloca el elemento x en el hueco.

La inserción requiere un tiempo constante como promedio, pero un tiempo logarítmico en el caso peor.

un hueco en la siguiente posición disponible en el montículo. Insertar 14 en el hueco violaría la propiedad de ordenación del montículo, así que desplazamos 31 hacia abajo para ocupar el hueco. Esta estrategia se continúa en la Figura 21.8 hasta encontrar la ubicación correcta para el valor 14.

Esta estrategia general se denomina *propagación hacia arriba*, y con ella la inserción se implementa creando un hueco en la siguiente ubicación disponible y haciendo que ascienda como una burbuja por el montículo hasta encontrar la ubicación correcta. La Figura 21.9 muestra el método add, que implementa la estrategia de propagación hacia arriba utilizando un bucle muy compacto. En la línea 13, colocamos x como centinela  $-\infty$  en la posición 0.

La instrucción de la línea 12 incrementa el tamaño actual y coloca el hueco en el nodo recién añadido. Iteramos el bucle en la línea 15 mientras el elemento en el nodo parente sea mayor que x. La línea 16 mueve el elemento del parente hacia abajo para ocupar el hueco y luego la tercera expresión del bucle for mueve el hueco hacia arriba, para ocupar el lugar del parente. Cuando el bucle termina, la línea 17 coloca el elemento x en el hueco.

El tiempo requerido para realizar la inserción podría ser hasta  $O(\log N)$ , si el elemento que hay que insertar fuera el nuevo mínimo. La razón es que se propagaría hacia arriba a lo largo de todo el camino que va hasta la raíz. Como promedio, la propagación termina antes: se ha demostrado que se necesitan 2,6 comparaciones como promedio para realizar una operación add, por lo que la operación add típica desplaza un elemento 1,6 niveles hacia arriba.

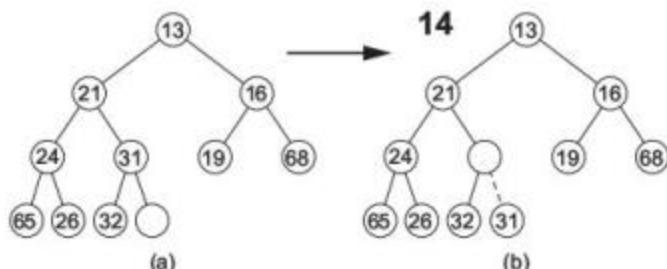


Figura 21.7 Intento de insertar 14, creando el hueco y haciendo que ascienda como una burbuja.

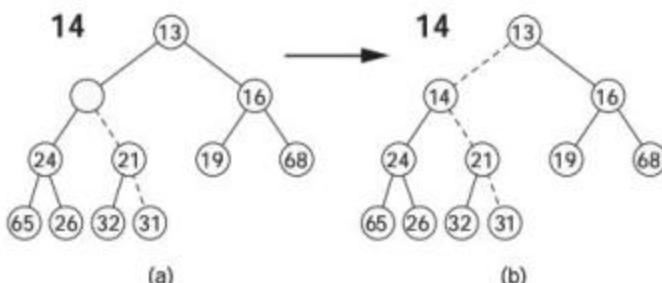


Figura 21.8 Los dos pasos restantes requeridos para insertar 14 en el montículo original mostrado en la Figura 21.7.

```

1  /**
2  * Añade un elemento a esta PriorityQueue.
3  * @param x cualquier objeto.
4  * @return true.
5  */
6  public boolean add( AnyType x )
7  {
8      if( currentSize + 1 == array.length )
9          doubleArray( );
10
11     // Propagar hacia arriba
12     int hole = ++currentSize;
13     array[ 0 ] = x;
14
15     for( ; compare( x, array[ hole / 2 ] ) < 0; hole /= 2 )
16         array[ hole ] = array[ hole / 2 ];
17     array[ hole ] = x;
18
19     return true;
20 }

```

Figura 21.9 El método add.

### 21.2.2 La operación deleteMin

La operación `deleteMin` se gestiona de una forma similar a la operación de inserción. Como ya hemos visto, encontrar el mínimo resulta fácil; la parte complicada es eliminarlo. Cuando se elimina el mínimo, se crea un hueco en la raíz. El tamaño del montículo se reduce con eso en una unidad, y la propiedad estructural nos dice que hay que eliminar el último nodo. La Figura 21.10 muestra la situación: el elemento mínimo es 13, la raíz tiene un hueco y el elemento que anteriormente era el último necesita ser colocado en algún lugar del montículo.

El borrado del mínimo implica colocar el elemento que anteriormente era el último en un hueco que se crea en la raíz. El hueco se propaga hacia abajo del árbol, a través de los hijos de valor mínimo, hasta poder colocar el elemento sin violar la propiedad de ordenación del montículo.

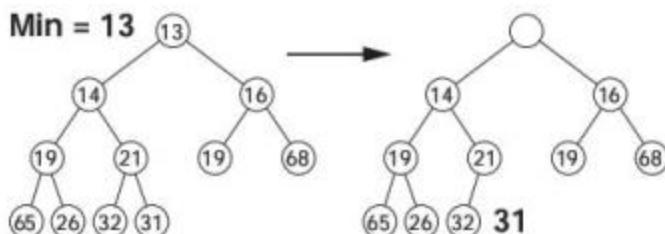
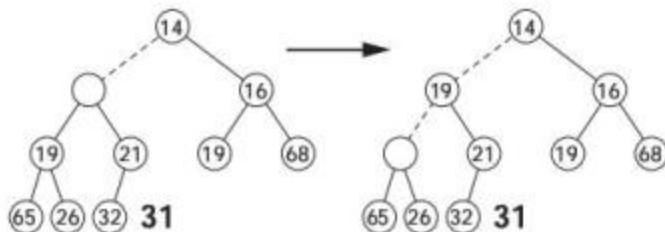
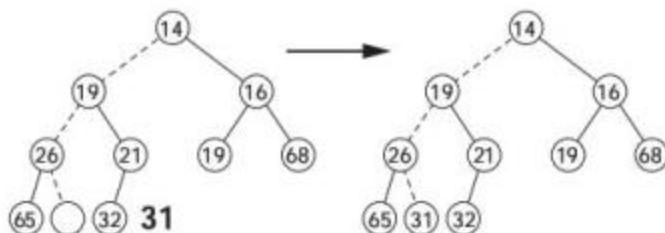


Figura 21.10 Creación del hueco en la raíz.

La operación `deleteMin` es logarítmica tanto en el caso promedio como en el caso peor.

Si el último elemento pudiera colocarse en el hueco, habríamos terminado. Sin embargo, eso es imposible a menos que el tamaño del montículo sea dos o tres, porque los elementos situados en la parte inferior tienen que ser mayores que los elementos del segundo nivel. Por tanto, debemos realizar la misma jugada que con la inserción: ponemos algún elemento en el hueco y después desplazamos el hueco. La única diferencia es que, para la operación `deleteMin`, nos movemos hacia abajo del árbol. Para ello, comprobamos cuál es el hijo más pequeño del hueco y, si ese hijo es menor que el elemento que estamos intentando colocar, movemos el hijo hasta el hueco, empujando el hueco hacia abajo un nivel y repitiendo estas acciones hasta que el elemento pueda ser correctamente situado –un proceso denominado *propagación hacia abajo*. En la Figura 21.11, colocamos el hijo más pequeño (14) en el hueco, deslizando hacia abajo el hueco un nivel. Repetimos esta acción colocando 19 en el hueco y creando un nuevo hueco un nivel más profundo. Después, situamos 26 en el hueco y creamos un nuevo hueco en el nivel inferior. Finalmente, podremos colocar 31 en el hueco, como se muestra en la Figura 21.12. Puesto que el

Figura 21.11 Los dos pasos siguientes en la operación `deleteMin`.Figura 21.12 Los dos últimos pasos en la operación `deleteMin`.

árbol tiene una profundidad logarítmica, `deleteMin` será una operación logarítmica en el caso peor. No es sorprendente que la propagación raramente termine más de uno o dos niveles antes de llegar a la parte inferior, de modo que `deleteMin` también es logarítmica en el caso promedio.

La Figura 21.13 muestra este método, que se denomina `remove` en la librería estándar. La comprobación de si el montículo está vacío durante la operación `remove` se realiza automáticamente mediante la llamada a `element` en la línea 8. El trabajo real se lleva a cabo en `percolateDown`, mostrado en la Figura 21.14. El código que allí se muestra es similar en su espíritu al código de propagación hacia arriba de la rutina `add`. Sin embargo, como hay dos hijos en lugar de un solo padre, el código es algo más complicado. El método `percolateDown` acepta un único parámetro que indica dónde hay que colocar el hueco. A continuación, el elemento del hueco se guarda en una variable temporal y da comienzo la propagación. Para `remove`, la variable `hole` que indica el hueco será la posición 1. El bucle `for` de la línea 10 termina cuando ya no hay un hijo izquierdo. La tercera expresión desplaza el hueco hasta el hijo. El hijo más pequeño se determina en las líneas 13 a 15. Tenemos que tener cuidado, porque el último nodo de un montículo de tamaño par es un hijo único; no siempre podemos asumir que habrá dos hijos, lo cual es la razón de que incluyamos la primera comparación de la línea 13.

## 21.3 La operación buildHeap: construcción de un montículo en tiempo lineal

La operación `buildHeap` toma un árbol completo que no cumple la propiedad de ordenación de los montículos y restaura dicha propiedad. Queremos que sea una propiedad en tiempo lineal, dado que se podrían realizar  $N$  inserciones en un tiempo  $O(N \log N)$ . Esperamos poder obtener  $O(N)$  porque  $N$  inserciones sucesivas requieren un tiempo total de  $O(N)$  como promedio, basándose en el resultado obtenido al final de la Sección 21.2.1. Las  $N$

La operación `buildHeap` se puede realizar en un tiempo lineal aplicando una rutina de propagación hacia abajo a los nodos, por orden inverso de niveles.

```

1  /**
2   * Elimina el elemento más pequeño de la cola con prioridad.
3   * @return el elemento más pequeño.
4   * @throws NoSuchElementException si la cola está vacía.
5   */
6  public AnyType remove( )
7  {
8      AnyType minItem = element( );
9      array[ 1 ] = array[ currentSize-- ];
10     percolateDown( 1 );
11
12     return minItem;
13 }
```

Figura 21.13 El método `remove`.

```
1  /**
2   * Método interno para propagar hacia abajo en el montículo.
3   * @param hole el índice en el que comienza la propagación.
4   */
5  private void percolateDown( int hole )
6  {
7      int child;
8      AnyType tmp = array[ hole ];
9
10     for( ; hole * 2 <= currentSize; hole = child )
11     {
12         child = hole * 2;
13         if( child != currentSize &&
14             compare( array[ child + 1 ], array[ child ] ) < 0 )
15             child++;
16         if( compare( array[ child ], tmp ) < 0 )
17             array[ hole ] = array[ child ];
18         else
19             break;
20     }
21     array[ hole ] = tmp;
22 }
```

---

Figura 21.14 El método `percolateDown` utilizado para `remove` y `buildHeap`.

inserciones sucesivas realizan más trabajo del que nosotros necesitamos porque se encargan de mantener la ordenación del montículo después de cada inserción, mientras que ahora lo que necesitamos es que la ordenación del montículo se respete únicamente al final de todas las inserciones.

La solución abstracta más sencilla se obtiene visualizando el montículo como una estructura definida recursivamente, como se muestra en la Figura 21.15: llamamos de forma recursiva a `buildHeap` para los submontículos izquierdo y derecho. En este punto, tendremos garantizado que la ordenación del montículo se habrá restaurado en todas partes, salvo en la raíz. Podemos entonces imponer la ordenación del montículo en todas partes invocando `percolateDown` para la raíz. La rutina recursiva funciona garantizando que, cuando aplicamos `percolateDown(i)`, todos los descendientes de `i` ya habrán sido procesados recursivamente mediante sus propias llamadas a `percolateDown`. Sin embargo, la recursión no es necesaria por la siguiente razón: si invocamos `percolateDown` para los nodos en un orden inverso de niveles, entonces en el momento de procesar `percolateDown(i)`, todos los descendientes del nodo `i` ya habrán sido procesados mediante una llamada anterior a `percolateDown`. Este proceso nos permite obtener un algoritmo increíblemente simple para `buildHeap`, el cual se muestra en la Figura 21.16. Observe que no hace falta realizar

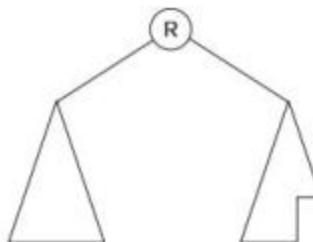


Figura 21.15 Vista recursiva del montículo.

`percolateDown` para un nodo hoja. Por tanto, comenzamos con el nodo de mayor número que no sea una hoja.

El árbol de la Figura 21.17(a) es el árbol desordenado. Los siete árboles restantes de las Figuras 21.17(b) a 21.20 muestran el resultado de cada una de las siete operaciones `percolateDown`. Cada línea de puntos se corresponde con dos comparaciones: una para encontrar el hijo más pequeño y otra para comparar el hijo más pequeño con el nodo. Observe que las diez líneas de puntos del algoritmo se corresponden con 20 comparaciones. (Podría haber habido una undécima línea.)

Para acotar el tiempo de ejecución de `buildHeap`, debemos acotar el número de líneas de puntos. Podemos hacerlo calculando la suma de las alturas de todos los nodos del montículo, que será el máximo número de líneas de puntos. Cabe esperar obtener un número pequeño, porque la mitad de los nodos son hojas y tienen altura 0 y una cuarta parte de los nodos tienen altura 1. Por tanto, solo una cuarta parte de los nodos (los que no caen en los dos primeros casos mencionados) pueden contribuir con más de 1 unidad de altura. En particular, solo un único nodo contribuirá con la máxima altura, que es de  $\lfloor \log N \rfloor$ .

Para obtener una cota de tiempo lineal para `buildHeap`, tenemos que establecer que la suma de las alturas de los nodos de un árbol binario completo es  $O(N)$ . Vamos a hacer esto en el Teorema 21.1, proporcionando la cota para árboles perfectos utilizando un argumento de marcado.

La cota de tiempo lineal se puede obtener calculando la suma de las alturas de todos los nodos del montículo.

Demostramos la cota para los árboles perfectos utilizando un argumento de marcado.

```

1  /**
2   * Establecer la propiedad de ordenación del montículo a partir de una
3   * disposición arbitraria de los elementos. Se ejecuta en un tiempo lineal.
4   */
5  private void buildHeap( )
6  {
7      for( int i = currentSize / 2; i > 0; i-- )
8          percolateDown( i );
9  }

```

Figura 21.16 Implementación del método `buildHeap` en tiempo lineal.

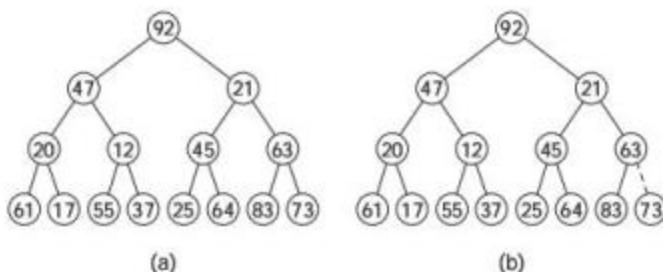


Figura 21.17 (a) Montículo inicial. (b) Despues de percolateDown(7).

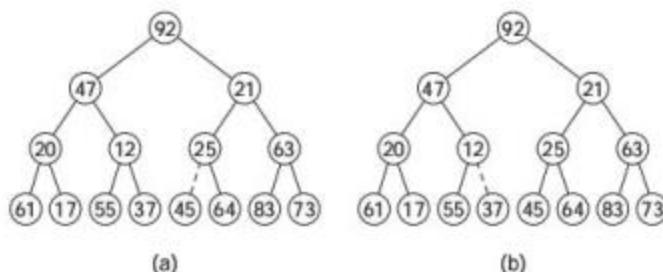


Figura 21.18 (a) Despues de percolateDown(6). (b) Despues de percolateDown(5).

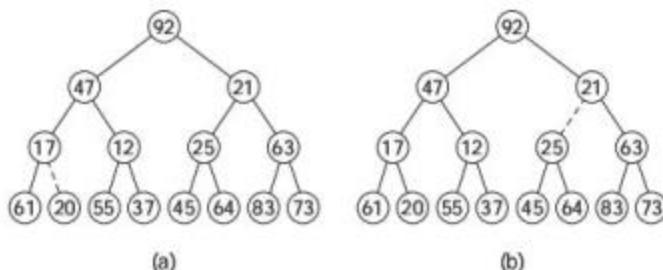


Figura 21.19 (a) Despues de percolateDown(4). (b) Despues de percolateDown(3).

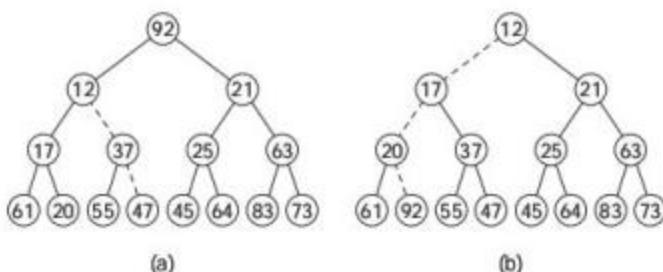


Figura 21.20 (a) Despues de percolateDown(2). (b) Despues de percolateDown(1) y de que buildHeap termine.

**Teorema 21.1**

Para el árbol binario perfecto de altura  $H$  que contiene  $N = 2^{H+1} - 1$  nodos, la suma de los nodos es  $N - H - 1$ .

**Demostración**

Utilizamos un argumento de marcado del árbol. (También se podría realizar un cálculo más directo por fuerza bruta, como en el Ejercicio 21.14.) Para cualquier nodo del árbol que tenga un cierta altura  $h$ , marcamos de color oscuro  $h$  aristas del árbol, de la forma siguiente: bajamos por el árbol recorriendo la arista izquierda y después únicamente aristas derechas. Cada arista recorrida se marca de color oscuro. Un ejemplo sería un árbol perfecto de altura 4. Para los nodos que tuvieran altura 1, se marcaría de color oscuro su arista izquierda, como se muestra en la Figura 21.21. A continuación, para los nodos de altura 2 se marcaría su arista izquierda y luego una arista derecha, en el camino que va desde ese nodo hasta la parte inferior, como se muestra en la Figura 21.22. En la Figura 21.23, se marcarían tres aristas por cada nodo de altura 3: la primera arista izquierda que sale del nodo y luego las dos aristas derechas en el camino que va hasta la parte inferior. Finalmente, en la Figura 21.24 se marcarían cuatro aristas: la arista izquierda que sale de la raíz y luego las tres aristas derechas que componen el camino hasta la parte inferior. Observe que ninguna arista se marca nunca dos veces y observe también que se marcan todas las aristas, excepto las que componen el camino de la derecha. Como hay  $(N - 1)$  aristas en el árbol (todo nodo tiene una arista entrante salvo la raíz) y como hay  $H$  aristas en el camino de la derecha, el número de aristas marcadas será  $N - H - 1$ . Esto demuestra el teorema.

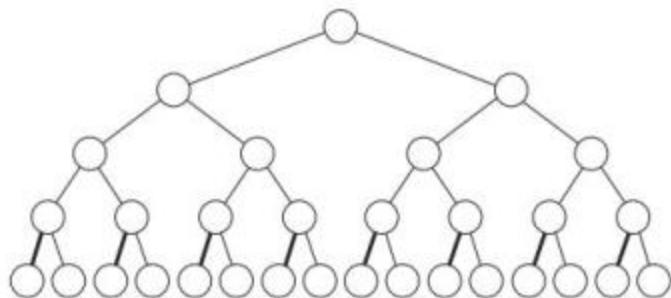


Figura 21.21 Marcado de las aristas izquierdas para los nodos de altura 1.

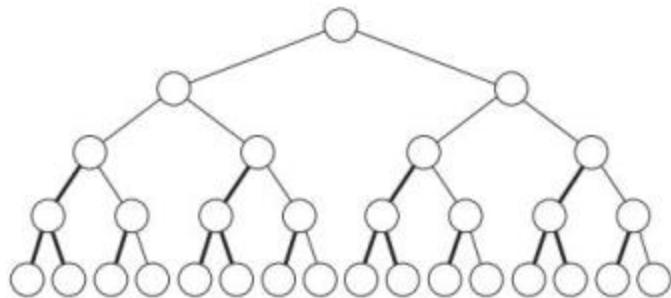


Figura 21.22 Marcado de la primera arista izquierda y de la posterior arista derecha para los nodos de altura 2.

Continúa

Demostración  
(cont.)

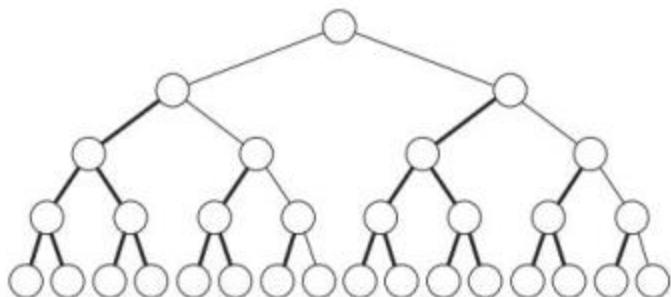


Figura 21.23 Marcado de la primera arista izquierda y de las dos aristas derechas posteriores para los nodos de altura 3.

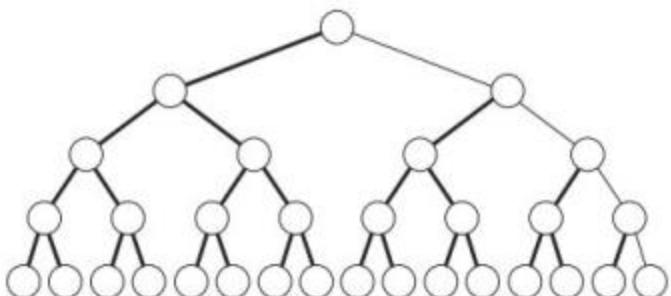


Figura 21.24 Marcado de la primera arista izquierda y de las tres aristas derechas posteriores para el nodo de altura 4.

Un árbol binario completo no es un árbol binario perfecto, pero el resultado que hemos obtenido proporciona una cota superior para la suma de las alturas de los nodos de un árbol binario completo. Un árbol binario completo tiene entre  $2^H$  y  $2^{H+1} - 1$  nodos, por lo que este teorema implica que la suma es  $O(N)$ . Un análisis más detenido permite establecer que la suma de las alturas es  $N - v(N)$ , donde  $v(N)$  es el número de unos (1) en la representación binaria de  $N$ . Dejamos la demostración de este enunciado como Ejercicio 21.13 para el lector.

## 21.4 Operaciones avanzadas: `decreaseKey` y `merge`

En el Capítulo 23 veremos colas con prioridad que soportan dos operaciones adicionales. La operación `decreaseKey` reduce el valor de un elemento de la cola con prioridad. Se supone que se conoce la posición del elemento. En un montículo binario, esta operación se puede implementar fácilmente mediante una propagación hacia arriba, hasta restablecer el orden del montículo. Sin embargo, debemos ser cuidadosos, porque, por suposición, la posición de cada elemento se almacena por separado, y se alterarán las posiciones de todos los elementos implicados en la propagación. Se puede incorporar `decreaseKey` a la clase `PriorityQueue`. Esto se deja como Ejercicio 21.27. La

operación `decreaseKey` resulta útil de implementar en algoritmos para grafos (por ejemplo, el algoritmo de Dijkstra presentado en la Sección 14.3).

La rutina `merge` permite combinar dos colas con prioridad. Puesto que el montículo está basado en una matriz, lo más que podemos esperar conseguir con una mezcla es copiar los elementos del montículo más pequeño al montículo mayor y llevar a cabo después una cierta reordenación. Hacer esto requiere al menos un tiempo lineal por operación. Si utilizamos árboles generales con nodos conectados mediante enlaces, podemos reducir la cota a un coste logarítmico por operación. La mezcla tiene aplicaciones en el diseño de algoritmos avanzados.

## 21.5 Ordenación interna: heapsort

La cola con prioridad se puede utilizar para ordenar  $N$  elementos mediante el siguiente procedimiento:

1. Insertar cada elemento en un montículo binario.
2. Extraer cada elemento invocando `deleteMin`  $N$  veces, ordenando así el resultado.

Se puede utilizar una cola con prioridad para ordenar en un tiempo  $O(N \log N)$ . Un algoritmo basado en esta idea es `heapsort`.

Utilizando la observación de la Sección 21.3, la forma más eficiente de implementar este procedimiento sería

1. Añadir cada elemento al montículo binario.
2. Aplicar `buildHeap`.
3. Invocar `deleteMin`  $N$  veces, con lo que los elementos saldrán del montículo por orden.

El paso 1 requiere un tiempo lineal en total y el paso 2 requiere un tiempo lineal. En el paso 3, cada llamada a `deleteMin` requiere un tiempo logarítmico, por lo que  $N$  llamadas tardan un tiempo  $O(N \log N)$ . En consecuencia, tenemos un algoritmo de ordenación con un tiempo de caso peor  $O(N \log N)$ , dicho algoritmo de ordenación se denomina `heapsort`. Este algoritmo es igual de bueno que lo que se puede conseguir con un algoritmo basado en comparaciones (véase la Sección 8.8). Un problema con el algoritmo, tal como lo hemos definido, es que ordenar una matriz requiere utilizar la estructura de datos de montículo binario, que a su vez necesita emplear una matriz adicional. Emular la estructura de datos de montículo en la matriz que se proporciona como entrada –en lugar de incurrir en el coste extra asociado a la clase que implementa el montículo– sería una solución preferible. En el resto de esta exposición vamos a asumir que eso es lo que se hace.

Aunque no utilizaremos directamente la clase que implementa el montículo, parece que seguimos necesitando una segunda matriz. La razón es que necesitamos registrar en una segunda matriz el orden en el que los elementos salen de nuestro equivalente del montículo, y luego copiar dicha ordenación de nuevo en la matriz original. El requisito de memoria se duplica, lo que podría resultar crucial en algunas aplicaciones. Observe que el tiempo adicional invertido en copiar de nuevo la segunda matriz en la primera es solo  $O(N)$ , por lo que, a diferencia de la ordenación por mezcla, la matriz adicional no afecta al tiempo de ejecución significativamente. El problema es el espacio.

Una forma inteligente de evitar tener que utilizar una segunda matriz se basa en el hecho de que después de cada `deleteMin`, el tamaño del montículo se reduce en 1. Por tanto, la última celda del montículo se puede emplear para almacenar el elemento que se acaba de borrar. Por ejemplo,

Utilizando partes vacías de la matriz podemos realizar la ordenación en la propia matriz.

Si utilizamos un montículo maximal obtendremos los elementos en orden creciente.

suponga que tenemos un montículo con seis elementos. La primera operación `deleteMin` produce  $A_1$ . Ahora el montículo tendrá solo cinco elementos, así que podemos colocar  $A_1$  en la posición 6. La siguiente operación `deleteMin` produce  $A_2$ . Puesto que ahora el montículo solo tiene cuatro elementos, podemos colocar  $A_2$  en la posición 5.

Cuando utilizamos esta estrategia, después de la última operación `deleteMin`, la matriz contendrá los elementos en orden *decreciente*. Si queremos que la matriz esté en orden *creciente*, que resulta más normal, podemos cambiar la propiedad de ordenación del montículo, de modo que el padre tenga una clave de mayor valor que los hijos. Es decir, podemos utilizar un montículo maximal. Por ejemplo, supongamos que queremos ordenar la secuencia de entrada 59, 36, 58, 21, 41, 97, 31, 16, 26 y 53. Después de añadir los elementos al montículo maximal y aplicar `buildHeap`, obtenemos la disposición mostrada en la Figura 21.25. (Observe que no hay nodo centinela; asumimos que los datos comienzan en la posición 0, como resulta normal para las otras ordenaciones descritas en el Capítulo 8.)

La Figura 21.26 muestra el montículo resultante después de la primera operación `deleteMax`. El último elemento del montículo es 21; el 97 ha sido colocado en una parte del montículo que, desde el punto de vista técnico, ya no forma parte del montículo.

La Figura 21.27 muestra que, después de una segunda operación `deleteMax`, 16 pasa a ser el último elemento. Ahora solo quedan ocho elementos en el montículo. El elemento máximo extraído,

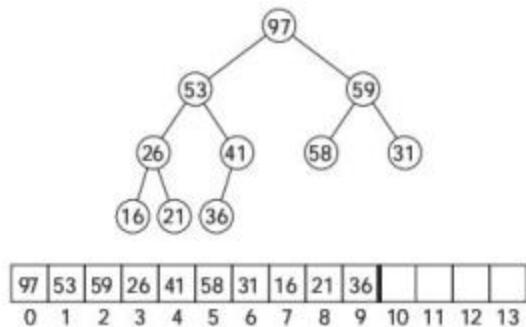


Figura 21.25 Montículo maximal de la fase `buildHeap`.

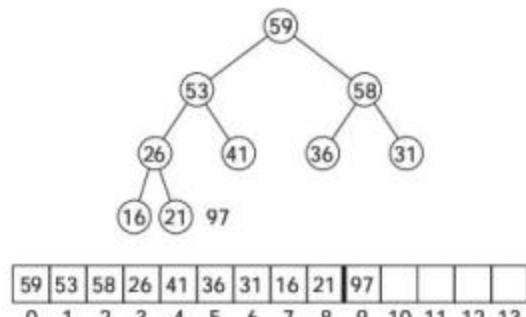


Figura 21.26 Montículo después de la primera operación `deleteMax`.

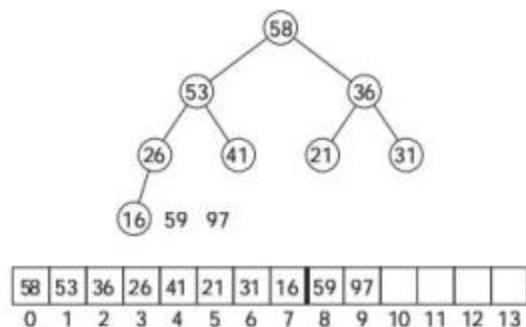


Figura 21.27 Montículo después de la segunda operación de `deleteMax`.

59, se coloca en el hueco que ha quedado en la matriz. Después de siete operaciones `deleteMax` adicionales, el montículo solo constará de un elemento, pero los elementos restantes en la matriz estarán ordenados en orden creciente.

La implementación de la operación heapsort es simple, porque básicamente se ajusta al modo de operación del montículo. Hay tres diferencias menores entre los dos tipos de operación. En primer lugar, puesto que estamos empleando un montículo maximal, necesitamos invertir la lógica de las comparaciones, de  $>$  a  $<$ . En segundo lugar, ya no podemos asumir que hay un nodo centinela en la posición 0. La razón es que todos nuestros restantes algoritmos de ordenación almacenan los datos en la posición 0, y debemos presuponer que `heapSort` no es diferente a este respecto. Aunque el nodo centinela no se necesita de todas formas (no existen operaciones de propagación hacia arriba), su ausencia afecta al cálculo de los hijos y los padres. Es decir, para un nodo en la posición  $i$ , el padre se encontrará en la posición  $(i - 1)/2$ , el hijo izquierdo estará en la posición  $2i + 1$  y el hijo derecho estará situado a continuación del hijo izquierdo. En tercer lugar, habrá que informar a `percDown` del tamaño actual del montículo (que se reduce en 1 en cada iteración de `deleteMax`). La implementación de `percDown` se deja como Ejercicio 21.23 para el lector. Suponiendo que ya hemos escrito `percDown`, podemos expresar fácilmente `heapSort` como se muestra en la Figura 21.28.

Hacen falta una serie de cambios menores para heapsort, porque la raíz se almacena en la posición 0.

```

1 // heapsort estándar.
2 public static <AnyType extends Comparable<? super AnyType>>
3 void heapsort( AnyType [ ] a )
4 {
5     for( int i = a.length / 2 - 1; i >= 0; i-- ) // Construir montículo
6         percDown( a, i, a.length );
7     for( int i = a.length - 1; i > 0; i-- )
8     {
9         swapReferences( a, 0, i );                      // deleteMax
10        percDown( a, 0, i );
11    }
12 }
  
```

Figura 21.28 La rutina `heapSort`.

Aunque heapsort no es tan rápido como el algoritmo de ordenación rápida, sigue teniendo utilidad. Como se explica en la Sección 8.6 (y se detalla en el Ejercicio 8.22), en el algoritmo de ordenación rápida podemos llevar la cuenta de la profundidad de cada llamada recursiva, y cambiar a una ordenación con un tiempo de caso peor  $O(N \log N)$  para cualquier llamada recursiva que sea demasiado profunda (aproximadamente  $2 \log N$  llamadas anidadas). El Ejercicio 8.22 sugería emplear la ordenación por mezcla, pero en realidad heapsort es el mejor candidato para esto.

## 21.6 Ordenación externa

La ordenación externa se emplea cuando la cantidad de datos es demasiado grande como para caber en la memoria principal.

Hasta ahora, todos los algoritmos de ordenación que hemos examinado exigían que la entrada cupiera en la memoria principal. Sin embargo, la entrada de algunas aplicaciones es demasiado grande como para caber en la memoria principal. En esta sección vamos a hablar de la *ordenación externa*, que se utiliza para gestionar esos conjuntos tan grandes de datos de entrada. Algunos de los algoritmos de ordenación externa implican el uso de montículos.

### 21.6.1 Por qué necesitamos nuevos algoritmos

La mayoría de los algoritmos de ordenación interna aprovechan el hecho de que la memoria es directamente accesible. Shellsort compara los elementos  $a[i]$  y  $a[i - \text{gap}]$  en una unidad de tiempo. Heapsort compara  $a[i]$  y  $a[\text{child} = i * 2]$  en una unidad de tiempo. La ordenación rápida con pivotes de la mediana de tres requiere comparar  $a[\text{first}]$ ,  $a[\text{center}]$  y  $a[\text{last}]$  en un número constante de unidades de tiempo. Si la entrada se encuentra en una cinta magnética, todas estas operaciones pierden su eficiencia, porque a los elementos de una cinta solo se puede acceder de manera secuencial. Incluso si los datos residen en un disco, la eficiencia sigue viéndose afectada, debido al retardo requerido para hacer girar el disco y mover el cabezal del mismo.

Para demostrar lo lentos que son realmente los accesos externos, podríamos crear un archivo aleatorio que fuera grande, pero no demasiado como para no caber en la memoria principal. Al leer el archivo y ordenarlo utilizando un algoritmo eficiente, el tiempo necesario para leer la entrada es probable que sea significativo con el tiempo requerido para ordenarla, aun cuando la ordenación sea una operación de  $O(N \log N)$  (o incluso peor para Shellsort) y leer la entrada sea solo  $O(N)$ .

### 21.6.2 Modelo para la ordenación externa

Vamos a suponer que las ordenaciones se realizan en cinta magnética. Solo se permite el acceso secuencial a los datos de entrada.

La amplia variedad de dispositivos de almacenamiento masivo hace que la ordenación externa sea mucho más dependiente del dispositivo que la ordenación interna. Los algoritmos considerados aquí funcionan con cintas magnéticas, que son probablemente el medio de almacenamiento más restrictivo. El acceso a un elemento almacenado en la cinta se consigue bobinando la cinta hasta la ubicación correcta, de modo que a las cintas magnéticas solo se puede acceder de manera eficiente en orden secuencial (en cualquier dirección).

Supongamos que tenemos al menos tres unidades de cinta magnética para realizar la ordenación. Necesitamos dos cintas para llevar a cabo una ordenación eficiente; la tercera cinta simplifica el procedimiento. Si solo hay disponible una cinta magnética, tendremos graves problemas: cualquier algoritmo requerirá  $\Omega(N^2)$  accesos a cinta.

### 21.6.3 El algoritmo simple

El algoritmo básico de ordenación externa implica el uso de la rutina de mezcla del algoritmo de ordenación por mezcla. Suponga que tenemos cuatro cintas, A1, A2, B1 y B2, que son dos cintas magnéticas de entrada y dos de salida. Dependiendo del punto del algoritmo, se utilizan las cintas A para entrada y las B para salida, o viceversa. Suponga también que los datos se encuentran inicialmente en A1 y que la memoria interna puede almacenar (y ordenar)  $M$  registros cada vez. El primer paso natural consiste en leer  $M$  registros cada vez de la cinta de entrada, ordenar los registros internamente y luego escribir los registros ordenados alternativamente en B1 y B2. Cada grupo de registros ordenados se denomina *lote*. Al terminar, rebobinamos todas las cintas. Si tenemos los mismos datos de entrada que nuestro ejemplo de Shellsort, la configuración inicial será como se muestra en la Figura 21.29. Si  $M = 3$ , después de haber construido los lotes, las cintas contendrán los datos en la configuración que se muestra en la Figura 21.30.

La ordenación externa básica utiliza una mezcla de dos vías repetida. Cada grupo de registros ordenados es un lote. Como resultado de cada pasada la longitud de los bloques se duplica, y al final termina quedando un único lote.

Ahora B1 y B2 contienen un grupo de lotes. Tomamos los primeros lotes de cada cinta, los combinamos y escribimos el resultado —que es un lote el doble de largo— en A1. Despues, tomamos los siguientes lotes de cada cinta, los mezclamos y escribimos el resultado en A2. Continuamos este proceso, alternando la salida entre A1 y A2 hasta que B1 o B2 estén vacías. En este punto, o ambas están vacías o nos queda un solo lote (posiblemente incompleto). En este último caso, copiamos dicho lote en la cinta apropiada. Rebobinamos las cuatro cintas y repetimos los mismos pasos, pero esta vez utilizando como entrada las cintas A y como salida las cintas B. Este proceso nos dará lotes de longitud  $4M$ . Continuamos con este procedimiento hasta obtener un único lote de longitud  $N$ , en cuyo momento dicho lote representará la disposición ordenada de los datos entrada. Las Figuras 21.31 a 21.33 muestran cómo funciona este proceso para nuestro ejemplo de datos de entrada.

El algoritmo requerirá  $\log \lceil (N/M) \rceil$  pasadas, más la pasada inicial de construcción de los lotes. Por ejemplo, si tenemos 10.000.000 de registros de 6.400 bytes cada uno y 200 MB de memoria interna, la primera pasada creará 320 lotes. Después necesitaremos nueve pasadas adicionales para completar la ordenación. Esta fórmula también nos dice, correctamente, que nuestro ejemplo de la Figura 21.30 requiere  $\lceil \log (13/3) \rceil$ , o tres pasadas más.

Necesitamos  $\log \lceil (N/M) \rceil$  pasadas con la entrada para llegar a obtener un único lote gigante.

A1	81	94	11	96	12	35	17	99	28	58	41	75	15
A2													
B1													
B2													

Figura 21.29 Configuración inicial de la cinta.

A1													
A2													
B1	11	81	94		17	28	99		15				
B2	12	35	96		41	58	75						

Figura 21.30 Distribución de los lotes de longitud 3 entre dos cintas.

A1	11	12	35	81	94	96	15	
A2	17	28	41	58	75	99		
B1								
B2								

Figura 21.31 Las cintas después de la primera ronda de mezcla (longitud de lote = 6).

A1													
A2													
B1	11	12	17	28	35	41	58	75	81	94	96	99	
B2	15												

Figura 21.32 Las cintas después de la segunda ronda de mezcla (longitud de lote = 12).

A1	11	12	15	17	28	35	41	58	75	81	94	96	99
A2													
B1													
B2													

Figura 21.33 Las cintas después de la tercera ronda de mezcla.

#### 21.6.4 Mezcla multivía

La mezcla de  $K$ -vías reduce el número de pasadas. La implementación obvia emplea  $2K$  cintas.

Si disponemos de cintas magnéticas adicionales, podemos reducir el número de pasadas necesarias para ordenar nuestra entrada, utilizando una *mezcla multivía* (o de  $K$ -vías). Podemos hacerlo ampliando la mezcla básica (de dos vías) a una mezcla de  $K$ -vías y utilizando  $2K$  cintas.

La mezcla de dos lotes se realiza bobinando cada cinta de entrada hasta el principio de cada lote. Después, se localiza el elemento más pequeño y se coloca en una cinta de salida, tras lo cual se hace avanzar la cinta de entrada apropiada. Si hay  $K$  cintas de entrada, esta estrategia funciona de la misma manera: la única diferencia es que es algo más complicado encontrar el mínimo de los  $K$  elementos. Podemos hacer esto utilizando una cola con prioridad. Para encontrar el siguiente elemento que hay que escribir en la cinta de salida, realizamos una operación `deleteMin`. Se hace avanzar la cinta de entrada apropiada y, si el lote de esa cinta de entrada todavía no ha sido completado, insertamos el nuevo elemento en la cola con prioridad. La Figura 21.34 muestra cómo se distribuye la entrada del ejemplo anterior entre tres cintas. Las Figuras 21.35 y 21.36 ilustran las dos pasadas de la mezcla de tres vías que permiten completar la ordenación.

Después de la fase inicial de construcción de los lotes, el número de pasadas requeridas utilizando la mezcla de  $K$ -vías es  $\lceil \log_K(NM) \rceil$ , porque la longitud de los lotes se hace  $K$  veces más grande en cada pasada. Podemos verificar la fórmula para nuestro ejemplo, porque  $\lceil \log_3(13/3) \rceil = 2$ . Si tenemos 10 cintas,  $K = 5$ . Para el ejemplo del conjunto de entrada de gran tamaño de la Sección 21.6.3, 320 lotes requerirían  $\log_5 320 = 4$  pasadas.

A1	
A2	
A3	
B1	11 81 94   41 58 75
B2	12 35 96   15
B3	17 28 99

Figura 21.34 Distribución inicial de los lotes de longitud 3 entre tres cintas.

A1	11 12 17 28 35 81 94 96 99
A2	15 41 58 75
A3	
B1	
B2	
B3	

Figura 21.35 Despues de una ronda de mezcla de tres vías (longitud de lote = 9).

A1	
A2	
A3	
B1	11 12 15 17 28 35 41 58 75 81 94 96 99
B2	
B3	

Figura 21.36 Despues de dos rondas de mezcla de tres vías.

## 21.6.5 Mezcla polifásica

La estrategia de mezcla de  $K$ -vías requiere el uso de  $2K$  cintas, lo que puede ser prohibitivo para algunas aplicaciones. Podemos salir del paso con solo  $K + 1$  cintas, utilizando lo que se denomina una *mezcla polifásica*. Un ejemplo sería realizar una mezcla de dos vías con solo tres cintas.

Suponga que tenemos tres cintas –T1, T2 y T3– y un archivo de entrada en T1 que puede generar 34 lotes. Una opción es colocar 17 lotes en T2 y otros 17 en T3. Después podríamos mezclar este resultado en T1, obteniendo así una cinta con 17 lotes. El problema es que, como todos los lotes se encuentran en una misma cinta, ahora tenemos que colocar algunos de esos lotes en T2 para poder realizar otra mezcla. La forma lógica de hacer esto consiste en copiar los primeros ocho lotes de T1 a T2 y luego realizar la mezcla. Este enfoque añade media pasada adicional por cada pasada que realicemos. La pregunta es: ¿podemos encontrar alguna solución mejor?

Un método alternativo consiste en dividir los 34 lotes originales de forma no equitativa. Si colocamos 21 lotes en T2 y 13 lotes en T3, podríamos mezclar 13 lotes en T1 antes de que T3

La *mezcla polifásica* implementa una mezcla de  $K$ -vías con  $K+1$  cintas.

La distribución de los lotes afecta al rendimiento.  
La mejor distribución está relacionada con los números de Fibonacci.

quedara vacía. Después podríamos rebobinar T1 y T3 y mezclar T1, que tiene 13 lotes, con T2, que tiene 8 lotes, almacenando el resultado en T3. Durante ese paso podríamos mezclar 8 lotes hasta que T2 quedara vacía, lo que nos dejaría 5 lotes en T1 y 8 lotes en T3. Después podríamos mezclar T1 y T3, y así sucesivamente. La Figura 21.37 muestra el número de lotes en cada cinta después de cada pasada.

La distribución original de lotes tiene una gran influencia en el resultado. Por ejemplo, si colocamos 22 lotes en T2 y 12 en T3, después de la primera mezcla obtenemos 12 lotes en T1 y 10 lotes en T2. Después de otra mezcla, habrá 10 lotes en T1 y 2 lotes en T3. En este punto, el procedimiento se ralentiza, porque solo podemos mezclar dos conjuntos de lotes antes de que T3 quede vacía. Entonces T1 tendrá 8 lotes y T2 tendrá 2 lotes. De nuevo, solo podemos mezclar dos conjuntos de lotes, obteniendo una cinta T1 con 6 lotes y una cinta T3 con 2 lotes. Después de tres pasadas más, T2 tendrá 2 lotes y las otras cintas estarán vacías. Deberemos entonces copiar 1 lote a otra cinta y luego podremos ya finalizar la mezcla.

Nuestra primera distribución resulta ser la óptima. Si el número de lotes es un número de Fibonacci,  $F_N$ , la mejor forma de distribuirlos es dividiendo los lotes en dos números de Fibonacci,  $F_{N-1}$  y  $F_{N-2}$ . Si el número original de lotes no fuera igual a un número de Fibonacci, habrá que llenar la cinta con lotes ficticios, con el fin de incrementar el número de lotes hasta igualarlo con un número de Fibonacci. Dejamos los detalles de cómo colocar en las cintas el conjunto inicial de lotes como ejercicio para el lector, en el Ejercicio 21.19. Podemos ampliar esta técnica a una mezcla de  $K$ -vías, para lo que necesitaremos utilizar el número de Fibonacci de  $K$ -ésimo orden para la distribución. El número de Fibonacci de  $K$ -ésimo orden se define como la suma de los  $K$  números de Fibonacci de  $K$ -ésimo orden anteriores.

$$\begin{aligned} F^{(K)}(N) &= F^{(K)}(N-1) + F^{(K)}(N-2) + \cdots + F^{(K)}(N-K) \\ F^{(K)}(0 \leq N \leq K-2) &= 0 \\ F^{(K)}(K-1) &= 1 \end{aligned}$$

### 21.6.6 Selección de sustitutos

El último tema que vamos a considerar en este capítulo es el de la construcción de los lotes. La estrategia utilizada hasta ahora es la más simple: leemos tantos elementos como sea posible y los ordenamos escribiendo el resultado en una cinta. Esta parece la mejor de las técnicas posibles, hasta que nos damos cuenta de que, en cuanto se escribe el primer elemento en la cinta de salida, la

Construcción lotes	Después						
	T3 + T2	T1 + T2	T1 + T3	T2 + T3	T1 + T2	T1 + T3	T2 + T3
T1	0	13	5	0	3	1	0
T2	21	8	0	5	2	0	1
T3	13	0	8	3	0	2	0

Figura 21.37 El número de lotes para una mezcla polifásica.

memoria que ese elemento utilizaba queda disponible para otro elemento. Si el siguiente elemento de la cinta de entrada es mayor que el elemento que acabamos de escribir como salida, se le puede incluir en el lote.

Utilizando esta observación, podemos escribir un algoritmo para construcción de lotes, al que comúnmente se denomina *selección de sustitutos*. Inicialmente, se leen  $M$  elementos en memoria y se colocan de manera eficiente en una cola con prioridad, con una única operación `buildHeap`. Realizamos una operación `deleteMin`, escribiendo el menor de los elementos 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 con prioridad; en caso contrario, no podremos incluirlo en el lote actual. Puesto que la cola con prioridad tiene ahora un tamaño 1 unidad menor, este elemento se almacena en el espacio muerto de la cola con prioridad hasta que se haya completado el lote, y luego se lo utiliza para el siguiente lote. Almacenar un elemento en el espacio muerto es exactamente lo que se hace en la ordenación `heapsort`. Continuamos con este proceso hasta que el tamaño de la cola con prioridad sea 0, en cuyo momento habremos acabado con el lote. Comenzamos entonces un nuevo lote reconstruyendo una nueva cola con prioridad mediante una operación `buildHeap`, utilizando durante el proceso todos los elementos almacenados en el espacio muerto.

La Figura 21.38 muestra el proceso de construcción de lotes para el ejemplo de pequeño tamaño que hemos venido utilizando, con  $M = 3$ . Los elementos reservados para el siguiente lote se muestran sombreados. Los elementos 11, 94 y 81 se colocan mediante `buildHeap`. El elemento 11 se escribe

Si somos inteligentes, podemos hacer que la longitud de los lotes que construyamos inicialmente sea mayor que la cantidad de memoria principal disponible. Esta técnica se denomina *selección de sustitutos*.

Tres elementos en la matriz del montículo			Salida	Siguiente elemento leído
	array[1]	array[2]		
Lote 1	11	94	81	11
	81	94	96	96
	94	96	12	12
	96	35	12	35
	17	35	12	Fin de lote
Lote 2	12	35	17	Reconstruir
	17	35	99	99
	28	99	35	28
	35	99	58	58
	41	99	58	41
	58	99	75	75
	75	99	15	15
	99		15	Fin de cinta
Lote 3	15		15	Reconstruir

Figura 21.38 Ejemplo de construcción de lotes.

como salida y luego se añade 96 al montículo mediante una inserción, porque es mayor que 11. A continuación, se proporciona como salida el elemento 81 y se lee el elemento 12. Como 12 es más pequeño que el 81 que acabamos de escribir, no se puede incluir en el lote actual. Por tanto, se coloca en el espacio muerto del montículo. El montículo contendrá ahora únicamente, desde el punto de vista lógico, los valores 94 y 96. Después de escribirlos como salida, solo tendremos elementos en el espacio muerto, por lo que construimos un nuevo montículo y comenzamos con el lote 2.

En este ejemplo, la técnica de selección de sustitutos genera solo 3 lotes, comparado con los 5 lotes obtenidos mediante la ordenación. Como resultado, una mezcla de tres vías permitirá terminar en una sola pasada en lugar de en dos. Si la entrada está distribuida aleatoriamente, la selección de sustitutos genera lotes con una longitud media  $2M$ . Para nuestro ejemplo del conjunto de datos de entrada de gran tamaño, cabría esperar obtener 160 lotes en lugar de 320, por lo que una mezcla de cinco vías seguiría requiriendo cuatro pasadas. En este caso, no nos hemos ahorrado ninguna pasada, aunque podríamos hacerlo si tuviéramos suerte y generáramos 125 lotes o menos. Puesto que las ordenaciones externas tardan tanto tiempo, cada pasada que nos ahorremos puede representar una diferencia significativa en cuanto a tiempo de ejecución.

Como hemos visto, la selección de sustitutos puede no tener un rendimiento mejor que el algoritmo estándar. Sin embargo, la entrada suele estar muchas veces casi ordenada, en cuyo caso la técnica de selección de sustitutos solo genera unos cuantos lotes anormalmente largos. Este tipo de entradas es común en el caso de las ordenaciones externas, lo que hace que la técnica de selección de sustitutos sea enormemente valiosa.

## Resumen

---

En este capítulo hemos mostrado una elegante implementación de la cola con prioridad. El montículo binario solo utiliza una matriz, a pesar de lo cual soporta las operaciones básicas en un tiempo logarítmico de caso peor. El montículo permite implementar un algoritmo de ordenación bastante popular, *heapsort*. En los Ejercicios 21.24 y 21.28 le pediremos que compare el rendimiento de heapsort con el del algoritmo de ordenación rápida. Hablando en términos generales, heapsort es más lento que la ordenación rápida, pero es más fácil de implementar. Finalmente, hemos mostrado que las colas con prioridad son estructuras de datos que tienen una gran importancia de cara a la ordenación externa.

Con esto completamos la implementación de las estructuras de datos fundamentales más clásicas. En la Parte Cinco examinaremos estructuras de datos más sofisticadas, comenzando con el árbol *splay*, un árbol de búsqueda binaria que presenta algunas propiedades bastante notables.



### Conceptos clave

**árbol binario completo** Un árbol completamente lleno y en el que no falta ningún nodo.

El montículo es un árbol binario completo, lo que permite representarlo mediante una matriz simple y garantiza que exista una profundidad logarítmica. (798)

**heapsort** Un algoritmo basado en la idea de que puede emplearse una cola con prioridad para ordenar en un tiempo  $O(N \log N)$ . (813)

**lote** Un grupo ordenado de elementos durante la ordenación externa. Al final de la ordenación solo queda un único lote. (817)

**mezcla multivía** Mezcla de  $K$ -vías que reduce el número de pasadas. La implementación obvia utiliza  $2K$  cintas magnéticas. (818)

**mezcla polifásica** Implementa una mezcla de  $K$ -vías con  $K + 1$  cintas. (819)

**montículo binario** El método clásico utilizado para implementar colas con prioridad. El montículo binario tiene dos propiedades: estructura y ordenación. (798)

**montículo maximal** Soporta el acceso al elemento máximo en lugar de al mínimo. (800)

**operación buildHeap** El proceso de reinstaurar la ordenación del montículo en un árbol completo, lo cual puede realizarse en un tiempo lineal aplicando una rutina de propagación hacia abajo a los nodos, en orden inverso de niveles. (807)

**ordenación externa** Una forma de ordenación utilizada cuando la cantidad de datos es demasiado grande como para caber en memoria principal. (816)

**propagación hacia abajo** El borrado del mínimo implica colocar el elemento que anteriormente era el último en un hueco creado en la raíz. El hueco es empujado hacia abajo del árbol, a través de los hijos de valor mínimo, hasta que se pueda colocar el elemento sin violar la propiedad de ordenación del montículo. (805)

**propagación hacia arriba** Implementa la inserción creando un hueco en la siguiente ubicación disponible y luego haciéndolo subir como una burbuja hasta que el nuevo elemento se pueda colocar en él sin introducir una violación del principio de ordenación del montículo con respecto al padre del hueco. (804)

**propiedad de ordenación del montículo** Establece que en un montículo (minimal), el elemento del parente nunca es mayor que el elemento de un nodo. (800)

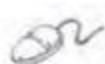
**representación implícita** Utilización de una matriz para almacenar un árbol. (799)

**selección de sustitutos** La longitud de los lotes inicialmente construidos puede ser mayor que la cantidad de memoria principal disponible. Si podemos almacenar  $M$  objetos en memoria principal, entonces podemos esperar obtener lotes de longitud  $2M$ . (821)



## Errores comunes

1. La parte más complicada del montículo binario es el caso de la propagación hacia abajo, cuando solo hay presente un hijo. Este caso se produce raras veces, por lo que detectar una implementación incorrecta es difícil.
2. 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  $2i + 2$ .



## Internet

El código para implementar `PriorityQueue` está disponible en un único archivo.

**PriorityQueue.java** Contiene la implementación de la clase `PriorityQueue`.



## Ejercicios

### EN RESUMEN

- 21.1** Muestre el resultado de insertar 14, 6, 5, 8, 1, 3, 12, 9, 7, 13 y 2, de uno en uno en un montículo inicialmente vacío. Después, muestre el resultado de utilizar en su lugar el algoritmo `buildHeap` de tiempo lineal.
- 21.2** Muestre el resultado del algoritmo heapsort después de la construcción inicial y de aplicar luego tres operaciones `deleteMax`, para los datos de entrada del Ejercicio 21.1.
- 21.3** Un montículo maximal soporta las operaciones `insert`, `deleteMax` y `findMax` (pero no `deleteMin` ni `findMin`). Describa en detalle cómo se pueden implementar los montículos maximales.
- 21.4** Describa las propiedades estructural y de ordenación del montículo binario.
- 21.5** ¿Dónde podría haber estado la undécima línea de puntos en las Figuras 21.17 a 21.20?
- 21.6** ¿Es heapsort una ordenación estable (es decir, si hay duplicados, esos elementos duplicados mantienen su ordenación inicial respectiva)?
- 21.7** En un montículo binario, ¿dónde están ubicados el padre, el hijo izquierdo y el hijo derecho de un elemento situado en la posición  $r$ ?

### EN TEORÍA

- 21.8** Demuestre las siguientes afirmaciones en relación con el elemento máximo del montículo.
- Debe estar en una de las hojas.
  - Hay exactamente  $\lceil N/2 \rceil$  hojas.
  - Es necesario examinar todas las hojas para encontrarlo.
- 21.9** Un árbol binario completo de  $N$  elementos utiliza las posiciones 1 a  $N$  de la matriz. Determina lo grande que debe ser la matriz para
- Un árbol binario que tenga dos niveles adicionales (es decir, que esté ligeramente desequilibrado).
  - Un árbol binario cuyo nodo más profundo esté a una profundidad  $2 \log N$ .
  - Un árbol binario cuyo nodo más profundo esté a una profundidad  $4,1 \log N$ .
- 21.10** Suponga que el montículo binario se almacena con la raíz en la posición  $r$ . Proporcione fórmulas que den las posiciones del padre y de los hijos del nodo situado en la posición  $i$ .
- 21.11** Para heapsort, en el caso peor se utilizan  $O(N \log N)$  comparaciones. Calcule el término multiplicador (es decir, determine si es  $N \log N$ ,  $2N \log N$ ,  $3N \log N$ , etc.)
- 21.12** Verifique que la suma de las alturas de todos los nodos de un árbol binario perfecto satisface  $N - v(N)$ , donde  $v(N)$  es el número de 1s en la representación binaria de  $N$ .

- 21.13** Demuestre la cota del Ejercicio 21.12 utilizando un argumento de inducción.
- 21.14** Demuestre el Teorema 21.1 utilizando un sumatorio directo. Haga lo siguiente:
- Demuestre que hay  $2^l$  nodos de altura  $H - l$ .
  - Escriba la ecuación correspondiente a la suma de las alturas utilizando el resultado del apartado (a).
  - Evalúe la suma del apartado (b).
- 21.15** Demuestre que hay entradas que obligan a que toda operación `percDown` en el algoritmo de ordenación heapsort recorra todo el camino hasta una hoja. (*Pista:* trabaje hacia atrás para encontrar la respuesta.)
- 21.16** Un montículo *d* es una estructura de datos implícita similar a un montículo binario, salvo porque los nodos tienen *d* hijos. Un montículo *d* es por tanto menos profundo que un montículo binario, pero encontrar el hijo mínimo requiere examinar *d* hijos en lugar de solo dos. Determine el tiempo de ejecución (en función de *d* y *N*) de las operaciones de inserción y `deleteMin` para un montículo *d*.
- 21.17** Suponga que los montículos binarios se representan mediante enlaces explícitos. Proporcione un algoritmo simple que permita encontrar el nodo del árbol almacenado en la posición implícita *I*.
- 21.18** El *montículo 2-D* es una estructura de datos que permite que cada elemento tenga dos claves individuales. La operación `deleteMin` se puede realizar con respecto a cualquiera de esas claves. La propiedad de ordenación del montículo 2-D consiste en que para cualquier nodo *X* situado a un profundidad par, el elemento almacenado en *X* tiene la clave #1 más pequeña de su subárbol y para cualquier nodo *X* situado a una profundidad impar, el elemento almacenado en *X* tiene la clave #2 más pequeña de su subárbol. Haga lo siguiente:
- Dibuje un posible montículo 2-D para los elementos (1, 8), (2, 7), (3, 6), (4, 5) y (5, 4).
  - Explique cómo encontrar el elemento con clave #1 mínima.
  - Explique cómo encontrar el elemento con clave #2 mínima.
- 21.19** Explique cómo colocar el conjunto inicial de lotes en dos cintas cuando el número de lotes no es un número de Fibonacci.
- 21.20** Suponga que representamos los montículos binarios mediante enlaces explícitos. Considere el problema de mezclar el montículo binario *lhs* con el *rhs*. Suponga que ambos montículos son árboles binarios completos perfectos, que contienen  $2^l - 1$  y  $2^r - 1$  nodos, respectivamente.
- Proporcione un algoritmo  $O(\log N)$  para mezclar los dos montículos si  $|l = r|$ .
  - Proporcione un algoritmo  $O(\log N)$  para mezclar los dos montículos si  $|l - r| = 1$ .
  - Proporcione un algoritmo  $O(\log^2 N)$  para mezclar los dos montículos independientemente de los valores de *l* y *r*.
- 21.21** Un *montículo minimal-maximal* es una estructura de datos que soporta tanto `deleteMin` como `deleteMax` con un coste logarítmico. La estructura es idéntica

a la del montículo binario. La propiedad de ordenación del montículo minimal-maximal consiste en que, para cada nodo  $X$  situado a una profundidad par, la clave almacenada en  $X$  es la más pequeña de su subárbol, mientras que para cada nodo  $X$  situado a una profundidad impar, la clave almacenada en  $X$  es la mayor de su subárbol. La raíz está situada a una profundidad par. Haga lo siguiente

- Dibuje un posible montículo minimal-maximal para los elementos 5, 6, 7, 9, 11, 13. Observe que existen muchos posibles montículos.
- Determine cómo encontrar los elementos mínimo y máximo.
- Proporcione un algoritmo para insertar un nuevo nodo en el montículo minimal-maximal.

**21.22** Un *treap* es un árbol de búsqueda binaria en el que cada nodo almacena un elemento, dos hijos y una prioridad aleatoriamente asignada que se genera en el momento de construir el nodo. Los nodos del árbol obedecen la ordenación usual de los árboles de búsqueda binaria, pero también tienen que mantener la ordenación de montículo con respecto a las prioridades. El *treap* es una buena alternativa al árbol de búsqueda equilibrado, porque el equilibrio se basa en las prioridades aleatorias en lugar de en los elementos. Por tanto, son de aplicación los resultados del caso promedio para los árboles de búsqueda binaria. Haga lo siguiente.

- Demuestre que una colección de elementos distintos, cada uno de los cuales tiene una prioridad diferente, puede representarse mediante un único *treap*.
- Indique cómo realizar la inserción en un *treap* utilizando un algoritmo de tipo abajo-arriba.
- Indique cómo realizar la inserción en un *treap* utilizando un algoritmo de tipo arriba-abajo.

## EN LA PRÁCTICA

**21.23** Escriba la rutina *percDown* con la declaración

```
static void percDown( AnyType [ ] a, int index, int size )
```

Recuerde que el montículo maximal comienza en la posición 0, no en la posición 1.

## PROYECTOS DE PROGRAMACIÓN

**21.24** Implemente tanto *heapsort* como el algoritmo de ordenación rápida y compare sus rendimientos para entradas tanto ordenadas como aleatorias. Utilice diferentes tipos de datos para las pruebas.

**21.25** Escriba un programa para comparar el tiempo de ejecución que se obtiene al utilizar el constructor de un parámetro de *PriorityQueue* para inicializar el montículo con  $N$  elementos con el que se obtiene en el caso en que comenzemos con una *PriorityQueue* vacía y efectuemos  $N$  inserciones separadas. Ejecute su programa para entradas ordenadas, ordenadas a la inversa y aleatorias.

**21.26** Suponga que tiene una serie de cajas, cada una de las cuales puede admitir un peso total de 1,0, y suponga también que dispone de una serie de elementos  $I_1, I_2, I_3, \dots$ ,

$i_N$  con pesos  $w_1, w_2, w_3, \dots, w_N$ , respectivamente. El objetivo es empaquetar todos los elementos utilizando el menor número de cajas posible, sin colocar más peso en ninguna de las cajas de lo que la caja puede soportar. Por ejemplo, si los elementos tienen pesos 0,4, 0,4, 0,6 y 0,6, el problema se puede resolver con dos cajas. Este problema es difícil y no se conoce ningún algoritmo eficiente. Existen varias estrategias que proporcionan embalajes buenos pero no óptimos. Escriba programas para implementar de manera eficiente las siguientes estrategias de aproximación.

- Explore los elementos en el orden dado, coloque cada nuevo elemento en la caja más llena que pueda aceptarlo sin que se sobrepase el peso permitido. Utilice una cola con prioridad para determinar la caja en la que debe ir cada elemento.
- Ordene los elementos, colocando primero el elemento más pesado; después utilice la estrategia del apartado (a).

**21.27** Haga que `PriorityQueue` soporte `decreaseKey` de la forma siguiente: defina una clase anidada que implemente `PriorityQueue.Position`. El montículo binario estará representado por una matriz de objetos, en la que cada objeto almacena un elemento de datos y su índice. Cada objeto `PriorityQueue.Position` almacena una referencia que apunta al objeto correspondiente de la matriz.

**21.28** Suponga que tiene un hueco en el nodo  $X$ . La rutina `percDown` normal consiste en comparar los hijos de  $X$  y luego desplazar hacia arriba el hijo hasta  $X$  si es mayor (en el caso de un montículo maximal) que el elemento que hay que colocar, desplazando así el hueco hacia abajo. Hay que detenerse cuando el colocar el nuevo elemento en el hueco sea seguro. Considere la siguiente estrategia alternativa para `percDown`. Mueva los elementos hacia arriba y el hueco hacia abajo lo más posible sin comprobar si puede insertarse la nueva celda. Estas acciones colocarían la nueva celda en una hoja y probablemente violarían la ordenación del montículo. Para corregir la ordenación del montículo, propague la nueva celda hacia arriba de la forma usual. La expectativa es que la propagación hacia arriba solo será de uno o dos niveles como promedio. Escriba una rutina para incluir esta idea. Compare el tiempo de ejecución con el de una implementación estándar de `heapsort`.



## Referencias

El montículo binario fue descrito por primera vez en el contexto del algoritmo de ordenación `heapsort` en [8]. El algoritmo `buildHeap` de tiempo lineal es de [4]. La referencia [7] proporciona resultados precisos sobre el número de comparaciones y movimientos de datos utilizados por `heapsort` en los casos mejor, peor y promedio. La ordenación externa se analiza en detalle en [6]. El Ejercicio 21.16 está resuelto en [5]. El Ejercicio 21.21 está resuelto en [2]. El Ejercicio 21.18 está resuelto en [3]. Los *treaps* 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 Heaps 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. Vol. 3: Sorting and Searching* 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 1998.
- 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  
cinco

# Estructuras de datos avanzadas



Capítulo 22 Árboles splay

Capítulo 23 Mezcla de colas con prioridad

Capítulo 24 La clase conjunto disjunto



## Árboles splay

En este capítulo vamos a describir una notable estructura de datos denominada *árbol splay*, que soporta todas las operaciones de los árboles de búsqueda binaria, pero no garantiza una velocidad  $O(\log N)$  de caso peor. En lugar de ello, sus cotas están *amortizadas*, lo que quiere decir que, aunque las operaciones individuales pueden ser costosas, se garantiza que cualquier secuencia de operaciones se comporte como si cada operación de la secuencia exhibiera un comportamiento logarítmico. Puesto que esta garantía es más débil que la proporcionada por los árboles de búsqueda equilibrados, solo hacen falta los datos y dos enlaces por nodo para cada elemento, y las operaciones son algo más simples de codificar. El árbol splay tiene algunas otras propiedades interesantes, que revelaremos en este capítulo.

En este capítulo veremos

- Los conceptos de amortización y autoajuste.
- El algoritmo básico abajo-arriba para árboles splay y una demostración de que tiene un coste amortizado logarítmico por cada operación.
- La implementación de los árboles splay con un algoritmo arriba-abajo, utilizando una implementación completa de árbol splay (incluyendo un algoritmo de borrado).
- La comparación de los árboles splay con otras estructuras de datos.

### 22.1 Autoajuste y análisis amortizado

Aunque los árboles de búsqueda equilibrados proporcionan un tiempo de ejecución logarítmico en caso peor por cada operación, presentan varias limitaciones.

- Requieren almacenar un elemento extra de información por cada nodo.
- Son complicados de implementar. Como resultado, las inserciones y borrados son costosos y potencialmente proclives a errores.
- No proporcionan ninguna ventaja cuando las entradas proporcionadas son favorables.

Examinemos las consecuencias de cada una de estas deficiencias. En primer lugar, los árboles de búsqueda equilibrados requieren un miembro de datos extra. Aunque en teoría este miembro

El problema real es que los miembros de datos adicionales añaden complicaciones.

puede ser tan pequeño como un único bit (como por ejemplo en un árbol rojo-negro), en la práctica el miembro de datos extra utiliza un entero completo para el almacenamiento, con el fin de satisfacer las restricciones de hardware. Puesto que las memorias de computadora están pasando a tener tamaños enormemente grandes, deberíamos preguntarnos si preocuparse acerca de la memoria tiene mucho sentido. La respuesta es que, en la mayoría de los casos, probablemente no lo tenga, salvo por el hecho de que mantener los miembros adicionales de datos requiere código más complejo y tiende a proporcionar tiempos de ejecución más largos y también un mayor número de errores. De hecho, identificar si es correcta la información de equilibrado para un árbol de búsqueda es difícil, porque los errores solo llevan a obtener un árbol desequilibrado. Si un cierto caso es ligeramente erróneo, detectar los errores puede ser difícil. Por tanto, en términos prácticos, los algoritmos que nos permitan eliminar algunas complicaciones sin sacrificar el rendimiento merecen un análisis atento.

La regla 90-10 afirma que el 90 por ciento de los accesos se refieren al 10 por ciento de los elementos de datos. Sin embargo, los árboles de búsqueda equilibrados no aprovechan las consecuencias de esta regla.

En segundo lugar, el rendimiento de caso peor, de caso promedio y de caso mejor de un árbol de búsqueda equilibrado son esencialmente idénticos. Un ejemplo sería una operación `find` para un cierto elemento  $X$ . Podríamos razonablemente esperar que no solo el coste de la operación `find` fuera logarítmico, sino también que si realizamos inmediatamente un segunda operación `find` para  $X$ , el segundo acceso fuera más barato que el primero. Sin embargo, en un árbol rojo-negro esta condición no se cumple. También esperaríamos que, si realizamos un acceso de  $X$ ,  $Y$  y  $Z$  en este orden, un segundo conjunto de accesos para la misma secuencia resultara sencillo. Esta suposición es importante debido a la *regla 90-10*. Como sugieren los estudios empíricos, la regla 90-10 afirma que, en la práctica, el 90 por ciento de los accesos se refieren al 10 por ciento de los elementos de datos. Por tanto, lo que queremos es simplificar el acceso para ese 90 por ciento de casos, pero los árboles de búsqueda equilibrados no aprovechan las consecuencias de esta regla.

La regla 90-10 se ha utilizado durante muchos años en los sistemas de E/S de disco. Una *caché* almacena en memoria principal el contenido de algunos de los bloques del disco. La esperanza es que cuando se solicita un acceso a disco, el bloque pueda encontrarse en la *caché* de la memoria principal, ahorrándonos así un costoso acceso a disco. Por supuesto, en la memoria solo se pueden almacenar un número relativamente pequeño de bloques de disco. Aun así almacenar en la *caché* los bloques de disco a los que se ha accedido más recientemente permite obtener grandes mejoras en el rendimiento, debido a que se suele acceder repetidamente a muchos de los mismos bloques de disco. Los exploradores web hacen uso de la misma idea: una *caché* almacena localmente las páginas web previamente visitadas.

### 22.1.1 Cotas de tiempo amortizadas

Estamos pidiendo mucho: queremos evitar la información de equilibrado y al mismo tiempo aprovecharnos de la regla 90-10. Naturalmente, cabe esperar que habrá que sacrificar a cambio algunas características del árbol de búsqueda equilibrado.

Lo que decidimos sacrificar es el rendimiento logarítmico de caso peor. Esperamos no tener que mantener la información de equilibrado, así que este sacrificio parece inevitable. Sin

embargo, no podemos aceptar el rendimiento típico de un árbol de búsqueda binaria no equilibrado. Pero existe un compromiso razonable: un tiempo  $O(N)$  para un único acceso puede ser aceptable, siempre y cuando no se produzca demasiado a menudo. En particular, si cualesquiera  $M$  operaciones (comenzando con la primera operación) requieren un tiempo total de caso peor de  $O(M \log N)$ , el hecho de que algunas operaciones sean costosas puede no tener ninguna importancia. Cuando podemos demostrar que una cota de caso peor para una secuencia de operaciones es mejor que la correspondiente cota obtenida considerando cada operación por separado, y ese coste total de la secuencia se puede distribuir equitativamente entre cada una de las operaciones que la forman, lo que hacemos es un *análisis amortizado* y decimos que el tiempo de ejecución está *amortizado*. En el ejemplo anterior, tenemos un coste amortizado logarítmico. Es decir, algunas operaciones aisladas pueden requerir un tiempo mayor que el logarítmico, pero tenemos garantizado que eso se compense mediante algunas operaciones menos costosas que se produzcan anteriormente en la secuencia.

Sin embargo, las cotas amortizadas no son siempre aceptables. Específicamente, si una única operación desfavorable consume demasiado tiempo, necesitaremos realmente obtener cotas de caso peor en lugar de cotas amortizadas. Aun así, en muchos casos, se utiliza una estructura de datos como parte de un algoritmo y solo es importante la cantidad de tiempo empleada por la estructura de datos en el curso de la ejecución del algoritmo.

Ya hemos presentado un ejemplo de cota amortizada. Cuando implementábamos la duplicación de la matriz en una pila o cola, el coste de una única operación podía ser constante, en caso de que no hiciera falta duplicación, o  $O(N)$ , en caso de que la duplicación fuera necesaria. Sin embargo, para cualquier secuencia de  $M$  operaciones con la pila o la cola, se garantizaba que el coste total fuera  $O(M)$ , dandonos un coste amortizado constante por operación. El hecho es que el paso de duplicación de la matriz no tiene ninguna importancia porque su coste se puede distribuir entre muchas operaciones anteriores poco costosas.

**El análisis amortizado**  
acota el coste de una secuencia de operaciones y distribuye ese coste equitativamente entre cada una de las operaciones de la secuencia.

### 22.1.2 Una estrategia simple de autoajuste (que no funciona)

En un árbol de búsqueda binaria no cabe esperar poder almacenar los elementos a los que se accede con frecuencia en una simple tabla. La razón es que la técnica de almacenamiento en caché aprovecha la gran discrepancia existente entre los tiempos de acceso a la memoria principal y a disco. Recuerde que el coste de un acceso en un árbol de búsqueda binaria es proporcional a la profundidad del nodo al que se accede. Entonces, lo que si podemos intentar es reestructurar el árbol, moviendo hacia la raíz los elementos a los que se accede frecuentemente. Aunque este proceso cuesta un tiempo adicional durante la primera operación `find`, podría merecer la pena a largo plazo.

La forma más fácil de mover hacia la raíz un elemento al que se accede frecuentemente consiste en rotarlo continuamente con su padre, acercando el elemento a la raíz, lo cual es un proceso que se denomina *estrategia de rotación hacia la raíz*. Entonces, cuando se acceda al elemento una segunda vez, el coste de acceso será barato, etc. Incluso si se realizaran algunas otras

**La estrategia de rotación hacia la raíz**  
reordena un árbol de búsqueda binaria después de cada acceso, con el fin de acercar a la raíz los elementos a los que se accede frecuentemente.

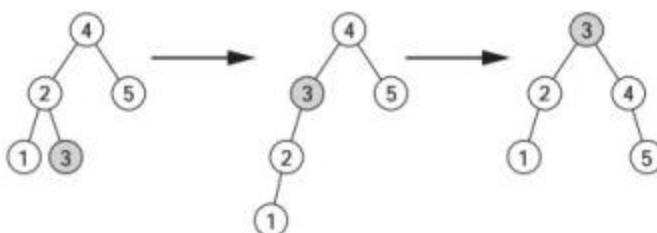


Figura 22.1 Estrategia de rotación hacia la raíz aplicada al acceder al nodo 3.

La estrategia de rotación hacia la raíz es buena si resulta aplicable la regla 90-10. Puede ser una mala estrategia cuando dicha regla no sea de aplicación.

operaciones antes de volver a acceder al elemento, dicho elemento seguiría estando próximo a la raíz y por tanto se podría localizar rápidamente. En la Figura 22.1 se muestra un ejemplo de aplicación de la estrategia de rotación hacia la raíz al nodo 3.<sup>1</sup>

Como resultado de la rotación, los accesos futuros al nodo 3 serán poco costosos (durante un tiempo). Lamentablemente, en el proceso de mover el nodo 3 dos niveles hacia arriba, los nodos 4 y 5 se han movido cada uno de ellos un nivel hacia abajo. Por tanto, si los patrones de acceso no siguen la regla 90-10, puede que se produzca una larga secuencia de accesos inconvenientes. Como resultado, la regla de rotación hacia la raíz no exhibe un comportamiento amortizado logarítmico, lo que probablemente sea inaceptable. En el Teorema 22.1 se ilustra un caso poco conveniente.

### Teorema 22.1

Hay secuencias arbitrariamente largas para las cuales *M* accesos de rotación hacia la raíz utilizan un tiempo  $\Theta(MN)$ .

#### Demostración

Considere el árbol formado por la inserción de  $1, 2, 3, \dots, N$  en un árbol inicialmente vacío. El resultado es un árbol compuesto únicamente por hijos izquierdos. Este resultado no es malo, ya que el tiempo requerido para construir el árbol solo es de  $O(N)$  en total.

Como se ilustra en la Figura 22.2, cada elemento recién añadido se hace hijo de la raíz. Entonces, solo hace falta una rotación para colocar el nuevo elemento en la raíz. El inconveniente, como se muestra en la Figura 22.3, es que acceder al nodo con clave 1 requiere  $N$  unidades de tiempo. Después de haber completado las rotaciones, el acceso al nodo con clave 2 requiere  $N$  unidades de tiempo y el acceso a la clave 3 requiere  $N - 1$  unidades de tiempo. El total para acceder a las  $N$  claves por orden es  $N + \sum_{i=2}^N i = \Theta(N^2)$ . Después de haber accedido a todas las claves, el árbol vuelve a su estado original y podemos repetir la secuencia. Por tanto, tenemos una cota amortizada de solo  $\Theta(N)$ .

Continúa

<sup>1</sup> Una inserción cuenta como un acceso. Por tanto, un elemento se insertaría siempre como una hoja y luego se rotaría inmediatamente hacia la raíz. Una búsqueda que no tenga éxito cuenta como un acceso a la hoja en la que la búsqueda termine.

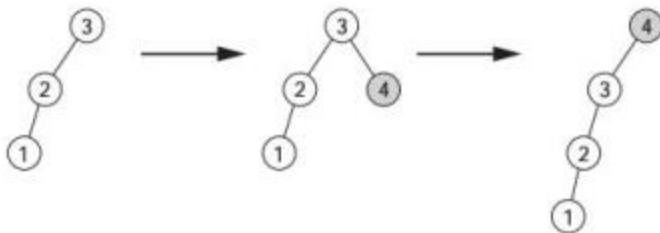
Demostración  
(cont.)

Figura 22.2 Inserción de 4 utilizando la estrategia de rotación hacia la raíz.

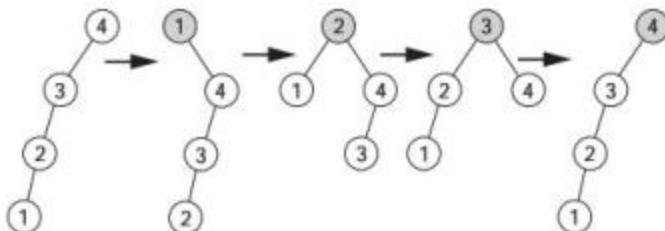


Figura 22.3 el acceso secuencial a los elementos requiere un tiempo cuadrático.

## 22.2 El árbol splay básico de tipo abajo-arriba

Conseguir un coste amortizado logarítmico parece imposible porque, cuando desplazamos un elemento hacia la raíz mediante rotaciones, otros elementos son empujados hacia abajo. Aparentemente, eso haría que siempre hubiera nodos muy profundos, si no mantenemos información de equilibrado. Sin embargo, sorprendentemente, podemos aplicar una simple corrección a la estrategia de rotación hacia la raíz que nos permite obtener la cota amortizada logarítmica. La implementación de este método ligeramente más complicado de rotación hacia la raíz, que se denomina *splaying*, conduce a la obtención del *árbol splay de tipo abajo-arriba*.

La estrategia para los árboles splay es similar a la estrategia simple de rotación hacia la raíz, pero presenta un sutil diferencia. Seguimos rotando de abajo hacia arriba a lo largo del camino de acceso (posteriormente en el capítulo describiremos una estrategia de tipo arriba-abajo). Si  $X$  es un nodo no raíz en el camino de acceso en el que estamos efectuando las rotaciones y si el padre de  $X$  es la raíz del árbol, simplemente rotamos  $X$  y la raíz, como se muestra en la Figura 22.4. Esta rotación es la última a lo largo del camino de acceso y coloca a  $X$  en la raíz. Observe que esta acción es exactamente la misma que en el algoritmo convencional de rotación hacia la raíz, y nos referiremos a ella como caso *zig*.

Si el padre de  $X$  no es la raíz del árbol,  $X$  tendrá tanto un parent  $P$  como un abuelo  $G$ , y debemos considerar dos casos y simetrías. El primer caso es el caso *zig-zag*, que se corresponde con el caso interior de los árboles AVL.

En un *árbol splay de tipo abajo-arriba* básico, los elementos se rotan hacia la raíz utilizando un método ligeramente más complicado que el que se emplea en la estrategia simple de rotación hacia la raíz.

Los casos *zig* y *zig-zag* son idénticos al mecanismo de rotación hacia la raíz.

Aquí  $X$  es un hijo derecho y  $P$  es un hijo izquierdo (o viceversa). Realizamos una doble rotación, exactamente como la doble rotación AVL, tal como se muestra en la Figura 22.5. Observe que, como una doble rotación es lo mismo que dos rotaciones simples abajo-arriba, este caso no difiere de la estrategia de rotación hacia la raíz. En la Figura 22.1, la modificación del nodo 3 es una única rotación en zig-zag.

**El caso zig-zig es original del árbol splay.**

El caso final, el caso *zig-zig*, es original de los árboles splay y se corresponde con el caso exterior 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 22.6 en el árbol de la derecha. Observe que este método difiere de la estrategia de rotación hacia la raíz. La modificación zig-zig realiza una rotación entre  $P$  y  $G$  y luego entre  $X$  y  $P$ , mientras que la estrategia de rotación hacia la raíz efectúa una rotación entre  $X$  y  $P$  y luego entre  $X$  y  $G$ .

**El splaying tiene el efecto de reducir aproximadamente a la mitad la profundidad de la mayoría de los nodos situados en el camino de acceso, incrementando en dos niveles como mucho la profundidad de algunos otros nodos.**

El caso final, el caso *zig-zig*, es original de los árboles splay y se corresponde con el caso exterior 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 22.6 en el árbol de la derecha. Observe que este método difiere de la estrategia de rotación hacia la raíz. La modificación zig-zig realiza una rotación entre  $P$  y  $G$  y luego entre  $X$  y  $P$ , mientras que la estrategia de rotación hacia la raíz efectúa una rotación entre  $X$  y  $P$  y luego entre  $X$  y  $G$ .

La diferencia parece bastante pequeña y el hecho de que tenga importancia resulta algo sorprendente. Para ver esta diferencia, considere la secuencia que nos proporcionó esos pobres resultados en el Teorema 22.1. De nuevo, insertamos las claves 1, 2, 3, ...,  $N$  en un árbol inicialmente vacío en un tiempo total lineal y obtenemos un árbol no equilibrado que solo tiene hijos izquierdos. Sin embargo, el resultado de las modificaciones del árbol splay es algo mejor, como se muestra en la Figura 22.7. Después de desplazar el nodo 1, que requiere  $N$  accesos a nodos, una modificación del nodo 2 solo precisa aproximadamente  $N/2$  accesos en lugar de  $N - 1$  accesos. El proceso de *splaying* no solo mueve hacia la raíz el nodo al que se accede, sino que también divide aproximadamente entre 2 la profundidad de la mayoría de los nodos en el camino de acceso (algunos nodos poco profundos son empujados hacia abajo como máximo dos niveles). Una modificación posterior del nodo 2 hará que los nodos estén a una distancia  $N/4$  de la raíz. El *splaying* se repite hasta que la profundidad pasa a ser aproximadamente  $\log N$ . De hecho, un complicado análisis permite demostrar que lo que solía ser un caso poco conveniente para el algoritmo de rotación hacia la raíz es, en realidad, un buen caso para la técnica de *splaying*: el acceso secuencial



Figura 22.4 El caso zig (rotación simple normal).

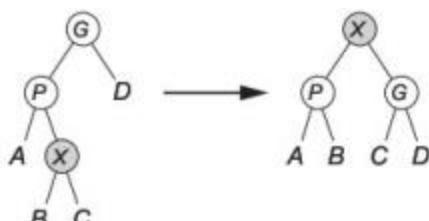


Figura 22.5 El caso zig-zag (igual a una doble rotación); el caso simétrico se ha omitido.

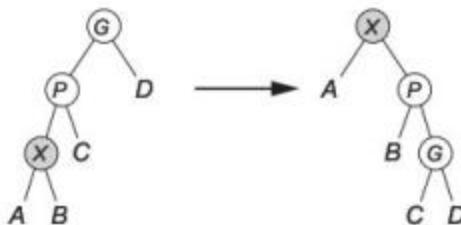


Figura 22.6 El caso zig-zig (original de los árboles splay); el caso simétrico se ha omitido.

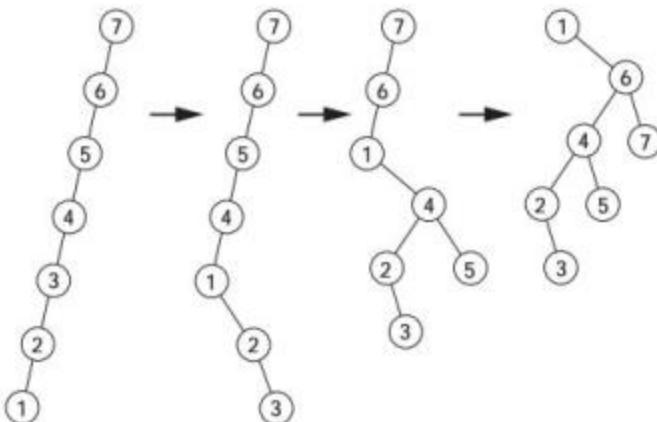


Figura 22.7 Resultado de aplicar la técnica de splaying al nodo 1 (tres zig-zig).

a los  $N$  elementos del árbol splay requiere un tiempo total de solo  $O(N)$ . Por tanto, salimos ganando con las entradas favorables. En la Sección 22.4 veremos, mediante un sutil análisis, que no existen secuencias de acceso inconvenientes.

## 22.3 Operaciones básicas con un árbol splay

Como hemos mencionado anteriormente, se realiza una operación splay después de cada acceso. Cuando se lleva a cabo una inserción, realizamos un splay. Como resultado, el elemento recién insertado pasa a ser la raíz del árbol. En caso contrario, podríamos invertir un tiempo cuadrático construyendo un árbol de  $N$  elementos.

Para la operación `find`, aplicamos el splay al último nodo al que se ha accedido durante la búsqueda. Si la búsqueda ha tenido éxito, el nodo encontrado será desplazado y pasará a convertirse en la nueva raíz. Si la búsqueda no tiene éxito, se aplica la operación de splay al último nodo al que se ha accedido antes de alcanzar la referencia `null`, con lo que ese nodo pasa a ser la nueva raíz. Este comportamiento es necesario porque, de no hacer las cosas así, podríamos estar repetidamente llevando a cabo una operación `find` en busca del elemento 0 en el árbol inicial de la Figura 22.7 y utilizar

Después de haber insertado como hoja un elemento, se desplaza con una operación splay hasta la raíz.

Todas las operaciones de búsqueda incorporan un splay.

un tiempo lineal por operación. De la misma forma, operaciones como `findMin` y `findMax` realizan también un splay después del acceso.

Las operaciones interesantes son los borrados. Recuerde que `deleteMin` y `deleteMax` son operaciones importantes en las colas con prioridad. Con los árboles splay, estas operaciones pasan a ser simples. Podemos implementar `deleteMin` de la forma siguiente. En primer lugar, realizamos una operación `findMin`. Esto lleva al elemento mínimo hasta la raíz y por la propiedad estructural de los árboles de búsqueda binaria, no habrá hijo izquierdo. Podemos utilizar el hijo derecho como nueva raíz. De forma similar, `deleteMax` se puede implementar invocando `findMax` y estableciendo como nueva raíz el hijo izquierdo de la raíz que queda después de la operación de splay.

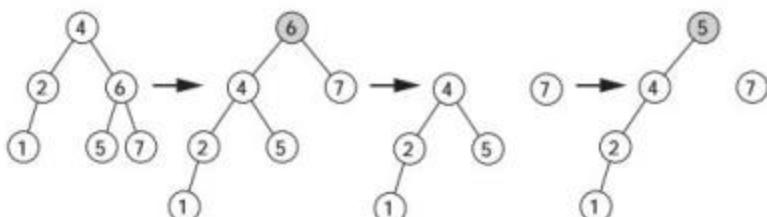
Incluso la operación `remove` es simple. Para realizar un borrado, accedemos al nodo que hay que borrar, lo que coloca a ese nodo en la raíz. Si se borra, obtenemos dos subárboles,  $L$  y  $R$  (izquierdo y derecho). Si encontramos el elemento mayor de  $L$ , mediante una operación `findMax`, rotamos ese elemento máximo hasta la raíz de  $L$  y la raíz de  $L$  no tendrá ningún hijo derecho. Finalizamos la operación `remove` haciendo que  $R$  pase a ser el hijo derecho de la raíz de  $L$ . En la Figura 22.8 se muestra un ejemplo de la operación `remove`.

Las operaciones de borrado  
son mucho más simples de  
lo normal. También contiene  
un paso de splaying  
(en ocasiones, dos).

El coste de la operación `remove` equivale a dos operaciones splay. Todas las demás operaciones cuestan una operación splay. Por tanto, necesitamos analizar el coste de una serie de pasos de splay. La siguiente sección muestra que el coste amortizado de un splay es igual, como máximo, a  $3 \log N + 1$  rotaciones simples. Entre otras cosas, esto significa que no tenemos que preocuparnos porque el algoritmo de borrado descrito anteriormente esté sesgado. La cota amortizada del árbol splay garantiza que cualquier secuencia de  $M$  operaciones splay utilizará como máximo  $3M \log N + M$  rotaciones de árbol. En consecuencia, cualquier secuencia de  $M$  operaciones que comience a partir de un árbol vacío, requerirá un tiempo total que será como máximo  $O(M \log N)$ .

## 22.4 Análisis del splaying abajo-arriba

El análisis del algoritmo para árboles splay es complicado porque cada operación splay puede variar entre unas pocas rotaciones y  $O(N)$  rotaciones. Cada operación splay puede modificar drásticamente la estructura del árbol. En esta sección vamos a demostrar que el coste amortizado de una operación splay es equivalente a como máximo  $3 \log N + 1$  rotaciones simples. La cota amortizada del árbol splay garantiza que cualquier secuencia de  $M$  splays utilizará como máximo  $3M \log N + M$



**Figura 22.8** La operación `remove` aplicada al nodo 6: en primer lugar, se lleva 6 hasta la raíz con un splay, lo que deja dos subárboles; se realiza una operación `findMax` con el subárbol izquierdo, llevando 5 hasta la raíz de ese subárbol; después se puede conectar el subárbol derecho (no mostrado en la figura).

rotaciones de árbol y, en consecuencia, cualquier secuencia de  $M$  operaciones que parte de un árbol vacío requerirá un tiempo total que será como máximo  $O(M \log N)$ .

Para demostrar la existencia de esta cota, introducimos una función de análisis denominada *función potencial*. Esta función, que no es mantenida por el algoritmo, es simplemente una herramienta de análisis utilizada para establecer la cota temporal requerida. Su elección no es obvia y es el resultado de un largo proceso de prueba y error.

Para cualquier nodo  $i$  del árbol splay, sea  $S(i)$  el número de descendientes de  $i$  (incluyendo al propio  $i$ ). La función potencial es la suma, para todos los nodos  $i$  del árbol  $T$ , del logaritmo de  $S(i)$ . Específicamente,

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

Para simplificar la notación, hacemos  $R(i) = \log S(i)$ , lo que nos da

$$\Phi(T) = \sum R(i)$$

El término  $R(i)$  representa el *rango* del nodo  $i$ , que es el logaritmo de su tamaño. Observe que el rango de la raíz es  $\log N$ . Recuerde que ni los rangos ni los tamaños son mantenidos por los algoritmos del árbol splay (a menos, por supuesto, que hagan falta estadísticas de orden). Cuando se realiza una rotación zig, solo varían los rangos de los dos nodos implicados en la rotación. Cuando se lleva a cabo una rotación zig-zig o zig-zag, solo varían los rangos de los tres nodos implicados en la rotación. Y, finalmente, un único splay está compuesto por un cierto número de rotaciones zig-zig o zig-zag seguidas quizás por una rotación zig. Cada rotación zig-zig o zig-zag puede contarse como dos rotaciones simples.

Para el Teorema 22.2, vamos a hacer que  $\Phi_i$  sea la función potencial del árbol inmediatamente después del  $i$ -ésimo splay y que  $\Phi_0$  sea la función potencial antes del primer splay.

### Teorema 22.2

Si la  $i$ -ésima operación splay utiliza  $r_i$  rotaciones,  $\Phi_i - \Phi_{i+1} + r_i \leq 3 \log N + 1$ .

Antes de demostrar el Teorema 22.2, vamos a ver lo que significa. El coste de  $M$  operaciones splay se puede tomar como  $\sum_{i=1}^M r_i$  rotaciones. Si las  $M$  operaciones splay son consecutivas (es decir, si no hay inserciones o borrados intermedios), el potencial de cada árbol después del  $i$ -ésimo splay es igual que antes del  $(i+1)$ -ésimo splay. Por tanto, podemos utilizar el Teorema 22.2  $M$  veces para obtener la siguiente secuencia de ecuaciones:

$$\begin{aligned} \Phi_1 - \Phi_0 + r_1 &\leq 3 \log N + 1 \\ \Phi_2 - \Phi_1 + r_2 &\leq 3 \log N + 1 \\ \Phi_3 - \Phi_2 + r_3 &\leq 3 \log N + 1 \\ &\dots \\ \Phi_M - \Phi_{M-1} + r_M &\leq 3 \log N + 1 \end{aligned} \tag{22.1}$$

El análisis del árbol splay es complicado y forma parte de una teoría mucho más amplia de análisis amortizado.

La función potencial es una herramienta de análisis utilizada para establecer la cota temporal requerida.

El rango de un nodo es el logaritmo de su tamaño. Los rangos y tamaños no son mantenidos por el algoritmo, sino que se trata de simples herramientas de análisis de cara a la demostración. Solo varían los rangos de los nodos situados en el camino de aplicación del splay.

En todas las demostraciones de esta sección vamos a utilizar el concepto de sumas telescópicas.

Estas ecuaciones están dispuestas telescópicamente, así que si las sumamos obtenemos

$$\Phi_M - \Phi_0 + \sum_{i=1}^M r_i \leq (3 \log N + 1) M \quad (22.2)$$

lo que nos da la siguiente cota para el número total de rotaciones

$$\sum_{i=1}^M r_i \leq (3 \log N + 1) M - (\Phi_M - \Phi_0)$$

Ahora consideremos lo que sucede cuando las inserciones están entremezcladas con las operaciones de búsqueda. El potencial de un árbol vacío es 0, por lo que cuando se inserta un nodo en el árbol como hoja, antes del splay, el potencial del árbol se incrementa en como máximo  $\log N$  (lo cual vamos a demostrar enseguida). Suponga que se utilizan  $r_i$  rotaciones para una inserción y que el potencial antes de la inserción es  $\Phi_{i-1}$ . Después de la inserción, el potencial será como máximo  $\Phi_{i-1} + \log N$ . Después del splay que mueve el nodo insertado hasta la raíz, el nuevo potencial satisfará

$$\begin{aligned} \Phi_i - (\Phi_{i-1} + \log N) + r_i &\leq 3 \log N + 1 \\ \Phi_i - \Phi_{i-1} + r_i &\leq 4 \log N + 1 \end{aligned} \quad (22.3)$$

Suponga además que se realizan  $F$  búsquedas e  $I$  inserciones y que  $\Phi_i$  representa el potencial después de la  $i$ -ésima operación. Entonces, como cada operación `find` está gobernada por el Teorema 22.2 y cada inserción está gobernada por la Ecuación 22.3, la lógica de la combinación telescópica nos indica que

$$\sum_{i=1}^M r_i \leq (3 \log N + 1) F + (4 \log N + 1) I - (\Phi_M - \Phi_0) \quad (22.4)$$

Además, antes de la primera operación el potencial es 0 y como nunca puede ser negativo,  $\Phi_M - \Phi_0 \geq 0$ . En consecuencia, obtenemos

$$\sum_{i=1}^M r_i \leq (3 \log N + 1) F + (4 \log N + 1) I \quad (22.5)$$

lo que demuestra que el coste de cualquier secuencia de búsquedas e inserciones es, como máximo, logarítmico por cada operación. Un borrado es equivalente a dos operaciones splay, así que también es logarítmico. Por tanto, lo que nos resta es demostrar las dos afirmaciones que tenemos pendientes: en concreto, el Teorema 22.2 y el hecho de que la inserción de un nodo añade como máximo  $\log N$  al potencial. Vamos a demostrar ambos teoremas utilizando argumentos telescópicos. Primero vamos a ocuparnos del enunciado relativo a la inserción, que repetimos como Teorema 22.3.

### Teorema 22.3

La inserción como hoja del  $N$ -ésimo nodo en un árbol añade como máximo  $\log N$  al potencial del árbol.

#### Demostración

Los únicos nodos cuyos rangos se ven afectados son aquellos que se encuentran en el camino que va desde la hoja insertada hasta la raíz. Sean  $S_1, S_2, \dots, S_k$  sus tamaños antes de la inserción y observe que  $S_k = N - 1$  y  $S_1 < S_2 < \dots < S_k$ . Sean  $S'_1, S'_2, \dots, S'_k$

Continúa

**Demostración  
(cont.)**

los tamaños después de la inserción. Claramente,  $S_i \leq S_{j+1}$  para  $i < j$ , dado que  $S_i = S_j + 1$ . En consecuencia,  $R'_i \leq R'_{j+1}$ . El cambio en el potencial será

$$\sum_{i=1}^M (R'_i - R_i) \leq R'_k - R_k + \sum_{i=1}^{k-1} (R'_{j+1} - R_i) \leq \log N - R_1 \leq \log N$$

Para demostrar el Teorema 22.2 vamos a descomponer cada splay en sus partes constituyentes zig, zig-zag y zig-zig y establecer una cota para el coste de cada tipo de rotación. Aplicando la lógica telescopica a estas cotas, obtendremos una cota para la operación de splay. Antes de continuar, necesitamos enunciar un teorema técnico, el Teorema 22.4.

**Teorema 22.4**

Si  $a + b \leq c$  y  $a$  y  $b$  son ambos enteros positivos, entonces  $\log a + \log b \leq 2 \log c - 2$ .

**Demostración**

Por la desigualdad de las medias aritmética y geométrica,  $\sqrt{ab} \leq (a + b)/2$ . Por tanto,  $\sqrt{ab} \leq c/2$ . Elevando ambos lados al cuadrado tenemos  $\sqrt{ab} \leq c^2/4$ , entonces, tomando logaritmos en ambos lados demostramos el teorema.

Ahora estamos preparados para demostrar el Teorema 22.2

### 22.4.1 Demostración de la cota de splaying

En primer lugar, si el nodo en el que hay que aplicar el splay ya se encuentra en la raíz, no se producirá ninguna rotación y no habrá ningún cambio de potencial. Por tanto, el teorema es trivialmente cierto y podemos asumir que se realizará al menos una rotación. Sea  $X$  el nodo implicado en el splay. Tenemos que demostrar que, si se realizan  $r$  rotaciones (cada zig-zig o zig-zag cuenta como dos rotaciones),  $r$  más el cambio en el potencial es como máximo  $3 \log N + 1$ . A continuación, sea  $\Delta$  el cambio en el potencial provocado por cualquiera de los pasos zig, zig-zag o zig-zig. Por último, sean  $R(X)$  y  $S(X)$  el rango y el tamaño de cualquier nodo  $X$  inmediatamente antes de un paso de splay y sean  $R_f(X)$  y  $S_f(X)$  el rango y el tamaño de cualquier nodo  $X$  inmediatamente después de un paso de splay. A continuación se muestran las cotas que tenemos que demostrar. Para un paso zig que promocione el nodo  $X$ ,  $\Delta \leq 3(R_f(X) - R_i(X))$ ; para los otros dos pasos,  $\Delta \leq 3(R_f(X) - R_i(X)) - 2$ . Cuando sumamos estas cotas para todos los pasos que componen un splay, la suma telescopica nos da la cota deseada. Vamos a demostrar separadamente cada cota en los Teoremas 22.5 a 22.7. Después podremos completar la demostración del Teorema 22.2 aplicando una suma telescopica.

**Teorema 22.5**

Para un paso zig,  $\Delta \leq 3(R_f(X) - R_i(X))$ .

**Demostración**

Como hemos mencionado anteriormente en esta sección, los únicos nodos cuyos rangos cambian en un paso zig son  $X$  y  $P$ . En consecuencia, el cambio de potencial es  $R_f(X) - R_i(X) + R_f(P) - R_i(P)$ . A partir de la Figura 22.4,  $S_f(P) < S_i(P)$ ; por tanto se deduce que  $R_f(P) - R_i(P) < 0$ . En consecuencia, el cambio de potencial satisface la desigualdad  $\Delta \leq R_f(X) - R_i(X)$ . Como  $S_f(X) > S_i(X)$ , se deduce que  $R_f(X) - R_i(X) > 0$ ; por tanto  $\Delta \leq 3(R_f(X) - R_i(X))$ .

Los pasos zig-zag y zig-zig son más complicados porque se ven afectados los rangos de tres nodos. En primer lugar, vamos a demostrar el teorema relativo al caso zig-zag.

**Teorema 22.6**

Para un paso zig-zag,  $\Delta \leq 3(R_f(X) - R_i(X)) - 2$ .

**Demostración**

Como antes, tenemos tres cambios, por lo que la variación de potencial está dada 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 22.5,  $S_f(X) = S_i(G)$ , por lo que sus rangos deben ser iguales. Por tanto, obtenemos

$$\Delta = -R_i(X) + R_f(P) - R_i(P) + R_f(G)$$

Asimismo,  $S_f(P) \geq S_i(X)$ . En consecuencia,  $R_f(P) \geq R_i(X)$ . Haciendo esta sustitución y reordenando los términos tenemos

$$\Delta \leq R_f(P) + R_f(G) - 2R_i(X) \quad (22.6)$$

A partir de la Figura 22.5,  $S_f(P) + S_f(G) \leq S_f(X)$ . Aplicando el Teorema 22.4, obtenemos  $S_f(P) + \log S_f(G) \leq 2 \log S_f(X) - 2$ , que por la definición de rango, se convierte en

$$R_f(P) + R_f(G) \leq 2R_f(X) - 2 \quad (22.7)$$

Sustituyendo la Ecuación 22.7 en la Ecuación 22.6 obtenemos

$$\Delta \leq 2R_f(X) - 2R_i(X) - 2 \quad (22.8)$$

En lo que respecta a la rotación zig,  $R_f(X) - R_i(X) > 0$ , por lo que podemos sumarla al lado derecho de la Ecuación 22.8, sacar factor común y obtener la expresión deseada

$$\Delta \leq 3(R_f(X) - R_i(X)) - 2$$

Finalmente, demostramos cuál es la cota para el caso zig-zig.

**Teorema 22.7**

Para un paso zig-zig,  $\Delta \leq 3(R_f(X) - R_i(X)) - 2$ .

**Demostración**

Como antes, tenemos tres cambios, por lo que la variación de potencial está dada 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 22.6,  $S_f(X) = S_i(G)$ , por lo que sus rangos deben ser iguales. Por tanto, obtenemos

$$\Delta = -R_i(X) + R_f(P) - R_i(P) + R_f(G)$$

Podemos obtener también que  $R_i(P) > R_i(X)$  y  $R_f(P) < R_f(X)$ . Haciendo esta sustitución y reordenando los términos tenemos

$$\Delta < R_f(X) + R_f(G) - 2R_i(X) \quad (22.9)$$

Continúa

Demostración (cont.)	A partir de la Figura 22.6, $S_i(X) + S_f(G) \leq S_f(X)$ , por lo que aplicando el Teorema 22.4, obtenemos $R_f(X) + R_f(G) \leq 2R_f(X) - 2 \quad (22.10)$ <p>Reordenando la Ecuación 22.10 nos queda</p> $R_f(G) \leq 2R_f(X) - R_f(X) - 2 \quad (22.11)$ <p>Sustituyendo la Ecuación 22.11 en la Ecuación 22.9, se obtiene</p> $\Delta \leq 3(R_f(X) - R_f(X)) - 2$
----------------------	--

Ahora que hemos establecido las cotas para cada paso de splaying, podemos finalmente completar la demostración del Teorema 22.2.

Demostración (Teorema 22.2)	Sea $R_0(X)$ el rango de $X$ antes del splay. Sea $R_i(X)$ el rango de $X$ después del $i$ -ésimo paso de splaying. Antes del último paso de splaying, todos los pasos de splaying deben ser zig-zag o zig-zig. Suponga que existen $k$ de tales pasos, entonces el número total de rotaciones realizadas hasta ese punto será $2k$ . La variación total de potencial será $\sum_{i=1}^k (3(R_i(X) - R_{i-1}(X)) - 2)$ . Esta suma nos da telescopicamente el resultado $3(R_k(X) - R_0(X)) - 2k$ . En este punto, el número total de rotaciones más la variación total de potencial estará acotado por $3R_k(X)$ , porque el término $2k$ se cancela y el rango inicial de $X$ no es negativo. Si la última rotación es un zig-zig o un zig-zag, entonces una continuación de la suma telescopica nos da un total de $3R(\text{raíz})$ . Observe que aquí, por un lado, el $-2$ del incremento de potencial cancela el coste de dos rotaciones. Por otro lado, esta cancelación no sucede en el zig, por lo que obtendríamos un total de $3R(\text{raíz}) + 1$ . El rango de la raíz es $\log N$ , por lo que entonces (en el caso peor) el número total de rotaciones más la variación de potencial durante un splay es como máximo $3 \log N + 1$ .
-----------------------------	--

Aunque es compleja, la demostración de la cota del árbol splay ilustra varios puntos interesantes. En primer lugar, el caso zig-zig es el aparentemente más costoso. Contribuye con una constante multiplicativa igual a 3, mientras que el zig-zag contribuye con una constante multiplicativa igual a 2. La demostración dejaría de ser válida si tratáramos de adaptarla al algoritmo de rotación hacia la raíz, porque en el caso zig, el número de rotaciones más la variación de potencial es  $R_f(X) - R(X) + 1$ . El 1 del final no puede eliminarse telescopicamente, por lo que no seríamos capaces de demostrar la existencia de una cota logarítmica. Y está bien que sea así, porque ya sabemos que una cota logarítmica sería incorrecta.

La técnica de análisis amortizado es muy interesante, y se han desarrollado algunos principios generales para formalizar este sistema de trabajo. Puede encontrar más detalles en las referencias incluidas al final del capítulo.

## 22.5 Árboles splay de tipo arriba-abajo

Una implementación directa de la estrategia de splay abajo-arriba requiere una pasada hacia abajo del árbol, para realizar un acceso, y luego una segunda pasada hacia arriba. Estas pasadas

Al igual que sucede con los árboles rojo-negro, los árboles splay arriba-abajo son más eficientes en la práctica que los correspondientes árboles abajo-arriba.

también que podemos utilizar nodos ficticios para evitar los casos especiales. En esta sección vamos a describir un *árbol splay de tipo arriba-abajo*, que mantiene la cota amortizada logarítmica, que es más rápido en la práctica y que solo utiliza un espacio adicional constante. Es el método recomendado por los inventores del árbol splay.

La idea básica que subyace al concepto de árbol splay de tipo arriba-abajo es que, a medida que descendemos por el árbol buscando un cierto nodo  $X$ , debemos tomar los nodos que se encuentran en el camino de acceso y quitar de en medio esos nodos y sus subárboles. También debemos realizar algunas rotaciones del árbol para garantizar la cota de tiempo amortizada.

Mantenemos tres árboles durante la pasada arriba-abajo.

En cualquier punto en mitad del splay, un cierto nodo actual  $X$  será la raíz de su subárbol; se representa en los diagramas como el árbol central. El árbol  $L$  almacena los nodos de valor inferior a  $X$ ; de forma similar, el árbol  $R$  almacena los nodos de valor superior a  $X$ . Inicialmente,  $X$  es la raíz de  $T$ , y  $L$  y  $R$  están vacíos. Descendiendo por el árbol dos niveles cada vez nos encontramos con un par de nodos. Dependiendo de si esos nodos son más pequeños o más grandes que  $X$ , los colocamos en  $L$  o en  $R$ , junto con los subárboles que no se encuentran en el camino de acceso hasta  $X$ . Así, el nodo actual del camino de búsqueda será *siempre* la raíz del árbol central. Cuando finalmente alcanzamos  $X$ , podemos conectar  $L$  y  $R$  a la parte inferior del árbol central. Como resultado,  $X$  habrá sido movido hasta la raíz. Las restantes tareas serán entonces colocar los nodos en  $L$  y  $R$  y realizar la reconexión del árbol al final, como ilustran los árboles mostrados en la Figura 22.9. Como suele ser habitual, omitimos tres casos simétricos.

En todos los diagramas,  $X$  es el nodo actual,  $Y$  es su hijo y  $Z$  es un nieto (si es que existe un nodo aplicable; el significado preciso del término *aplicable* quedará claro durante el análisis del caso zig).

Si la rotación debe ser un zig, el árbol que tiene  $Y$  como raíz pasa a ser la nueva raíz del árbol central. El nodo  $X$  y el subárbol  $B$  se conectan como hijo izquierdo del menor de los elementos de  $R$ ; el hijo izquierdo de  $X$  se hace lógicamente igual a `null`.<sup>2</sup> Como resultado,  $X$  será el nuevo elemento más pequeño de  $R$ , lo que facilita las futuras conexiones.

Observe que  $Y$  no tiene por qué ser una hoja para que el caso zig sea aplicable. Si el elemento buscado se encuentra en  $Y$ , el caso zig sería aplicable incluso si  $Y$  tuviera hijos. El caso zig también es aplicable si el elemento buscado es menor que  $Y$  y  $Y$  no tiene hijo izquierdo, incluso aunque  $Y$  tenga un hijo derecho, y también para el caso simétrico.

Podemos aplicar un análisis similar al caso zig-zig. El punto crucial es que se realiza una rotación entre  $X$  e  $Y$ . El caso zig-zag lleva hasta la parte superior del árbol central el nodo  $Z$  de la parte inferior y conecta los subárboles  $X$  e  $Y$  a  $R$  y  $L$ , respectivamente. Observe que  $Y$  se conecta a  $L$  y pasa a ser el elemento máximo de  $L$ .

<sup>2</sup> En el código que vamos a escribir aquí, el nodo más pequeño de  $R$  no tiene un enlace izquierdo `null`, porque no es necesario.

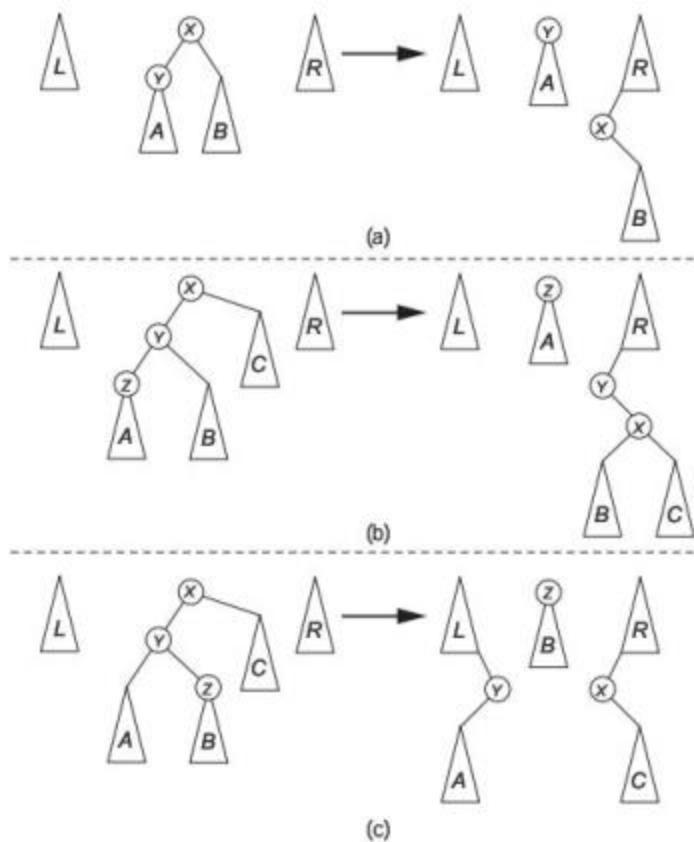


Figura 22.9 Rotaciones en un árbol splay de tipo arriba-abajo: (a) zig, (b) zig-zig y (c) zig-zag.

El paso zig-zag se puede simplificar en cierta medida, porque no se realizan rotaciones. En lugar de hacer que  $Z$  sea la raíz del árbol central, hacemos que lo sea  $Y$ , como se muestra en la Figura 22.10. Esta acción simplifica la codificación, porque la acción realizada para el caso zig-zag pasa a ser idéntica a la del caso zig. Asimismo, parece que tiene sus ventajas el hacer las cosas así, ya que comprobar una gran cantidad de casos consume mucho tiempo. La desventaja es que, como solo descendemos un nivel, el número de iteraciones dentro del procedimientos de splaying se incrementa.

Una vez realizado el paso de splaying final, se conectan  $L$ ,  $R$  y el árbol central para formar un único árbol, como se muestra en la Figura 22.11. Observe que el resultado es diferente del obtenido con el splaying abajo-arriba. El hecho crucial es que se preserva la cota amortizada  $\mathcal{O}(\log N)$  (véase el Ejercicio 22.5).

Al final, los tres árboles se reconectan para formar uno solo.

En la Figura 22.12 se muestra un ejemplo del algoritmo arriba-abajo simplificado. Cuando tratamos de acceder a 19, el primer paso es un zig-zag. De acuerdo con una versión simétrica de la Figura 22.10, llevamos el subárbol cuya raíz está en 25 hasta la raíz del árbol central y conectamos 12 y su subárbol izquierdo a  $L$ .

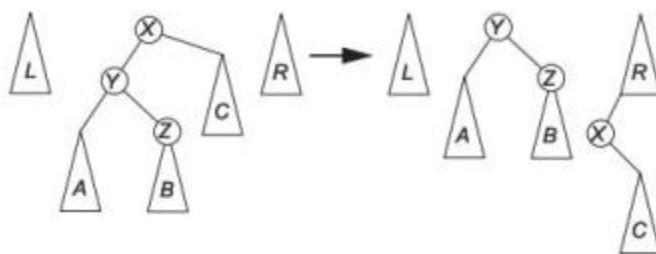


Figura 22.10 Zig-zag arriba-abajo simplificado.

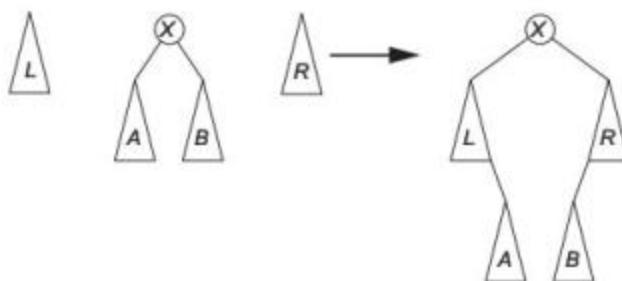


Figura 22.11 Disposición final para el splaying arriba-abajo.

A continuación, tenemos un zig-zig: 15 se eleva a la raíz del árbol central y se lleva a cabo una rotación entre 20 y 25, conectando el subárbol resultante a *R*. La búsqueda de 19 provoca después un zig terminal. La nueva raíz del árbol central es 18, y 15 y su subárbol izquierdo se conectan como hijo derecho del nodo de mayor valor de *L*. La reconexión final, de acuerdo con lo mostrado en la Figura 22.11, pone fin al paso de splaying.

## 22.6 Implementación de los árboles splay de tipo abajo-arriba

En la Figura 22.13 se muestra el esqueleto de la clase del árbol splay. Tenemos los métodos usuales, salvo porque `find` es un método mutador en lugar de accesor. La clase `BinaryNode` es nuestra clase estándar de nodo con visibilidad de paquete, que contiene datos y dos referencias a los hijos; en la figura no se muestra dicha clase. Para eliminar molestos casos especiales, mantenemos un centinela `nullNode`. Asignamos e inicializamos el centinela en el constructor, como se muestra en la Figura 22.14.

La Figura 22.15 muestra el método para la inserción de un elemento *x*. Se asigna un nuevo nodo (`newNode`) y, si el árbol está vacío, se crea un árbol de un único nodo. En caso contrario, aplicamos un splay a *x*. Si el dato en la nueva raíz del árbol es igual a *x*, tendremos un duplicado. En este caso, no deseamos insertar *x*; en su lugar, generamos una excepción en la línea 39. Utilizamos una variable de instancia para que la siguiente llamada a `insert` pueda evitar invocar `new`, en caso de que la operación `insert` falle debido a un elemento duplicado. (Normalmente, no nos preocuparíamos tanto de este caso excepcional; sin embargo, una alternativa razonable consiste en emplear un valor de retorno booleano en lugar de utilizar excepciones.)

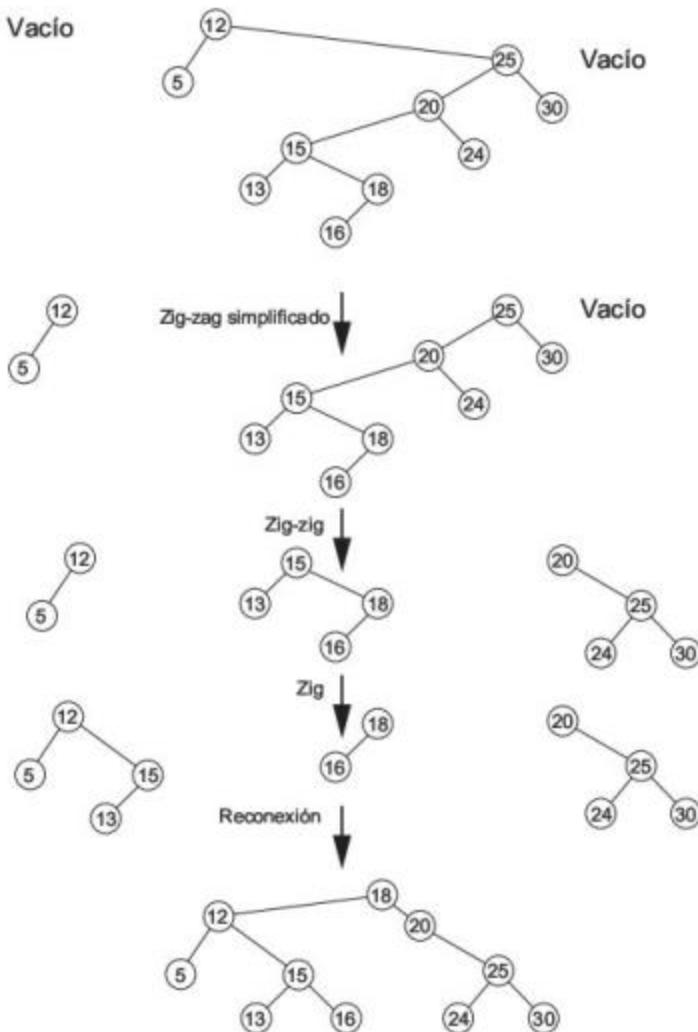


Figura 22.12 Pasos en una operación de splay de tipo arriba-abajo (intentando acceder al elemento de valor 19 en el árbol de la parte superior).

Si la nueva raíz contiene un valor mayor que  $x$ , la nueva raíz y su subárbol derecho pasan a ser subárbol derecho de `newNode`, y el subárbol izquierdo de la raíz pasa a ser subárbol izquierdo de `newNode`. Una lógica similar se aplica si la nueva raíz contiene un valor menor que  $x$ . En cualquiera de los casos, `newNode` se asigna a `root` para indicar que es la nueva raíz. Después hacemos `newNode` igual a `null` en la línea 41, para que la siguiente llamada a `insert` vuelva a invocar a `new`.

La Figura 22.16 muestra la rutina de borrado para los árboles splay. Un procedimiento de borrado raramente es más corto que el correspondiente procedimiento de inserción. A continuación, se muestra la rutina de splaying arriba-abajo.

Nuestra implementación, mostrada en la Figura 22.17, utiliza una cabecera con enlaces izquierdo y derecho para almacenar al final las raíces de los árboles izquierdo y derecho. Estos árboles están inicialmente vacíos; se utiliza una cabecera para que se corresponda con el nodo mínimo o máximo

```

1 package weiss.nonstandard;
2
3 // Clase SplayTree
4 //
5 // CONSTRUCCIÓN: sin ningún inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void insert( x )      --> Insertar x
9 // void remove( x )     --> Eliminar x
10 // Comparable find( x ) --> Devolver el elemento que se corresponde con x
11 // boolean isEmpty( )   --> Devolver true si está vacío; en caso contrario false
12 // void makeEmpty( )   --> Eliminar todos los elementos
13 // *****ERRORES*****
14 // insert y remove generan excepciones en caso necesario
15
16 public class SplayTree<AnyType extends Comparable<AnyType>>
17 {
18     public SplayTree( )
19         { /* Figura 22.14 */ }
20
21     public void insert( AnyType x )
22         { /* Figura 22.15 */ }
23     public void remove( AnyType x )
24         { /* Figura 22.16 */ }
25     public AnyType find( AnyType x )
26         { /* Figura 22.18 */ }
27
28     public void makeEmpty( )
29         { root = nullNode; }
30     public boolean isEmpty( )
31         { return root == nullNode; }
32
33     private BinaryNode<AnyType> splay( AnyType x, BinaryNode<AnyType> t )
34         { /* Figura 22.17 */ }
35
36     private BinaryNode<AnyType> root;
37     private BinaryNode<AnyType> nullNode;
38 }

```

Figura 22.13 Esqueleto de la clase SplayTree de tipo arriba-abajo.

```

1 /**
2  * Construir el árbol.
3  */
4 public SplayTree( )
5 {
6     nullNode = new BinaryNode<AnyType>( null );
7     nullNode.left = nullNode.right = nullNode;
8     root = nullNode;
9 }

```

Figura 22.14 Constructor de la clase SplayTree.

```

1 // Utilizado entre operaciones insert diferentes
2 private BinaryNode<AnyType> newNode = null;
3
4 /**
5 * Insertar en el árbol.
6 * @param x el elemento que hay que insertar.
7 * @throws DuplicateItemException si x ya está presente.
8 */
9 public void insert( AnyType x )
10 {
11     if( newNode == null )
12         newNode = new BinaryNode<AnyType>( null );
13     newNode.element = x;
14
15     if( root == nullNode )
16     {
17         newNode.left = newNode.right = nullNode;
18         root = newNode;
19     }
20     else
21     {
22         root = splay( x, root );
23         if( x.compareTo( root.element ) < 0 )
24         {
25             newNode.left = root.left;
26             newNode.right = root;
27             root.left = nullNode;
28             root = newNode;
29         }
30         else
31         if( x.compareTo( root.element ) > 0 )
32         {
33             newNode.right = root.right;
34             newNode.left = root;
35             root.right = nullNode;
36             root = newNode;
37         }
38         else
39             throw new DuplicateItemException( x.toString() );
40     }
41     newNode = null; // Para que el siguiente insert invoque new
42 }

```

Figura 22.15 Rutina de inserción de la clase SplayTree de tipo arriba-abajo.

```

1  /**
2   * Eliminar del árbol.
3   * @param x el elemento que hay que eliminar.
4   * @throws ItemNotFoundException si no se encuentra x.
5   */
6  public void remove( AnyType x )
7  {
8      BinaryNode<AnyType> newTree;
9
10     // Si se encuentra x, estará en la raíz
11     root = splay( x, root );
12     if( root.element.compareTo( x ) != 0 )
13         throw new ItemNotFoundException( x.toString() );
14
15     if( root.left == nullNode )
16         newTree = root.right;
17     else
18     {
19         // Encontrar el máximo en el subárbol izquierdo
20         // Aplicarle un splay para llevarlo hasta la raíz y conectar hijo dcho.
21         newTree = root.left;
22         newTree = splay( x, newTree );
23         newTree.right = root.right;
24     }
25     root = newTree;
26 }

```

Figura 22.16 Rutina de borrado de la clase SplayTree de tipo arriba-abajo.

```

1  private BinaryNode<AnyType> header = new BinaryNode<AnyType>( null );
2
3  /**
4   * Método interno para realizar un splay arriba-abajo.
5   * El último nodo al que se acceda pasa a ser la nueva raíz.
6   * @param x elemento objetivo alrededor del que hay que aplicar el splay.
7   * @param t la raíz del subárbol al que hay que aplicar el splay.
8   * @return el subárbol después del splay.
9   */
10  private BinaryNode<AnyType> splay( AnyType x, BinaryNode<AnyType> t )

```

Continúa

Figura 22.17 Un algoritmo splay de tipo arriba-abajo.

```
11  {
12      BinaryNode<AnyType> leftTreeMax, rightTreeMin;
13
14      header.left = header.right = nullNode;
15      leftTreeMax = rightTreeMin = header;
16
17      nullNode.element = x; // Garantiza que se encuentre una correspondencia
18
19      for( ; ; )
20          if( x.compareTo( t.element ) < 0 )
21          {
22              if( x.compareTo( t.left.element ) < 0 )
23                  t = Rotations.rotateWithLeftChild( t );
24              if( t.left == nullNode )
25                  break;
26              // Enlazar a la derecha
27              rightTreeMin.left = t;
28              rightTreeMin = t;
29              t = t.left;
30          }
31          else if( x.compareTo( t.element ) > 0 )
32          {
33              if( x.compareTo( t.right.element ) > 0 )
34                  t = Rotations.rotateWithRightChild( t );
35              if( t.right == nullNode )
36                  break;
37              // Enlazar a la izquierda
38              leftTreeMax.right = t;
39              leftTreeMax = t;
40              t = t.right;
41          }
42          else
43              break;
44
45      leftTreeMax.right = t.left;
46      rightTreeMin.left = t.right;
47      t.left = header.right;
48      t.right = header.left;
49      return t;
50 }
```

Figura 22.17 (Continuación).

del árbol derecho o izquierdo, respectivamente, en este estado inicial. De esta forma, podemos evitar comprobar si los árboles están vacíos. La primera vez que el árbol izquierdo deja de estar vacío, se inicializa el enlace derecho de la cabecera y ya no cambia en lo sucesivo. De ese modo, contendrá la raíz del árbol izquierdo de la búsqueda arriba-abajo. De forma similar, el lado izquierdo de la cabecera terminará por contener la raíz del árbol derecho. La variable `header` no es local, porque queremos asignarla un valor únicamente una vez a lo largo de toda la secuencia de operaciones splay.

Antes de reconectar los árboles al final del splay, `header.left` y `header.right` harán referencia a  $R$  y  $L$ , respectivamente (no, esto no es un error tipográfico –siga los enlaces). Observe que estamos utilizando el splay arriba-abajo. El método `find`, mostrado en la Figura 22.18, completa la implementación del árbol splay.

## 22.7 Comparación del árbol splay con otros árboles de búsqueda

La implementación que acabamos de presentar sugiere que los árboles splay no son tan complicados como los árboles rojo-negro y son casi tan simples como los árboles AA. ¿Merece la pena utilizarlos? La respuesta todavía no está del todo clara, pero si los patrones de acceso no son del todo aleatorios, los árboles splay parecen presentar un buen rendimiento en la práctica. Algunas propiedades relativas a su rendimiento también se puede demostrar analíticamente. Entre los accesos no aleatorios incluimos aquellos que se ajustan a la regla 90–10, así como varios casos especiales como el acceso secuencial, el acceso por los dos extremos y, aparentemente, los patrones de acceso que son típicos de las colas con prioridad durante algunos tipos de simulaciones de sucesos. En los ejercicios le pediremos que examine esta cuestión con mayor detalle.

Los árboles splay no son perfectos. Un problema con ellos es que la operación `find` resulta costosa debido al splay. Por tanto, cuando las secuencias de acceso sean aleatorias y uniformes, los árboles splay no tienen un comportamiento tan bueno como otros árboles equilibrados.

```

1  /**
2   * Buscar un elemento en el árbol.
3   * @param x el elemento que hay que buscar.
4   * @return el elemento correspondiente o null si no se encuentra.
5   */
6  public AnyType find( AnyType x )
7  {
8      root = splay( x, root );
9
10     if( isEmpty( ) || root.element.compareTo( x ) != 0 )
11         return null;
12
13     return root.element;
14 }
```

Figura 22.18 Rutina `find` de la clase `SplayTree` de tipo arriba-abajo.

## Resumen

En este capítulo hemos descrito el árbol splay, que es una reciente alternativa al árbol de búsqueda equilibrado. Los árboles splay tienen varias propiedades notables que se pueden demostrar, incluyendo su coste logarítmico por cada operación. En los ejercicios se sugieren otras propiedades. Algunos estudios apuntan a que los árboles splay se pueden utilizar en un amplio rango de aplicaciones, debido a su aparente capacidad para adaptarse a secuencias de acceso favorables.

En el Capítulo 23 describiremos dos colas con prioridad que, al igual que el árbol splay, presentan un mal comportamiento de caso peor, pero buen comportamiento amortizado. Una de esas estructuras de datos, el montículo de emparejamiento, parece ser una excelente elección para algunos tipos de aplicaciones.



## Conceptos clave

**análisis amortizado** Acota el coste de una secuencia de operaciones y distribuye equitativamente el coste entre cada una de las operaciones de la secuencia. (833)

**árbol splay de tipo abajo-arriba** Un árbol en el que los elementos se rotan hacia la raíz utilizando un método ligeramente más complicado que el que se emplea en una estrategia simple de rotación hacia la raíz. (835)

**árbol splay de tipo arriba-abajo** Un tipo de árbol splay que es más eficiente en la práctica que el correspondiente árbol de tipo abajo-arriba, exactamente igual que lo que sucede en el caso de los árboles rojo-negro. (844)

**estrategia de rotación hacia la raíz** Reordena un árbol de búsqueda binaria después de cada acceso, con el fin de acercar hacia la raíz los elementos a los que se accede frecuentemente. (833)

**función potencial** Una herramienta de análisis utilizada para establecer una cota de tiempo amortizado. (839)

**regla 90-10** Afirma que el 90 por ciento de los accesos afecta al 10 por ciento de los elementos de datos. Sin embargo, los árboles de búsqueda equilibrados no aprovechan esta regla. (832)

**splaying** Una estrategia de rotación hacia la raíz que permite conseguir una cota amortizada logarítmica. (835)

**zig** y **zig-zag** Casos que son idénticos a los correspondientes casos de la estrategia de rotación hacia la raíz. zig se utiliza cuando  $X$  es un hijo de la raíz y zig-zag se emplea cuando  $X$  es un nodo (nieto) interno. (835)

**zig-zig** Un caso original de los árboles splay que se utiliza cuando  $X$  es un nodo (nieto) externo. (836)



## Errores comunes

1. Se debe realizar un splay después de cada acceso, incluso aunque no tenga éxito ese acceso, porque de lo contrario no serían válidas las cotas de rendimiento.
2. El código está todavía lleno de complicaciones.
3. No se pueden utilizar de manera segura métodos privados recursivos en la clase `SplayTree`, a pesar de que el rendimiento sería aceptable, porque la profundidad del árbol puede ser demasiado grande.



## Internet

La clase `SplayTree` está disponible en línea. El código incluye versiones de `findMin` y `findMax` que son eficientes en un sentido amortizado, pero que no están completamente optimizadas.

### `SplayTree.java`

Contiene la implementación de la clase `SplayTree`.



## Ejercicios

### EN RESUMEN

- 22.1 Muestre el resultado de insertar 5, 3, 6, 7, 4, 11, 8 y 10 en
  - a. Un árbol splay abajo-arriba.
  - b. Un árbol splay arriba-abajo.
- 22.2 Indique el resultado de borrar 7 en el árbol splay obtenido en el Ejercicio 22.1, tanto para la versión de tipo abajo-arriba como para la de tipo arriba-abajo.

### EN TEORÍA

- 22.3 Demuestre que si se accede en orden secuencial a los nodos de un árbol splay, el árbol resultante está compuesto por una cadena de hijos izquierdos.
- 22.4 Suponga que, en un intento de ahorrar tiempo aplicamos un splay solo en una de cada dos operaciones con el árbol, ¿seguirá siendo logarítmico el coste amortizado?
- 22.5 Demuestre que el coste amortizado para un splay de tipo arriba-abajo es  $O(\log N)$ .
- 22.6 Cambiando la función potencial, se pueden identificar diferentes cotas para el splaying. Sea la función de ponderación  $W(i)$  alguna función asignada a cada nodo del árbol y sea  $S(i)$  la suma de los pesos de todos los nodos del subárbol que tiene como raíz el nodo  $i$ , incluyendo al propio nodo  $i$ . El caso especial  $W(i) = 1$  para todos los nodos se corresponde con la función utilizada en la demostración de la

cota para el splaying. Sea  $N$  el número de nodos del árbol y sea  $M$  el número de accesos. Demuestre los dos teoremas siguientes.

- El tiempo total de acceso es  $O(M + (M + N) \log N)$ .
  - Si  $q_i$  es el número total de veces que se accede al elemento  $i$  y  $q_i > 0$  para todo  $i$ , entonces el tiempo total de acceso es  $O(M + \sum_{i=1}^N q_i \log(M/q_i))$ .
- 22.7** Los nodos 1 a  $N = 512$  forman un árbol splay de hijos izquierdos.
- ¿Cuál es (exactamente) la longitud interna de camino del árbol?
  - Calcule la longitud interna de camino después de cada una de las operaciones `find(3)`, `find(4)` y `find(5)`, cuando se realiza un splay de tipo abajo-abajo.

## EN LA PRÁCTICA

**22.8** Modifique el árbol splay para permitir estadísticas de orden.

**22.9** Utilice el árbol splay para implementar una clase para colas con prioridad.

## PROYECTOS DE PROGRAMACIÓN

- 22.10** Compare empíricamente una implementación de cola con prioridad basada en árbol splay de tipo arriba-abajo con otra implementación basada en montículo binario, utilizando
- Operaciones `insert` y `deleteMin` aleatorias.
  - Operaciones `insert` y `deleteMin` que se correspondan con una simulación dirigida por sucesos.
  - Operaciones `insert` y `deleteMin` que se correspondan con el algoritmo de Dijkstra.

**22.11** Compare empíricamente la operación splay arriba-abajo simplificada que hemos implementado en la Sección 22.6 con el splay arriba-abajo original analizado en la Sección 22.5

**22.12** A diferencia de los árboles de búsqueda equilibrados, los árboles splay presentan un coste adicional durante una operación `find` que puede no ser tolerable si la secuencia de acceso es lo suficientemente aleatoria. Experimente con una estrategia que realice un splay con una operación `find` únicamente después de haber recorrido una cierta profundidad  $d$  en la búsqueda arriba-abajo. El splay no desplaza el elemento al que se accede hasta la raíz, sino solo hasta el punto de profundidad  $d$  donde ha comenzado el splaying.



## Referencias

El árbol splay se describe en el artículo [3]. El concepto de análisis amortizado se analiza en el artículo recopilatorio [4] y también, con mayor detalle, en [5]. En [1] se proporciona una comparación entre los árboles splay y AVL y en [2] se muestra que los árboles splay tienen un buen comportamiento en algunos tipos de simulaciones dirigidas por sucesos.

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 the 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 Java, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 2007.

## Mezcla de colas con prioridad

En este capítulo vamos a examinar las colas con prioridad que soportan una operación adicional: la operación `merge`, que es importante en el diseño avanzado de algoritmos. Esta operación combina dos colas con prioridad en una sola (destruyendo desde el punto de vista lógico los originales). Representamos las colas con prioridad como árboles generales, lo que simplifica en cierta medida la operación `decreaseKey` y es importante en algunas aplicaciones.

En este capítulo veremos

- Cómo funciona el *montículo sesgado* o *montículo autoajustable*, una cola con prioridad mezclable implementada mediante árboles binarios.
- Cómo funciona el *montículo de emparejamiento*, una cola con prioridad mezclable basada en el árbol *Mario*. El montículo de emparejamiento parece ser una alternativa práctica al montículo binario, incluso si no hace falta la operación `merge`.

### 23.1 El montículo sesgado

El *montículo sesgado* es un árbol binario ordenado como montículo y que no tiene una condición de equilibrado. Sin esta restricción estructural en el árbol –a diferencia del montículo o de los árboles de búsqueda binaria equilibrados– no hay ninguna garantía de que la profundidad del árbol sea logarítmica. Sin embargo, soporta todas las operaciones en un tiempo amortizado logarítmico. El montículo sesgado es por tanto similar en cierta forma al árbol splay.

*El montículo sesgado* es un árbol binario con ordenación de montículo y sin condición de equilibrado que soporta todas las operaciones en un tiempo amortizado logarítmico.

#### 23.1.1 El mezclado es fundamental

Si se utiliza un árbol binario con ordenación de montículo y sin restricciones estructurales para representar una cola con prioridad, el mezclado se convierte en la operación fundamental. Esto se debe a que podemos realizar otras operaciones de la forma siguiente:

- `h.insert( x )`: crear un árbol de un nodo que contenga `x` y mezclar dicho árbol con la cola con prioridad.
- `h.findMin( )`: devolver el elemento situado en la raíz.

- `h.deleteMin()`: borrar la raíz y mezclar sus subárboles izquierdo y derecho.
- `h.decreaseKey(p, newVal)`: asumiendo que `p` es una referencia a un nodo de la cola con prioridad, podemos reducir el valor de clave de `p` apropiadamente y luego desconectar `p` de su padre. Hacer esto nos da colas con prioridad que luego se pueden mezclar. Observe que `p` (que hace referencia a la posición) no cambia como resultado de esta operación (a diferencia de la operación equivalente en un montículo binario).

La operación `decreaseKey` se implementa desconectando un subárbol de su padre y luego utilizando `merge`.

Necesitamos mostrar únicamente cómo implementar la mezcla; las restantes operaciones serán entonces triviales. La operación `decreaseKey` es importante en algunas aplicaciones avanzadas. Ya hemos presentado una ilustración de esto en la Sección 14.3 –el algoritmo de Dijkstra para encontrar los caminos más cortos en un grafo. Allí no utilizamos la operación `decreaseKey` en nuestra implementación debido a las complicaciones de mantener la posición en cada elemento dentro del montículo binario. En un montículo de mezcla, la posición se puede mantener mediante una referencia al nodo del árbol y, a diferencia del montículo binario, la posición nunca cambia.

En esta sección vamos a analizar una implementación de una cola con prioridad mezclable que utiliza un árbol binario: el montículo sesgado. En primer lugar, veremos que, si no nos preocupa la eficiencia, mezclar dos árboles con ordenación de montículo es sencillo. Despues, veremos una modificación sencilla (el montículo sesgado) que corrige la evidente falta de eficiencia del algoritmo original. Por último, proporcionaremos una demostración de que la operación `merge` para montículos sesgados es logarítmica en un sentido amortizado, y haremos algunos comentarios acerca de la importancia práctica que tiene este resultado.

### 23.1.2 Mezcla simplista de árboles con ordenación de montículo

Se pueden mezclar fácilmente dos árboles de manera recursiva.

El resultado es que se mezclan los caminos derechos. Tenemos que tener cuidado para no crear caminos derechos indebidamente largos.

Vamos a suponer que disponemos de dos subárboles con ordenación de montículo,  $H_1$  y  $H_2$ , y que necesitamos mezclarlos. Claramente, si cualquiera de los dos árboles está vacío, el otro árbol coincidirá con el resultado de la mezcla. En caso contrario, para mezclar los dos árboles lo que hacemos es comparar sus raíces. Mezclaremos recursivamente el árbol que tenga la raíz más grande con el subárbol derecho del árbol que tenga la raíz más pequeña.<sup>1</sup>

La Figura 23.1 muestra el efecto de esta estrategia recursiva: los caminos derechos de las dos colas con prioridad se mezclan para formar la nueva cola con prioridad. Cada nodo del camino derecho retiene su subárbol izquierdo original y solo se tocan los nodos situados en el camino derecho. El resultado mostrado en la Figura 23.1 no se puede conseguir utilizando únicamente inserciones y mezclas porque, como ya hemos mencionado, no se pueden añadir hijos izquierdos mediante una mezcla. El efecto práctico es que lo que parece ser un árbol binario con ordenación de montículo es, de hecho, una ordenación consistente únicamente en un solo camino derecho. Por

<sup>1</sup> Claramente, podría utilizarse cualquiera de los subárboles. Hemos decidido arbitrariamente emplear el subárbol derecho.

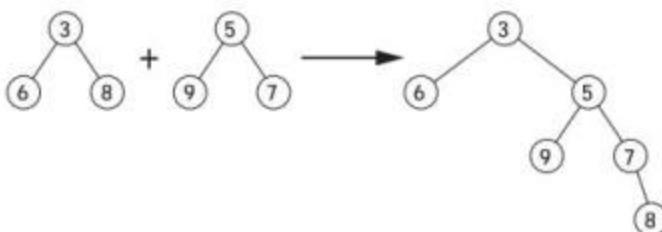


Figura 23.1 Mezcla simplista de árboles con ordenación de montículo: se mezclan los caminos derechos.

tanto, todas las operaciones requieren un tiempo lineal. Afortunadamente, una sencilla modificación nos permite asegurarnos de que el camino derecho no sea siempre largo.

### 23.1.3 El montículo sesgado: una modificación simple

La mezcla mostrada en la Figura 23.1 crea un árbol mezclado temporal. Podemos hacer una sencilla modificación en la operación de la forma siguiente. Antes de completar una mezcla, intercambiamos el hijo derecho y el hijo izquierdo para todos los nodos del camino derecho resultante del árbol temporal. De nuevo, solo los nodos de los caminos derechos originales se encontrarán en el camino derecho del árbol temporal. Como resultado del intercambio, mostrado en la Figura 23.2, estos nodos formarán entonces el camino izquierdo del árbol resultante. Cuando se realiza una mezcla de esta forma, al árbol con ordenación de montículo se le denomina también *montículo sesgado*.

Para evitar los problemas de los caminos indebidamente largos, transformamos en un camino izquierdo el camino derecho resultante después de una operación merge. Dicha mezcla nos da lo que se denomina *montículo sesgado*.

Una visión recursiva del asunto sería la siguiente. Si llamamos  $L$  al árbol con la raíz más pequeña y  $R$  al otro árbol, los siguientes enunciados son ciertos.

1. Si uno de los árboles está vacío, el otro puede utilizarse como resultado de la mezcla.
2. En caso contrario, sea  $Temp$  el subárbol derecho de  $L$ .
3. Hacemos que el subárbol izquierdo de  $L$  sea el nuevo subárbol derecho.
4. Hacemos que el resultado de la mezcla recursiva de  $Temp$  y  $R$  sea el nuevo subárbol izquierdo de  $L$ .

Esperamos que el resultado del intercambio de los hijos sea que la longitud del camino derecho no siempre resulte indebidamente larga. Por ejemplo, si mezclamos un par de árboles con caminos

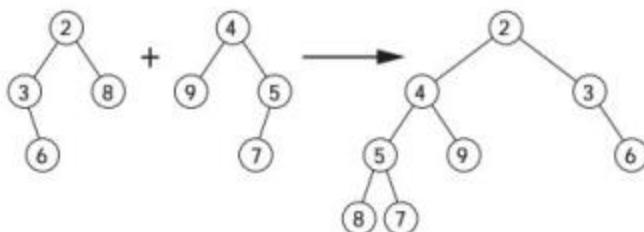


Figura 23.2 Mezcla de montículo sesgado; se mezclan los caminos derechos y el resultado se convierte en un camino izquierdo.

Sigue siendo posible que aparezca un camino derecho largo. Sin embargo, raramente aparece y, si lo hace, debe estar precedido por muchas mezclas en las que están involucrados caminos derechos de corta longitud.

derechos largos, los nodos que forman el camino no reaparecerán en un camino derecho durante un cierto tiempo futuro. Sigue siendo posible obtener árboles que exhiban la propiedad de que todo nodo aparezca en un camino derecho, pero eso solo se puede hacer como resultado de un gran número de mezclas relativamente poco costosas. En la Sección 23.1.4 demostraremos este enunciado de forma rigurosa, estableciendo que el coste amortizado de una operación de mezcla es únicamente logarítmico.

### 23.1.4 Análisis del montículo sesgado

El coste efectivo de una mezcla es igual al número de nodos existentes en los caminos derechos de los dos árboles que se están mezclando.

Suponga que tenemos dos montículos,  $H_1$  y  $H_2$ , y que hay  $r_1$  y  $r_2$  nodos en sus respectivos caminos derechos. Entonces el tiempo requerido para realizar la mezcla es proporcional a  $r_1 + r_2$ . Cuando cargamos una unidad de coste por cada nodo existente en los caminos derechos, el coste de la mezcla es proporcional al número de unidades cargadas. Puesto que los árboles no tienen estructura, todos los nodos de ambos árboles pueden estar en el camino derecho. Esta condición nos daría una cota  $\Theta(N)$  de caso peor para la mezcla de los árboles

(en el Ejercicio 23.7 le pediremos que construya uno de esos árboles). Como vamos a demostrar en breve, el tiempo amortizado necesario para mezclar dos montículos sesgados es  $O(\log N)$ .

Al igual que con el árbol splay, vamos a introducir una función potencial que cancele los costes variables de las operaciones con montículos sesgados. Queremos que la función potencial se incremente en un total de  $O(\log N) - (r_1 + r_2)$ , de modo que la suma del coste de la mezcla y de la variación del potencial sea solo  $O(\log N)$ . Si el potencial es mínimo antes de la primera operación, aplicar la suma telescópica garantizará que el coste total invertido para cualesquiera  $M$  operaciones será  $O(M \log N)$ , como con el árbol splay.

Lo que necesitamos es alguna función potencial que capture el efecto de las operaciones con los montículos sesgados. Encontrar tal función es bastante complicado. Sin embargo, una vez que hemos encontrado una, la demostración es relativamente corta.

#### Definición:

Un nodo es un nodo pesado si el tamaño de su subárbol derecho es mayor que el de su subárbol izquierdo. En caso contrario, será un nodo ligero; si los dos subárboles son de igual tamaño, el nodo es ligero.

La función potencial es el número de nodos pesados. Solo se ve modificada la condición de pesado o ligero de los nodos situados en el camino mezclado. El número de nodos ligeros en un camino derecho es logarítmico.

En la Figura 23.3, antes de la mezcla, los nodos 3 y 4 son pesados. Después de la mezcla, solo es pesado el nodo 3. Podemos percatarnos fácilmente de tres hechos. En primer lugar, como resultado de una mezcla, solo puede variar la condición de ligero o pesado de los nodos situados en el camino derecho, porque no se modifican los subárboles de ningún otro nodo. En segundo lugar, una hoja siempre es ligera. En tercer lugar, el número de nodos ligeros en el camino derecho de un árbol de  $N$  nodos es como máximo  $\lfloor \log N \rfloor + 1$ . La razón es que el tamaño del hijo derecho de un nodo ligero es menor que la mitad del tamaño del propio nodo ligero, por lo que se aplica el principio de división por la mitad. El  $+1$  adicional es resultado de que la hoja es ligera. Con estos preliminares podemos ahora enunciar y demostrar los Teoremas 23.1 y 23.2.

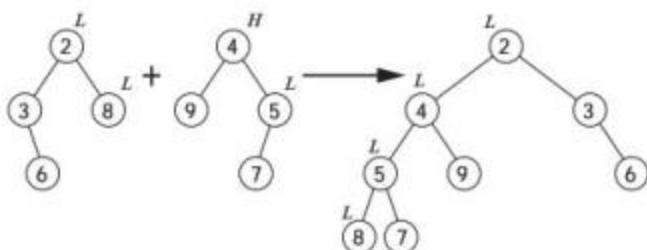


Figura 23.3 Cambio en el carácter ligero o pesado de los nodos después de una mezcla.

**Teorema 23.1**

Sean  $H_1$  y  $H_2$  dos montículos sesgados con  $N_1$  y  $N_2$  nodos, respectivamente y sea  $N$  su tamaño combinado (es decir,  $N_1 + N_2$ ). Suponga que el camino derecho de  $H_1$  tiene  $l_1$  nodos ligeros y  $h_1$  nodos pesados, lo que da un total de  $l_1 + h_1$ , mientras que el camino derecho de  $H_2$  tiene  $l_2$  nodos ligeros y  $h_2$  nodos pesados, lo que da un total de  $l_2 + h_2$ . Si definimos el potencial como el número total de nodos pesados en la colección de montículos sesgados, entonces la mezcla cuesta como máximo  $2 \log N + (h_1 + h_2)$ , mientras que la variación de potencial es como máximo  $2 \log N - (h_1 + h_2)$ .

**Demostración**

El coste de la mezcla es simplemente el número total de nodos en los caminos derechos,  $l_1 + l_2 + h_1 + h_2$ . El número de nodos ligeros es logarítmico, por lo que  $l_i \leq \lfloor \log N_i \rfloor + 1$ . Por tanto,  $l_1 + l_2 \leq \log N_1 + \log N_2 + 2 \leq 2 \log N$ , donde la última desigualdad se deduce del Teorema 22.4. El coste de la mezcla es por tanto, como máximo  $2 \log N + (h_1 + h_2)$ . La cota de la variación de potencial se deduce del hecho de que solo puede verse modificada la condición de pesado/ligero de los nodos implicados en la mezcla, y del hecho de que cualquier nodo pesado del camino deberá transformarse en ligero, debido a que sus hijos serán intercambiados. Incluso si todos los nodos ligeros pasaran a ser pesados, la variación del potencial seguiría estando limitada a  $l_1 + l_2 - (h_1 + h_2)$ . Basándonos en el mismo argumento que antes, eso es como máximo  $2 \log N - (h_1 + h_2)$ .

**Teorema 23.2**

El coste amortizado del montículo sesgado es como máximo  $4 \log N$  para las operaciones `merge`, `insert` y `deleteMin`.

**Demostración**

Sea  $\Phi$  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 crea un árbol de un solo nodo cuya raíz es, por definición, ligera y que por tanto no altera el potencial antes de la mezcla resultante. Una operación `deleteMin` descarta la raíz antes de la mezcla, así que no puede aumentar el potencial (de hecho, puede reducirlo). Lo único que nos queda por considerar son los costes de la mezcla. Sea  $c_i$  el coste de la mezcla que tiene lugar como resultado de la  $i$ -ésima operación, entonces  $c_i + \Phi_i - \Phi_{i-1} \leq 4 \log N$ . Aplicando la suma telescopica para  $M$  operaciones obtenemos  $\sum_{i=1}^M c_i \leq 4 M \log N$  porque  $\Phi_M - \Phi_0$  es no negativa.

El montículo sesgado es un notable ejemplo de algoritmo sencillo con un análisis que no resulta obvio. Sin embargo, el análisis es fácil de realizar una vez que hemos identificado la función

Encontrar una función potencial útil es la parte más difícil del análisis.

Debería utilizarse un algoritmo no recursivo, debido a la posibilidad de quedarnos sin espacio de pila.

potencial apropiada. Lamentablemente, sigue sin existir una teoría general que nos permita determinar fácilmente una función potencial. Normalmente hay que probar con muchas funciones diferentes antes de encontrar una utilizable.

Hay un comentario que tenemos que hacer: aunque la descripción inicial del algoritmo utiliza recursión y la recursión proporciona el código más simple, no se puede utilizar en la práctica. La razón es que el tiempo lineal de caso peor para una operación podría provocar un desbordamiento en la pila de tiempo de ejecución al implementar la recursión. En consecuencia, es necesario utilizar un algoritmo no recursivo. En lugar de explorar dichas posibilidades, hablaremos de una estructura de datos alternativa, que es

ligeramente más complicada: el montículo de emparejamiento. Esta estructura de datos no ha sido analizada completamente, pero parece tener un buen rendimiento en la práctica.

## 23.2 El montículo de emparejamiento

El *montículo de emparejamiento* es un árbol  $M$ -ario con ordenación de montículo y sin restricciones estructurales para el que todas las operaciones, excepto el borrado, requieren un tiempo constante de caso peor.

Aunque `deleteMin` podría requerir un tiempo lineal de caso peor, cualquier secuencia de operaciones con un montículo de emparejamiento tiene un rendimiento amortizado logarítmico. Se ha establecido como conjectura –pero no se ha demostrado– que podría estar garantizado un rendimiento aun mejor. Sin embargo, el mejor de los escenarios posibles –que todas las operaciones salvo `deleteMin` tengan un coste amortizado constante, mientras que `deleteMin` tiene un coste amortizado logarítmico– se ha demostrado recientemente que no es cierto.

La Figura 23.4 muestra un montículo de emparejamiento abstracto. La implementación real utiliza una representación de tipo hijo izquierdo/hermano derecho (véase el Capítulo 18). El método `decreaseKey`, como veremos en breve, requiere que cada nodo contenga un enlace adicional. Un nodo que sea un hijo izquierdo contendrá un enlace a su padre; en caso contrario, el nodo será un hermano derecho y contendrá un enlace a su hermano izquierdo. Esta representación se muestra en la Figura 23.5, donde la línea oscura indica que hay dos enlaces (uno en cada dirección) conectando cada pareja de nodos.

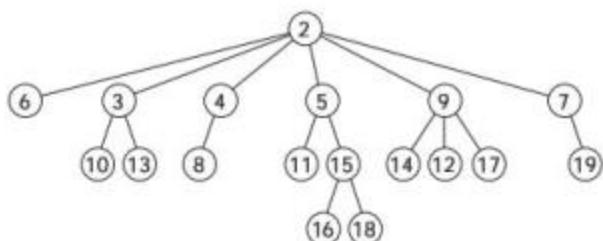
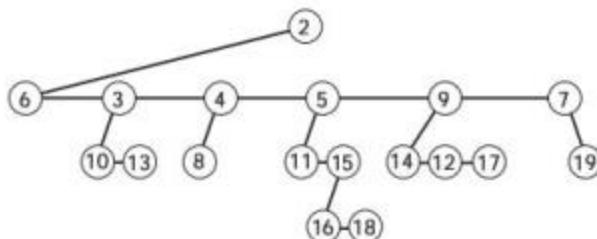


Figura 23.4 Representación abstracta de un ejemplo de montículo de emparejamiento.



**Figura 23.5** Representación real del montículo de emparejamiento mostrado en la Figura 23.4; las líneas oscuras representan parejas de enlaces que conectan los nodos en ambas direcciones.

### 23.2.1 Operaciones con el montículo de emparejamiento

En principio, las operaciones básicas con un montículo de emparejamiento son simples, razón por la cual el montículo de emparejamiento tiene un buen comportamiento en la práctica. Para mezclar dos montículos de emparejamiento, hacemos que el montículo con raíz mayor sea el nuevo primer hijo del montículo con raíz menor. La inserción es un caso especial de mezcla. Para realizar una operación `decreaseKey`, reducimos el valor del nodo solicitado. Puesto que no estamos manteniendo enlaces al padre en todos los nodos, no sabemos si esta acción viola el orden del montículo. Por tanto, lo que hacemos es desconectar de su padre el nodo ajustado y completar `decreaseKey` mezclando los dos montículos de emparejamiento resultantes. La Figura 23.5 muestra que desconectar un nodo de su padre implica eliminarlo de lo que es, esencialmente, una lista enlazada de hijos. Hasta el momento los resultados que hemos obtenido son buenos; todas las operaciones descritas requieren un tiempo constante. Sin embargo, no tenemos tanta suerte con la operación `deleteMin`.

Para realizar una operación `deleteMin`, debemos eliminar la raíz del árbol, creando una colección de montículos. Si hay  $c$  hijos de la raíz, combinar estos montículos en un solo montículo requiere  $c - 1$  mezclas. Por tanto, si la raíz tiene un montón de hijos, la operación `deleteMin` tiene un coste grande en términos de tiempo. Si la secuencia de inserción es  $1, 2, \dots, N$ , entonces 1 estará en la raíz y todos los restantes elementos estarán en nodos que serán hijos de la raíz. En consecuencia, `deleteMin` requerirá un tiempo  $O(N)$ . Lo mejor que cabe esperar es disponer las mezclas de modo que no tengamos de manera repetida costosas operaciones `deleteMin`.

El orden en el que se mezclan los subárboles del montículo de emparejamiento es importante. La manera más simple y práctica de entre las muchas variantes existentes para hacer esto que se han propuesto es la *mezcla de dos pasadas*, en la que una primera exploración mezcla parejas de hijos de izquierda a derecha<sup>2</sup> y luego se realiza una segunda pasada, de derecha

La mezcla es simple:  
conectamos el árbol que  
tenga la raíz mayor como  
hijo izquierdo del árbol que  
tenga la raíz menor. La  
inserción y la operación de  
reducción de claves son  
también simples.

La operación `deleteMin`  
es costosa porque la nueva  
raíz podría ser cualquiera  
de los  $c$  hijos de la antigua  
raíz. Necesitamos  $c - 1$   
mezclas.

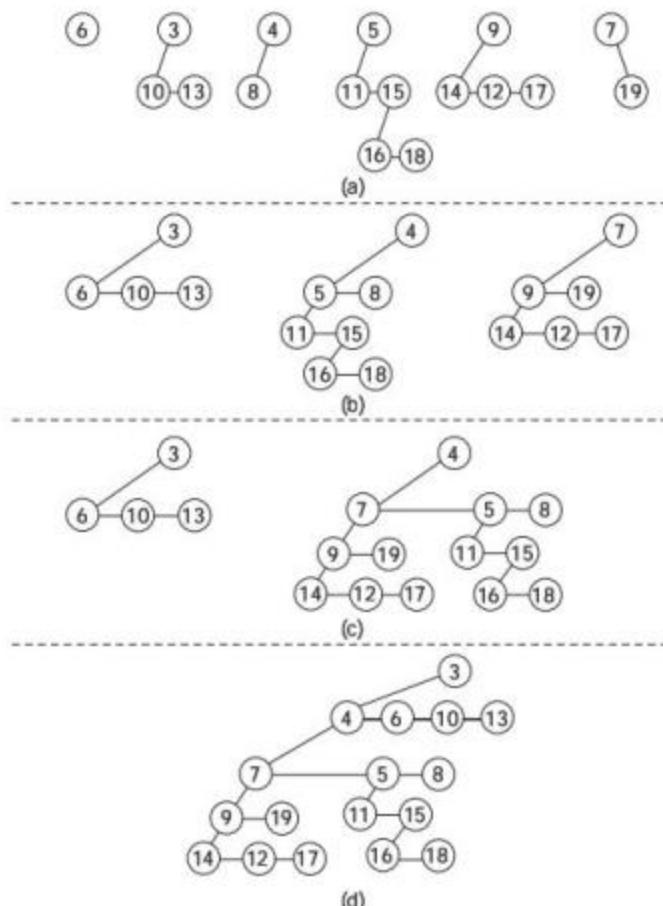
El orden en el que se  
mezclan los subárboles  
del montículo de  
emparejamiento es  
importante. El algoritmo más  
simple se denomina *mezcla  
de dos pasadas*.

<sup>2</sup> Hay que tener cuidado si existe un número impar de hijos. Cuando eso sucede, mezclamos el último hijo con el resultado de la mezcla de más a la derecha para completar la primera pasada.

a izquierda, para completar la mezcla. Después de la primera pasada, tendremos la mitad de los árboles para mezclar. En la segunda pasada, en cada paso, mezclamos el árbol de más a la derecha resultante de la primera pasada con el resultado actual de la mezcla. Por ejemplo, si tenemos los hijos  $c_1$  a  $c_8$ , la primera pasada realizará las mezclas  $c_1$  y  $c_2$ ,  $c_3$  y  $c_4$ ,  $c_5$  y  $c_6$ , y  $c_7$  y  $c_8$ . El resultado será  $d_1$ ,  $d_2$ ,  $d_3$  y  $d_4$ . Realizamos la segunda pasada mezclando  $d_3$  y  $d_4$ ; a continuación se mezcla  $d_2$  con dicho resultado y finalmente se mezcla  $d_1$  con el resultado de esa mezcla, completando la operación `deleteMin`. La Figura 23.6 muestra el resultado de utilizar `deleteMin` con el montículo de emparejamiento mostrado en la Figura 23.5.

Se han propuesto varias alternativas. La mayoría de ellas son indistinguibles, pero utilizar una única pasada de izquierda a derecha no es una buena idea.

También son posibles otras estrategias de mezcla. Por ejemplo, podemos colocar cada subárbol (que se corresponde con un hijo) en una cola, extraer de la cola repetidamente dos árboles y luego introducir en la cola el resultado de dicha mezcla. Después de  $c - 1$  mezclas, solo quedará un árbol en la cola, que será el resultado de la operación `deleteMin`. Sin embargo, utilizar una pila en lugar de una cola es un desastre, porque la raíz del árbol resultante



**Figura 23.6** Recombinación de hermanos después de una operación `deleteMin`. En cada mezcla, se hace que el árbol con la raíz mayor sea el hijo izquierdo del árbol con la raíz más pequeña: (a) los árboles resultantes; (b) después de la primera pasada; (c) después de la primera mezcla de la segunda pasada; (d) después de la segunda mezcla de la segunda pasada.

podría tener posiblemente  $c - 1$  hijos. Si ocurriera eso en una secuencia, la operación `deleteMin` tendría un coste amortizado lineal, en lugar de logarítmico, por cada operación. En el Ejercicio 23.4 le pediremos que construya una de tales secuencias.

### 23.2.2 Implementación del montículo de emparejamiento

El esqueleto de la clase `PairingHeap` se muestra en la Figura 23.7. La clase anidada `PairNode` implementa la interfaz anidada `Position` que se declara en las líneas 16 y 17.

En el montículo de emparejamiento, `insert` devuelve una posición `Position` que es el `PairNode` recién creado.

En la Figura 23.8 se muestra el nodo básico de un montículo de emparejamiento, `PairNode` y está compuesto por un elemento y tres enlaces. Dos de estos enlaces son al hijo izquierdo y al siguiente hermano. El tercer enlace es `prev`, que hace referencia al padre si el nodo es un primer hijo o a un hermano izquierdo en caso contrario.

La rutina `findMin` está codificada en la Figura 23.9. El mínimo se encuentra en la raíz, así que esta rutina se implementa fácilmente. La rutina `insert`, mostrada en la Figura 23.10, crea un árbol de un solo nodo y lo mezcla con `root` para obtener un nuevo árbol. Como hemos dicho anteriormente en esta sección, `insert` devuelve una referencia al nodo recién asignado. Observe que tenemos que encargarnos de gestionar el caso especial de una inserción en un árbol vacío.

La Figura 23.11 implementa la rutina `deleteMin`. Si el montículo de emparejamiento está vacío, tenemos un error. Después de guardar el valor encontrado en la raíz (en la línea 11) y borrar dicho valor en la línea 12, hacemos una llamada a `combineSiblings` en la línea 16 para mezclar los subárboles de la raíz y hacemos que el resultado sea la nueva raíz. Si no hay subárboles, simplemente asignamos a `root` el valor `null` en la línea 14.

El método `decreaseKey` está implementado en la Figura 23.12. Si el nuevo valor es mayor que el original, podríamos destruir el orden del montículo. No tenemos forma de saber eso sin examinar todos los hijos. Puesto que pueden existir muchos hijos, hacer esto sería ineficiente. Por tanto, asumimos que siempre es un error tratar de incrementar la clave utilizando `decreaseKey`. (En el Ejercicio 23.5 le pediremos que describa un algoritmo para `increaseKey`.) Después de efectuar esta comprobación, reducimos el valor del nodo. Si el nodo es la raíz, habremos terminado. En caso contrario, desconectamos el nodo de la lista de hijos a la que pertenece, utilizando el código de las líneas 21 a 28. Después de hacer eso, simplemente mezclamos el árbol resultante con la raíz.

El miembro de datos `prev` enlaza a un hermano izquierdo o al padre.

La operación `deleteMin` se implementa mediante una llamada a `combineSiblings`.

```

1 package weiss.nonstandard;
2
3 // Clase PairingHeap
4 //
5 // CONSTRUCCIÓN: sin inicializador
6 //
7 // *****OPERACIONES PÚBLICAS*****

```

Continúa

Figura 23.7 El esqueleto de la clase `PairingHeap`.

```
8 // Métodos generales para colas con prioridad y también:
9 // void decreaseKey( Position p, newVal )
10 //           --> Reducir valor del nodo p
11 // *****ERRORES*****
12 // Se generan excepciones en los casos necesarios
13
14 public class PairingHeap<AnyType extends Comparable<? super AnyType>>
15 {
16     public interface Position<AnyType>
17         { AnyType getValue( ); }
18
19     private static class PairNode<AnyType> implements Position<AnyType>
20         { /* Figura 23.8 */ }
21
22     private PairNode<AnyType> root;
23     private int theSize;
24
25     public PairingHeap( )
26         { root = null; theSize = 0; }
27
28     public boolean isEmpty( )
29         { return root == null; }
30     public int size( )
31         { return theSize; }
32     public void makeEmpty( )
33         { root = null; theSize = 0; }
34
35     public Position<AnyType> insert( AnyType x )
36         { /* Figura 23.10 */ }
37     public AnyType findMin( )
38         { /* Figura 23.9 */ }
39     public AnyType deleteMin( )
40         { /* Figura 23.11 */ }
41     public void decreaseKey( Position<AnyType> pos, AnyType newVal )
42         { /* Figura 23.12 */ }
43
44     private PairNode<AnyType> compareAndLink( PairNode<AnyType> first,
45                                               PairNode<AnyType> second )
46         { /* Figura 23.14 */ }
47     private PairNode [ ] doubleIfFull( PairNode [ ] array, int index )
48         { /* Implementación de la forma usual; véase el código en línea */ }
49     private PairNode<AnyType> combineSiblings( PairNode<AnyType> firstSibling )
50         { /* Figura 23.15 */ }
51 }
```

---

Figura 23.7 (Continuación).

```

1  /**
2   * Clase estática privada para utilizar con PairingHeap.
3   */
4  private static class PairNode<AnyType> implements Position<AnyType>
5  {
6      /**
7       * Construir el PairNode.
8       * @param theElement el valor almacenado en el nodo.
9       */
10    public PairNode( AnyType theElement )
11    {
12        element = theElement;
13        leftChild = null;
14        nextSibling = null;
15        prev = null;
16    }
17
18    /**
19     * Devuelve el valor almacenado en esta posición.
20     */
21    public AnyType getValue( )
22    {
23        return element;
24    }
25
26    public AnyType element;
27    public PairNode<AnyType> leftChild;
28    public PairNode<AnyType> nextSibling;
29    public PairNode<AnyType> prev;
30 }

```

Figura 23.8 La clase anidada PairNode.

```

1  /**
2   * Encontrar el elemento más pequeño en la cola con prioridad.
3   * @return el elemento más pequeño.
4   * @throws UnderflowException si el montículo de emparej. está vacío.
5   */
6  public AnyType findMin( )
7  {
8      if( isEmpty( ) )
9          throw new UnderflowException( );
10     return root.element;
11 }

```

Figura 23.9 El método findMin para la clase PairingHeap.

```

1  /**
2   * Insertar en la cola con prioridad y devolver una Position
3   * que pueda ser utilizada por decreaseKey.
4   * Los duplicados están permitidos.
5   * @param x el elemento que hay que insertar.
6   * @return el nodo que contiene el elemento recién insertado.
7   */
8  public Position<AnyType> insert( AnyType x )
9  {
10     PairNode<AnyType> newNode = new PairNode<AnyType>( x );
11
12     if( root == null )
13         root = newNode;
14     else
15         root = compareAndLink( root, newNode );
16
17     theSize++;
18     return newNode;
19 }

```

---

**Figura 23.10** La rutina `insert` para la clase `PairingHeap`.

```

1  /**
2   * Eliminar el elemento más pequeño de la cola con prioridad.
3   * @return el elemento más pequeño.
4   * @throws UnderflowException si el montículo de emparej. está vacío.
5   */
6  public AnyType deleteMin( )
7  {
8      if( isEmpty( ) )
9          throw new UnderflowException( );
10
11     AnyType x = findMin( );
12     root.element = null; // Que decreaseKey pueda detec. Position no válida
13     if( root.leftChild == null )
14         root = null;
15     else
16         root = combineSiblings( root.leftChild );
17
18     theSize--;
19     return x;
20 }

```

---

**Figura 23.11** El método `deleteMin` para la clase `PairingHeap`.

```

1  /**
2   * Cambiar el valor del elemento almacenado en el montículo de emparej.
3   * @param pos cualquier Position devuelta por insert.
4   * @param newVal el nuevo valor, que tiene que ser más pequeño que
5   * el valor actualmente almacenado.
6   * @throws IllegalArgumentException si pos es null.
7   * @throws IllegalStateException si nuevo valor es mayor que el anterior.
8   */
9  public void decreaseKey( Position<AnyType> pos, AnyType newVal )
10 {
11     if( pos == null )
12         throw new IllegalArgumentException( );
13
14     PairNode<AnyType> p = (PairNode<AnyType>) pos;
15
16     if( p.element == null || p.element.compareTo( newVal ) < 0 )
17         throw new IllegalStateException( );
18     p.element = newVal;
19     if( p != root )
20     {
21         if( p.nextSibling != null )
22             p.nextSibling.prev = p.prev;
23         if( p.prev.leftChild == p )
24             p.prev.leftChild = p.nextSibling;
25         else
26             p.prev.nextSibling = p.nextSibling;
27
28         p.nextSibling = null;
29         root = compareAndLink( root, p );
30     }
31 }

```

**Figura 23.12** El método decreaseKey para la clase PairingHeap.

Las dos rutinas que quedan son compareAndLink, que combina dos árboles y combineSiblings, que combina todos los hermanos, cuando se le pasa el primer hermano. La Figura 23.13 muestra cómo se combinan dos submontículos. El procedimiento está generalizado para permitir que el segundo submontículo tenga hermanos (lo que es necesario para la segunda pasada en la mezcla de dos pasadas). Como hemos mencionado anteriormente en el capítulo, el submontículo con la mayor raíz se convierte en hijo izquierdo del otro submontículo, mostrándose el código correspondiente en la Figura 23.14. Observe que en varios casos se comprueba si la referencia de un enlace es null antes de acceder a su miembro de datos prev. Esta acción sugiere que podría ser útil disponer de un centinela nullNode –como era habitual en las implementaciones avanzadas de árboles de búsqueda. Dejamos esta posibilidad para que la explore el lector como Ejercicio 23.11.

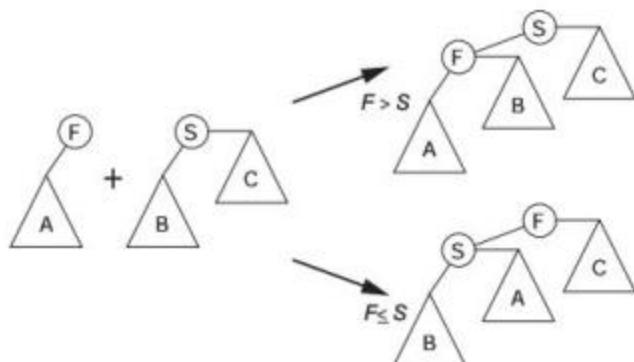


Figura 23.13 El método compareAndLink mezcla dos árboles.

Finalmente, la Figura 23.15 implementa `combineSiblings`. Utilizamos la matriz `treeArray` para almacenar los subárboles. Comenzamos separando los subárboles y almacenándolos en `treeArray`, utilizando el bucle de las líneas 16 a 22. Suponiendo que tengamos más de un hermano para mezclar, hacemos una pasada izquierda-derecha en las líneas 28 y 29. El caso especial de que exista un número impar de árboles se trata en las líneas 31 a 36. Finalizamos la mezcla con una pasada derecha-izquierda en las líneas 40 y 41. Después de terminar, el resultado aparece en la posición 0 de la matriz y puede ser devuelto.

### 23.2.3 Aplicación: algoritmo de Dijkstra para el cálculo del camino más corto

La operación `decreaseKey` es una mejora del algoritmo de Dijkstra en aquellos casos en los que existen muchas llamadas al mismo.

Como ejemplo de la manera en que se utiliza la operación `decreaseKey`, vamos a escribir de nuevo el algoritmo de Dijkstra (véase la Sección 14.3). Recuerde que en cualquier punto estamos siempre manteniendo una cola con prioridad de objetos `Path`, ordenada según el miembro de datos `dist`. Para cada vértice del grafo, necesitábamos un solo objeto `Path` en la cola con prioridad en cualquier instante, pero por conveniencia teníamos muchos. En esta sección vamos a rehacer el código de modo que si se reduce la distancia de un vértice `w`, se encuentra su posición en la cola con prioridad y se ejecuta una operación `decreaseKey` para su correspondiente objeto `Path`.

El nuevo código se muestra en la Figura 23.16, y todos los cambios son relativamente menores. En primer lugar, en la línea 6 declaramos que `pq` es un montículo de emparejamiento en lugar de un montículo binario. Observe que el objeto `Vertex` tiene un miembro de datos adicional `pos`, que representa su posición en la cola con prioridad (y que es `null` si el `Vertex` no está en la cola con prioridad). Inicialmente, todas las posiciones son `null` (lo que se lleva a cabo en `clearAll`). Cada vez que se inserta un vértice en el montículo de emparejamiento, ajustamos su miembro de datos `pos`, en las líneas 13 y 35. El propio algoritmo está simplificado. Ahora simplemente invocamos `deleteMin` mientras que el montículo de emparejamiento no esté vacío, en lugar de invocar repetidamente `deleteMin` hasta que emerja un vértice que no hayamos visto. En consecuencia, ya no necesitamos el miembro de datos `scratch`. Compare las líneas 15 a 18 con el código correspondiente

```

1  /**
2   * Método interno que es la operación básica para mantener el orden.
3   * Enlaza el primero y el segundo para satisfacer el orden del montículo.
4   * @param first raíz del árbol 1, que no puede ser null.
5   * first.nextSibling DEBE ser null al entrar.
6   * @param second raíz del árbol 2, que puede ser null.
7   * @return resultado de la mezcla del árbol.
8   */
9  private PairNode<AnyType> compareAndLink( PairNode<AnyType> first,
10                                         PairNode<AnyType> second )
11 {
12     if( second == null )
13         return first;
14
15     if( second.element.compareTo( first.element ) < 0 )
16     {
17         // Conectar el primero como hijo izquierdo del segundo
18         second.prev = first.prev;
19         first.prev = second;
20         first.nextSibling = second.leftChild;
21         if( first.nextSibling != null )
22             first.nextSibling.prev = first;
23         second.leftChild = first;
24         return second;
25     }
26     else
27     {
28         // Conectar el segundo como hijo izquierdo del primero
29         second.prev = first;
30         first.nextSibling = second.nextSibling;
31         if( first.nextSibling != null )
32             first.nextSibling.prev = first;
33         second.nextSibling = first.leftChild;
34         if( second.nextSibling != null )
35             second.nextSibling.prev = second;
36         first.leftChild = second;
37         return first;
38     }
39 }

```

Figura 23.14 La rutina compareAndLink.

```
1 // La matriz del árbol para combineSiblings
2 private PairNode<AnyType> treeArray = new PairNode[ 5 ];
3
4 /**
5 * Método interno que implementa la mezcla en dos pasadas.
6 * @param firstSibling la raíz del conglomerado;
7 * se presupone que no es null.
8 */
9 private PairNode<AnyType> combineSiblings( PairNode<AnyType> firstSibling )
10 {
11     if( firstSibling.nextSibling == null )
12         return firstSibling;
13
14     // Almacena los subárboles en una matriz
15     int numSiblings = 0;
16     for( ; firstSibling != null; numSiblings++ )
17     {
18         treeArray = doubleIfFull( treeArray, numSiblings );
19         treeArray[ numSiblings ] = firstSibling;
20         firstSibling.prev.nextSibling = null; // break links
21         firstSibling = firstSibling.nextSibling;
22     }
23     treeArray = doubleIfFull( treeArray, numSiblings );
24     treeArray[ numSiblings ] = null;
25
26     // Combina los subárboles de dos en dos, yendo de izquierda a derecha
27     int i = 0;
28     for( ; i + 1 < numSiblings; i += 2 )
29         treeArray[ i ] = compareAndLink( treeArray[ i ], treeArray[ i + 1 ] );
30
31     int j = i - 2;
32
33     // j tiene el resultado del último compareAndLink.
34     // Si hay un número impar de árboles, obtener el último.
35     if( j == numSiblings - 3 )
36         treeArray[ j ] = compareAndLink( treeArray[ j ], treeArray[ j + 2 ] );
37
38     // Ahora ir de derecha a izquierda, mezclando el último árbol con el
39     // penúltimo. El resultado será el nuevo último árbol.
40     for( ; j >= 2; j -= 2 )
41         treeArray[ j - 2 ] = compareAndLink( treeArray[ j - 2 ], treeArray[ j ] );
42
43     return (PairNode<AnyType>) treeArray[ 0 ];
44 }
```

Figura 23.15 El corazón del algoritmo del montículo de emparejamiento: Implementación de una mezcla de dos pasadas para combinar todos los hermanos, dado el primer hermano.

```

1  /**
2  * Algoritmo camino pond. más corto con un único origen usando mont. emparej.
3  */
4  public void dijkstra( String startName )
5  {
6      PairingHeap<Path> pq = new PairingHeap<Path>();
7
8      Vertex start = vertexMap.get( startName );
9      if( start == null )
10         throw new NoSuchElementException( "Start vertex not found" );
11
12     clearAll( );
13     start.pos = pq.insert( new Path( start, 0 ) ); start.dist = 0;
14
15     while ( !pq.isEmpty( ) )
16     {
17         Path vrec = pq.deleteMin( );
18         Vertex v = vrec.dest;
19
20         for( Edge e : v.adj )
21         {
22             Vertex w = e.dest;
23             double cvw = e.cost;
24
25             if( cvw < 0 )
26                 throw new GraphException( "Graph has negative edges" );
27
28             if( w.dist > v.dist + cvw )
29             {
30                 w.dist = v.dist + cvw;
31                 w.prev = v;
32
33                 Path newVal = new Path( w, w.dist );
34                 if( w.pos == null )
35                     w.pos = pq.insert( newVal );
36                 else
37                     pq.decreaseKey( w.pos, newVal );
38             }
39         }
40     }
41 }

```

Figura 23.16 Algoritmo de Dijkstra, utilizando el montículo de emparejamiento y la operación decreaseKey.

al presentado en la Figura 14.27. Lo único que nos queda por hacer son las actualizaciones después de la línea 28 que indican que hace falta un cambio. Si el vértice no ha sido insertado nunca en la cola con prioridad, lo insertamos por primera vez actualizando su miembro de datos `pos`. En caso contrario, simplemente llamamos a `decreaseKey` en la línea 37.

El que la implementación con montículo binario del algoritmo de Dijkstra sea más rápida que la implementación con montículo de emparejamiento dependerá de varios factores. Un estudio (véase la sección Referencias) sugiere que el montículo de emparejamiento es ligeramente mejor que el montículo binario cuando los dos se implementan cuidadosamente. Los resultados dependen en gran medida de los detalles de codificación y de la frecuencia de las operaciones `decreaseKey`. Son necesarios más estudios para terminar de decidir si el montículo de emparejamiento resulta adecuado en la práctica.

## Resumen

En este capítulo hemos descrito dos estructuras de datos que soportan la mezcla y que son eficientes en el sentido amortizado: el montículo sesgado y el montículo de emparejamiento. Ambos son fáciles de implementar, porque carecen de una propiedad estructural rígida. El montículo de emparejamiento parece tener utilidad práctica, aunque su análisis sigue constituyendo un problema abierto bastante intrigante.

En el Capítulo 24, que es el último, describiremos una estructura de datos que se utiliza para mantener conjuntos disjuntos y que también tiene un notable análisis amortizado.



### Conceptos clave

**mezcla de dos pasadas** El orden en el que se mezclan los subárboles del montículo de emparejamiento es importante. El algoritmo más simple es la mezcla de dos pasadas, en la que los subárboles se mezclan por parejas en una pasada de izquierda a derecha y luego se realiza otra pasada de derecha a izquierda para finalizar la mezcla. (863)

**montículo de emparejamiento** Un árbol  $M$ -ario con ordenación de montículo y sin restricciones estructurales para el que todas las operaciones, salvo el borrado, requieren un tiempo constante de caso peor. Su análisis no se ha completado, pero parece comportarse bien en la práctica. (862)

**montículo sesgado** Un árbol binario con ordenación de montículo y sin condición de equilibrado que soporta todas las operaciones en un tiempo amortizado logarítmico. (857)



### Errores comunes

1. No se puede utilizar en la práctica una implementación recursiva del montículo sesgado, porque la profundidad de recursión podría ser lineal.

- 2 Tenga cuidado de no perder la cuenta de los enlaces `prev` en el montículo sesgado.
- 3 Efectúe comprobaciones para asegurarse de que las referencias no sean `null` a lo largo de todo el código del montículo de emparejamiento.
- 4 Cuando se realiza una mezcla, no debe haber ningún nodo que forme parte de dos montículos de emparejamiento.



## Internet

Esta disponible la clase para el montículo de emparejamiento, junto con un programa de prueba. La Figura 23.16 forma parte de la clase `Graph` mostrada en el Capítulo 14 (**`Graph.java`**).

**`PairingHeap.java`** Contiene la implementación para la clase `PairingHeap`.



## Ejercicios

### EN RESUMEN

- 23.1** Muestre el resultado de un montículo de emparejamiento construido a partir de la secuencia de inserción
  - a. 3, 4, 5, 6, 7, 8, 9.
  - b. 4, 3, 5, 2, 6, 7, 1.
- 23.2** Muestre el resultado de un montículo sesgado construido a partir de la secuencia de inserción
  - a. 3, 4, 5, 6, 7, 8, 9.
  - b. 4, 3, 5, 2, 6, 7, 1.
- 23.3** Para cada montículo de los Ejercicios 23.1 y 23.2, muestre el resultado de dos operaciones `deleteMin`.

### EN TEORÍA

- 23.4** Demuestre que no es una buena idea utilizar una pila para implementar la operación `combineSiblings` en los montículos de emparejamiento. Hágalo construyendo una secuencia que tenga un coste amortizado lineal por cada operación.
- 23.5** Describa cómo implementar `increaseKey` para montículos de emparejamiento.
- 23.6** Demuestre que tanto la operación `decreaseKey` como la operación `increaseKey` puede ser soportadas por los montículos sesgados en un tiempo amortizado logarítmico.
- 23.7** Demuestre que la cota amortizada logarítmica para operaciones con montículos sesgados no es una cota de caso peor, proporcionando una secuencia de operaciones que conduzcan a una operación `merge` que requiera un tiempo lineal.

- 23.8** Describa un algoritmo `buildHeap` de tiempo lineal para el montículo sesgado.
- 23.9** Demuestre que almacenar la longitud del camino derecho para cada nodo del árbol nos permite imponer una condición de equilibrado que proporciona un tiempo logarítmico de caso peor por cada operación. Dicha estructura se denomina *montículo izquierdista*.

### EN LA PRÁCTICA

- 23.10** Añada el método público `merge` a la clase `PairingHeap`. Asegúrese de que cada nodo aparezca en un único árbol.

### PROYECTOS DE PROGRAMACIÓN

- 23.11** Implemente el algoritmo del montículo de emparejamiento con un centinela `nullNode`.
- 23.12** Si la operación `decreaseKey` no está soportada, los enlaces al padre no son necesarios. Implemente el algoritmo del montículo de emparejamiento sin enlaces a los padres y compare su rendimiento con el algoritmo correspondiente al montículo binario y/o el montículo sesgado y/o el árbol splay.



### Referencias

El *montículo izquierdista* [1] fue la primera cola con prioridad mezclable con una eficiencia suficiente. Es la variante de caso peor del montículo sesgado sugerido en el Ejercicio 23.9. Los montículos sesgados se describen en [6], que también contiene soluciones a los Ejercicios 23.6 y 23.7.

[3] describe el montículo de emparejamiento y demuestra que, cuando se utiliza la mezcla de dos pasadas, el coste amortizado de todas las operaciones es logarítmico. Ya se había conjecturado hace tiempo que el coste amortizado de todas las operaciones, salvo `deleteMin`, es en realidad constante y que el coste amortizado de `deleteMin` es logarítmico, de modo que cualquier secuencia de  $D$  `deleteMin` y de otras  $I$  operaciones tarda un tiempo  $O(I + D \log N)$ . Sin embargo, recientemente, se ha demostrado que esta conjectura es falsa [2]. Una estructura de datos que sí que consigue esta cota, pero es demasiado complicada como para ser práctica es el *montículo de Fibonacci* [4]. La esperanza es que el montículo de emparejamiento constituya una alternativa práctica al tan interesante, desde el punto de vista teórico, montículo de Fibonacci, aun cuando su caso peor es ligeramente peor. En [7] se analizan los montículos izquierdistas y los montículos de Fibonacci.

En [5] hay una comparación de diversas colas con prioridad en el contexto de resolución del problema del árbol mínimo de recubrimiento (del que hablamos en la Sección 24.2.2) utilizando 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, CA, 1972.

- 2 M. L. Fredman, "On the Efficiency of Pairing Heaps and Related Data Structures", *Journal of the ACM* 46 (1999), 473–501.
- 3 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.
- 4 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.
- 5 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.
- 6 D. D. Sleator y R. E. Tarjan, "Self-adjusting Heaps", *SIAM Journal on Computing* 15 (1986), 52–69.
- 7 M. A. Weiss, *Data Structures and Algorithm Analysis in Java*, 2<sup>a</sup> ed., Addison-Wesley, Reading, MA, 2007.



## La clase conjunto disjunto

En este capítulo vamos a describir una eficiente estructura de datos para resolver el problema de equivalencia: la clase conjunto disjunto. Esta estructura de datos es sencilla de implementar, requiriendo cada rutina solo unas pocas líneas de código. Su implementación es también extremadamente rápida, necesitando un tiempo medio constante por cada operación. Esta estructura de datos es también interesante desde el punto de vista teórico, porque su análisis es extremadamente difícil. La forma funcional para el caso peor no se parece en nada a ninguna de las que hemos analizado hasta ahora en este texto.

En este capítulo veremos

- Tres aplicaciones simples de la clase conjunto disjunto.
- Una forma de implementar un conjunto disjunto con un esfuerzo mínimo de codificación.
- Un método para incrementar la velocidad del conjunto disjunto, utilizando dos observaciones simples.
- Un análisis del tiempo de ejecución de una implementación rápida del conjunto disjunto.

### 24.1 Relaciones de equivalencia

Decimos que en un conjunto  $S$  existe una *relación*  $R$  si para todo par de elementos  $(a, b)$ ,  $a R b$  es verdadero o falso. Si  $a R b$  es verdadero, decimos que  $a$  está relacionado con  $b$ .

Una *relación de equivalencia* es una relación  $R$  que satisface tres propiedades:

1. *Reflexiva*:  $a R a$  es verdadero para todo  $a \in S$ .
2. *Simétrica*:  $a R b$  si y solo si  $b R a$ .
3. *Transitiva*:  $a R b$  y  $b R c$  implican que  $a R c$ .

En un conjunto decimos que hay definida una relación si cada par de elementos está relacionado o no lo está. Una relación de equivalencia es reflexiva, simétrica y transitiva.

La conectividad eléctrica, donde todas las conexiones se realizan mediante hilos metálicos es una relación de equivalencia. La relación es claramente reflexiva, ya que cada componente está conectado consigo mismo. Si  $a$  está eléctricamente conectado con  $b$ , entonces  $b$  tiene que estar eléctricamente conectado con  $a$ , por lo que la relación es simétrica. Finalmente, si  $a$  está conectado con  $b$  y  $b$  está conectado con  $c$ , entonces  $a$  está conectado con  $c$ .

De forma similar, la conectividad a través de una red bidireccional forma clases de equivalencia de componentes conectados. Sin embargo, si las conexiones de la red son dirigidas (es decir, una conexión de  $v$  a  $w$  no implica una conexión de  $w$  a  $v$ ), no tendremos una relación de equivalencia, porque no se cumple la propiedad simétrica. Un ejemplo sería una relación en la que la ciudad  $a$  estuviera relacionada con la ciudad  $b$  si es posible viajar por carretera de  $a$  hasta  $b$ . Esta relación sería una relación de equivalencia si las carreteras fueran de dos sentidos.

## 24.2 Equivalencia dinámica y aplicaciones

Para cualquier relación de equivalencia, denotada mediante  $\sim$ , el problema natural consiste en decidir, para cualesquiera  $a$  y  $b$  si  $a \sim b$ . Si la relación se almacena como una matriz bidimensional de variables booleanas, podemos comprobar la equivalencia en un tiempo constante. El problema es que la relación se suele definir implícitamente, en lugar de explícitamente.

Por ejemplo, podemos definir una relación de equivalencia para el conjunto de cinco elementos  $\{a_1, a_2, a_3, a_4, a_5\}$ . Este conjunto nos da 25 pares de elementos, cada uno de los cuales podrá o no estar relacionado. Sin embargo, la información de que  $a_1 \sim a_2, a_3 \sim a_4, a_1 \sim a_5$  y  $a_4 \sim a_2$  están relacionados implica que todos los pares están relacionados. Lo que queremos es ser capaces de deducir esta condición rápidamente.

La *clase de equivalencia* de un elemento  $x \in S$  es el subconjunto de  $S$  que contiene todos los elementos relacionados con  $x$ . Observe que las clases de equivalencia forman una partición de  $S$ :

cada miembro de  $S$  aparece exactamente en una sola clase de equivalencia. Para decidir si  $a \sim b$ , solo necesitamos comprobar si  $a$  y  $b$  pertenecen a la misma clase de equivalencia. Esta información nos proporciona la estrategia para resolver el problema de equivalencia.

La entrada es, inicialmente, una colección de  $N$  conjuntos, cada uno con un elemento. En esta representación inicial, todas las relaciones son falsas (excepto las relaciones reflexivas). Cada conjunto tiene un elemento distinto, por lo que  $S_i \cap S_j = \emptyset$  y dichos conjuntos (en los que ninguna pareja de conjuntos contendrá elementos comunes) se denominan *conjuntos disjuntos*.

Las dos *operaciones básicas para la clase conjunto disjunto* son *find*, que devuelve el nombre del conjunto (es decir, la clase de equivalencia) que contiene un elemento especificado, y *union*, que añade relaciones. Si queremos añadir el par  $(a, b)$  a la lista de relaciones, determinamos primero si  $a$  y  $b$  ya están relacionados. Para hacer esto, ejecutamos operaciones *find* tanto para  $a$  como para  $b$  y comprobamos si los dos elementos se encuentran

en la misma clase de equivalencia; si no lo están, aplicamos *union*. Esta operación mezcla las dos clases de equivalencia que contienen  $a$  y  $b$  en una nueva clase de equivalencia. En términos de la teoría de conjuntos, el resultado es un nuevo conjunto  $S_k = S_i \cup S_j$  que creamos destruyendo simultáneamente los originales y preservando el carácter disjunto de todos los conjuntos. La estructura de datos para hacer esto se denomina comúnmente *estructura de datos union/find para conjuntos disjuntos*. El *algoritmo union/find* se ejecuta procesando solicitudes *union/find* dentro de la estructura de datos de conjuntos disjuntos.

El algoritmo es *dinámico* porque, durante el curso de la ejecución del mismo, los conjuntos pueden cambiar mediante la operación *union*. El algoritmo también debe operar como un

La clase de equivalencia de un elemento  $x$  en el conjunto  $S$  es el subconjunto de  $S$  que contiene todos los elementos relacionados con  $x$ . Las clases de equivalencia forman conjuntos disjuntos.

Las dos operaciones básicas de la clase conjunto disjunto son *union* y *find*.

*algoritmo en línea*, de modo que cuando se realiza una operación `find`, hay que proporcionar una respuesta antes de poder acceder al resultado de la siguiente consulta. Otra posibilidad sería utilizar un *algoritmo fuera de línea*, en el que se proporcionara el resultado de la secuencia completa de solicitudes `union` y `find`. La respuesta que proporcione para cada `find` debe ser coherente con todas las operaciones `union` realizada antes del `find`. Sin embargo, el algoritmo puede proporcionar todas las respuestas después de haber procesado *todas* las solicitudes. Esta distinción es similar a la diferencia existente entre realizar un examen escrito (que generalmente es un examen fuera de línea, porque solo estamos obligados a dar las respuestas antes de que se termine el tiempo) o someterse a un examen oral (que es en línea, porque hay que responder a la pregunta actual antes de pasar a la siguiente).

Observe que no realizamos ninguna operación para comparar los valores relativos de los elementos, sino que simplemente necesitamos conocer su ubicación. Por esta razón, podemos asumir que todos los elementos están numerados secuencialmente, partiendo de 0, y que la numeración se puede determinar fácilmente mediante algún tipo de esquema hash.

Antes de describir cómo implementar las operaciones `union` y `find`, vamos a hablar de tres aplicaciones de esta estructura de datos.

En un *algoritmo en línea* hay que proporcionar la respuesta a cada pregunta antes de procesar la consulta siguiente.

Los elementos del conjunto están numerados secuencialmente comenzando por 0.

### 24.2.1 Aplicación: generación de laberintos

Un ejemplo de la estructura de datos `union/find` sería la generación de laberintos, como el mostrado en la Figura 24.1. El punto de partida es la esquina superior izquierda, mientras que el punto de llegada es la esquina inferior derecha. Podemos contemplar el laberinto como un rectángulo de  $50 \times 88$  celdas, en el que la celda superior izquierda está conectada con la celda inferior derecha, y las celdas están separadas de sus vecinas mediante paredes.

Un algoritmo simple para generar el laberinto consiste en comenzar con paredes en todas partes (salvo a la entrada y a la salida). Después, seleccionamos continuamente una pared de manera

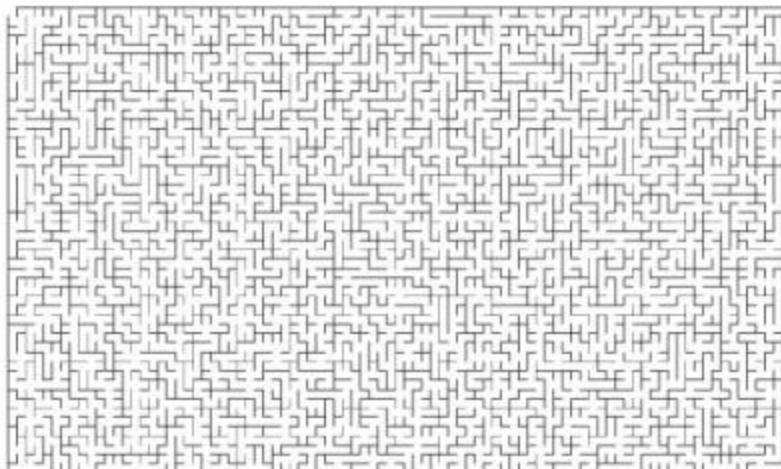


Figura 24.1 Un laberinto de  $50 \times 88$ .

aleatoria y la eliminamos si las celdas separadas por la pared no están ya conectadas entre sí. Si repetimos este proceso hasta que estén conectadas las celdas inicial y final, tendremos un laberinto. Pero en realidad es mejor continuar eliminando paredes hasta que toda celda sea alcanzable desde cualquier otra celda, porque al hacer esto abrimos más caminos sin salida dentro del laberinto.

Vamos a ilustrar el algoritmo con un laberinto  $5 \times 5$  y la Figura 24.2 muestra la configuración inicial. Utilizamos la estructura de datos union/find para representar conjuntos de celdas que están conectadas entre sí. Inicialmente, hay paredes en todas partes y cada celda pertenece a su propia clase de equivalencia.

La Figura 24.3 muestra una etapa posterior del algoritmo, después de haber eliminado unas cuantas paredes. Suponga, en esa etapa, que seleccionamos aleatoriamente la pared que conecta las celdas 8 y 13. Puesto que 8 y 13 ya están conectadas (pertenecen al mismo conjunto), no eliminamos la pared, porque si lo hicieramos estaríamos simplemente trivializando el laberinto. Suponga que a continuación seleccionamos aleatoriamente las celdas 18 y 13. Realizando dos operaciones `find`, determinamos que estas celdas pertenecen a conjuntos distintos; por tanto, 18 y 13 no están conectadas. En consecuencia, eliminamos la pared que las separa, como se muestra en la Figura 24.4. Como resultado de esta operación, los conjuntos que contienen las celdas 18 y 13 se combinan mediante una operación `union`. La razón es que todas las celdas anteriormente conectadas con 18

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14}  
{15} {16} {17} {18} {19} {20} {21} {22} {23} {24}

Figura 24.2 Estado inicial: todas las paredes están levantadas y cada celda pertenece a su propio conjunto.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12}  
{16, 17, 18, 22} {19} {20} {21} {22} {23} {24}

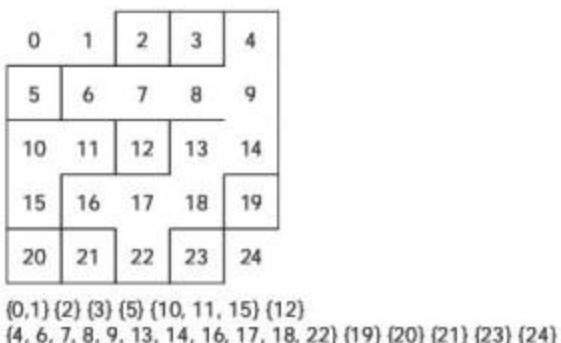
Figura 24.3 En un cierto punto del algoritmo, se han eliminado varias paredes y se han mezclado diversos conjuntos. En este punto, si seleccionamos aleatoriamente la pared entre 8 y 13, no eliminaremos esa pared, porque las celdas 8 y 13 ya están conectadas.

están ahora conectadas a todas las celdas anteriormente conectadas con 13. Al final del algoritmo, como se muestra en la Figura 24.5, todas las celdas estarán conectadas y habremos terminado.

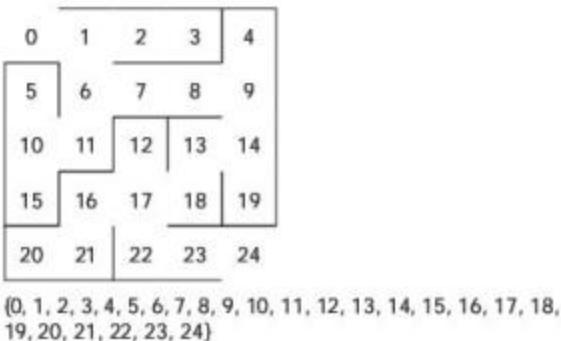
El tiempo de ejecución del algoritmo está dominado por los costes de las operaciones `union`/`find`. El tamaño del universo `union/find` es el número de celdas. El número de operaciones `find` es proporcional al número de celdas, porque el número de paredes eliminadas es 1 menos que el número de celdas. Sin embargo, si examinamos el problema cuidadosamente, podemos ver que solo hay aproximadamente el doble de paredes que de celdas para empezar. Por tanto, si  $N$  es el número de celdas y se realizan dos operaciones `find` por cada pared seleccionada aleatoriamente, obtenemos una estimación de entre (aproximadamente)  $2N$  y  $4N$  operaciones `find` a lo largo de todo el algoritmo. Por tanto, el tiempo de ejecución del algoritmo dependerá del coste de  $O(N)$  operaciones `union` y  $O(N)$  operaciones `find`.

### 24.2.2 Aplicación: árboles mínimos de recubrimiento

Un *árbol de recubrimiento* en un grafo no dirigido es un árbol formado por aristas del grafo que permiten conectar todos los vértices del grafo. A diferencia de los grafos del Capítulo 14, una arista  $(u, v)$  en un grafo  $G$  es idéntica a una arista  $(v, u)$ . El coste de un árbol de recubrimiento es igual a la



**Figura 24.4** Seleccionamos aleatoriamente la pared entre las celdas 18 y 13 de la Figura 24.3; esta pared se elimina, porque las celdas 18 y 13 no estaban ya conectadas. A continuación, mezclamos sus conjuntos.



**Figura 24.5** Finalmente, se habrán eliminado 24 paredes y todos los elementos pertenecerán al mismo conjunto.

El *árbol mínimo de recubrimiento* es un subgrafo conectado de  $G$  que abarca todos los vértices con un coste total mínimo.

suma de los costes de las aristas que forman el árbol. El *árbol mínimo de recubrimiento* es un subgrafo conectado de  $G$  que abarca todos los vértices con un coste mínimo. Solo podrá existir un árbol mínimo de recubrimiento si el subgrafo de  $G$  es conectado. Como veremos enseguida, la comprobación de la conectividad de un grafo puede realizarse como parte del cálculo del árbol mínimo de recubrimiento.

En la Figura 24.6(b), el grafo es un árbol mínimo de recubrimiento del grafo de la Figura 24.6(a) (sucede además que ese recubrimiento es único, lo que resulta inusual si el grafo tiene muchas aristas de igual coste). Observe que el número de aristas en el árbol mínimo de recubrimiento es  $|V| - 1$ . El árbol mínimo de recubrimiento es un *árbol*, porque es acíclico, es *de recubrimiento* porque abarca todos los vértices y es *mínimo* por la razón obvia. Suponga que necesitamos conectar varias ciudades con carreteras minimizando el coste total de construcción, con la condición de que podamos pasar de una carretera a otra únicamente en una ciudad (en otras palabras, no se permiten conexiones adicionales). Entonces, necesitaremos resolver un problema de árbol mínimo de recubrimiento, en el que cada vértice será una ciudad y cada arista equivaldrá al coste de construir una carretera entre las dos ciudades que conecte.

Un problema relacionado es el *problema del árbol mínimo de Steiner*, que es como el problema del árbol mínimo de recubrimiento, salvo porque se pueden crear uniones nuevas como parte de la solución. El problema del árbol mínimo de Steiner es mucho más difícil de resolver. Sin embargo, se puede demostrar que si el coste de una conexión es proporcional a la distancia euclídea, el árbol mínimo de recubrimiento es como máximo un 15 por ciento más costoso que el árbol mínimo de Steiner. Por tanto, un árbol mínimo de recubrimiento, que resulta fácil de calcular, proporciona una buena aproximación para el árbol mínimo de Steiner, cuyo cálculo es mucho más complicado.

Se utiliza el algoritmo de Kruskal para seleccionar aristas pro orden creciente de coste y añadirlas al árbol si no crean un ciclo.

Se utiliza un algoritmo sencillo, comúnmente denominado *algoritmo de Kruskal*, para seleccionar de modo continuo aristas por orden de menor peso y añadir una arista al árbol si esta no hace que se forme un ciclo. Formalmente, lo que hace el algoritmo de Kruskal es mantener un bosque (una colección de árboles). Inicialmente, hay  $|V|$  árboles de un solo nodo. Añadir una arista hace que se combinen dos árboles en uno solo. Cuando el algoritmo termina, solo queda un árbol, que es el árbol mínimo de recubrimiento.<sup>1</sup> Contando el número de aristas añadidas, podemos determinar cuándo debería finalizar el algoritmo.

La Figura 24.7 muestra la operación del algoritmo de Kruskal sobre el grafo mostrado en la Figura 24.6. Las primeras cinco aristas se aceptan, porque no crean ciclos. Las siguientes dos aristas,  $(v_1, v_3)$  (de coste 3) y luego  $(v_0, v_2)$  (de coste 4), se rechazan porque cada una de ellas crearía un ciclo en el árbol. La siguiente arista considerada se acepta y, puesto que es la sexta arista en un grafo de siete vértices, podemos finalizar el algoritmo.

Ordenar las aristas para las comprobaciones es bastante sencillo. Podemos ordenarlas con un coste de  $|E| \log |E|$  y luego recorrerla la matriz ordenada de aristas. Alternativamente, podemos construir una cola con prioridad de  $|E|$  aristas e ir extrayendo repetidamente aristas, invocando `deleteMin`. Aunque la cota de caso peor no varía, en ocasiones es mejor emplear una cola con prioridad, porque el algoritmo de Kruskal tiende a comprobar únicamente una pequeña fracción de

<sup>1</sup> Si el grafo no está conectado, el algoritmo terminará con más de un árbol. Cada árbol representará entonces un árbol mínimo de recubrimiento para uno de los componentes conectados del grafo.

las aristas en grafos aleatorios. Por supuesto, en el caso peor puede que sea necesario tener que comprobar todas las aristas. Por ejemplo, si hubiera un vértice adicional  $v_8$  y una arista  $(v_5, v_8)$  de coste 100, habría que examinar todas las aristas. En este caso, terminaríamos antes realizando una ordenación rápida al principio. En la práctica, la elección entre una cola con prioridad y una ordenación inicial es una apuesta relativa al número de aristas que probablemente tengamos que examinar.

Podemos ordenar las aristas o utilizar una cola con prioridad.

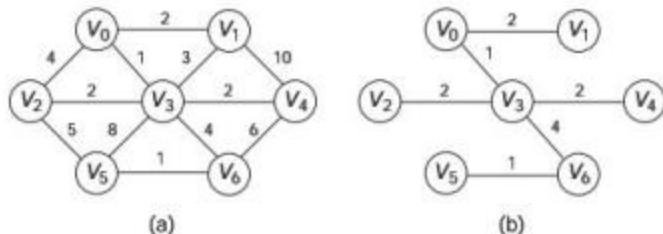


Figura 24.6 (a) Un grafo  $G$  y (b) su árbol mínimo de recubrimiento.

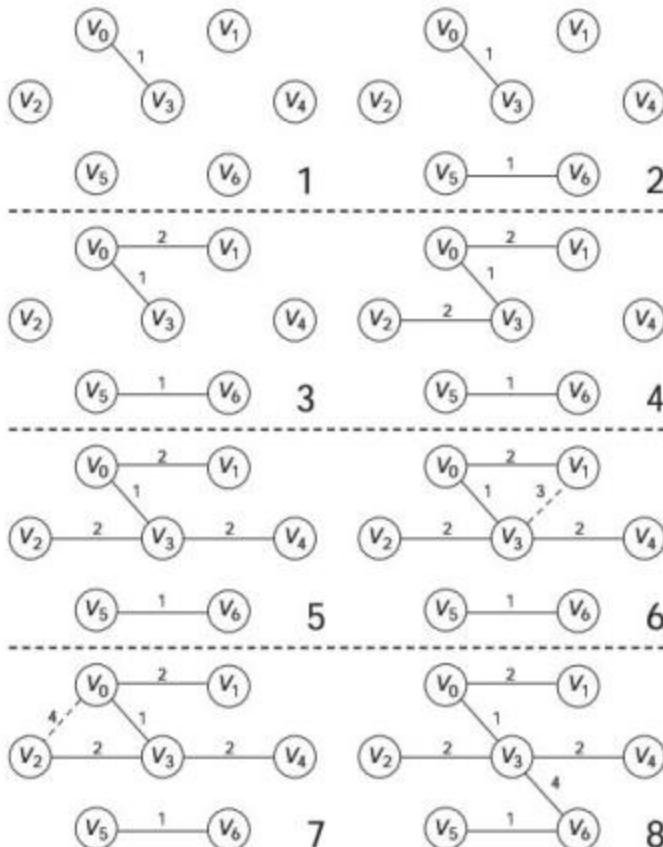


Figura 24.7 Algoritmo de Kruskal después de tomar en consideración cada arista. Las etapas están ordenadas de izquierda a derecha y de arriba a abajo, como indican los números.

La comprobación de la aparición de ciclos se hace utilizando una estructura de datos union/find.

Más interesante es la cuestión de cómo decidir si una arista  $(u, v)$  debe ser aceptada o rechazada. Claramente añadir la arista  $(u, v)$  hará que aparezca un ciclo si (y solo si)  $u$  y  $v$  ya están conectadas en el *bosque* de recubrimiento actual, que es una colección de árboles. Por tanto, lo que haremos es simplemente mantener cada componente conectado del bosque de recubrimiento en forma de un conjunto disjunto. Inicialmente, cada vértice se encuentra en su propio conjunto disjunto. Si  $u$  y  $v$  están en el mismo conjunto disjunto, como se puede comprobar realizando dos operaciones `find`, rechazaremos la arista porque  $u$  y  $v$  ya están conectadas. En caso contrario, aceptaremos la arista y realizaremos una operación `union` con los dos conjunto disjuntos que contienen  $u$  y  $v$ , combinando en la práctica los dos componentes conectados. Este resultado es precisamente lo que andamos buscando porque una vez que la arista  $(u, v)$  se ha añadido al bosque de recubrimiento, si  $w$  estaba conectado a  $u$  y  $x$  estaba conectado a  $v$ ,  $x$  y  $w$  tienen que estar conectados y por tanto pertenecer al mismo conjunto.

### 24.2.3 Aplicación: el problema del ancestro común más próximo

Otro ejemplo que permite ilustrar el uso de la estructura de datos `union/find` es el *problema del ancestro común más próximo (NCA, nearest common ancestor)*.

#### Problema del ancestro común más próximo en versión fuera de línea

Dado un árbol y una lista de pares de nodos del árbol, encontrar el ancestro común más próximo para cada par de nodos.

La solución del problema del ancestro común más próximo es importante en los algoritmos para grafos y en las aplicaciones de biología computacional.

Por ejemplo, la Figura 24.8 muestra un árbol con una lista de pares que contiene cinco solicitudes. Para el par de nodos  $x$  y  $z$ , el nodo  $C$  es el ancestro común más próximo a ambos. ( $A$  y  $B$  también son ancestros, pero no son el más próximo.) El problema es de tipo fuera de línea, porque podemos conocer la secuencia completa de solicitudes antes de proporcionar la primera respuesta. La solución a este problema es importante en las aplicaciones de la teoría de grafos y en las aplicaciones de biología computacional (en las que los árboles representan la evolución).

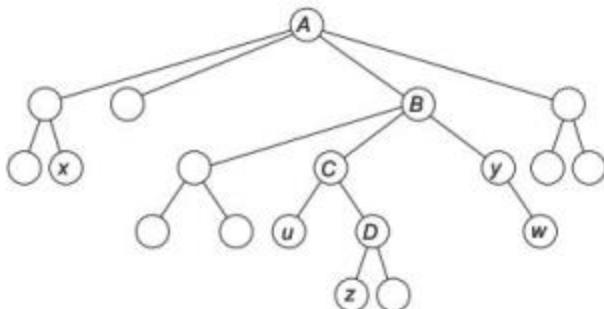


Figura 24.8 El ancestro común más próximo para cada solicitud de la secuencia de pares  $(x, y)$ ,  $(u, z)$ ,  $(w, x)$ ,  $(z, w)$  y  $(w, y)$  es  $A$ ,  $C$ ,  $A$ ,  $B$  e  $y$ , respectivamente.

El algoritmo funciona realizando un recorrido del árbol en postorden. Cuando estemos a punto de volver después de procesar un nodo, examinamos la lista de pares para determinar si hay que realizar algún cálculo de ancestro. Si  $u$  es el nodo actual, si  $(u, v)$  se encuentra en la lista de pares y si hemos terminado ya la llamada recursiva a  $v$ , tendremos la suficiente información para determinar el ancestro común más próximo (NCA) de  $(u, v)$ .

La Figura 24.9 nos ayuda a entender cómo funciona este algoritmo. Aquí, estamos a punto de finalizar la llamada recursiva a  $D$ . Todos los nodos sombreados han sido visitados por una llamada recursiva  $y$ , excepción hecha de los nodos situados en el camino hasta  $D$ ; todas las llamadas recursivas ya han sido finalizadas. Marcamos cada nodo después de haber completado su llamada recursiva. Si  $v$  está marcado, entonces  $\text{NCA}(D, v)$  será algún nodo situado en el camino hasta  $D$ . El *ancla* de un nodo  $v$  visitado (pero no necesariamente marcado) es el nodo del camino de acceso actual que esté más próximo a  $v$ . En la Figura 24.9, el ancla de  $p$  es  $A$ , el ancla de  $q$  es  $B$  y  $r$  no tiene ancla porque todavía no ha sido visitado; podríamos decir que el ancla de  $r$  es  $r$  en el momento de visitar  $r$  por primera vez. Cada nodo del camino de acceso actual es un ancla (al menos de sí mismo). Además, los nodos visitados forman clases de equivalencia: dos nodos estarán relacionados si tienen el mismo ancla, y podemos considerar cada nodo no visitado como perteneciente a su propia clase de equivalencia. Ahora suponga de nuevo que  $(D, v)$  se encuentra en la lista de pares. Entonces tenemos tres casos:

- $v$  no está marcado, así que no tenemos información para calcular  $\text{NCA}(D, v)$ . Además, cuando marquemos  $v$  seremos capaces de determinar  $\text{NCA}(v, D)$ .
- $v$  está marcado pero no se encuentra en el subárbol de  $D$ , por lo que  $\text{NCA}(D, v)$  será igual al ancla de  $v$ .
- $v$  se encuentra en el subárbol de  $D$ , por lo que  $\text{NCA}(D, v) = D$ . Observe que esto no es un caso especial porque el ancla de  $v$  es  $D$ .

Lo único que nos resta por hacer es cerciorarnos de poder determinar en todo instante el ancla de cualquier nodo visitado. Podemos hacer esto fácilmente mediante el algoritmo union/find. Después de volver de una llamada recursiva, invocamos `union`. Por ejemplo, después de volver de la llamada recursiva a  $D$  en la Figura 24.9, cambiará de  $D$  a  $C$  el ancla de todos los nodos en  $D$ . La nueva situación se muestra en la Figura 24.10. Por tanto, necesitaremos mezclar las dos clases de equivalencia en una sola. En cualquier punto, podemos obtener el ancla de un vértice  $v$  mediante una llamada a la operación `find` para conjuntos disjuntos. Puesto que `find` devuelve un número de conjunto, utilizamos una matriz `anchor` para almacenar el nodo ancla correspondiente a cada conjunto concreto.

En la Figura 24.11 se muestra una implementación en pseudocódigo del algoritmo NCA. Como hemos mencionado anteriormente en el capítulo, generalmente la operación `find` se basa en la suposición de que los elementos del conjunto son  $0, 1, \dots, N - 1$ , así que asignamos un número de orden en cada nodo del árbol, en un paso de preprocesamiento que calcula el tamaño del árbol. Un enfoque orientado a objetos podría tratar de incorporar una correspondencia dentro del método `find`, pero

Se puede utilizar un recorrido en postorden para solucionar el problema.

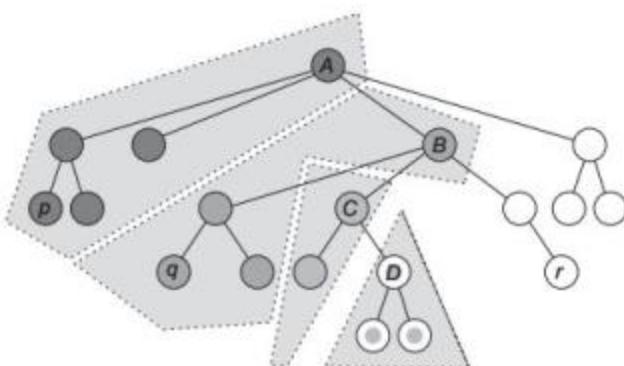
El ancla de un nodo  $v$  visitado (pero no necesariamente marcado) es el nodo del camino de acceso actual que esté más próximo a  $v$ .

Se utiliza el algoritmo union/find para mantener los conjuntos de nodos con anclas comunes.

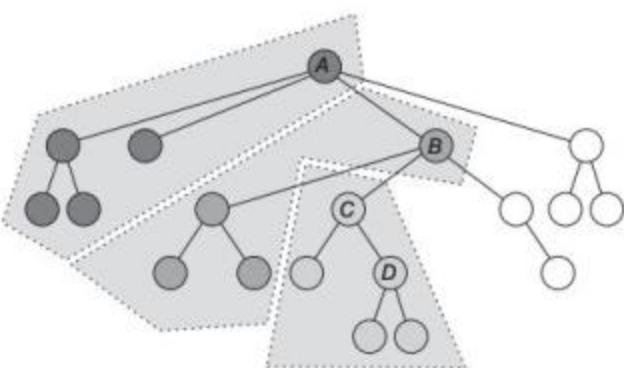
El pseudocódigo es compacto.

nosotros no vamos a hacerlo así. También vamos a asumir que tenemos una matriz de listas en las que almacenar las solicitudes NCA; es decir, la lista  $/$  almacena las solicitudes correspondientes al nodo  $/$  del árbol. Teniendo en cuenta estos detalles, el código es notablemente corto.

Cuando se visita un nodo  $u$  por primera vez, este se convierte en ancla de sí mismo, como en la línea 18 de la Figura 24.11. Después procesa recursivamente sus hijos  $v$  realizando la llamada de la línea 23. Después de volver de cada llamada recursiva, el subárbol se combina en la clase de equivalencia actual de  $u$  y nos cercioramos de que se actualice el ancla en las líneas 24 y 25. Cuando todos los hijos han sido procesados recursivamente, podemos marcar  $u$  como procesado en la línea 29 y finalizar comprobando todas las solicitudes NCA relativas a  $u$  en las líneas 30 a 33.<sup>2</sup>



**Figura 24.9** Los conjuntos inmediatamente antes de volver de la llamada recursiva a  $D$ ;  $D$  está marcado como visitado y  $\text{NCA}(D, v)$  es el ancla de  $v$  en el camino actual.



**Figura 24.10** Después de volver de la llamada recursiva de  $D$ , mezclamos el conjunto anclado por  $D$  en el conjunto anclado por  $C$  y luego calculamos todos los valores  $\text{NCA}(C, v)$  para los nodos  $v$  marcados anteriormente, con el fin de completar la llamada recursiva a  $C$ .

<sup>2</sup> Hablando estrictamente,  $u$  debería ser marcado en la última instrucción, pero marcarlo antes permite gestionar el molesto caso  $\text{NCA}(u, u)$ .

```

1 // Algoritmo de los ancestros comunes más próximos
2 //
3 // Precondiciones (y objetos globales):
4 // 1. La estructura union/find está inicializada
5 // 2. Todos los nodos inicialmente no están marcados
6 // 3. Los números de preorden ya están asignados en el campo num
7 // 4. Cada nodo puede almacenar su estado de marcado
8 // 5. Hay disponible globalmente una lista de pares
9
10 DisjSets s = new DisjSets( treeSize ); // union/find
11 Node [ ] anchor = new Node[ treeSize ]; // Nodo ancla de cada conjunto
12
13 // main hace la llamada NCA( root )
14 // después de las inicializaciones requeridas
15
16 void NCA( Node u )
17 {
18     anchor[ s.find( u.num ) ] = u;
19
20     // Realizar llamadas postorden
21     for( each child v of u )
22     {
23         NCA( v );
24         s.union( s.find( u.num ), s.find( v.num ) );
25         anchor[ s.find( u.num ) ] = u;
26     }
27
28     // Realizar cálculo nca para los pares donde aparezca u
29     u.marked = true;
30     for( each v such that NCA( u, v ) is required )
31         if( v.marked )
32             System.out.println( "NCA( " + u + ", " + v +
33                                 " ) es " + anchor[ s.find( v.num ) ] );
34 }

```

Figura 24.11 Pseudocódigo para el problema de los ancestros comunes más próximos.

## 24.3 El algoritmo rápido de búsqueda

En esta sección y en la Sección 24.4 vamos a establecer las bases para la eficiente implementación de la estructura de datos union/find. Existen dos estrategias básicas para resolver el problema union/find. El primer enfoque, el *algoritmo rápido de búsqueda*, garantiza que pueda ejecutarse la instrucción `find` en un tiempo constante de caso peor. El otro enfoque, el *algoritmo rápido de unión*, garantiza que pueda ejecutarse la operación `union` en un tiempo constante de caso peor. Se ha

demostrado que no pueden ejecutarse ambas operaciones simultáneamente en un tiempo constante de caso peor (ni siquiera amortizado).

Para que la operación `find` sea rápida, podríamos mantener en una matriz el nombre de la clase de equivalencia para cada elemento. Entonces `find` sería una simple búsqueda en tiempo constante. Suponga que queremos realizar `union(a, b)`. Suponga también que  $a$  se encuentra en la clase de equivalencia  $i$  y que  $b$  se encuentra en la clase de equivalencia  $j$ . Entonces podemos explorar la matriz, cambiando todos los  $i$  por  $j$ . Lamentablemente, esta exploración requiere un tiempo lineal. Por tanto, una secuencia de  $N - 1$  operaciones `union` (el máximo, porque entonces todos los elementos se encontrarán en un solo conjunto) requeriría un tiempo cuadrático. En el caso típico en el que el número de operaciones `find` es subcuadrático, este tiempo es claramente inaceptable.

Una posibilidad consiste en mantener en una lista enlazada todos los elementos que pertenezcan a una misma clase de equivalencia. Este enfoque nos permite ahorrar tiempo durante las actualizaciones, porque no tenemos que explorar la matriz completa. Aunque no garantiza, por sí mismo, la reducción del tiempo asintótico de ejecución, ya que sigue siendo posible realizar  $\Theta(N^2)$  actualizaciones de clases de equivalencia a lo largo de la ejecución del algoritmo.

Si también llevamos la cuenta del tamaño de las clases de equivalencia –y a la hora de realizar una operación `union` cambiamos el nombre de la clase más pequeña por el de la mayor– el tiempo total invertido para  $N$  operaciones `union` será  $O(N \log N)$ . La razón es que cada elemento podrá ver cambiar su clase de equivalencia como máximo  $\log N$  veces, porque cada vez que su clase cambia, su nueva clase de equivalencia es al menos dos veces mayor que su antigua clase (por lo que aplica el principio de la duplicación repetida).

Esta estrategia permite que cualquier secuencia de como máximo  $M$  operaciones `find` y  $N - 1$  operaciones `union` requiera como máximo un tiempo de  $O(M + N \log N)$ . Si  $M$  es lineal (o ligeramente no lineal), esta solución sigue siendo costosa. También es un poco complicada, porque tenemos que mantener listas enlazadas. En la Sección 24.4 examinamos una solución del problema `union/find` que hace que `union` sea sencilla pero que `find` resulte complicada –el algoritmo rápido de unión. Aun así, el tiempo de ejecución para cualquier secuencia de como máximo  $M$  operaciones `find` y  $N - 1$  operaciones `union` es solo ligeramente superior a  $O(M + N)$ . Además, solo se utiliza una única matriz de enteros.

## 24.4 El algoritmo rápido de unión

Recuerde que el problema `union/find` no requiere que la operación `find` devuelva ningún nombre específico; simplemente requiere que las búsquedas realizadas con `find` para dos elementos devuelvan la misma respuesta si y solo si ambos elementos se encuentran en el mismo conjunto. Una posibilidad podría ser utilizar un árbol para representar un conjunto, ya que todos los elementos de un árbol tienen la misma raíz y dicha raíz se puede usar para denominar el conjunto.

Cada conjunto está representado por un árbol (recuerde que a una colección de árboles se la denomina *bosque*). El nombre de cada conjunto está dado por el nodo situado en la correspondiente raíz. Nuestros árboles no son necesariamente árboles binarios, pero su representación resulta sencilla,

El argumento de que una clase de equivalencia solo puede cambiar como máximo  $\log N$  veces por cada elemento se utiliza también en el algoritmo rápido de `union`. El algoritmo rápido de búsqueda es un algoritmo sencillo, pero el algoritmo rápido de `union` es mejor.

Un árbol está representado por una matriz de enteros que representan a los nodos padre. El nombre del conjunto al que pertenece cada nodo de un árbol es la raíz de uno de los subárboles.

porque la única información que necesitamos es el padre. Por tanto, tan solo nos hace falta una matriz de enteros: cada entrada  $p[i]$  de la matriz representará el padre del elemento  $i$ , y podemos utilizar  $-1$  como padre para indicar una raíz. La Figura 24.12 muestra un bosque y la matriz empleada para representarlo.

Para llevar a cabo una operación `union` de dos conjuntos, mezclamos los dos árboles haciendo que la raíz de un árbol sea hijo de la raíz del otro. Esta operación requiere, claramente, un tiempo constante. Las Figuras 24.13 a 24.15 representan el bosque después de ejecutar cada una de las operaciones `union(4, 5)`, `union(6, 7)` y `union(4, 6)`, donde hemos adoptado el convenio de que la nueva raíz después de `union(x, y)` es  $x$ .

Una operación `find` para el elemento  $x$  se realiza devolviendo la raíz del árbol que contiene a  $x$ . El tiempo para realizar esta operación es proporcional al número de nodos contenidos en el camino que va desde  $x$  hasta la raíz. La estrategia para `union` esbozada anteriormente nos permite crear un árbol cuyos nodos estén todos en el camino hasta  $x$ , lo que nos da un tiempo de ejecución de caso peor de  $\Theta(N)$  por cada operación `find`. Típicamente (como se muestra en las aplicaciones anteriores), el tiempo de ejecución se calcula para una secuencia de  $M$  instrucciones entremezcladas. En el caso peor,  $M$  operaciones consecutivas podrían tardar un tiempo  $\Theta(MN)$ .

El tiempo cuadrático de ejecución para una secuencia de operaciones es generalmente inaceptable. Afortunadamente, hay varias maneras de garantizar fácilmente que este tiempo de ejecución no llegue a ser necesario.

La operación `union` es de tiempo constante.

El coste de una operación `find` depende de la profundidad del nodo al que se acceda y podría ser lineal.



Figura 24.12 Un bosque y sus ocho elementos, inicialmente situados en conjuntos diferentes.

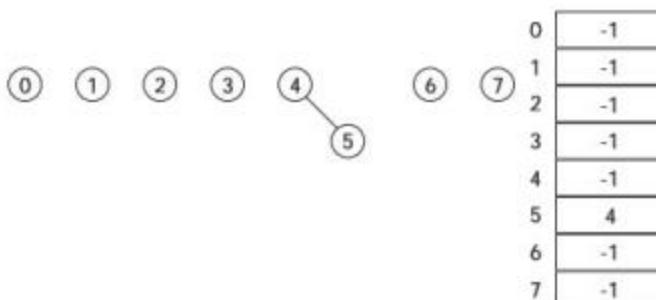
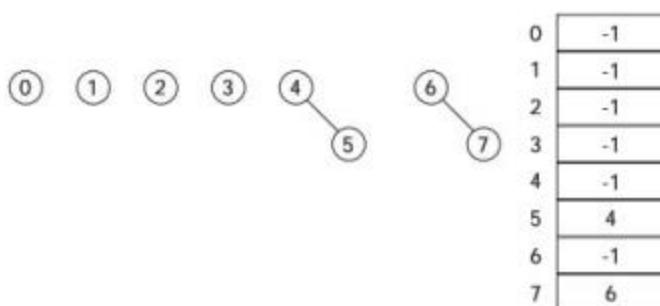
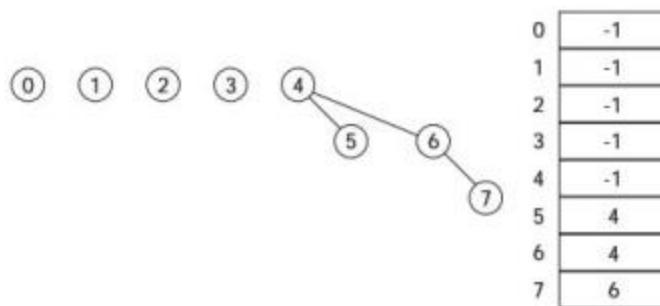


Figura 24.13 El bosque después de la operación `union` de los árboles con raíces 4 y 5.

Figura 24.14 El bosque después de la operación `union` de los árboles con raíces 6 y 7.Figura 24.15 El bosque después de la operación `union` de los árboles con raíces 4 y 6.

#### 24.4.1 Algoritmos inteligentes de unión

Hemos realizado las operaciones `union` anteriores de manera bastante arbitraria haciendo que el segundo árbol pasase a ser un subárbol del primero. Una mejora simple consiste en hacer siempre que el árbol más pequeño pase a ser un subárbol del otro más grande, rompiendo los empates por cualquier método. A esta técnica se la denomina *unión por tamaño*. Las tres operaciones `union` anteriores representaban todas ellas un empate, así que podemos considerar que se llevaron a cabo por tamaño. Si la siguiente operación fuera `union(3, 4)`, se formaría el bosque mostrado en la Figura 24.16. Si no hubiéramos empleado el heuristicó de tamaño, el resultado hubiera sido un bosque más profundo (hubiera habido tres nodos en lugar de uno, a un nivel adicional de profundidad).

Si la operación `union` se realiza por tamaño, la profundidad de cualquier nodo no será nunca mayor que  $\log N$ . Cada nodo se encuentra inicialmente a profundidad 0 y cuando su profundidad se incrementa como resultado de una operación `union`, se coloca en un árbol que será al menos el doble de grande que antes. Por tanto, su profundidad se puede incrementar como máximo  $\log N$  veces. (Hemos utilizado este argumento también para el algoritmo rápido de búsqueda en la Sección 24.3.) Este resultado implica que el tiempo de ejecución para una operación `find` es  $O(\log N)$  y que una secuencia de  $M$  operaciones tarda como máximo un tiempo de  $O(M \log N)$ . El árbol mostrado en la Figura 24.17 ilustra cuál es el peor árbol posible después de 15 operaciones `union` y se obtiene si todas las uniones se realizan entre árboles de igual tamaño (el árbol de caso peor se

La *unión por tamaño* garantiza la obtención de búsquedas logarítmicas.

denomina *árbol binomial*. Los árboles binomiales tienen otras aplicaciones en estructuras de datos avanzadas.)

Para implementar esta estrategia, tenemos que llevar la cuenta de cuál es el tamaño de cada árbol. Puesto que estamos utilizando simplemente una matriz, podemos hacer que la entrada de la matriz correspondiente a la raíz contenga el *negado* del tamaño del árbol, como se muestra en la Figura 24.16. Así, la representación inicial del árbol tendrá todos los valores iguales a  $-1$ .

Cuando se realiza una operación `union`, comprobamos los tamaños; el nuevo tamaño será igual a la suma de los antiguos. Por tanto, la unión por tamaño no es difícil de implementar y no requiere espacio adicional. También es rápida como promedio porque, cuando se realizan operaciones `union` aleatorias, generalmente se mezclan conjuntos muy pequeños (normalmente de un único elemento) con grandes conjuntos a todo lo largo del algoritmo. El análisis matemático de este proceso es muy complejo; las referencias incluidas al final de capítulo proporcionan la literatura en la que podrá encontrar más información.

Una implementación alternativa que también garantiza una profundidad logarítmica es la *unión por altura*, en la que llevamos la cuenta de la altura de los árboles en lugar de su tamaño y realizamos las operaciones `union` haciendo que los árboles menos profundos pasen a ser subárboles de los árboles más profundos. Este algoritmo es fácil de escribir y de utilizar, porque la altura de un árbol solo se incrementa cuando se unen dos árboles de la misma profundidad (en cuyo caso la altura se incrementa en 1). Por tanto, la unión por altura constituye una modificación trivial de la

En lugar de almacenar  
-1 para las raíces, se  
almacena el negado del  
tamaño.

La unión por altura también  
garantiza la obtención  
de operaciones `find`  
logarítmicas.

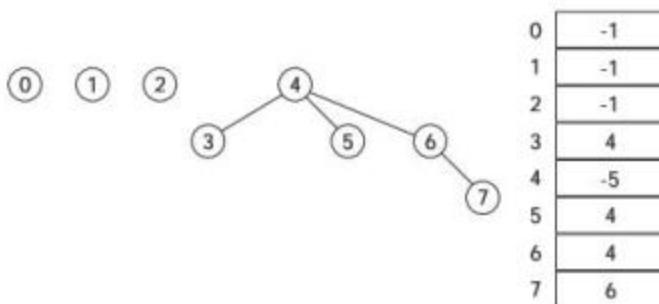


Figura 24.16 El bosque formado mediante una unión por tamaño, en el que los tamaños están codificados mediante números negativos.

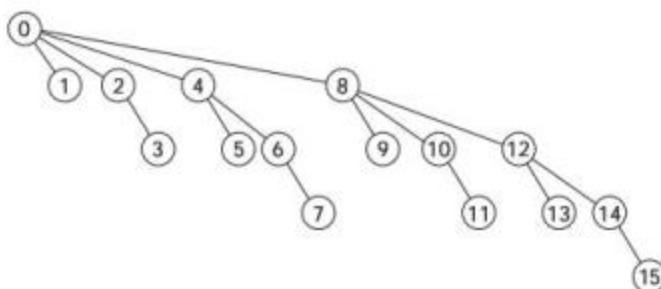


Figura 24.17 Árbol de caso peor para  $N = 16$ .

unión por tamaño. Como las alturas comienzan en 0, almacenamos el negado del número de nodos que componen el camino más profundo, en lugar de la altura, como se muestra en la Figura 24.18.

#### 24.4.2 Compresión de caminos

El algoritmo union/find, tal como lo hemos descrito hasta ahora, es bastante aceptable para la mayoría de los casos. Es muy simple y lineal como promedio para una secuencia de  $M$  instrucciones. Sin embargo, el caso peor sigue siendo bastante poco convincente. La razón es que una secuencia de operaciones `union` que se produzca en alguna aplicación concreta (como por ejemplo en el problema NCA) no es obviamente aleatoria (de hecho, para ciertos árboles, dista bastante de ser aleatoria). Por tanto, tenemos que buscar una cota mejor para el caso peor de una secuencia de  $M$  operaciones. Aparentemente, no se pueden realizar más mejoras en el algoritmo de unión, porque podemos llegar al caso peor cuando se mezclan árboles idénticos. La única forma de acelerar entonces el algoritmo, sin volver a diseñar por completo la estructura de datos, consiste en hacer alguna cosa inteligente con la operación `find`.

**La compresión de caminos**  
hace que todo nodo al que  
se acceda sea hijo de la  
raíz hasta que se produzca  
otra operación `union`.

Esa cosa inteligente que podemos hacer es la *compresión de caminos*. Claramente, después de realizar una operación `find` para  $x$ , tendría sentido cambiar el padre de  $x$  y hacer que fuera la raíz. De esta forma, una segunda operación `find` para  $x$  o para cualquier elemento del subárbol de  $x$  resultará más sencilla. Sin embargo, no tenemos ninguna necesidad de detenernos ahí. También podríamos cambiar los padres de todos los nodos situados en el camino de acceso. En la técnica de compresión de caminos se cambia el padre de todos los nodos situados en el camino que va desde la raíz hasta  $x$ , haciendo que el nuevo padre sea la raíz. La Figura 24.19 muestra el efecto de la compresión de caminos después de ejecutar `find(14)` en el árbol genérico de caso peor mostrado en la Figura 24.17. Con dos cambios adicionales de parentesco, los nodos 12 y 13 estarán ahora una posición más cerca de la raíz y los nodos 14 y 15 estarán ahora dos posiciones más cerca. La futura rapidez adicional en el acceso a esos nodos compensará (esperamos) el trabajo adicional requerido para realizar la compresión de caminos. Observe que las operaciones `union` subsiguientes empujarán los nodos a mayores profundidades.

Cuando las operaciones `union` se realizan arbitrariamente, la compresión de caminos es una buena idea, debido a la abundancia de nodos profundos; la compresión de caminos permite acercarnos a la raíz. Se ha demostrado que, cuando se efectúa la compresión de caminos en este caso,

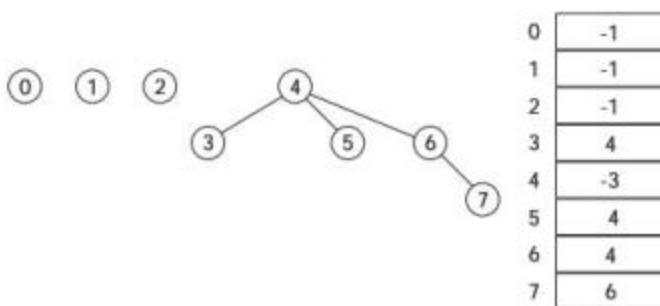


Figura 24.18 Un bosque formado mediante unión por altura, en el que la altura se codifica mediante un número negativo.

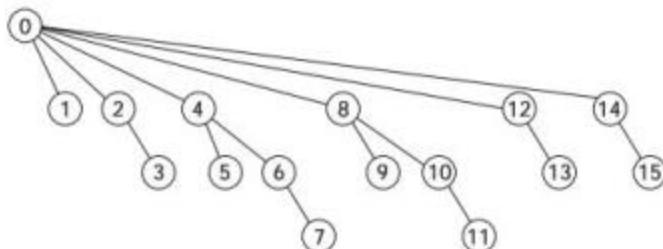


Figura 24.19 Compresión de caminos resultante de ejecutar una operación `find(14)` para el árbol mostrado en la Figura 24.17.

una secuencia de  $M$  operaciones requiere como máximo un tiempo  $O(M \log N)$ , por lo que la compresión de caminos garantiza por sí misma un coste amortizado logarítmico para la operación `find`.

La compresión de caminos es perfectamente compatible con la unión por tamaño. Por tanto, podemos implementar ambas rutinas al mismo tiempo. Sin embargo, la compresión de caminos no es del todo compatible con la unión por alturas, porque la compresión puede hacer variar las alturas de los árboles. No sabemos cómo recalcular esas alturas de manera eficiente, así que no intentamos hacerlo. En consecuencia, las alturas almacenadas para cada árbol pasan a ser alturas estimadas, denominadas *rangos*, lo cual no constituye un problema. El algoritmo resultante, *unión por rango*, se obtiene por tanto de combinar la unión por alturas y la compresión de caminos. Como veremos en la Sección 24.6, la combinación de una regla inteligente de unión y del mecanismo de compresión de caminos nos da una cota casi lineal para el tiempo de ejecución correspondiente a una secuencia de  $M$  operaciones.

La compresión de caminos garantiza un coste amortizado logarítmico para la operación `find`.

La compresión de caminos y una regla de unión inteligente garantizan un coste amortizado esencialmente constante por operación. (Es decir, una secuencia larga se puede ejecutar en un tiempo casi lineal.)

## 24.5 Implementación Java

En la Figura 24.20 se proporciona el esqueleto de la clase para conjuntos disjuntos, completándose la implementación en la Figura 24.21. El algoritmo completo es sorprendentemente corto.

Los conjuntos disjuntos son relativamente fáciles de implementar.

```

1 package weiss.nonstandard;
2
3 // Clase DisjointSets
4 //
5 // CONSTRUCCIÓN: con int que representa el número inicial de conjuntos
6 //
7 // *****OPERACIONES PÚBLICAS*****
8 // void union( root1, root2 ) --> Mezclar dos conjuntos
9 // int find( x )           --> Devolver conjunto que contiene x
10 // *****ERRORES*****
11 // Se realiza comprobación de errores de los parámetros
  
```

Continúa

Figura 24.20 El esqueleto de la clase para conjuntos disjuntos.

```

12
13 public class DisjointSets
14 {
15     public DisjointSets( int numElements )
16         { /* Figura 24.21 */ }
17
18     public void union( int root1, int root2 )
19         { /* Figura 24.21 */ }
20
21     public int find( int x )
22         { /* Figura 24.21 */ }
23
24     private int [ ] s;
25
26
27     private void assertIsRoot( int root )
28     {
29         assertIsItem( root );
30         if( s[ root ] >= 0 )
31             throw new IllegalArgumentException();
32     }
33
34     private void assertIsItem( int x )
35     {
36         if( x < 0 || x >= s.length )
37             throw new IllegalArgumentException();
38     }
39 }

```

Figura 24.20 (Continuación).

En nuestra rutina, `union` se realiza en las raíces de los árboles. En ocasiones, la operación se implementa pasando cualesquiera dos elementos y haciendo que `union` ejecute la operación `find` para determinar las raíces.

El procedimiento interesante es `find`. Después de haber llevado a cabo la operación `find` recursivamente, se almacena en `array[x]` la raíz y luego se devuelve ese valor. Puesto que este procedimiento es recursivo, todas las entradas correspondientes a los nodos que forman el camino se configurarán con el valor correspondiente a la raíz.

```

1    /**
2     * Construir el objeto de conjuntos disjuntos.
3     * @param numElements el número inicial de conjuntos disjuntos.
4     */
5    public DisjointSets( int numElements )

```

Continúa

Figura 24.21 Implementación de una clase para conjuntos disjuntos.

```

6  {
7      s = new int[ numElements ];
8      for( int i = 0; i < s.length; i++ )
9          s[ i ] = -1;
10 }
11
12 /**
13 * Unir dos conjuntos disjuntos mediante la heurística de altura.
14 * root1 y root2 son diferentes y representan nombres de conjuntos.
15 * @param root1 la raíz del conjunto 1.
16 * @param root2 la raíz del conjunto 2.
17 * @throws IllegalArgumentException si root1 o root2
18 * son iguales.
19 */
20 public void union( int root1, int root2 )
21 {
22     assertIsRoot( root1 );
23     assertIsRoot( root2 );
24     if( root1 == root2 )
25         throw new IllegalArgumentException();
26
27     if( s[ root2 ] < s[ root1 ] ) // root2 es más profunda
28         s[ root1 ] = root2;        // Hacer de root2 la nueva raíz
29     else
30     {
31         if( s[ root1 ] == s[ root2 ] )
32             s[ root1 ]--;           // Actualizar altura si son iguales
33         s[ root2 ] = root1;       // Hacer de root1 la nueva rafz
34     }
35 }
36
37 /**
38 * Realizar una búsqueda con compresión de camino.
39 * @param x el elemento que se está buscando.
40 * @return el conjunto que contiene x.
41 * @throws IllegalArgumentException si x no es válido.
42 */
43 public int find( int x )
44 {
45     assertIsItem( x );
46     if( s[ x ] < 0 )
47         return x;
48     else
49         return s[ x ] = find( s[ x ] );
50 }

```

Figura 24.21 (Continuación).

## 24.6 Caso peor para la unión por rango con compresión de caminos

Cuando se utilizan ambos heurísticos, el algoritmo es casi lineal en el caso peor. Específicamente, el tiempo requerido para procesar una secuencia de como máximo  $N - 1$  operaciones `union` y  $M$  operaciones `find` es, en caso peor, de  $\Theta(M\alpha(M, N))$  (supuesto que  $M \geq N$ ), donde  $\alpha(M, N)$  es una inversa funcional de la *función de Ackermann*, que crece muy rápidamente y está definida de la forma siguiente:<sup>3</sup>

$$\begin{aligned} A(1, j) &= 2^j & j \geq 1 \\ A(i, 1) &= A(i-1, 2) & i \geq 2 \\ A(i, j) &= A(i-1, A(i, j-1)) & i, j \geq 2 \end{aligned}$$

A partir de esto, definimos

$$\alpha(M, N) = \min \left\{ i \geq 1 \mid \left( A\left(i, \lfloor M/N \rfloor\right) > \log N \right) \right\}$$

*La función de Ackermann crece muy rápidamente y su inversa tiene, esencialmente, un valor que como máximo es 4.*

Puede probar a calcular algunos valores, pero para todos los efectos prácticos,  $\alpha(M, N) \leq 4$ , que es lo único que realmente nos importa aquí. Por ejemplo, para cualquier  $j > 1$ , tenemos

$$\begin{aligned} A(2, j) &= A(1, A(2, j-1)) \\ &= 2^{A(2, j-1)} \\ &= 2^{2^{2^{\dots}}} \end{aligned}$$

donde el número de doses (2) en el exponente es  $j$ . La función  $F(N) = A(2, N)$  se denomina comúnmente *función de Ackermann de una sola variable*. La inversa de una sola variable de la función de Ackermann, que en ocasiones se escribe como  $\log^* N$ , es el número de veces que hay que aplicar el logaritmo de  $N$  para que  $N \leq 1$ . Así,  $\log^* 65536 = 4$ , porque  $\log \log \log \log 65536 = 1$  y  $\log^* 2^{65536} = 5$ . Sin embargo, recuerde que  $2^{65536}$  tiene más de 20.000 dígitos. La función  $\alpha(M, N)$  crece aun más lentamente que  $\log^* N$ . Por ejemplo,  $A(3, 1) = A(2, 2) = 2^{2^2} = 16$ . Por tanto, para  $N < 2^{16}$ ,  $\alpha(M, N) \leq 3$ . Además, puesto que  $A(4, 1) = A(3, 2) = A(2, A(3, 1)) = A(2, 16)$ , que es 2 elevado a una potencia de 16 doses (2) apilados, en la práctica resultará que  $\alpha(M, N) \leq 4$ . Sin embargo,  $\alpha(M, N)$  no es una constante cuando  $M$  es ligeramente superior a  $N$ , por lo que el tiempo de ejecución no es lineal.<sup>4</sup>

<sup>3</sup> La función de Ackermann se define frecuentemente como  $A(1, j) = j + 1$  para  $j \geq 1$ . La forma que utilizamos en este texto crece más rápidamente, por tanto la inversa crece de manera más lenta.

<sup>4</sup> Sin embargo, observe que si  $M = N \log^* N$ , entonces  $\alpha(M, N)$  es como máximo 2. Por tanto, siempre que  $M$  sea ligeramente más que lineal, el tiempo de ejecución será lineal con respecto a  $M$ .

En el resto de esta sección vamos a demostrar un resultado ligeramente más débil. Demostraremos que cualquier secuencia de  $M = \Omega(N)$  operaciones `union` y `find` requiere un tiempo total de  $O(M \log^* N)$ . Esta misma cota es válida si sustituimos la unión por rango por la unión por tamaño. Este análisis es probablemente el más complejo de este libro y es uno de los primeros análisis realmente complejos que jamás se hayan realizado para un algoritmo cuya implementación es, esencialmente, trivial. Ampliando esta técnica podemos demostrar la cota más fuerte que hemos enunciado anteriormente.

### 24.6.1 Análisis del algoritmo `union/find`

En esta sección vamos a establecer una cota bastante estricta para el tiempo de ejecución de una secuencia de  $M = \Omega(N)$  operaciones `union` y `find`. Las operaciones `union` y `find` se pueden ejecutar en cualquier orden, pero `union` se realiza por rango y `find` se realiza con compresión de caminos.

Comenzamos con algunos teoremas relativos al número de nodos de rango  $r$ . Intuitivamente, debido a la regla de unión por rango, hay muchos más nodos de rango pequeño que de rango grande. En particular, puede haber como máximo un nodo de rango  $\log N$ . Lo que queremos hacer es obtener una cota tan precisa como sea posible para el número de nodos con un rango concreto  $r$  cualquiera. Puesto que los rangos solo varían cuando se realizan operaciones `union` (y en ese caso únicamente cuando los dos árboles tienen el mismo rango), podemos determinar esta cota ignorando la compresión de caminos. Vamos a hacerlo en el Teorema 24.1.

<b>Teorema 24.1</b>	En ausencia de compresión de caminos, cuando se está ejecutando una secuencia de instrucciones <code>union</code> , un nodo de rango $r$ debe tener $2^r$ descendientes (incluido él mismo).
Demostración	<p>La demostración se hace por inducción. La base <math>r = 0</math> es claramente cierta. Sea <math>T</math> el árbol de rango <math>r</math> con el menor número de descendientes y sea <math>x</math> la raíz de <math>T</math>. Suponga que la última operación <code>union</code> en la que <math>x</math> estuvo implicado se realizó entre <math>T_1</math> y <math>T_2</math>. Suponga que la raíz de <math>T_1</math> fuera <math>x</math>. Si <math>T_1</math> tuviera rango <math>r</math>, entonces <math>T_1</math> sería un árbol con rango <math>r</math> con menos descendientes que <math>T</math>. Esta condición contradice la suposición de que <math>T</math> es el árbol con el menor número de descendientes. Por tanto, el rango de <math>T_1</math> es como máximo <math>r - 1</math>. El rango de <math>T_2</math> es como máximo igual al rango de <math>T_1</math>, debido a la unión por rango. Como <math>T</math> tiene rango <math>r</math> y el rango solo podría incrementarse debido a <math>T_2</math>, se deduce que el rango de <math>T_2</math> es <math>r - 1</math>. Entonces el rango de <math>T_1</math> es también <math>r - 1</math>. Por la hipótesis de inducción, cada árbol tiene al menos <math>2^{r-1}</math> descendientes, lo que nos da un total de <math>2^r</math> y permite demostrar el teorema.</p>

El Teorema 24.1 establece que si no se realiza compresión de caminos, cualquier nodo de rango  $r$  debe tener al menos  $2^r$  descendientes. La compresión de caminos puede por supuesto variar esta condición, ya que puede eliminar descendientes de un nodo. Sin embargo, cuando se realizan operaciones `union` –incluso con compresión de caminos– estamos utilizando rangos, o alturas estimadas. Dichos rangos se comportan como si no hubiera compresión de caminos. Por tanto, cuando estamos intentando acotar el número de nodos de rango  $r$ , podemos ignorar la compresión de caminos, como en el Teorema 24.2.

**Teorema 24.2**

El número de nodos de rango  $r$  es como máximo  $N^{2^r}$ .

**Demostración**

Sin compresión de caminos, cada nodo de rango  $r$  es la raíz de un subárbol de al menos  $2^r$  nodos. Ningún otro nodo del subárbol puede tener rango  $r$ . Por tanto, todos los subárboles de nodos de rango  $r$  son disjuntos. En consecuencia, existen como máximo  $N^{2^r}$  subárboles disjuntos y, por tanto,  $N^{2^r}$  nodos de rango  $r$ .

El Teorema 24.3 establece algo que en cierto modo es obvio, pero que resulta crucial para el análisis.

**Teorema 24.3**

En cualquier punto del algoritmo union/find, los rangos de los nodos que forman un camino que va desde una hoja hasta la raíz son monótonamente crecientes.

**Demostración**

El teorema es obvio si no existe compresión de caminos. Si después de la compresión de caminos, algún nodo  $v$  es descendiente de  $w$ , entonces claramente  $v$  debe haber sido descendiente de  $w$  si solo se toman en consideración las operaciones union. Por tanto, el rango de  $v$  es estrictamente menor que el rango de  $w$ .

No existen demasiados nodos de rango grande, y los rangos se incrementan a lo largo de cualquier camino ascendente hacia la raíz.

A continuación proporcionamos un resumen de los resultados preliminares. El Teorema 24.2 describe el número de nodos a los que se les puede asignar el rango  $r$ . Puesto que los rangos se asignan únicamente mediante operaciones union, que no dependen de la compresión de caminos, el Teorema 24.2 es válido en cualquier etapa del algoritmo union/find –incluso en mitad de una compresión de caminos. El Teorema 24.2 es estricto en el sentido de que puede haber  $N^{2^r}$  nodos para cualquier rango  $r$ . También es ligeramente débil,

porque la cota puede que no se cumpla para todos los rangos  $r$  simultáneamente. Mientras que el Teorema 24.2 describe el número de nodos de un rango  $r$ , el Teorema 24.3 indica la distribución de los nodos dentro de un rango  $r$ . Como cabía esperar, el rango de los nodos se incrementa de manera estricta a lo largo del camino que va desde una hoja hasta la raíz.

Ahora estamos listos para demostrar el teorema principal, y nuestro plan básico es el siguiente. Una operación find sobre cualquier nodo  $v$  cuesta un tiempo que es proporcional al número de nodos existentes en el camino que va desde  $v$  hasta la raíz. Cargamos 1 unidad de coste por cada nodo comprendido en el camino que va desde  $v$  hasta la raíz durante cada operación find. Para ayudarnos a contar el coste, depositamos una moneda imaginaria en cada nodo del camino. Esto es estrictamente un truco de contabilidad que no forma parte del programa.

En cierto modo es equivalente al uso de una función potencial en el análisis amortizado para árboles splay y montículos sesgados. Cuando el algoritmo finalice, recopilaremos todas las monedas que hayamos ido depositando para determinar el tiempo total.

Como truco de contabilidad adicional vamos a depositar tanto dólares como euros. Vamos a demostrar que, durante la ejecución del algoritmo, solo podemos depositar un cierto número de dólares durante cada operación find (independientemente de cuántos nodos haya). También demostraremos que podemos depositar únicamente un cierto número de euros en cada

Utilizamos monedas como si fueran una función potencial. El número total de monedas será el tiempo total.

Tenemos tanto dólares como euros. Los euros representan las primeras veces que se comprime un nodo, los dólares representan las posteriores compresiones o no compresiones.

nodo (independientemente de cuántas operaciones `find` se realicen). Sumando estos dos totales, obtendremos una cota para el número total de monedas que se pueden depositar.

Ahora esbozaremos nuestro esquema de contabilización de manera más detallada. Comenzamos dividiendo los nodos según sus rangos. Después dividimos los rangos en grupos de rangos. En cada `find`, depositamos algunas monedas de dólar en un bote general y algunos euros en nodos específicos. Para calcular el número total de euros depositados, calculamos los depósitos por nodo. Sumando todos los depósitos para cada nodo de rango  $r$ , obtenemos el número total de depósitos por rango  $r$ . Después sumamos todos los depósitos para cada rango  $r$  del grupo  $g$  y obtendremos así el total de depósitos para cada grupo de rangos  $g$ . Finalmente, sumamos todos los depósitos para cada grupo de rangos  $g$ , con el fin de obtener el número total de euros depositados en el bosque. Sumando ese total al número de dólares existentes en el bote general obtendremos la respuesta.

Como hemos mencionado anteriormente, particionamos los rangos en grupos. El rango  $r$  irá al grupo  $G(r)$  y  $G$  se determinará posteriormente (para equilibrar los depósitos de dólares y euros). El rango más alto de cada grupo de rangos  $g$  es  $F(g)$ , donde  $F = G^{-1}$  es la inversa de  $G$ . El número de rangos en cualquier grupo de rangos,  $g > 0$ , será por tanto  $F(g) - F(g - 1)$ . Claramente,  $G(N)$  es una cota superior muy poco estricta para el grupo de rangos mayor. Suponga que particionamos los rangos como se muestra en la Figura 24.22. En este caso,  $G(r) = \lfloor \sqrt{r} \rfloor$ . El rango máximo dentro del grupo  $g$  será  $F(g) = g^2$ . Asimismo, observe que el grupo  $g > 0$  contiene los rangos comprendidos entre  $F(g - 1) + 1$  y  $F(g)$ . Esta fórmula no se aplica al grupo de rangos 0, así que por comodidad nos cercioraremos de que el grupo de rangos 0 contenga únicamente los elementos de rango 0. Observe que los grupos comprenden rangos consecutivos.

Los rangos se partitionan en grupos. La composición real de los grupos se determinará al final de la demostración. El grupo 0 contiene únicamente el rango 0.

Como hemos mencionado anteriormente en el capítulo, cada instrucción `union` requiere un tiempo constante, siempre y cuando cada raíz lleve la cuenta de cuál es su rango. Por tanto, las operaciones `union` son esencialmente gratuitas en lo que a esta demostración concierne.

Cada operación `find` requiere un tiempo proporcional al número de nodos comprendidos en el camino que va de la raíz hasta el nodo que representa al elemento accedido  $i$ . Depositamos por tanto una moneda por cada vértice del camino. Sin embargo, si nos limitamos a hacer esto, no podremos esperar mucho de una cota, porque no estaremos aprovechando el mecanismo de compresión de caminos. Por tanto, debemos emplear algún hecho relativo a la compresión de caminos en nuestro análisis. La observación clave es que, como resultado de la compresión de caminos, un nodo obtiene un nuevo padre y está garantizado que el nuevo padre tenga un rango más alto que el padre anterior.

Cuando se comprime un nodo, su nuevo padre tendrá un rango más alto que su nodo anterior.

Para incorporar este hecho en la demostración, empleamos la siguiente técnica inteligente de contabilidad: para cada nodo  $v$  del camino que va desde el nodo  $i$  al que se ha accedido hasta la raíz, depositamos una moneda en una de dos cuentas.

Reglas para los depósitos de dólares y euros.

- Si  $v$  es la raíz o el padre de  $v$  es la raíz, o si el padre de  $v$  pertenece a un grupo de rangos distinto del grupo al que pertenece  $v$ , entonces cargamos 1 unidad según esta regla y depositamos un dólar en el bote general.
- En caso contrario, depositamos un euro en el nodo.

Grupo	Rango
0	0
1	1
2	2, 3, 4
3	5 a 9
4	10 a 16
$i$	$(i - 1)^2$ a $i^2$

Figura 24.22 Posible particionamiento de los rangos en grupos.

El Teorema 24.4 establece que esta contabilización es precisa.

<b>Teorema 24.4</b>	Para cualquier operación <code>find</code> , el número total de monedas depositadas en el bote general o en un nodo es exactamente igual al número de nodos a los que se accede durante la operación <code>find</code> .
Demostración	Obvia.

Los depósitos de dólares están limitados por el número de grupos distintos existentes. Los depósitos de euros están limitados por el tamaño de los grupos. Al final, necesitamos equilibrar ambos costes.

Por tanto, solo necesitamos sumar todos los dólares depositados según la regla 1 y todos los euros depositados según la regla 2. Antes de continuar con la demostración, vamos a esbozar las ideas. Se depositan euros en un nodo cuando este se comprime y su padre se encuentra en el mismo grupo de rangos que ese nodo. Puesto que el nodo pasa a tener un parente de mayor rango después de cada compresión de caminos y puesto que el tamaño de un grupo de rangos es finito, el nodo terminará por obtener un parente que no se encuentre en su grupo de rangos. En consecuencia, por un lado, solo podrá colocarse un número limitado de euros en cada nodo. Este número es aproximadamente igual al tamaño del grupo de rangos al que pertenece el nodo. Por otro lado, los depósitos de dólares también están limitados, esencialmente por el número de grupos de rangos. Por tanto, queremos seleccionar tanto grupos de rangos pequeños (para limitar la cantidad de euros depositada) como un número pequeño de grupos de rangos (para limitar los depósitos de dólares). Ahora estamos listos para llenar los detalles con una rápida serie de teoremas, los Teoremas 24.5 a 24.10.

<b>Teorema 24.5</b>	A lo largo de todo el algoritmo, el número total de dólares depositados según la regla 1 asciende a $M(G(N) + 2)$ .
Demostración	Para toda operación <code>find</code> , se depositan como máximo dos dólares debido a la raíz y a su hijo. Según el Teorema 24.3, los vértices que forman el camino ascendente tienen un rango que se incrementa monótonamente y por tanto el grupo de rangos nunca disminuye a medida que vamos ascendiendo por el camino. Puesto que existen como máximo $G(N)$ grupos de rangos (aparte del grupo 0), solo un número $G(N)$ de otros vértices podrá dar lugar a un depósito según la regla 1 para cualquier operación <code>find</code> concreta. Por tanto, durante cualquier operación <code>find</code> , como máximo se podrán depositar en el bote general $G(N) + 2$ dólares. En consecuencia, para una secuencia de $M$ operaciones <code>find</code> se podrán depositar según la regla 1 un máximo de $M(G(N) + 2)$ dólares.

<b>Teorema 24.6</b>	Para todo nodo del grupo de rangos $g$ , el número total de euros depositados es como máximo $F(g)$ .
---------------------	---

<b>Demostración</b>	Si se deposita un euro en un vértice $v$ según la regla 2, $v$ será desplazado por el mecanismo de compresión de caminos y obtendrá un nuevo padre de rango superior a su padre anterior. Puesto que el rango mayor de su grupo es $F(g)$ , tenemos garantizado que después de depositar $F(g)$ monedas, el padre de $v$ ya no pertenecerá al grupo de rangos al que pertenece $v$ .
---------------------	--

La cota del Teorema 24.6 se puede mejorar utilizando únicamente el tamaño del grupo de rangos, en lugar de su miembro de mayor valor. Sin embargo, esta modificación no mejora la cota obtenida para el algoritmo union/find.

<b>Teorema 24.7</b>	El número de nodos, $N(g)$ , en el grupo de rangos $g \geq 0$ es como máximo $N/2^{F(g-1)}$ .
---------------------	---

<b>Demostración</b>	Por el Teorema 24.2, hay como máximo $N/2^r$ nodos de rango $r$ . Sumando para todos los grupos del grupo $g$ , obtenemos
---------------------	---

$$\begin{aligned} N(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{N}{2^r} \\ &\leq \sum_{r=F(g-1)+1}^{\infty} \frac{N}{2^r} \\ &\leq N \sum_{r=F(g-1)+1}^{\infty} \frac{1}{2^r} \\ &\leq \frac{N}{2^{F(g-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\ &\leq \frac{2N}{2^{F(g-1)+1}} \\ &\leq \frac{N}{2^{F(g-1)}} \end{aligned}$$

<b>Teorema 24.8</b>	El número máximo de euros depositado en todos los vértices del grupo de rangos $g$ es como máximo $NF(g)/2^{F(g-1)}$ .
---------------------	--

<b>Demostración</b>	El resultado se obtiene mediante un simple multiplicación de las cantidades obtenidas en los Teoremas 24.6 y 24.7.
---------------------	--

<b>Teorema 24.9</b>	El número total de monedas depositadas según la regla 2 es como máximo $N \sum_{g=1}^{G(N)} F(g) / 2^{F(g-1)}$ euros.
---------------------	---

<b>Demostración</b>	Puesto que el grupo de rangos 0 contiene únicamente elementos de rango 0, no puede contribuir a la cantidad de monedas depositadas según la regla 2 (no puede tener un padre del mismo grupo de rangos). La cota se obtiene sumando los otros grupos de rangos.
---------------------	---

Con esto tenemos los depósitos efectuados según las reglas 1 y 2. El total es

$$M(G(N)+2) + N \sum_{g=1}^{G(N)} \frac{F(g)}{2^{F(g-1)}} \quad (24.1)$$

Ahora podemos especificar los grupos de rangos para minimizar la cota. Nuestra elección no es del todo mínima, pero se aproxima bastante.

Todavía no hemos especificado  $G(N)$  o su inversa  $F(N)$ . Obviamente, somos libres de elegir prácticamente cualquier cosa que queramos, pero tiene bastante sentido seleccionar  $G(N)$  de modo que se minimice la cota expresada en la Ecuación 24.1. Sin embargo, si  $G(N)$  es demasiado pequeña,  $F(N)$  será grande, afectando negativamente a la cota. Una elección aparentemente buena es hacer que  $F(j)$  sea la función recursivamente definida mediante  $F(0)$  y  $F(j) = 2^{F(j-1)}$ , que nos da  $G(N) = 1 + \lfloor \log^* N \rfloor$ .

La Figura 24.23 muestra cómo quedan particionados los rangos con esta elección. Observe que el grupo 0 contiene únicamente el rango 0, lo que era requerido para la demostración del Teorema 24.9. Observe también que  $F$  es muy similar a la función de Ackermann de una sola variable, difiriendo solo en la definición del caso base. Con esta elección de  $F$  y  $G$ , podemos completar el análisis en el Teorema 24.10.

#### Teorema 24.10

El tiempo de ejecución del algoritmo union/find con  $M = \Omega(N)$  operaciones find es  $O(M \log^* N)$ .

#### Demostración

Inserte las definiciones de  $F$  y  $G$  en la Ecuación 24.1. El número total de dólares es  $O(MG(N)) = O(M \log^* N)$ . Puesto que  $F(g) = 2^{F(g-1)}$ , el número total de euros es  $NG(N) = O(N \log^* N)$ , y como  $M = \Omega(N)$ , se obtiene la cota indicada.

Observe que tenemos más dólares que euros. La función  $\alpha(M, N)$  equilibra las cosas, lo cual es la razón de que proporcione una mejor cota.

Grupo	Rango
0	0
1	1
2	2
3	3, 4
4	5 a 6
5	17 a 65.536
6	65.537 a $2^{65.536}$
7	Rangos verdaderamente grandes

Figura 24.23 Particionamiento real de los rangos en grupos, utilizado en la demostración.

## Resumen

En este capítulo hemos analizado una estructura de datos simple para el mantenimiento de conjuntos disjuntos. Cuando se realiza la operación `union`, no importa, en lo que a la corrección respecta, qué conjunto retenga su nombre. Una valiosa lección que habría que aprender aquí es que puede ser muy importante tomar en consideración todas las alternativas existentes, cuando un paso no esté completamente especificado. El paso `union` es flexible. Aprovechando esta flexibilidad, podemos obtener un algoritmo mucho más eficiente.

La compresión de caminos es una de las primeras formas de autoajuste, que es una técnica que ya hemos utilizado en otros lugares (árboles splay y montículos sesgados). Su uso aquí es extremadamente interesante desde un punto de vista teórico, porque fue uno de los primeros ejemplos de un algoritmo simple con un análisis de caso peor no tan simple.



## Conceptos clave

**algoritmo de Kruskal** Un algoritmo utilizado para seleccionar aristas en orden creciente de coste y que añade una arista al árbol si el añadirla no crea un ciclo. (884)

**algoritmo en línea** Un algoritmo en el que hay que proporcionar una respuesta para cada solicitud antes de conocer la solicitud siguiente. (881)

**algoritmo fuera de línea** Un algoritmo en el que se conoce la secuencia completa de consultas antes de tener que proporcionar la primera respuesta. (881)

**algoritmo rápido de búsqueda** Implementación `union/find` en la que `find` es una operación de tiempo constante. (889)

**algoritmo rápido de unión** Implementación `union/find` en la que `union` es una operación de tiempo constante. (889)

**algoritmo union/find** Un algoritmo que se ejecuta procesando operaciones `union` y `find` en una estructura de datos `union/find`. (880)

**árbol de recubrimiento** Un árbol formado por aristas del grafo que conectan todos los vértices de un grafo no dirigido. (884)

**árbol mínimo de recubrimiento** Un sugrafo conectado de  $G$  que abarca todos los vértices con un coste total mínimo. Se trata de un problema fundamental en la teoría de grafos. (884)

**bosque** Una colección de árboles. (890)

**clase de equivalencia** La clase de equivalencia de un elemento  $x$  en el conjunto  $S$  es el subconjunto de  $S$  que contiene todos los elementos relacionados con  $x$ . (880)

**compresión de caminos** Hace que cada nodo al que se accede pase a ser un hijo de la raíz, hasta que se produzca otra operación `union`. (894)

**conjuntos disjuntos** Conjuntos que tienen la propiedad de que cada elemento aparece únicamente en un solo conjunto. (880)

**estructura union/find** Un método utilizado para manipular conjuntos disjuntos. (880)

**función de Ackermann** Una función que crece muy rápidamente. Su inversa tiene, esencialmente, un valor máximo igual a 4. (898)

**operaciones de la clase para conjuntos disjuntos** Las dos operaciones básicas necesarias para la manipulación de conjuntos disjuntos son `union` y `find`. (880)

**problema del ancestro común más próximo** Dado un árbol y una lista de pares de nodos del árbol, encontrar el ancestro común más próximo para cada par de nodos. La solución de este problema es importante en los algoritmos de grafos y en las aplicaciones de biología computacional. (886)

**rangos** En el algoritmo para conjuntos disjuntos, son las alturas estimadas de los nodos. (895)

**relación** Decimos que en un conjunto hay definida una relación si cada par de elementos están relacionados o no lo están. (879)

**relación de equivalencia** Una relación que es reflexiva, simétrica y transitiva. (879)

**unión por altura** Hace que el árbol menos profundo pase a ser hijo de la raíz del árbol más profundo durante una operación `union`. (893)

**unión por rango** Unión por altura cuando se realiza compresión de caminos. (895)

**unión por tamaño** Hace que el árbol más pequeño pase a ser hijo de la raíz del árbol mayor durante una operación `union`. (892)



## Errores comunes

- Al utilizar `union` asumimos a menudo que sus parámetros son raíces de árboles. Pueden producirse errores en el código si invocamos esa operación `union` utilizando como parámetros nodos que no sean raíces.



## Internet

Está disponible en línea la clase para conjuntos disjuntos. El nombre de archivo es el siguiente.

### **DisjointSets.java**

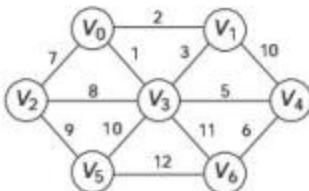
Contiene la clase para conjuntos disjuntos.



## Ejercicios

### EN RESUMEN

- Determine el árbol mínimo de recubrimiento para el grafo mostrado en la Figura 24.24.
- Muestre la operación del algoritmo NCA para los datos proporcionados en la Figura 24.8.

Figura 24.24 Grafo  $G$  para el Ejercicio 24.1.

- 24.3** Determine el resultado de la siguiente secuencia de instrucciones:  $\text{union}(1, 2)$ ,  $\text{union}(3, 4)$ ,  $\text{union}(3, 5)$ ,  $\text{union}(1, 7)$ ,  $\text{union}(3, 6)$ ,  $\text{union}(8, 9)$ ,  $\text{union}(1, 8)$ ,  $\text{union}(3, 10)$ ,  $\text{union}(3, 11)$ ,  $\text{union}(3, 12)$ ,  $\text{union}(3, 13)$ ,  $\text{union}(14, 15)$ ,  $\text{union}(16, 0)$ ,  $\text{union}(14, 16)$ ,  $\text{union}(1, 3)$  y  $\text{union}(1, 14)$  cuando las operaciones `union` se realizan
- Arbitrariamente.
  - Por altura.
  - Por tamaño.
- 24.4** Para cada uno de los árboles del Ejercicio 24.3, realice una operación `find` para el nodo más profundo, con compresión de caminos.

## EN TEORÍA

- 24.5** Demuestre que el algoritmo de Kruskal es correcto. ¿Ha asumido en su demostración que los costes de las aristas son no negativos?
- 24.6** Demuestre que para los laberintos generados por el algoritmo de la Sección 24.2.1, el camino que va desde el punto de partida al de llegada es único.
- 24.7** Diseñe un algoritmo que genere un laberinto que no contenga ningún camino desde la entrada a la salida, pero que tenga la propiedad de que la eliminación de una pared *pre-especificada* cree un camino único.
- 24.8** Suponga que quiere añadir una operación adicional, `remove(x)`, que elimina  $x$  de su conjunto actual y lo coloca en su propio conjunto. Muestre cómo modificar el algoritmo `union/find` de modo que el tiempo de ejecución de una secuencia de  $M$  operaciones `union`, `find` y `remove` siga siendo  $O(M\alpha(M, N))$ .
- 24.9** Demuestre que, si se realizan operaciones `union` por tamaño y se aplica la compresión de caminos, el tiempo de ejecución de caso peor sigue siendo  $O(M \log^* N)$ .
- 24.10** Suponga que implementa una compresión parcial de caminos en `find(i)`, cambiando el padre de cada uno de dos nodos del camino que va desde  $i$  hasta la raíz, de modo que el nuevo padre de cada nodo que se cambie sea ahora el abuelo de ese nodo (cuando hacer esto tenga sentido). Este proceso se denomina *división de caminos por la mitad*. Demuestre que, si realizamos la división de caminos por la mitad en las operaciones `find` y empleamos cualquiera de los heurísticos de `union`, el tiempo de ejecución de caso peor sigue siendo  $O(M \log^* N)$ .

## EN LA PRÁCTICA

- 24.11** Suponga que quiere añadir una operación extra, `deunion`, que deshace la última operación `union` que no se haya deshecho todavía. Una forma de hacer esto consiste en emplear la unión por rango, pero con operaciones `find` sin compresión y utilizar una pila para almacenar el estado anterior a una operación `union`. Podemos implementar la operación `deunion` extrayendo de la pila un estado anterior.
- ¿Por qué no podemos utilizar compresión de caminos?
  - Implemente el algoritmo `union/find/deunion`.
- 24.12** Implemente la operación `find` de forma no recursiva. ¿Hay una diferencia perceptible en el tiempo de ejecución?

## PROYECTOS DE PROGRAMACIÓN

- 24.13** Implemente el algoritmo de Kruskal.
- 24.14** Escriba un programa para determinar los efectos de la compresión de caminos y de las diversas estrategias de unión. Su programa debe procesar una larga secuencia de operaciones de equivalencia, utilizando todas las estrategias analizadas (incluyendo la división de caminos a la mitad, presentada en el Ejercicio 24.10).
- 24.15** Un algoritmo alternativo para el cálculo del árbol mínimo de recubrimiento es el desarrollado por Prim [12]. Funciona haciendo crecer un único árbol en etapas sucesivas. Comience seleccionando cualquier nodo como raíz. Al principio de una etapa, algunos nodos formarán parte del árbol y el resto no. En cada etapa, añadimos la arista de coste mínimo que conecte un nodo del árbol con un nodo que no sea del árbol. Una implementación del algoritmo de Prim es esencialmente idéntica al algoritmo de cálculo del camino más corto de Dijkstra proporcionado en la Sección 14.3, con una regla de actualización:

$$d_w = \min(d_w, c_{v,w})$$

(en lugar de  $d_w = \min(d_w, d_v + c_{v,w})$ ). Asimismo, como el grafo es no dirigido, cada arista aparece en dos listas de adyacencia. Implemente el algoritmo de Prim y compare su rendimiento con el del algoritmo de Kruskal.



## Referencias

La representación de cada conjunto mediante un árbol se propuso en [8]. En [1] se atribuye la compresión de caminos a McIlroy y Morris y se proporcionan varias aplicaciones de la estructura de datos `union/find`. El algoritmo de Kruskal se presenta en [11] y la alternativa expuesta en el Ejercicio 24.15 está tomada de [12]. El algoritmo NCA se describe en [2]. Otras aplicaciones se describen en [15].

La cota  $O(M \log^* N)$  para el problema `union/find` está tomada de [9]. Tarjan [13] obtuvo la cota  $O(M\alpha(M, N))$  y demostró que esa cota es estricta. Que la cota es intrínseca al problema general y que no puede ser mejorada mediante un algoritmo alternativo se

demuestra en [14]. Una cota más precisa para  $M < N$  aparece en [3] y [16]. Diversas otras estrategias de conversión de caminos y de unión permiten conseguir las mismas cotas, consulte los detalles en [16]. Si la secuencia de operaciones `union` se conoce de antemano, el problema `union/find` se puede resolver en un tiempo  $O(M)$  [7]. Este resultado se puede emplear para demostrar que el problema NCA fuera de línea es resoluble en un tiempo lineal.

Resultados del caso promedio para el problema `union/find` aparecen en [6], [10], [17] y [5]. En [4] se proporcionan resultados que acotan el tiempo de ejecución de cualquier operación aislada (en lugar de acotar para toda la secuencia).

1. A. V. Aho, J. E. Hopcroft y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 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. N. Blum, "On the Single-Operation Worst-Case Time Complexity of the Disjoint Set Union Problem", *SIAM Journal on Computing* 15 (1986), 1021–1024.
5. B. Bollobas e I. Simon, "Probabilistic Analysis of Disjoint Set Union Algorithms", *SIAM Journal on Computing* 22 (1993), 1053–1086.
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. Gallery 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.

# Apéndice A

## Operadores

La Figura A.1 muestra la precedencia y asociatividad de los operadores Java comunes de los que hemos hablado. Los operadores bit a bit se presentan en el Apéndice C.

Categoría	Ejemplos	Asociatividad
Operaciones con referencias	<code>+ []</code>	Izquierda a derecha
Unarios	<code>++ -- ! - (tipo)</code>	Derecha a izquierda
Multiplicativos	<code>* / %</code>	Izquierda a derecha
Aditivos	<code>+ -</code>	Izquierda a derecha
Desplazamiento (bit a bit)	<code>&lt;&lt; &gt;&gt;</code>	Izquierda a derecha
Relacionales	<code>&lt; &lt;= &gt; &gt;= instanceof</code>	Izquierda a derecha
Igualdad	<code>== !=</code>	Izquierda a derecha
AND booleano (o bit a bit)	<code>&amp;</code>	Izquierda a derecha
XOR booleano (o bit a bit)	<code>^</code>	Izquierda a derecha
OR booleano (o bit a bit)	<code> </code>	Izquierda a derecha
AND lógico	<code>&amp;&amp;</code>	Izquierda a derecha
OR lógico	<code>  </code>	Izquierda a derecha
Condicional	<code>? :</code>	Derecha a izquierda
Asignación	<code>= *= /= %= += -=</code>	Derecha a izquierda

Figura A.1 Operadores Java enumerados de mayor a menor precedencia.



# Interfaces gráficas de usuario

Una *interfaz gráfica de usuario* (GUI, *graphical user interface*) es la alternativa moderna a la E/S por terminal que permitía a un programa comunicarse con su usuario. En una GUI, se crea una aplicación con ventanas. Entre las formas que permiten proporcionar al programa la entrada se incluyen la técnica de selección entre una lista de posibles alternativas, el pulsar botones, el marcar casillas de verificación, el escribir en campos de texto y el utilizar el ratón. La salida se puede realizar escribiendo en campos del texto o dibujando gráficos. En Java 1.2 o versiones superiores, la programación de interfaces GUI se lleva a cabo utilizando el paquete *Swing*.

En este apéndice veremos

- Los componentes GUI básicos en Swing.
- Cómo comunican información dichos componentes.
- Cómo pueden disponerse esos componentes en una ventana.
- Cómo dibujar gráficos.

Una *interfaz gráfica de usuario* (GUI) es la alternativa moderna a la E/S por terminal que permite a un programa comunicarse con su usuario.

## B.1 Abstract Window Toolkit y Swing

*Abstract Window Toolkit* (AWT) es un conjunto de herramientas GUI que se suministra con todos los sistemas Java. Proporciona las clases básicas para desarrollar interfaces de usuario. Estas clases se encuentran en el paquete `java.awt`.<sup>1</sup> El AWT está diseñado para ser portable y funciona en múltiples plataformas. Para interfaces relativamente simples, el AWT es fácil de utilizar. Pueden escribirse interfaces GUI sin recurrir a herramientas de desarrollo visual, consiguiendo una gran mejora con respecto a las interfaces básicas de terminal.

Abstract Window Toolkit (AWT) es un conjunto de herramientas GUI que se suministra con todos los sistemas Java.

En un programa que utilice E/S de terminal, el programa suele pedir al usuario que introduzca los datos de entrada y luego ejecuta una instrucción que lee una línea del terminal. Después de leer la línea, se procesa esa

La programación de interfaces GUI está dirigida por sucesos.

<sup>1</sup> El código de este apéndice utiliza la directiva de importación comodín para ahorrar espacio.

entrada. El flujo de control en esta situación es fácil de seguir. La programación de interfaces GUI por su parte es completamente distinta. En la programación GUI, los componentes de entrada se disponen en una ventana. Después de mostrar la ventana, el programa espera a que se produzca un suceso, como por ejemplo la pulsación de un botón, en cuyo momento se invoca una rutina de tratamiento de sucesos. Esto quiere decir que el flujo de control es menos obvio en un programa con interfaz GUI. El programador debe suministrar la rutina de tratamiento de sucesos encargada de ejecutar un cierto fragmento de código.

El modelo de sucesos  
cambió entre Java 1.0 y  
Java 1.1 creando algunas  
incompatibilidades. Aquí  
describiremos la última  
versión.

Swing es un paquete GUI  
proporcionado en Java 1.2  
que está construido sobre  
AWT y que proporciona  
componentes más  
sofisticados.

Java 1.0 proporcionaba un modelo de sucesos que era muy engorroso de utilizar. Dicho modelo se sustituyó en Java 1.1 por otro modelo de sucesos más robusto. No es sorprendente que dichos modelos no sean completamente compatibles. Específicamente, un compilador de Java 1.0 no permitirá compilar adecuadamente el código que utilice el nuevo modelo de sucesos. Por su parte, los compiladores de Java 1.1 proporcionarán mensajes de diagnóstico para las estructuras de tipo Java 1.0. Sin embargo, el código Java 1.0 ya compilado se puede ejecutar sin problemas en una máquina virtual Java 1.1. En este apéndice se describe el modelo de sucesos más reciente. Muchas de las clases requeridas por el nuevo modelo de sucesos se encuentran en el paquete `java.awt.event`.

AWT proporcionaba una GUI simple, pero fue muy criticada por su falta de capacidades y por su inadecuado rendimiento. En Java 1.2, se añadió un conjunto mejorado de componentes en un nuevo paquete denominado `javax.swing`. Estos componentes se conocen colectivamente con el nombre

de *Swing*. Los componentes de Swing tienen un aspecto mucho mejor que sus equivalentes AWT; hay nuevos componentes Swing que no existían en AWT (como deslizadores y barras de progreso) y los componentes tienen ahora muchas más opciones (como por ejemplo sencillas sugerencias y mnemónicos). Además, Swing proporciona la noción de aspecto y estilo, con la que un programador puede visualizar la GUI en estilo Windows, X-Motif, Macintosh, independiente de la plataforma (metal) o incluso personalizado, independientemente de la plataforma subyacente (aunque, debido a cuestiones de derechos de propiedad intelectual y, quizás, a las malas relaciones entre Sun y Microsoft, el aspecto y estilo Windows solo funciona en sistemas Windows).

Swing está construido sobre AWT y, como resultado, el modelo de tratamiento de sucesos no varía. La programación en Swing es muy similar a la programación en el AWT de Java 1.1, salvo porque muchos nombres han variado. En este apéndice vamos a escribir únicamente la programación con Swing. Swing es una librería de gran tamaño; no es inusual encontrarse con libros completos dedicados a este tema, así que hay que dejar claro que la presentación que vamos a hacer aquí subestima enormemente los distintos aspectos involucrados en el diseño de interfaces de usuario.

La Figura B.1 ilustra algunos de los componentes básicos suministrados por Swing. Entre estos se incluyen `JComboBox` (actualmente está seleccionada la opción *Circle*), una lista `JList` (en la que actualmente está seleccionada *blue*), campos de texto `JTextField` básicos para entrada, tres botones de opción `JRadioButton`, una casilla de verificación `JCheckBox` y un botón `JButton` (denominado *Draw*). Junto al botón hay un campo de texto `JTextField` que se usa solo para salida (por eso tiene un color más oscuro que los campos `JTextField` de entrada situados encima de él). En la esquina superior izquierda hay un objeto `JPanel` que se puede utilizar para dibujar imágenes y gestionar la entrada de ratón.

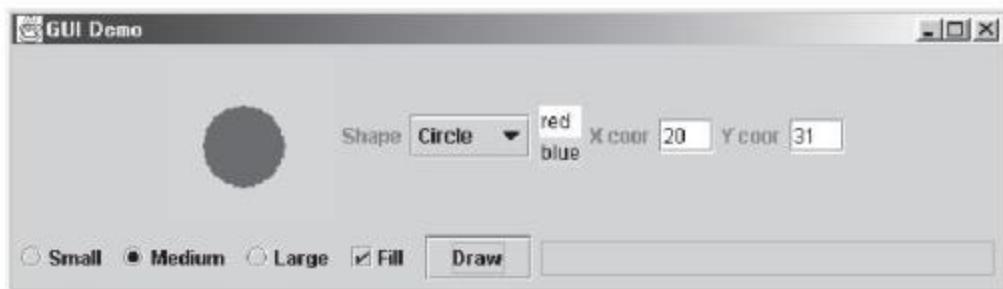


Figura B.1 Una GUI que ilustra algunos de los componentes Swing básicos.

Este apéndice describe la organización básica de la API de Swing. En él se cubren los distintos tipos de objetos, se indica cómo se pueden utilizar para llevar a cabo la entrada y la salida, se explica cómo se disponen dentro de una ventana y se analiza la forma en que se tratan los sucesos.

## B.2 Objetos básicos en Swing

AWT y Swing están organizados utilizando una jerarquía de herencia de clases. En la Figura B.2 se muestra una versión comprimida de esta jerarquía. Decimos que está comprimida porque algunas clases intermedias no se muestran. En la jerarquía completa, `JTextField` y `JTextArea`, por ejemplo, se amplían a partir de `JTextComponent`, mientras que muchas clases que tratan con fuentes, colores y otros objetos no se encuentran en la jerarquía `Component` y no se muestran en absoluto. Las clases `Font` y `Color`, que están definidas en el paquete `java.awt`, están ampliadas a partir de `Object`.

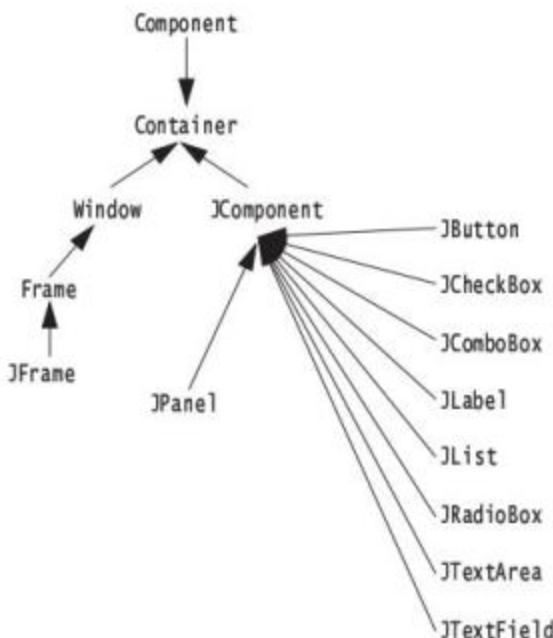


Figura B.2 Jerarquía comprimida de Swing.

### B.2.1 Component

La clase `Component` es una clase abstracta que actúa como superclase de muchos objetos AWT y por tanto de muchos objetos Swing. Puesto que es abstracta, no se puede instanciar. Un `Component` representa algo que tiene una posición y un tamaño y que se puede pintar en la pantalla, además de recibir sucesos de entrada.

La clase `Component` es una clase abstracta que actúa como superclase de muchos objetos AWT y por tanto de muchos objetos Swing. Puesto que es abstracta, no se puede instanciar. Un `Component` representa algo que tiene una posición y un tamaño y que se puede pintar en la pantalla, además de recibir sucesos de entrada. A partir de la Figura B.2 se pueden deducir algunos ejemplos de `Component`.

La clase `Component` contiene muchos métodos. Algunos de estos métodos se pueden utilizar para especificar el color o la fuente; otros se emplean para tratar sucesos. Algunos de los métodos importantes son

```
void setSize( int width, int height );
void setBackground( Color c );
void setFont( Font f );
void setVisible( boolean isVisible );
```

El método `setSize` se utiliza para cambiar el tamaño de un objeto. Funciona con objetos `JFrame`, pero no debe invocarse para objetos que empleen un mecanismo de disposición automática como por ejemplo `JButton`. Para estos, utilice `setPreferredSize`; este método admite un objeto `Dimension` que a su vez es construido con una longitud y una anchura (y está definido en `JComponent`). Los métodos `setBackground` y `setFont` se utilizan para cambiar el color de fondo y la fuente asociados con un componente `Component`. Requieren un objeto `Color` y un objeto `Font`, respectivamente. Finalmente, el método `show` hace que un componente sea visible. Su uso típico es para un `JFrame`.

### B.2.2 Container

Un `Container` es una superclase abstracta que representa todos los componentes que pueden albergar otros componentes.

En AWT, un contenedor `Container` es la superclase abstracta que representa todos los componentes que pueden albergar a otros componentes. Un ejemplo de `Container` AWT sería la clase `Window`, que representa una ventana de nivel superior. Como muestra la jerarquía de herencia, un `Container` ES-UN `Component`. Una instancia concreta de un objeto `Container` almacenará una colección de objetos `Component` además de otros objetos `Container`.

El contenedor cuenta con un útil objeto auxiliar denominado `LayoutManager`, que es una clase que permite posicionar componentes dentro del contenedor. Algunos métodos útiles son:

```
void setLayout( LayoutManager mgr );
void add( Component comp );
void add( Component comp, Object where );
```

Los gestores de disposición (*layout managers*) se describen en la Sección B.3.1. Un contenedor debe definir primero cómo disponer los objetos dentro del contenedor. Esto se hace utilizando `setLayout`. Después se añaden los objetos al contenedor, uno a uno, utilizando `add`. Piense en el contenedor como si fuera una maleta en la que podemos meter ropa. Piense en el gestor de disposición como en el experto en hacer maletas que nos explicará cómo meter la ropa en la maleta.

### B.2.3 Contenedores de nivel superior

Como muestra la Figura B.2 hay dos tipos de objetos `Container`:

1. Las ventanas de nivel superior, que al final terminan abarcando a `JFrame`.
2. El `JComponent`, que al final termina abarcando el resto de los componentes Swing.

`JFrame` es un ejemplo de "componente pesado", mientras que todos los componentes Swing de la jerarquía `JComponent` son "ligeros". La diferencia básica entre componentes pesados y ligeros es que los componentes ligeros son dibujados en un lienzo completamente por Swing mientras que los componentes pesados interactúan con el sistema de ventanas nativo. Como resultado, los componentes ligeros pueden agregar otros componentes ligeros (por ejemplo, se puede utilizar `add` para colocar varios objetos `JButton` en un `JPanel`), pero no se puede agregar con `add` directamente dentro de un componente pesado. En lugar de ello, lo que hacemos es obtener un `Container` que representa su "panel de contenidos" y agregar con `add` dentro del panel de contenido, permitiendo así que Swing se encargue de actualizar dicho panel de contenido. De este modo, el sistema de ventanas nativo no se ve involucrado en la actualización (obtendrá una excepción de tiempo de ejecución si trata de agregar algo dentro de un componente pesado), incrementando así la velocidad de las actualizaciones.

Los contenedores básicos son la ventana de nivel superior `Window` y `JComponent`. Los componentes típicos pesados son `JWindow`, `JFrame` y `JDialog`.

Solo hay unas cuantas ventanas básicas de nivel superior, entre las que se incluyen

1. `JWindow`: una ventana de nivel superior que no tiene borde.
2. `JFrame`: una ventana de nivel superior que tiene borde y que también puede tener un `JMenuBar` asociado.<sup>2</sup>
3. `JDialog`: una ventana de nivel superior utilizada para crear cuadros de diálogo.

Una aplicación que utilice una interfaz Swing debe tener un `JFrame` (o una clase ampliada a partir de `JFrame`) como contenedor más externo.

### B.2.4 JPanel

La otra subclase `Container` es `JComponent`. Uno de esos componentes `JComponent` es el `JPanel`, que se utiliza para almacenar una colección de objetos, pero no crea bordes, por lo que se trata de la más simple de las clases de contenedores.

El `JPanel` se utiliza para almacenar una colección de objetos, pero no crea bordes. Por tanto, es la más simple de las clases `Container`.

El principal uso del `JPanel` consiste en organizar los objetos en una sola unidad. Por ejemplo, piense en un formulario de registro que requiera un nombre, una dirección, un número de la seguridad social y los números de teléfono del domicilio y del trabajo. Todos estos componentes del formulario podrían darnos un panel `PersonPanel`. Entonces, el formulario de registro podría contener varias entidades `PersonPanel` para permitir la posibilidad de que varias personas utilicen un mismo formulario.

<sup>2</sup> Los menús no se explican en este apéndice.

Por ejemplo, la Figura B.3 muestra cómo se agrupan en una clase JPanel los componentes mostrados en la Figura B.1 e ilustra la técnica general de creación de una subclase de JPanel. Lo que quedará por hacer después es construir los objetos, disponerlos de una forma adecuada y tratar el suceso de pulsación del botón.

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5 class GUI extends JPanel implements ActionListener
6 {
7     public GUI( )
8     {
9         makeTheObjects( );
10        doTheLayout( );
11        theDrawButton.addActionListener( this );
12    }
13    // Crear todos los objetos
14    private void makeTheObjects( )
15    { /* Implementación en la Figura B.4 */ }
16
17    // Disponer todos los objetos
18    private void doTheLayout( )
19    { /* Implementación en la Figura B.7 */ }
20
21    // Tratar la pulsación del botón Draw
22    public void actionPerformed( ActionEvent evt )
23    { /* Implementación en la Figura B.9 */ }
24
25    private GUICanvas theCanvas;
26    private JComboBox theShape;
27    private JList theColor;
28    private JTextField theXCoor;
29    private JTextField theYCoor;
30    private JRadioButton smallPic;
31    private JRadioButton mediumPic;
32    private JRadioButton largePic;
33    private JCheckBox theFillBox;
34    private JButton theDrawButton;
35    private JTextField theMessage;
36 }
```

Figura B.3 Clase GUI básica mostrada en la Figura B.1.

Observe que `GUI` implementa la interfaz `ActionListener`. Esto quiere decir que entiende cómo tratar un *suceso de acción* (en este caso, una pulsación de botón). Para implementar la interfaz `ActionListener`, una clase debe proporcionar un método `actionPerformed`. Asimismo, cuando el botón genere un suceso de acción, debe saber qué componente debe recibir el suceso. En este caso, al hacer la llamada de la línea 11 (en la Figura B.3), el objeto `GUI` que contiene el `JButton` le dice a `Button` que le envíe el suceso. Estos detalles de tratamiento de sucesos se analizan en la Sección B.3.3.

Un segundo uso del `JPanel` es agrupar los objetos en una unidad con el propósito de simplificar la disposición en pantalla. Esto se explica en la Sección B.3.5.

Casi toda la funcionalidad de `JPanel` está heredada de `JComponent`. Esto incluye rutinas para dibujar, cambiar de tamaño y tratar los sucesos, así como el método para configurar las sugerencias para los componentes:

```
void setToolTipText( String txt );
void setPreferredSize( Dimension d );
```

## B.2.5 Componentes de E/S importantes

Swing proporciona un conjunto de componentes que se pueden usar para llevar a cabo la entrada y la salida. Estos componentes son fáciles de configurar y de utilizar. El código de la Figura B.4 ilustra cómo se construyen cada uno de los componentes básicos mostrados en la Figura B.1. Generalmente, esto implica invocar un constructor y aplicar un método para personalizar el componente. Este código no especifica cómo se disponen los elementos en el `JPanel` o cómo se examinan los estados de los componentes. Recuerde que la programación de las interfaces GUI consiste en dibujar la interfaz y luego esperar a que ocurran sucesos. La disposición de los componentes en pantalla y el tratamiento de sucesos se analizan en la Sección B.3.

### JLabel

Un `JLabel` es un componente que sirve para colocar texto en un contenedor. Su uso principal es el de etiquetar otros componentes tales como `JComboBox`, `JList`, `JTextField` o `JPanel` (muchos otros componentes ya se encargan de mostrar sus nombres de una u otra manera). En la Figura B.1, las frases *Shape*, *X Coore* e *Y Coor* son etiquetas. Un componente `JLabel` se construye con una cadena de caracteres `String` opcional y puede modificarse con el método `setText`. Estos métodos son:

```
JLabel();
JLabel( String theLabel );
void setText( String theLabel );
```

`JLabel` es un componente que permite colocar texto en un contenedor. Su uso principal es el de etiquetar otros componentes.

### JButton

El botón `JButton` se utiliza para crear un botón etiquetado. La Figura B.1 contiene un `JButton` con la etiqueta *Draw*. Cuando se pulsa el `JButton`, se genera un *suceso de acción*. En la Sección B.3.3 se describe cómo se tratan los sucesos de acción. El `JButton` es similar al `JLabel` en el sentido de que

`JButton` se utiliza para crear un botón etiquetado. Cuando se pulsa, se genera un suceso de acción.

```
1 // Construir todos los objetos
2 private void makeTheObjects( )
3 {
4     theCanvas = new GUICanvas( );
5     theCanvas.setBackground( Color.green );
6     theCanvas.setPreferredSize( new Dimension( 99, 99 ) );
7
8     theShape = new JComboBox( new String [ ]
9                             { "Circle", "Square" } );
10
11    theColor = new JList( new String [ ] { "red", "blue" } );
12    theColor.setSelectionMode(
13        ListSelectionModel.SINGLE_SELECTION );
14    theColor.setSelectedIndex( 0 ); // make red default
15
16    theXCoor = new JTextField( 3 );
17    theYCoor = new JTextField( 3 );
18
19    ButtonGroup theSize = new ButtonGroup( );
20    smallPic = new JRadioButton( "Small", false );
21    mediumPic = new JRadioButton( "Medium", true );
22    largePic = new JRadioButton( "Large", false );
23    theSize.add( smallPic );
24    theSize.add( mediumPic );
25    theSize.add( largePic );
26
27    theFillBox = new JCheckBox( "Fill" );
28    theFillBox.setSelected( false );
29
30    theDrawButton = new JButton( "Draw" );
31
32    theMessage = new JTextField( 25 );
33    theMessage.setEditable( false );
34 }
```

Figura B.4 Código que construye los objetos de la Figura B.1.

un JButton se construye con una cadena de caracteres opcional. La etiqueta del JButton se puede modificar con el método setText. Estos métodos son:

```
JButton( );
JButton( String theLabel );
void setText( String theLabel );
void setMnemonic( char c );
```

### JComboBox

El recuadro combinado `JComboBox` se utiliza para seleccionar un único objeto (típicamente una cadena de caracteres) mediante una lista emergente de opciones. Solo se puede seleccionar una opción en cada momento, y de manera predeterminada solo se puede elegir un objeto que sea una de las opciones. Si el `JComboBox` se hace editable, el usuario puede escribir una entrada distinta de las opciones proporcionadas. En la Figura B.1, el tipo de forma geométrica mostrado es un objeto `JComboBox`; actualmente está seleccionada la opción *Circle*. Algunos de los métodos `JComboBox` son:

```
JComboBox();
JComboBox( Object [ ] choices );
void addItem( Object item );
Object getSelectedItem();
int getSelectedIndex();
void setEditable( boolean edit );
void setSelectedIndex( int index );
```

`JComboBox` se utiliza para seleccionar una única cadena de caracteres a través de una lista emergente de opciones.

Un `JComboBox` se construye sin ningún parámetro o con una matriz de opciones. Después se pueden añadir objetos `Object` (típicamente cadenas de caracteres) a la lista de opciones `JComboBox` o pueden eliminarse opciones de la lista. Cuando se invoca `getSelectedItem`, se devuelve un `Object` que representa el elemento actualmente seleccionado (o `null`, si no hay seleccionada ninguna opción).

En lugar de devolver el objeto `Object` actual, se puede devolver su índice (calculado a partir del orden de llamadas a `addItem`) invocando `getSelectedIndex`. El primer elemento añadido tiene índice 0, etc. Esto puede ser útil, porque si una matriz almacena información correspondiente a cada una de las opciones, se puede utilizar `getSelectedIndex` para indexar dicha matriz. El método `setSelectedIndex` se emplea para especificar una selección predeterminada.

### JList

El componente `JList` permite seleccionar de una lista desplazable de objetos `Object`. En la Figura B.1, la opción de color se presenta como una `JList`. La `JList` difiere del `JComboBox` en tres aspectos fundamentales:

1. `JList` se puede configurar para permitir que se seleccione un único elemento o múltiples elementos (la opción predeterminada es la selección múltiple).
2. `JList` permite al usuario ver más de una opción a la vez.
3. `JList` ocupa más espacio en pantalla que `JComboBox`.

El componente `JList` permite seleccionar de una lista desplazable de objetos `Object`. Puede configurarse para seleccionar solo un elemento o múltiples elementos.

Los métodos básicos de `JList` son:

```
JList();
JList( Object [ ] items );
void setListData( Object [ ] items );
int getSelectedIndex();
```

```

int [ ] getSelectedIndices( );
Object getSelectedValue( );
Object [ ] getSelectedValues( );
void setSelectedIndex( int index );
void setSelectedValue( Object value );
void setSelectionMode( int mode );

```

Una `JList` se construye o bien sin parámetros o con una matriz de elementos (hay también otros constructores más sofisticados). La mayoría de los métodos indicados tienen el mismo comportamiento (posiblemente con diferentes nombres) que los métodos correspondientes de `JComboBox`. `getSelectedValue` devuelve `null` si no hay ningún elemento seleccionado. `getSelectedValues` se emplea para gestionar las selecciones múltiples; devuelve una matriz de `Object` (posiblemente de longitud 0) que se corresponde con los elementos seleccionados. Al igual que sucede con `JComboBox`, se pueden obtener índices en lugar de referencias `Object` mediante otros métodos públicos.

El método `setSelectionMode` se utiliza para permitir únicamente la selección de un solo elemento. El código típico sería

```
lst.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
```

### JCheckBox y JRadioButton

Una casilla de verificación es un componente GUI que tiene un estado *on* y un estado *off*. Un `ButtonGroup` puede contener un grupo de botones de los que solo uno puede ser *true* en cualquier momento determinado.

Una casilla de verificación `JCheckBox` es un componente de la GUI que tiene un estado *on* y un estado *off*. El estado *on* es `true` y el estado *off* es `false`. Se considera un botón (en la API de Swing hay definida una clase `AbstractButton` de la que derivan `JButton`, `JCheckBox` y `JRadioButton`). Un botón `JRadioButton` es similar a una casilla de verificación, salvo porque los `JRadioButton` son redondos; utilizaremos el término *casilla de verificación* como término genérico para describir a ambos. La Figura B.1 contiene cuatro objetos casilla de verificación. En esta figura, la casilla de verificación *Fil* es actualmente `true` y las otras tres casillas de verificación pertenecen a un grupo de botones `ButtonGroup`: solo puede estar activada una de las casillas de verificación del grupo de tres. Cuando se selecciona una casilla de verificación de un grupo, las restantes casillas del grupo quedan desactivadas. Un grupo `ButtonGroup` se construye con cero parámetros. Observe que no es un `Component`; simplemente es una clase auxiliar que amplía `Object`.

Los métodos comunes para `JCheckBox` son similares a los de `JRadioButton` y son:

```

JCheckBox( );
JCheckBox( String theLabel );
JCheckBox( String theLabel, boolean state );
boolean isSelected( );
void setLabel( String theLabel );
void setSelected( boolean state );

```

Una `JCheckBox` independiente se construye con una etiqueta opcional. Si no se proporciona la etiqueta, se puede añadir posteriormente mediante `setLabel`. El método `setLabel` también se

puede utilizar para cambiar la etiqueta existente de la `JCheckBox`. El método `setSelected` se suele utilizar fundamentalmente para establecer un valor predeterminado para una `JCheckBox`. El método `isSelected` devuelve el estado de una `JCheckBox`.

Una `JCheckBox` que es parte de un `ButtonGroup` se construye de la manera usual y luego se la añade al objeto `ButtonGroup` utilizando el método `add` de `ButtonGroup`. Los métodos de `ButtonGroup` son:

```
ButtonGroup( );
void add( AbstractButton b );
```

## Lienzos

En AWT, un componente `Canvas` representa un área rectangular en blanco de la pantalla, en la que una aplicación puede dibujar. Las primitivas gráficas se describen en la Sección B.3.2. Un `Canvas` podría también recibir entradas del usuario, en forma de sucesos de ratón y de teclado. El `Canvas` nunca se utilizaba directamente; en lugar de ello, el programador definía una subclase de `Canvas` con la funcionalidad apropiada. La subclase sustituía el método público

```
void paint( Graphics g );
```

En Swing, esto ya no se utiliza. Se puede conseguir el mismo efecto ampliando `JPanel` y sustituyendo el método público

```
void paintComponent( Graphics g );
```

Aunque esto funciona para cualquier componente, utilizando un `JPanel` de un tamaño específico, evitaremos dibujar por accidente más allá de la frontera del "lienzo".

## JTextField y JTextArea

`JTextField` es un componente que presenta al usuario una única línea de texto. Un campo `JTextArea` permite varias líneas y tiene una funcionalidad similar. Por tanto, aquí solo vamos a hablar de `JTextField`. De manera predeterminada el texto puede ser editado por el usuario, pero es posible hacer que el texto no sea editable. En la Figura B.1, hay tres objetos `JTextField`: dos para las coordenadas y uno, que el usuario no puede editar, que se utiliza para comunicar mensajes de error. El color de fondo de un campo de texto no editable difiere del color de un campo de texto editable. Algunos de los métodos comunes asociados con `JTextField` son:

```
JTextField( );
JTextField( int cols );
JTextField( String text, int cols );
String getText( );
boolean isEditable( );
void setEditable( boolean editable );
void setText( String text );
```

Un componente lienzo (canvas) representa un área rectangular en blanco de la pantalla, en la que una aplicación puede dibujar o a través de la cual puede recibir sucesos de entrada.

Un `JTextField` es un componente que presenta al usuario una sola línea de texto. Un campo `JTextArea` permite múltiples líneas y tiene una funcionalidad similar.

Un `JTextField` se construye sin ningún parámetro o especificando un texto inicial opcional y el número de columnas. El método `setEditable` se puede utilizar para impedir la entrada en el campo `JTextField`. `setText` puede emplearse para imprimir mensajes en el campo `JTextField` y `getText` para leer del `JTextField`.

## B.3 Principios básicos

En esta sección examinamos tres facetas importantes de la programación con AWT: en primer lugar, cómo se disponen los objetos dentro de un contenedor, seguido de la manera en que se tratan los sucesos, como por ejemplo la pulsación de botones. Finalmente, se describe la forma de dibujar gráficos dentro de objetos lienzo.

### B.3.1 Gestores de disposición

Un gestor de disposición (*layout manager*) coloca automáticamente los componentes de un contenedor. Un gestor de disposición se asocia con un contenedor mediante el método `setLayout`.

Un *gestor de disposición (layout manager)* se encarga de colocar automáticamente los componentes de un contenedor. Se asocia con un contenedor ejecutando el comando `setLayout`. Un ejemplo de utilización de `setLayout` sería la llamada

```
setLayout( new FlowLayout( ) );
```

Observe que no es necesario guardar una referencia al gestor de disposición. El contenedor al que se aplica el comando `setLayout` se encarga de almacenarlo como un miembro de datos privado. Cuando se utiliza un gestor de disposición, las solicitudes para cambiar el tamaño de muchos de los componentes, como los botones, no funcionan, porque el gestor de disposición se encargará de seleccionar sus propios tamaños para los componentes, según considere apropiado. La idea es que el gestor de disposición determinará los mejores tamaños que permiten ajustar la disposición en pantalla a las especificaciones.

Piense en el gestor de disposición como si fuera un empaquetador experto contratado por el contenedor para tomar las decisiones finales acerca de cómo empaquetar los elementos añadidos al contenedor.

#### FlowLayout

La más simple de las disposiciones es `FlowLayout`. Cuando el contenido de un contenedor se coloca utilizando `FlowLayout`, sus componentes se añaden en una fila de izquierda a derecha. Cuando no queda espacio en una fila, se forma una fila nueva. De manera predeterminada, cada fila está centrada. Esto puede modificarse proporcionando un parámetro adicional en el constructor con el valor `FlowLayout.LEFT` o `FlowLayout.RIGHT`.

El problema de utilizar `FlowLayout` es que una fila se puede partir en algún lugar que no deseemos. Por ejemplo, si no hay espacio en una fila podría producirse un salto a la fila siguiente entre un componente `JLabel` y un componente `JTextField`, aunque ambos componentes deberían siempre permanecer adyacentes desde el punto de vista lógico. Una forma de evitar esto es crear un `JPanel` separado con estos dos

La más simple de las disposiciones es `FlowLayout`, que añade los componentes en una fila, de izquierda a derecha.

elementos y luego añadir el JPanel al contenedor. Otro problema con FlowLayout es que resulta difícil alinear las cosas verticalmente.

El gestorFlowLayout es el predeterminado para un JPanel.

### BorderLayout

El gestor BorderLayout es el predeterminado para los objetos de la jerarquía Window, como por ejemplo JFrame. Dispone los objetos de un contenedor situando los componentes en una de cinco ubicaciones posibles. Para que esto suceda, el método add debe proporcionar como segundo parámetro una de las cadenas de caracteres "North", "South", "East", "West" y "Center"; Si no se especifica, el segundo parámetro toma la opción predeterminada "Center" (por lo que un método add con un solo parámetro funcionará, pero varios métodos add colocarían los elementos unos encima de otros). La Figura B.5 muestra cinco botones añadidos a un Frame utilizando un BorderLayout. El código para generar este tipo de disposición en pantalla se muestra en la Figura B.6. Observe que utilizamos la técnica típica consistente en añadir un JPanel ligero y luego añadir el JPanel al panel de contenido JFrame de nivel superior. Típicamente, alguna de las cinco ubicaciones puede quedar sin utilizar. Asimismo, el componente colocado en una ubicación suele ser un JPanel que a su vez contiene otros componentes y que utiliza algún otro tipo de disposición.

**BorderLayout** es el gestor de disposición predeterminado para los objetos de la jerarquía Window, como por ejemplo JFrame y JDialo g. Dispone el contenedor colocando los componentes en una de cinco ubicaciones posibles.

Por ejemplo, el código de la Figura B.7 muestra cómo se disponen los objetos de la Figura B.1. Aquí, tenemos dos filas, pero queremos asegurarnos de que las casillas de verificación, los botones y el campo de texto de salida se coloquen debajo del resto de la GUI. La idea es crear un JPanel que almacene los elementos que deben estar en la mitad superior y otro JPanel que almacene los elementos de la mitad inferior. Estos dos paneles JPanel se pueden disponer uno encima de otro colocándolos mediante un BorderLayout.

Las líneas 4 y 5 crean los dos objetos JPanel llamados topHalf y bottomHalf. Cada uno de los objetos JPanel se dispone a continuación por separado utilizando un FlowLayout. Observe que los métodos setLayout y add se aplican al JPanel apropiado. Puesto que los JPanel se disponen mediante FlowLayout, pueden consumir más de una fila si no hay suficiente espacio horizontal disponible. Esto podría provocar un salto de fila incorrecto entre un JLabel y un JTextField. Dejamos como ejercicio para el lector la creación de paneles JPanel adicionales para cerciorarse de que ningún salto de fila desconecte un JLabel del componente al que está etiquetando. Una vez terminados los JPanel, utilizamos un BorderLayout para alinearlos. Esto se hace en las líneas 28 a 30. Observe también que el contenido de ambos JPanel está centrado. Esto es así gracias al



Figura B.5 Cinco botones dispuestos en pantalla mediante BorderLayout.

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 // Generar Figura B.5
5 public class BorderTest extends JFrame
6 {
7     public static void main( String [ ] args )
8     {
9         JFrame f = new BorderTest( );
10        JPanel p = new JPanel( );
11
12        p.setLayout( new BorderLayout( ) );
13        p.add( new JButton( "North" ), "North" );
14        p.add( new JButton( "East" ), "East" );
15        p.add( new JButton( "South" ), "South" );
16        p.add( new JButton( "West" ), "West" );
17        p.add( new JButton( "Center" ), "Center" );
18
19        Container c = f.getContentPane( );
20        c.add( p );
21        f.pack( );           // Redimensionar marco a tamaño mínimo
22        f.setVisible( true ); // Mostrar el marco
23    }
24 }

```

**Figura B.6** Código que ilustra el uso de BorderLayout.

Cuando se utiliza BorderLayout, todo comando add que se ejecute sin un parámetro String tomará como opción predeterminada "Center".

FlowLayout. Para hacer que los contenidos del JPanel estén alineados a la izquierda, las líneas 8 y 19 deberían construir el FlowLayout con el parámetro adicional FlowLayout.LEFT.

Cuando se utiliza el BorderLayout, cualquier comando add que se ejecute sin una String utilizará "Center" como opción predeterminada. Si se proporciona una String pero no es una de las cinco cadenas aceptables (deben respetarse las mayúsculas y las minúsculas), se genera una excepción de tiempo de ejecución.<sup>3</sup>

<sup>3</sup> Observe que en Java 1.0, los argumentos de add estaban invertidos, y que los objetos String que faltaban o que eran incorrectos se ignoraran sin más, lo que hacía que la depuración fuera difícil. Este estilo antiguo sigue estando permitido, pero se recomienda no utilizarlo.

```

1      // Colocar todos los objetos
2  private void doTheLayout( )
3  {
4      JPanel topHalf = new JPanel( );
5      JPanel bottomHalf = new JPanel( );
6
7          // Colocar la mitad superior
8      topHalf.setLayout( new FlowLayout( ) );
9      topHalf.add( theCanvas );
10     topHalf.add( new JLabel( "Shape" ) );
11     topHalf.add( theShape );
12     topHalf.add( theColor );
13     topHalf.add( new JLabel( "X coor" ) );
14     topHalf.add( theXCoor );
15     topHalf.add( new JLabel( "Y coor" ) );
16     topHalf.add( theYCoor );
17
18         // Colocar la mitad inferior
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         // Ahora colocar la GUI
28     setLayout( new BorderLayout( ) );
29     add( topHalf, "North" );
30     add( bottomHalf, "South" );
31 }

```

Figura B.7 Código que coloca los objetos de la Figura B.1.

## Gestor de disposición null

El gestor de disposición `null` se utiliza para realizar un posicionamiento preciso. En el gestor `null`, cada objeto se añade al contenedor mediante `add`. Después, su posición y su tamaño se pueden configurar llamando al método `setBounds`:

```
void setBounds( int x, int y, int width, int height );
```

El gestor de disposición `null` se utiliza para realizar un posicionamiento preciso.

Aquí, `x` e `y` representan la ubicación de la esquina superior izquierda del objeto, en relación a la esquina superior izquierda de su contenedor. `width` y `height`, por su parte, representan el tamaño

del objeto, especificando respectivamente la altura y la anchura. Todas las unidades están en píxeles.

El gestor de disposición `null` es dependiente de la plataforma; normalmente, esto constituye un problema.

### Gestores de disposición más sofisticados

Otros gestores de disposición simulan las ventanas de fichas con pestanas o permiten disponer los objetos en una cuadrícula arbitraria.

Java también proporciona los gestores `CardLayout`, `GridLayout` y `GridBagLayout`. El gestor de disposición `CardLayout` simula las fichas con pestaña que tan populares son en las aplicaciones Windows. `GridLayout` añade componentes en una cuadrícula, pero hace que cada elemento de la cuadrícula tenga el mismo tamaño. Esto provoca que los componentes se estiren en ocasiones de manera poco natural. Resulta útil en aquellos casos en los que esto no es un problema, como por ejemplo cuando lo que se quiere pintar es un teclado que calculadora que está compuesto por una cuadrícula bidimensional de botones. `GridBagLayout` añade componentes a una cuadrícula, pero permite que los componentes cubran varias celdas de la cuadrícula. Es más complicado que los otros gestores de disposición.

### Herramientas visuales

Los productos comerciales incluyen herramientas que permiten al programador dibujar la disposición utilizando un sistema tipo CAD. La herramienta genera a continuación el código Java para construir los objetos y disponerlos de la manera adecuada. Incluso con este sistema, el programador sigue teniendo que escribir la mayor parte del código, incluyendo el tratamiento de sucesos, aunque queda liberado del trabajo sucio requerido para calcular posiciones precisas de los objetos.

#### B.3.2 Gráficos

Los gráficos se dibujan definiendo una clase que amplía `JPanel`. La nueva clase sustituye al método `paintComponent` y proporciona un método público que se puede invocar desde el contenedor del lienzo.

`Graphics` es una clase abstracta que define varios métodos de dibujo.

Como hemos mencionado en la Sección B.2.5, los gráficos se dibujan utilizando un objeto `JPanel`. Específicamente, para generar gráficos, el programador debe definir una nueva clase que amplíe `JPanel`. Esta nueva clase proporciona un constructor (si el predeterminado no es aceptable), sustituye un método denominado `paintComponent` y proporciona un método público que se puede invocar desde el contenedor del lienzo. El método `paintComponent` es

```
void paintComponent( Graphics g );
```

`Graphics` es una clase abstracta que define varios métodos. Entre estos se encuentran

```
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 y1, int x2, int y2 );
void drawString( String str, int x, int y );
void setColor( Color c );
```

En Java, las coordenadas se miden respecto a la esquina superior izquierda del componente. `drawOval`, `drawRect`, `fillOval` y `fillRect` dibujan todas ellas un objeto de anchura `width` y altura `height` especificadas, estando la esquina superior izquierda situada en las coordenadas indicadas por `x` e `y`. `drawLine` y `drawString` dibujan línea y texto, respectivamente. `setColor` se utiliza para cambiar el color actual; el nuevo color será utilizado por todas las rutinas de dibujo hasta que se vuelva a cambiar.

Es importante que la primera línea de `paintComponent` invoque al método `paintComponent` de la superclase.

La Figura B.8 ilustra cómo se implementa el lienzo de la Figura B.1. La nueva clase `GUICanvas` amplía `JPanel`. Proporciona varios miembros de datos privados que describen el estado actual del lienzo. El constructor `GUICanvas` predeterminado es razonable, así que lo aceptamos.

Los miembros de datos se configuran mediante el método público `setParams`, que es proporcionado para que el contenedor (es decir, la clase `GUI` que almacena el lienzo `GUICanvas`) pueda comunicar al lienzo `GUICanvas` el estado de sus diversos componentes de entrada. El método `setParams` se muestra en las líneas 3 a 13. La última línea de `setParams` llama al método `repaint`.

El método `repaint` planifica un borrado del componente y una subsiguiente llamada a `paintComponent`. Por tanto, lo único que tenemos que hacer es escribir un método `paintComponent` que dibuje el lienzo tal como se especifique en los miembros de datos de la clase. Como puede verse por su implementación en las líneas 15 a 35, después de efectuar un encadenamiento hacia arriba, hacia la superclase, `paintComponent` simplemente llama al método `Graphics` descrito anteriormente en este apéndice.

En Java, las coordenadas se miden respecto a la esquina superior izquierda del componente.

Es importante que la primera línea de `paintComponent` llame al método `paintComponent` de la superclase.

El método `repaint` planifica el borrado de un componente y una posterior llamada a `paintComponent`.

### B.3.3 Sucesos

Cuando el usuario escribe en el teclado o utiliza el ratón, el sistema operativo genera un suceso. El sistema original de tratamiento de sucesos de Java era muy engorroso y ha sido completamente rehecho. El nuevo modelo, en vigor desde Java 1.1, es mucho más simple de programar que el antiguo. Observe que los dos modelos son incompatibles: los sucesos Java 1.1 no son comprendidos por los compiladores Java 1.0. La reglas básicas son las siguientes:

1. Cualquier clase que quiera proporcionar código para gestionar un suceso debe implementar (con `implement`) un interfaz `escucha`. Ejemplos de interfaces escucha serían `ActionListener`, `WindowListener` y `MouseListener`. Como es usual, implementar una interfaz significa que la clase debe definir todos los métodos de la misma.
2. Un objeto que quiera tratar el suceso generado por un componente debe registrar su deseo de hacerlo mediante un mensaje `addListener` enviado al componente que genera el suceso. Cuando un componente genera un suceso, este es enviado al objeto que se ha registrado para recibarlo. Si ningún objeto se ha registrado para recibarlo, entonces es ignorado.

El sistema original de tratamiento de sucesos de Java era muy engorroso y ha sido rehecho por completo.

Por ejemplo, considere el suceso de acción generado cuando el usuario pulsa un `JButton`, cuando pulsa `Intro` mientras está en un `JTextField` o efectúa una selección en una `JList` o un `JMenuItem`.

```
1 class GUICanvas extends JPanel
2 {
3     public void setParams( String aShape, String aColor, int x,
4                               int y, int size, boolean fill )
5     {
6         theShape = aShape;
7         theColor = aColor;
8         xcoor = x;
9         ycoor = y;
10        theSize = size;
11        fillOn = fill;
12        repaint();
13    }
14
15    public void paintComponent( Graphics g )
16    {
17        super.paintComponent( g );
18        if( theColor.equals( "red" ) )
19            g.setColor( Color.red );
20        else if( theColor.equals( "blue" ) )
21            g.setColor( Color.blue );
22
23        theWidth = 25 * ( theSize + 1 );
24
25        if( theShape.equals( "Square" ) )
26            if( fillOn )
27                g.fillRect( xcoor, ycoor, theWidth, theWidth );
28            else
29                g.drawRect( xcoor, ycoor, theWidth, theWidth );
30        else if( theShape.equals( "Circle" ) )
31            if( fillOn )
32                g.fillOval( xcoor, ycoor, theWidth, theWidth );
33            else
34                g.drawOval( xcoor, ycoor, theWidth, theWidth );
35    }
36
37    private String theShape = "";
38    private String theColor = "";
39    private int xcoor;
40    private int ycoor;
41    private int theSize; // 0 = pequeño, 1 = mediano, 2 = grande
42    private boolean fillOn;
43    private int theWidth;
44 }
```

Figura B.8 Lienzo básico mostrado en la esquina superior izquierda de la Figura B.1.

La forma más simple de tratar un clic sobre un JButton es hacer que su contenedor implemente ActionListener proporcionando un método actionPerformed y registrándose a sí mismo ante el JButton como rutina de tratamiento del suceso.

Esto se aplica a nuestro ejemplo de la Figura B.1 de la forma siguiente: recuerde que en la Figura B.3 ya hemos hecho dos cosas: en la línea 5, GUI declara que implementa ActionListener y en la línea 11, una instancia de GUI se registra a sí misma como rutina de tratamiento del suceso de acción del JButton. En la Figura B.9, implementamos la rutina de escucha haciendo que actionPerformed llame a setParam en la clase GUICanvas. Este ejemplo está simplificado por el hecho de que solo hay un JButton, por lo que cuando se invoca actionPerformed, sabemos lo que hay que hacer. Si GUI contuviera varios JButton y se registrara para recibir sucesos de todos esos JButton, entonces actionPerformed tendría que examinar el parámetro evt para determinar qué suceso de JButton hay que procesar; esto podría implicar una secuencia de comprobaciones if/else.<sup>4</sup> El parámetro evt, que en este caso es una referencia a ActionEvent, se pasa siempre a una rutina de tratamiento de sucesos. El suceso será específico del tipo de rutina de tratamiento (ActionEvent, WindowEvent, etc.), pero siempre será una subclase de AWTEvent.

Un suceso importante que hay que procesar es el suceso de cierre de ventana. Este suceso se genera cuando se cierra una aplicación pulsando en el botón ✕ situado en la esquina superior

```

1 // Tratar la pulsación del botón Draw
2 public void actionPerformed( ActionEvent evt )
3 {
4     try
5     {
6         theCanvas.setParams(
7             (String) theShape.getSelectedItem( ),
8             (String) theColor.getSelectedItem( ),
9             Integer.parseInt( theXCoor.getText( ) ),
10            Integer.parseInt( theYCoor.getText( ) ),
11            smallPic.isSelected( ) ? 0 :
12                mediumPic.isSelected( ) ? 1 : 2,
13            theFillBox.isSelected( ) );
14
15         theMessage.setText( "" );
16     }
17     catch( NumberFormatException e )
18     { theMessage.setText( "Incomplete input" ); }
19 }
```

Cuando el usuario pulsa un JButton se genera un suceso de acción; dicho suceso es tratado mediante actionPerformed.

Figura B.9 Código para tratar la pulsación del botón Draw de la Figura B.1.

<sup>4</sup> Una forma de hacer esto consiste en usar evt.getSource(), que devuelve una referencia al objeto que ha generado el suceso.

Cuando se cierra una aplicación se genera un suceso de cierre de ventana.

El un suceso de cierre de ventana se trata implementando la interfaz WindowListener.

CloseableFrame amplía JFrame e implementa WindowListener.

derecha de la ventana de la aplicación. Lamentablemente, este suceso se ignora de manera predeterminada, de modo que si no se proporciona una rutina de tratamiento de sucesos, el mecanismo normal para cerrar una aplicación no funcionará.

El cierre de la ventana es uno de los varios sucesos asociados con una interfaz `WindowListener`. Puesto que implementar la interfaz requiere que proporcionemos muchos métodos (que tienen una gran probabilidad de tener un cuerpo vacío), el curso de acción más razonable consiste en definir una clase que amplíe `JFrame` e implemente la interfaz `WindowListener`. Esta clase, `CloseableFrame`, se muestra en la Figura B.10. La rutina de tratamiento del suceso de cierre de ventana es sencilla de escribir: simplemente invoca a `System.exit`. Los otros métodos permanecen sin ninguna implementación especial. El constructor registra el hecho de que está dispuesto a aceptar el suceso de cierre de ventana. Ahora podemos utilizar `CloseableFrame` en lugar de `JFrame` en todas partes.

Observe que el código para `CloseableFrame` es engorroso; enseguida volveremos a echar un vistazo y encontraremos una manera de utilizar las clases internas anónimas.

```

1 // Un marco que se cierre ante un suceso de cierre de ventana
2
3 public class CloseableFrame extends JFrame
4     implements WindowListener
5 {
6     public CloseableFrame()
7     { addWindowListener( this ); }
8
9     public void windowClosing( WindowEvent event )
10    { System.exit( 0 ); }
11    public void windowClosed( WindowEvent event )
12    { }
13    public void windowDeiconified( WindowEvent event )
14    { }
15    public void windowIconified( WindowEvent event )
16    { }
17    public void windowActivated( WindowEvent event )
18    { }
19    public void windowDeactivated( WindowEvent event )
20    { }
21    public void windowOpened( WindowEvent event )
22    { }
23 }
```

**Figura B.10** Clase `CloseableFrame`: igual que `JFrame` pero se encarga de tratar el suceso de cierre de ventana.

La Figura B.11 proporciona una rutina `main` que se puede utilizar para iniciar la aplicación de la Figura B.1. Colocamos esa rutina en una clase separada, a la que denominamos `BasicGUI`. La clase `BasicGUI` amplía la clase `CloseableFrame`. El método `main` simplemente crea un `JFrame` en el que colocamos un objeto `GUI`. Después añadimos un objeto `GUI` sin nombre al panel de contenido de `JFrame` y ejecutamos el método `pack` del `JFrame`. El método `pack` simplemente comprime al máximo el tamaño del `JFrame`, teniendo en cuenta sus componentes constituyentes. El método `show` muestra el `JFrame`.

El método `pack` simplemente hace que el `JFrame` sea lo más compacto posible, dados sus componentes constituyentes. El método `show` muestra el `JFrame`.

### B.3.4 Tratamiento de sucesos: clases adaptadoras y clases internas anónimas

La clase `CloseableFrame` es un lío. Para ponerse a la escucha de un suceso `WindowEvent`, debemos declarar una clase que implemente la interfaz `WindowListener`, instanciar la clase y luego registrar dicho objeto ante el `CloseableFrame`. Puesto que la interfaz `WindowListener` tiene siete métodos, debemos implementar los siete, aunque solo estemos interesados en uno de ellos.

Se puede imaginar fácilmente lo liso que se puede volver el código cuando un programa de gran envergadura tenga que tratar numerosos sucesos. El problema es que cada estrategia de tratamiento de sucesos se corresponde con una nueva clase, y resultaría bastante absurdo tener numerosas clases con un montón de métodos que simplemente declaran {}.

Como resultado, el paquete `java.awt.event` define un conjunto de *clases adaptadoras de escucha*. Cada interfaz de escucha que disponga de más de un método se implementa mediante una clase adaptadora de escucha correspondiente, con cuerpos de métodos vacíos. Así, en lugar de proporcionar nosotros mismos esos cuerpos de métodos vacíos, podemos simplemente ampliar la clase adaptadora y sustituir los métodos en los que estemos interesados. En nuestro caso, necesitamos ampliar `WindowAdapter`. Esto nos da la implementación (errónea) de `CloseableFrame`, mostrada en la Figura B.12.

Las clases adaptadoras de escucha proporcionan implementaciones predeterminadas de todos los métodos de escucha.

```

1 class BasicGUI extends CloseableFrame
2 {
3     public static void main( String [ ] args )
4     {
5         JFrame f = new BasicGUI( );
6         f.setTitle( "GUI Demo" );
7
8         Container contentPane = f.getContentPane( );
9         contentPane.add( new GUI( ) );
10        f.pack( );
11        f.show( );
12    }
13 }
```

Figura B.11 La rutina `main` para la Figura B.1.

```

1 // Marco que se cierra con un suceso de cierre de ventana: (no funciona)
2 public class CloseableFrame extends JFrame, WindowAdapter
3 {
4     public CloseableFrame( )
5         { addWindowListener( this ); }
6
7     public void windowClosing( WindowEvent event )
8         { System.exit( 0 ); }
9 }

```

**Figura B.12** Clase CloseableFrame utilizando WindowAdapter. No funciona porque en Java no existe la herencia múltiple.

El código de la Figura B.12 falla porque la herencia múltiple de implementación es ilegal en Java. No obstante, este no es un problema serio, porque no necesitamos que el CloseableFrame sea el objeto que trate sus propios sucesos. En lugar de ello, podemos delegar esa tarea en un objeto función.

La Figura B.13 ilustra este enfoque. La clase ExitOnClose implementa la interfaz WindowListener ampliando WindowAdapter. Se crea una instancia de dicha clase y se registra como escucha de ventana del marco. ExitOnClose se declara como una clase interna en lugar de como una clase anidada. Esto le proporciona acceso a cualquiera de los miembros de instancia de CloseableFrame, en caso necesario. El modelo de tratamiento de sucesos es un ejemplo clásico del uso de objetos función y es la razón de que las clases internas se consideraran un añadido esencial al lenguaje (recuerde que las clases internas y el nuevo modelo de sucesos aparecieron simultáneamente en Java 1.1).

La Figura B.14 muestra la continuación lógica utilizando clases interna anónimas. Aquí estamos añadiendo un WindowListener y explicando, básicamente en la siguiente línea de código, lo que hace el WindowListener. Este es un uso típico de las clases internas anónimas. Todo el lio de llaves,

```

1 // Marco que se cierra con un suceso de cierre de ventana: (funciona)
2 public class CloseableFrame extends JFrame
3 {
4     public CloseableFrame( )
5         { addWindowListener( new ExitOnClose( ) ); }
6
7     private class ExitOnClose extends WindowAdapter
8     {
9         public void windowClosing( WindowEvent event )
10            { System.exit( 0 ); }
11     }
12 }

```

**Figura B.13** Clase CloseableFrame que utiliza WindowAdapter y una clase interna.

```

1 // Marco que se cierra con un suceso de cierre de ventana: (funciona)
2 public class CloseableFrame extends JFrame
3 {
4     public CloseableFrame( )
5     {
6         addWindowListener( new WindowAdapter( )
7             {
8                 public void windowClosing( WindowEvent event )
9                 { System.exit( 0 ); }
10            }
11        );
12    }
13 }

```

Figura B.14 Clase `CloseableFrame` que utiliza `WindowAdapter` y una clase interna anónima.

paréntesis y puntos y comas es horrible, pero los lectores experimentados de código Java se saltan esos detalles sintácticos y pueden ver fácilmente qué es lo que está haciendo el código de tratamiento de sucesos. La principal ventaja de esto es que si existe un montón de métodos de tratamiento de sucesos, esos métodos no tienen por qué estar dispersos en clases de nivel superior, sino que pueden colocarse en su lugar cerca de los objetos en los que esos sucesos se originan.

### B.3.5 Resumen: encajando la piezas

He aquí el resumen sobre cómo crear una aplicación con interfaz GUI. Coloque la funcionalidad GUI en una clase que amplíe `JPanel`. Para esa clase, haga lo siguiente:

1. Decida los elementos de entrada y los elementos de salida de texto básicos. Si se utilizan los mismo elementos dos veces, defina una clase adicional para almacenar la funcionalidad común y aplique estos principios a dicha clase.
2. Si se utilizan gráficos, defina una clase adicional que amplíe `JPanel`. Esta clase debe proporcionar un método `paintComponent` y un método público que pueda ser utilizado por el contenedor para comunicarse con ella. También es posible que necesite proporcionar un constructor.
3. Seleccione un tipo de disposición y ejecute un comando `setLayout`.
4. Añada componentes a la GUI utilizando `add`.
5. Trate los sucesos. La forma más simple de hacer esto es emplear un `JButton` y atrapar la pulsación del botón mediante `actionPerformed`.

Una vez escrita una clase GUI, una aplicación define una clase que amplía `CloseableFrame` con una rutina `main`. La rutina `main` simplemente crea una instancia de esta clase ampliada de marco, coloca el panel GUI dentro del panel de contenidos del marco y ejecuta un comando `pack` y un comando `show` sobre el marco.

### B.3.6 ¿Es esto todo lo que necesito saber acerca de Swing?

Lo que hemos descrito hasta ahora funcionará bien para interfaces de usuario simples y representa una mejora respecto a las aplicaciones basadas en consola. Pero existen complicaciones significativas con las que un programador profesional de aplicaciones se verá obligado a tratar.

Resultará raro que el gestor de disposición le satisfaga completamente. A menudo necesitará jugar con la interfaz añadiendo subpaneles adicionales. Como ayuda, Swing define elementos como espaciadores, barras, etc. que permiten posicionar los elementos de manera más precisa, junto con gestores de disposición elaborados. Utilizar estos elementos es bastante complicado.

Otros componentes Swing incluyen los deslizadores, las barras de progreso, el desplazamiento (que puede añadirse a cualquier `JComponent`), los campos de texto de contraseña, los seleccionadores de archivo, los paneles de opciones y los cuadros de diálogo, las estructuras de árbol (como las que se pueden ver en el Administrador de archivos de los sistemas Windows), tablas, etc. Swing también soporta la adquisición y visualización de imágenes. Además, a menudo es necesario saber acerca de fuentes, colores y el entorno de pantalla con el que uno está trabajando.

Por si fuera poco, está la importante cuestión de qué ocurre si se produce un suceso mientras nos encontramos en una rutina de tratamiento de sucesos. Resulta que los sucesos se ponen en cola. Sin embargo, si nos quedamos atrapados en una rutina de tratamiento de sucesos durante mucho tiempo, puede que la aplicación parezca no responder; todos hemos tenido oportunidad de ver este tipo de comportamiento en el código de alguna aplicación. Por ejemplo, si el código de tratamiento de las pulsaciones de botón tiene un bucle infinito, seremos incapaces de cerrar una ventana. Para resolver este problema, los programadores típicos utilizan una técnica conocida con el nombre de *programación multihilo*, que abre la puerta a un campo enormemente complicado.

## Resumen

---

En este apéndice hemos examinado los fundamentos del paquete Swing, que permite la programación de interfaces GUI. Esto hace que el programa parezca mucho más profesional que con la simple E/S de terminal.

Las aplicaciones GUI difieren de las aplicaciones de E/S de terminal en que están dirigidas por sucesos. Para diseñar una interfaz GUI, escribimos una clase. Debemos decidir sobre los elementos de entrada y de salida básicos, seleccionar una disposición y ejecutar un comando `setLayout`, añadir componentes a la GUI utilizando `add` y llevar a cabo el tratamiento de sucesos. Todo esto forma parte de la clase. A partir de Java 1.1, el tratamiento de sucesos se realiza mediante escuchas de sucesos.

Una vez escrita esta clase, una aplicación define una clase que amplía `JFrame` con una rutina `main` y otra rutina para el tratamiento de sucesos. La rutina de tratamiento de sucesos procesa el suceso de cierre de ventana. La forma más fácil de hacer esto es utilizando la clase `CloseableFrame` de la Figura B.14. La rutina `main` simplemente crea una instancia de esta clase ampliada de marco, coloca una instancia de la clase (cuyo constructor probablemente cree el panel GUI) dentro del panel de contenidos del marco y ejecuta un comando `pack` y un comando `show` sobre el marco.

Aquí solo hemos visto los componentes básicos de Swing. Swing es un tema al que están dedicados libros completos.



## Conceptos clave

**Abstract Window Toolkit (AWT)** Un conjunto de herramientas GUI que se suministra con todos los sistemas Java. Proporciona las clases básicas para permitir las interfaces de usuario. (913)

**ActionEvent** Un suceso generado cuando un usuario pulsa un JButton, pulsa *Intro* en un JTextField o selecciona una opción en una JList o un JMenuItem. Debe ser tratado por un método actionPerformed en una clase que implemente la interfaz ActionListener. (931)

**actionPerformed** Un método utilizado para tratar sucesos de acción. (931)

**AWTEvent** Un objeto que almacena información acerca de un suceso. (931)

**BorderLayout** El gestor de disposición predeterminado para los objetos de la jerarquía Window. Se utiliza para disponer los objetos de un contenedor colocando los componentes en una de cinco ubicaciones ("North", "South", "East", "West", "Center"). (925)

**ButtonGroup** Un objeto utilizado para agrupar una colección de objetos botón y garantizar que solo uno de ellos pueda estar *on* en cualquier momento determinado. (922)

**clase adaptadora de escucha** Proporciona implementaciones predeterminadas para una interfaz de escucha que disponga de más de un método. (933)

**Component** Una clase abstracta que actúa como superclase de muchos objetos AWT. Representa algo que tiene una posición y un tamaño y que puede ser pintado en la pantalla, pudiendo también recibir sucesos de entrada. (916)

**Container** La superclase abstracta que representa a todos los componentes que pueden albergar a otros componentes. Normalmente tiene un gestor de disposición asociado. (916)

**FlowLayout** Un gestor de disposición que es el predeterminado para JPanel. Utilizado para disponer los objetos de un contenedor añadiendo los componentes en fila de izquierda a derecha. Cuando no queda espacio en una fila se forma otra fila nueva. (924)

**gestor de disposición** Un objeto auxiliar que dispone automáticamente los componentes de un contenedor. (924)

**gestor de disposición null** Un gestor de disposición utilizado para llevar a cabo un posicionamiento preciso. (927)

**Graphics** Una clase abstracta que define varios métodos que se pueden emplear para dibujar formas geométricas. (928)

**interfaz** ActionListener Una interfaz utilizada para tratar sucesos de acción. Contiene el método abstracto actionPerformed. (929)

**interfaz gráfica de usuario (GUI)** Alternativa moderna a la E/S de terminal, que permite al programa comunicarse con el usuario a través de botones, casillas de verificación, campos de texto, listas de opciones, menús y el ratón. (913)

**interfaz WindowListener** Una interfaz utilizada para especificar el tratamiento de sucesos de ventana, como el cierre de ventana. (932)

- JButton** Un componente utilizado para crear un botón etiquetado. Cuando se pulsa el botón se genera un suceso de acción. (919)
- JCheckBox** Un componente que tiene un estado *on* y un estado *off*. (922)
- JComboBox** Un componente utilizado para seleccionar una única cadena de caracteres a través de una lista emergente de opciones. (921)
- JComponent** Una clase abstracta que actúa como superclase de los objetos Swing ligeros. (917)
- JDialog** Una ventana de nivel superior utilizada para crear cuadros de diálogo. (917)
- JFrame** Una ventana de nivel superior que tiene un borde y que también puede tener una **JMenuBar** asociada. (917)
- JLabel** Un componente que se utiliza para etiquetar a otros componentes, como **JComboBox**, **JList**, **JTextField** o **JPanel**. (919)
- JList** Un componente que permite efectuar una selección en una lista desplazable de caracteres. Puede permitir que se elija un solo elemento o varios, pero ocupa más espacio de pantalla que **JComboBox**. (921)
- JPanel** Un contenedor utilizado para almacenar una colección de objetos, pero que no crea bordes. También utilizado para lienzos. (917)
- JTextArea** Un componente que presenta al usuario varias líneas de texto. (923)
- JTextField** Un componente que presenta al usuario una única línea de texto. (923)
- JWindow** Una ventana de nivel superior que no tiene bordes. (917)
- lienzo** Un área rectangular en blanco de la pantalla, en la que una aplicación puede dibujar y recibir entradas procedentes del usuario, en forma de sucesos de teclado y de ratón. En Swing, esto se implementa ampliando **JPanel**. (923)
- pack** Un método empleado para comprimir un **JFrame** al tamaño más pequeño posible, dados sus componentes constituyentes. (933)
- paintComponent** Un método utilizado para dibujar en un componente. Normalmente sustituido por clases que amplían **JPanel**. (928)
- repaint** Un método utilizado para borrar y volver a dibujar un componente. (929)
- setLayout** Un método que asocia un gestor de disposición con un contenedor. (924)
- show** Un método que hace que un componente sea visible. (933)
- suceso** Generado por el sistema operativo para varias situaciones, como por ejemplo operaciones de entrada. El suceso generado por el SO es pasado a Java. (929)
- WindowAdapter** Una clase que proporciona implementaciones predeterminadas de la interfaz **WindowListener**. (933)



## Errores comunes

1. Olvidarse de configurar un gestor de disposición es uno de los errores más comunes. Si se olvida de hacerlo, se le asignará el predeterminado. Sin embargo, puede que eso no sea lo que deseemos.

- 2 El gestor de disposición debe aparecer antes de las llamadas a add.
- 3 Otro error común es aplicar add o configurar un gestor de disposición en el contenedor equivocado. Por ejemplo, en un contenedor que contenga paneles, aplicar el método add sin especificar el panel, implica que el método add se aplica al contenedor principal.
- 4 Si falta un argumento String en el método add de BorderLayout se utiliza "Center" como opción predeterminada. Un error común consiste en especificarlo sin respetar las mayúsculas y minúsculas, como por ejemplo en "north". Los cinco argumentos válidos son "North", "South", "East", "West" y "Center". En Java 1.1, si el objeto String es el segundo parámetro, una excepción de tiempo de ejecución permitirá detectar el error. Si utiliza el antiguo estilo de llamada, en el que el objeto String va primero, ese error podría no ser detectado.
- 5 Hace falta código especial para procesar el suceso de cierre de ventana.



## Internet

Todo el código mostrado en este apéndice está disponible en:

**BorderTest.java**

Ilustración simple del BorderLayout mostrado en la Figura B.6.

**BasicGUIL.java**

El ejemplo principal para la aplicación GUI empleada en este capítulo, con CloseableFrame de la Figura B.14.



## Ejercicios

### EN RESUMEN

- B.1** ¿Qué es una GUI?
- B.2** Enumere las distintas clases JComponent que se pueden utilizar para la entrada de datos a través de la GUI.
- B.3** Describa la diferencia entre componentes pesados y ligeros y proporcione ejemplos de cada uno de esos tipos.
- B.4** ¿Cuáles son las diferencias entre componentes JList y JComboBox?
- B.5** ¿Para qué se utiliza un ButtonGroup?
- B.6** Explique los pasos requeridos para diseñar una GUI.
- B.7** Explique cómo disponen los componentes los gestores FlowLayout, BorderLayout y null.
- B.8** Describa los pasos necesarios para incluir un componente gráfico en un JPanel.
- B.9** ¿Cuál es el comportamiento predeterminado cuando se produce un suceso? ¿Cómo se cambia ese comportamiento predeterminado?
- B.10** ¿Qué sucesos generan un ActionEvent?

**B.11** ¿Cómo se trata el suceso de cierre de ventana?

### EN LA PRÁCTICA

- B.12** Podemos escribir un método `paintComponent` para cualquier componente. Indique lo que sucede cuando se pinta un círculo en la clase `GUI` en lugar de en su propio lienzo.
- B.13** Escriba el código necesario para tratar la pulsación de la tecla `Intro` en el campo de texto relativo a la coordenada `y` en la clase `GUI`.
- B.14** Añada un valor predeterminado de `(0,0)` para las coordenadas de una forma geométrica en la clase `GUI`.

### PROYECTOS DE PROGRAMACIÓN

- B.15** Escriba un programa que se pueda utilizar para introducir dos fechas y que proporcione como salida el número de días existentes entre ellas. Utilice la clase `Date` del Ejercicio 3.29.
- B.16** Escriba un programa que le permita dibujar líneas dentro de un lienzo utilizando el ratón. Un clic hace que comience el dibujo de la línea; un segundo clic hace que termine la línea. Se pueden dibujar varias líneas en el lienzo. Para hacer esto, amplíe la clase `JPanel` y trate los sucesos de ratón implementando `MouseListener`. Debe mantener un `ArrayList` donde se almacene el conjunto de líneas que se han dibujado y utilizarlo para guiar la actuación de `paintComponent`. Añada un botón para borrar el lienzo.
- B.17** Escriba una aplicación que contenga dos objetos `GUI`. Cuando se produzcan acciones en uno de los objetos `GUI`, el otro objeto `GUI` tiene que guardar su antiguo estado. Tendrá que añadir un método `copyState` a la clase `GUI` que se encargue de copiar todos los estados de los campos de `GUI` y redibujar el lienzo.
- B.18** Escriba un programa que contenga un único lienzo y un conjunto de diez componentes de entrada a través de la `GUI`, cada uno de los cuales debe especificar una forma, un color, unas coordenadas y un tamaño, junto con una casilla de verificación que indique que el componente está activo. Después dibuje todos los componentes de entrada en un lienzo. Represente el componente de entrada a través de la `GUI` utilizando una clase con funciones accesorias. El programa principal debe tener una matriz de estos componentes de entrada, más el lienzo.



### Referencias

Además del conjunto de referencias estándar del Capítulo 1, podrá encontrar un tutorial completo sobre Swing en el libro de 950 páginas [1].

1. K. Walrath y M. Campione, *The JFC Swing Tutorial*, Addison-Wesley, Reading, MA, 1999.

# Apéndice C

## Operadores bit a bit

Java proporciona *operadores bit a bit* para la manipulación de cada uno de los bits que componen un valor entero. Este proceso permite empaquetar varios objetos booleanos dentro de un tipo entero. Esos operadores son: ~ (complemento unario), << y >> (desplazamiento a la izquierda y a la derecha), & (AND bit a bit), ^ (OR exclusivo bit a bit), | (OR bit a bit) y los operadores de asignación correspondientes a todos estos operadores, salvo el complemento unario. La Figura C.1 ilustra el resultado de aplicar estos operadores. Observe que >> se considera un desplazamiento de bit con signo: el valor que se inserta en el bit de mayor peso puede depender de la extensión de signo. >>> se considera un desplazamiento de bits sin signo y se emplea para garantizar que el bit de mayor peso se complete con el valor 0.

La precedencia y asociatividad de los operadores bit a bit es en cierto modo arbitraria. Al trabajar con estos operadores, debería utilizar paréntesis.

La Figura C.2 muestra cómo se utilizan los operadores bit a bit para empaquetar información en un entero de 16 bits. Dicha información es mantenida por una universidad típica por una amplia variedad de razones, incluyendo normas estatales y federales. Muchos elementos requieren respuestas simples de tipo sí/no y son, por tanto, representables mediante un único bit desde el punto de vista lógico. Como muestra la Figura C.2, se utilizan 10 bits para representar 10 categorías. Un profesor puede tener uno de cuatro rangos posibles (ayudante, asociado, titular y sin salario), por lo que se requieren dos bits. Los 4 bits restantes se utilizan para representar una de 16 posibles facultades dentro de la universidad.

Java proporciona operadores bit a bit para la manipulación de cada uno de los bits que componen un valor entero. Este proceso permite empaquetar varios objetos booleanos dentro de un tipo entero.

```
//Suponga que los valores int son de 16 bits
int a = 3737;      // 0000111010011001
int b = a << 1;    // 0001110100110010
int c = a >> 2;    // 0000001110100110
int d = 1 << 15;   // 1000000000000000
int e = a | b;     // 000111110111011
int f = a & b;     // 0000110000010000
int g = a ^ b;     // 0001001110101011
int h = ~g;
```

Figura C.1 Ejemplos de operadores bit a bit.

```

1 // Campos que definen el perfil de un profesor.
2 static int SEX = 0x0001;      // On si es mujer
3 static int MINORITY = 0x0002; // On si pertenece a una minoría
4 static int VETERAN = 0x0004;  // On si es veterano
5 static int DISABLED = 0x0008; // On si es discapacitado
6 static int US_CITIZEN = 0x0010; // On si es ciudadano de EE.UU.
7 static int DOCTORATE = 0x0020; // On si tiene un doctorado
8 static int TENURED = 0x0040;  // On si tiene salario
9 static int TWELVE_MON = 0x0080; // On si tiene contrato de 12 meses
10 static int VISITOR = 0x0100; // On si es visitante
11 static int CAMPUS = 0x0200; // On si está en el campus principal
12
13 static int RANK = 0xc000; // 2 bits para representar el rango
14 static int ASSISTANT = 0x0400; // Profesor ayudante
15 static int ASSOCIATE = 0x0800; // Profesor asociado
16 static int FULL = 0xc000; // Profesor titular
17
18 static int COLLEGE = 0xf000; // Representa 16 facultades
19 ...
20 static int ART_SCIENCE = 0x3000; // Artes & Ciencias: facultad 3
21 ...
22
23 // Más adelante en el método se inicializan los campos apropiados.
24 tim = ART_SCIENCE | ASSOCIATE | CAMPUS | TENURED |
25           TWELVE_MON | DOCTORATE | US_CITIZEN;
26
27 // Promocionar a tim a Profesor titular
28 tim &= ~RANK; // Desactivar todos los campos de rango
29 tim |= FULL; // Activar campos de rango

```

Figura C.2 Bits empaquetados para definir el perfil de los profesores.

Las líneas 24 y 25 muestran cómo se representaría a un profesor concreto, `tim`. `Tim` es un profesor asociado sin salario en la Facultad de Artes y Ciencias. Tiene un título doctoral, es ciudadano de Estados Unidos y trabaja en el campus principal de la universidad. No es miembro de ninguna minoría, no es discapacitado ni tampoco es veterano. Tiene un contrato de 12 meses. Por tanto, el patrón de bits para `tim` está dado por

0011 10 1 0 1 1 1 1 0 0 0 0

es decir, `0x3af0`. El patrón de bits se forma aplicando el operador OR a los campos apropiados.

Las líneas 28 y 29 muestran la lógica utilizada cuando Tim es merecidamente promocionado a profesor titular. La categoría que indica el rango, `RANK`, tiene dos bits de rango puestos a 1 y todos los demás bits puestos a 0; es decir

0000 11 0 0 0 0 0 0 0 0 0 0

El complemento, `~RANK`, será por tanto

1111 00 1 1 1 1 1 1 1 1 1 1

Aplicando una operación AND bit a bit entre este patrón y la configuración actual de `tim` se desactivan los bits de rango de `tim`, obteniendo

0011 00 1 0 1 1 1 1 0 0 0 0

El resultado del operador OR bit a bit en la línea 29 hace así de `tim` profesor titular sin modificar ningún otro bit, lo que nos da

0011 11 1 0 1 1 1 1 0 0 0 0

Sabemos que Tim tiene salario porque `tim&TENURED` da un resultado distinto de cero. También podemos averiguar que Tim está en la Facultad #3 desplazando hacia la derecha 12 bits y examinando luego los 4 bits de menor peso resultantes. Observe que hacen faltar paréntesis. La expresión es `(tim>>12)&0xf`.



# Índice

– (doble menos), 10  
- (menos), 9  
! (signo de exclamación), 11, 22  
!= (signo e exclamación e igual), 11, 12  
% (porcentaje), 9  
%= (porcentaje e igual), 911  
& (ampersand), 11, 23  
&& (doble ampersand), 11-12, 22  
\* (asterisco), 9  
\*= (asterisco e igualdad), 8

## A

Abstract Window Toolkit (AWT), 913-915  
componentes de E/S, 919-924

contenedores de nivel superior, 917-918  
definición, 937  
gráficos, 928-929  
principios básicos, 924-936  
Swing, objetos básicos, 915-916  
tratamiento de sucesos, 929-935  
**AbstractCollection**, clase  
definición, 579  
introducción, 567-571  
acceso con visibilidad de paquete, 73, 94, 97  
accesores, 76-78, 97  
Ackermann, función de, 898, 906  
Ackermann, función de una sola variable, 898  
acoplamiento  
    dinámico, 115  
    estático, 164, 167  
**ActionEvent**, 937  
**ActionListener**, interfaz, 929, 937  
**actionPerformed**, 931, 937  
adaptadora, clase, 141, 167  
adaptadores  
    patrones, 144-145  
    tratamiento de sucesos, 929-935  
**advance**, rutina, 665  
agregado, 37, 59  
agrupamiento primario, 771-772, 792  
agrupamiento secundario, 785, 788, 792  
ajedrez por computadora, 425-426  
*Deep Blue*, programa, 426  
nivel de gran maestro, 425  
posiciones terminales y, 425  
aleatorización, 383-406. *Véase también* números aleatorios  
    algoritmos, 395-398  
    conceptos clave, 402-403  
    ejercicios y proyectos, 404-406  
    en Internet, 403  
    errores comunes, 403  
    generación de permutación aleatoria, 394-395  
    prueba de primalidad, 398-401  
    resumen, 402  
    selección rápida, 397

- aleatorizado, algoritmo, 384, 395–398, 402  
**álgebra booleana**, concepto, 11  
**algoritmo**  
 aleatorizado, 384, 395–398, 402  
 algoritmos de cifrado y descifrado, 311–312  
 algoritmos inteligentes de unión, 892–894  
 análisis de precedencia de operadores, 275, 444–448, 459  
 caso peor lineal, 372  
 de Bellman-Ford, 541  
 de Dijkstra, 535–538, 551, 870–874  
 de división, 399, 402  
 de Euclides, 308  
 de fuerza bruta, 191  
 de Kruskal, 884, 905  
 de ordenación basado en comparaciones, 343, 374  
 de ordenación por mezcla, 352–354  
 de ordenación rápida, 355–358  
 del camino más corto, 520–521  
 en línea, 881, 905  
 fuera de línea, 881, 905  
 genéricos y la API de Colecciones, 238–243  
 lineal, 195–198  
 montículo binario, necesidad de nuevos algoritmos, 816  
 para el problema de Josefo, 500–502  
 rápido de búsqueda, 889–890, 905  
 rápido de unión, 890–892, 905  
 simple, montículo binario, 817–818  
 un único origen, 522  
 union/find, 880, 899–904, 905  
 voraz, 322, 331  
**algoritmo rápido de unión**, 890–892  
 compresión de caminos, 894–895  
 definición, 905  
 inteligente, 892–894  
**algoritmo union/find**  
 análisis de, 899–904  
 definición, 880, 905  
**algoritmos de ordenación**, 341–381  
 conceptos clave, 374  
 cota inferior, 372–373  
 detección de duplicados, 342  
 ejercicios y proyectos, 375–380  
 en Internet, 375  
 externa, 341  
 importancia, 341–343  
 introducción, 341  
 por inserción, 343–347  
 por mezcla, 350–354  
 rápida, 355–370  
 Shell, 347–350  
 preliminares, 343  
 resumen, 373–374  
**algoritmos genéricos**, 238–243  
 búsqueda binaria, 242  
**Collections**, clase, 239–241  
**Comparator**, objetos función, 238  
 introducción, 238  
 ordenación, 242–243  
 alias, 81, 97, 653  
 almacenamiento en caché, 367  
 almacenamiento en caché del código hash, 768  
 altura de un nodo, 642, 671  
 amortización, 584  
 análisis de algoritmos, 185–224  
 comprobación, 209–211  
 conceptos clave, 212–213  
 cuadrático, 201–202  
 de fuerza bruta, 191  
 ejercicios y proyectos, 214–223  
 en Internet, 213–214  
 errores comunes, 213  
 introducción, 185–189  
 limitaciones del análisis O mayúscula, 211  
 lineal, 195–198  
 notación O mayúscula y, 198–202  
 O ( $N^2$ ), 194  
 O ( $N^3$ ), 191–194  
 problema de la búsqueda estática y, 204–209  
 problema de la suma máxima de una  
 subsecuencia contigua, 191  
 resumen, 211  
 subcuadrático, 199  
 tiempos de ejecución, ejemplos, 189–190  
**análisis**  
 de precedencia de operadores, 275, 444–448, 459  
 del camino crítico, 548–550, 551  
 del caso peor, 211  
 del caso promedio, 211  
 intuitivo, 315  
 léxico, 435, 458  
 ancestro, 642, 671  
 ancestro propio, 642, 671  
 API de Colecciones, 225–286

- algoritmos genéricos, 238–243  
 clase `LinkedList`, implementación, 621–635  
 colas con prioridad y, 270–273  
 colas y, 256–257  
 conceptos clave, 275–276  
 conjuntos y, 257–264  
 contenedores e iteradores, 232–238  
 ejercicios y proyectos, 277–286  
 en Internet, 277  
 errores comunes, 276  
 introducción, 225–227  
`List`, interfaz, 244–253  
 mapas, 264–270  
 patrón iterador, 227–232  
 pilas y, 254–256  
 resumen, 274–275  
`TreeSet` y `TreeMap`, implementación de las clases, 726–746  
 vistas, 273–274  
 aplicaciones numéricas, 305–312  
   aritmética modular, 305–306  
   criptosistema RSA, 309–312  
   exponenciación modular, 305, 306–307  
   inversa multiplicativa, 305, 307–309  
   máximo común divisor, 305, 307–309  
   prueba de primalidad, 305, 311  
 árbol binario completo, 798, 822  
 árbol completo, 465, 489  
 árbol de búsqueda binaria, 677–761  
   AA. Véase árboles AA  
   análisis de las operaciones con, 692–696  
   árboles AVL. Véase árboles AVL  
   árboles M-arios, 748, 754  
   árboles rojo-negro. Véase árboles rojo-negro  
   árboles-B, 747–752, 753  
   clases `TreeSet` y `TreeMap` clases, implementación, 726–746  
   conceptos clave, 753–754  
   definición, 275, 677  
   ejercicios y proyectos, 755–759  
   en Internet, 755  
   equilibrado, 696, 754  
   errores comunes, 754–755  
   estadísticas de orden, 687–692  
   implementación Java, 680–687  
   introducción, 677–678  
   operaciones, 678–680  
   propiedad de orden en, 678  
   resumen, 752–753  
   y tablas hash, 789–791  
 árbol de expresión, 457, 459, 649  
 árbol enhebrado, 727  
 árbol splay de tipo abajo-arriba  
   análisis, 838–843  
   definición, 853  
   introducción, 835–837  
 árbol-B+, 748, 760  
 árboles, 641–675  
   alias, 653  
   árboles-B, 747–752, 753  
   AVL. Véase árboles AVL  
   binarios, 649–654  
   binomiales, 893  
   conceptos clave, 671–672  
   de codificación de Huffman, 649–650  
   de expresión, 649  
   definición, 331, 641, 671  
   ejercicios y proyectos, 673–675  
   en Internet, 672  
   enhebrados, 727  
   errores comunes, 672  
   generales, 641–648  
   implementación Java, 646–648  
   implementación, 643  
   M-ario, 748, 754  
   previsualización de, 299–300  
   recorrido. Véase recorrido de árboles y clases  
     iteradoras  
     recursión y, 654–656  
     resumen, 671  
     rojo-negro. Véase árboles rojo-negro  
     sistemas de archivos, aplicación, 644–648  
 árboles AA, 718–726, 753  
   borrado, 722–723  
   definición, 753  
   implementación Java y, 723–726  
   inserción, 719–722  
   introducción, 718–719  
 árboles AVL, 696–704  
   definición, 696, 753  
   introducción, 696  
   propiedades, 696–699  
   resumen, 704  
   rotación doble, 701–704  
   rotación simple, 699–701  
 árboles mínimos de recubrimiento, 883–886, 905

- 
- árboles rojo-negro, 704–718, 754  
 arriba-abajo, 707–709  
 borrado arriba-abajo, 716–718  
 definición, 754  
 implementación Java, 709–716  
 inserción arriba-abajo, 705–707  
 introducción, 704–705
- árboles splay, 831–856  
 análisis del splaying abajo-arriba, 838–843  
 autoajuste y análisis amortizado, 831–835  
 básico de tipo abajo-arriba, 835–837  
 conceptos clave, 853  
 de tipo arriba-abajo, 843–846  
 ejercicios y proyectos, 854–855  
 en Internet, 854  
 errores comunes, 854  
 implementación de los árboles splay arriba-abajo, 846–852  
 introducción, 831  
 operaciones básicas, 837–838  
 resumen, 853  
 y otros árboles de búsqueda, 852  
 AA. Véase árboles AA
- archivos secuenciales de E/S, 56–59
- arcos, 515
- argumentos de la línea de comandos  
 en la invocación de la Máquina Virtual, 4  
 en matrices, 46, 59
- argumentos reales, 18–19
- aritmética modular, 305–306
- aritméticos binarios, operadores, 9, 21
- ArrayList*, iteradores, 572–578
- ArrayList*. Véase también clases internas e implementación de *ArrayList*  
 implementación con un iterador, 572–578  
 introducción, 41–43, 249  
 y *LinkedList*, 250–251
- ArrayStack*, clase, 586–587
- arriba-abajo, árboles rojo-negro, 707–709
- asignación de matrices, 37–40
- asignación, operadores de, 8–9, 21
- asociatividad usada para romper empates de precedencia, 447
- atomicidad, principio, 70
- autoajuste y análisis amortizado, 831–835, 853
- autoboxing/unboxing*, 143–144
- autodecremento (–), operador, 10, 21
- autoincremento (++), operador, 10, 21
- AWTEvent*, 931, 937
- B**
- base, 289, 331
- Bellman-Ford, algoritmo de, 541, 551
- BigInteger*, 78–80
- BigRational*, clase, 86–90
- binario, árbol, 299, 649–654, 672. Véase también árboles de búsqueda binaria
- BinaryNode*, clase, 650–651, 655, 681
- BinaryOp*, rutina, 454
- BinarySearchTree*, 680
- BinarySearchTreeWithRank*, clase, 689
- BinaryTree*, clase, 650–652, 654
- binomial, árbol, 893
- bit a bit, operadores, 941–943
- bloques  
 catch, 47  
 definición, 20  
 if, instrucción, 12–13  
 try, 47
- boolean*, tipo primitivo, 6
- BorderLayout*, 925–926, 937
- borrado  
 árboles AA, 722–723  
 arriba-abajo en árboles rojo-negro, 716–718  
 de tipos, 152, 167  
 perezoso, 718, 754, 770, 792
- bosque, 890, 905
- break etiquetada, instrucción, 16, 20
- break, instrucción, 15–16, 20
- bucles  
 anidamiento, 14  
 control de flujo y, 11  
 for avanzado, 46–47, 59  
 formas, 13
- BufferedReader*, 137
- buildHeap*, operación, 807–812, 823
- buildTree*, 501–502
- búsqueda binaria  
 algoritmos genéricos y, 242  
 definición, 212  
 en búsquedas estáticas, 205–207  
 recursiva, 300–302
- búsqueda en anchura, 530, 551, 667
- búsqueda por interpolación, 207–209, 212
- búsqueda secuencial, 205, 212

`ButtonGroup`, 922, 937  
`byte`, tipo primitivo, 6

## C

cabecera de método, 18, 20  
 caché, 832  
 cadenas de caracteres (`String`), 34–37  
     comparación de, 35  
     concatenación de, 35, 59  
     conceptos básicos de la manipulación de, 34  
     conversión de otros tipos a, 36  
     definición, 61  
     inmutable, 34, 60  
     introducción, 34  
     longitud, 36  
     otros métodos `String`, 36  
 calculadora usada en pilas, 444–458  
     árboles de expresión y, 457–458  
     conversión de notación infija a postfija, 445–448  
     implementación, 448–457  
     introducción, 444  
     máquinas postfijas, 445  
`CallSim`, constructor, 508  
 camino. Véase también grafos y caminos  
     algoritmos del camino más corto, 520–521  
     análisis del camino crítico, 548–550, 551  
     definición, 516, 551  
     longitud no ponderada de un camino, 516, 552  
     longitud ponderada de un camino, 516, 533, 552  
     longitud, 300, 516, 552  
     por la mitad, 907  
     simple, 516, 551  
 campos  
     definición, 97  
     en programación orientada a objetos, 71–72  
     estáticos, 82–85  
 Carmichael, números de, 400  
 caso base, 292, 331  
 catch, bloque, 47, 59  
 cero parámetros, constructor, 116–117  
 char, tipo primitivo, 6  
 checkBalance, rutina, 441, 442  
 chooseMove, método, 421–425  
 ciclo, 516, 551  
     de coste negativo, 541, 552  
     de coste positivo, 549, 552  
 cifrado, 309, 331

circularmente enlazadas, listas, 618–621, 635  
 clase, 69–106  
     `AbstractCollection`, 567–571, 579  
     adaptadora, 141, 167  
     anidada y functor, 162, 163, 167  
     anónima, 161–162, 167  
     `ArrayList`, 586–587  
     `BigInteger`, 78–80  
     `BigRational`, ejemplo e implementación de, 86–90  
     campos, 71–73, 82–85, 97  
     clase en bruto, 152, 167  
     `Collections`, 239–241, 275  
     conceptos clave, 97–99  
     constructores adicionales, 80–85  
     creación de nuevas clases y herencia, 108–113  
     de equivalencia de un elemento, 880–881, 905  
     declaración, parámetros de tipo y, 148  
     definición, 97  
     definición de nuevos tipos objetos, 6  
     ejercicios y proyectos, 100–106  
     en Internet, 100  
     errores comunes, 99  
     especificación, 73, 98  
     `EvalTokenizer`, clase anidada, 451  
     `Evaluator`, 448, 450  
     functor y, 161–163  
     `Graph`, 524, 526  
     hoja, 118, 167  
     `IntCell`, 71–72  
     interfaz y, 134  
     interna, 232  
     `java.math.BigInteger`, ejemplo, 78–80  
     javadoc y, 73–74  
     jerarquía de herencia y, 124–128  
     local y functor, 160–161, 167  
     métodos, 71, 74–78  
     métodos final y, 117–118, 167, 168  
     miembros, 71–72, 116  
     miembros protegidos, 116, 168  
     `Object`, 134–135  
     objeto como instancia de, 71  
     paquetes, 90–94  
     patrón de diseño compuesto (par), 94–95, 99  
     programación orientada a objetos y, 69–71  
     puente, 137  
     relaciones subclase/superclase, 115, 169  
     resumen, 95–97

- dase (cont.)  
 Tokenizer, 433–434, 451  
 un ejemplo simple de, 71–73  
 Vertex, 523  
 visibilidad de paquete, 94
- dase abstracta  
 definición, 127, 167  
 en jerarquías de herencia, 124–128  
 interfaz como, 134  
 parámetros de sustitución y, 124, 126  
 resumen, 128
- dase anónima  
 definición, 167  
 functor y, 161–162
- clase base  
 clase derivada y, 112–113  
 daúsula extends y, 112  
 definición, 167  
 implementación predeterminada y, 127, 128  
 relación *ES-UNy*, 107  
 relación *TIENE-UNy*, 107
- dase derivada  
 compatible en cuanto a tipo, 115  
 con clase base, 108, 112, 113  
 definición, 167  
 descripción, 112–113  
 en la herencia, 108
- dases adaptadoras escucha, 933, 937
- dases anidadas  
 functor y, 162, 168  
 iteradores y, 561–563  
 y clases internas, 564
- dases de almacenamiento, 20
- dases genéricas  
 definición, 168  
 functor y, 162
- dases internas  
 definición, 579  
 iteradores y, 563–567  
 tratamiento de sucesos, 929–935  
 y clases anidadas, 564
- dases internas e implementación de `ArrayList`, 561–582  
`AbstractCollection`, clase, 567–571  
 conceptos clave, 579  
 ejercicios y proyectos, 580–582  
 en Internet, 579  
 errores comunes, 579
- implementación de `ArrayList` con un iterador, 572–578  
 introducción, 561  
 iteradores y clases anidadas, 561–563  
 iteradores y clases internas, 563–567  
 resumen, 579  
`StringBuilder`, 571–572, 579
- dases para entrada y salida de flujos de bits 470–472  
 CLASSPATH, variable de entorno, 93–94, 98  
 claves iguales al pivote, 365  
`clearAll`, rutina, 523  
 clon, 291  
`ClosableFrame`, clase, 933–935  
 código de bytes, 4, 21  
 código genérico, 70  
 códigos prefijo, 464–466, 489  
 colas, 256–257. Véase también pilas y colas  
 con prioridad, 270–273, 275  
 de doble terminación, 604  
 definición, 275  
 en la API de Colecciones, 257  
 implementaciones basadas en matrices dinámicas y, 588–594  
 implementaciones con lista enlazada y, 597–601  
 introducción, 256–257  
 simulación dirigida por sucesos y, 272
- colas con prioridad, definición, 275
- colas con prioridad, mezcla de, 857–877  
 conceptos clave, 874  
 ejercicios y proyectos, 875–876  
 en Internet, 875  
 errores comunes, 874–875  
 introducción, 270–273, 857  
 montículo de emparejamiento y, 863–874  
 montículo sesgado y, 857–862  
 resumen, 874
- colisión, 764, 792
- `Collection`, Interfaz, 232–236, 275
- `Collections`, clase, 239–241, 275
- comentarios  
 definición, 21  
 formatos, 4
- comodines  
 como parámetros de tipo, 148–150  
 definición, 169
- `Comparator`, objetos función, 238
- compatibilidad descendente, 43, 233
- compatibilidad en cuanto a tipo

- clases derivada y, 115
- matrices y, 119–122
- compiladores. Véase pilas y compiladores
- Component**, 916, 937
- componentes genéricos implementados usando genéricos java, 147–155
  - borrado de tipos y, 152
  - comodines con límites y, 148–150
  - interfaces y clases genéricas simples, 148
  - límites de tipo y, 151
  - métodos estáticos genéricos, 150–151
  - restricciones a los genéricos, 152–155
- componentes genéricos, implementación mediante herencia, 140–147
  - autoboxing/unboxing* 143–144
  - envoltorios para tipos primitivos, 141–143
  - Object**, 140–141
  - patrones adaptadores y, 144–145
  - tipos de interfaz, 145–147
- composición, 108, 168
- compresión, 463, 489
  - clases para flujos de datos comprimidos, 479–481
  - definición, 464, 489
  - esquema estándar de codificación, 464
- compresión de archivos, 463–484
  - algoritmo de Huffman, 466–468, 489
  - códigos prefijo, 464–466
  - introducción, 463
- compresión de archivos, implementación, 469–484
  - clases para entrada y salida de flujos de bits 470–472
  - clases para flujos de datos comprimidos, 479–481
  - HuffmanTree**, clase, 470, 474–479
  - introducción, 469
  - main**, rutina, 481, 483
  - mejora del programa, 484
  - recuento de caracteres, clase, 470, 473
- compresión de caminos, 894–895, 905
- comprobador de equilibrado de los símbolos, 431–443
  - algoritmo básico, 431–433
  - implementación y, 433–443
  - introducción, 431
- concatenación de cadenas, 35, 59
- conjunción, 11
- conjunto de sucesos, 504
- conjunto disjunto, clase, 879–910
  - algoritmo rápido de búsqueda, 889–890
  - algoritmo rápido de unión, 890–892
  - algoritmo union/find, 880, 905
  - algoritmos inteligentes de unión, 892–894
  - caso peor para la unión por rango con comprensión de camino, 898–904
  - compresión de caminos, 894–895
  - conceptos clave, 905–906
  - definición, 905
  - ejercicios y proyectos, 906–908
  - en Internet, 906
  - equivalencia dinámica y aplicaciones, 880–889
  - errores comunes, 906
  - implementación Java, 895–897
  - introducción, 879
  - relaciones de equivalencia, 879
  - resumen, 905
- conjuntos, 257–264
  - HashSet**, clase, 260–264
  - introducción, 257–259
  - Set**, definición, 276
  - SortedSet**, 259, 276
  - TreeSet**, 259–260, 276
- constante de cadena, 7, 21
- constante de caracteres, 7
- constantes enteras, 6
  - octales y hexadecimales, 21
- construcción de un montículo en tiempo lineal, 807–812
- constructor
  - abreviatura para **this**, 82
  - CallSite**, 508
  - de cero parámetros, 116–117
  - definición, 98
  - introducción, 76
  - predeterminado, 76
  - this**, llamada a constructor, 82, 99
  - y **super**, 116, 168
- construir
  - clases, 80–85
  - initializadores estáticos, 85
  - instanceof**, 82
  - métodos y campos estáticos, 82–85
  - miembros de instancia y miembros estáticos, 82
  - this**, abreviatura para constructores, 82
  - this**, referencia, 81
- Container**, 916, 937

contenedores de nivel superior, 917–918  
**JPanel**, 917  
 contenedores e iteradores, 232–238  
**Collection**, interfaz, 232–236  
 introducción, 232  
**Iterator**, interfaz, 236–238  
 contextos estáticos, restricciones sobre los genéricos, 153  
**continue**, instrucción, 15–16, 21  
 conversión de tipo, operador de, 10, 21  
 copiar y pegar, 108–110  
 cortar y pegar, 637  
 coste de la arista (peso), 515, 552  
 cota de caso peor, 200, 212  
 cota de caso promedio, 200, 212  
 cota inferior para la ordenación, demostración, 346, 372–373, 374  
 cotas de tiempo amortizadas, 832  
 criptografía de clave pública, 312, 331  
 criptosistema RSA, 309–312, 331  
     algoritmos de cifrado y descifrado, 311–312  
     cálculo de las constantes RSA, 311  
 cuadrático, algoritmo, 201–202  
 cuerpo del método, 18

## D

declaración  
     de clases y parámetros de tipo, 148  
     de matrices, 37–39  
     de objetos, 30–31  
     de tipos primitivos, 7–8  
     del método, 18–19, 21  
 decorador, patrón, 4  
     definición, 139, 168  
     en entrada/salida (E/S), 136–139  
**decreaseKey**, operación, 812, 858, 869  
*Deep Blue*, programa, 426  
**deleteMin**, operación, 805–807  
 demostración por inducción, 288–290  
**dequeue**, 588–589, 591–593, 600–601  
 DES, 312  
 descendiente, 642, 671  
 descendiente propio, 642, 671  
 despacho dinámico  
     definición, 168  
     introducción, 163–166  
     métodos estáticos y, 163  
     polimorfismo y, 114–115

desplazamiento al frente, 638  
 desreferenciar el puntero, 28  
 digrafos, 515  
 Dijkstra, algoritmo de, 535–538, 551, 870–874  
 directiva de importación estática, 92  
 diseño, patrón de diseño compuesto (par), 94–95, 99  
 disjunción, 11  
 distribución  
     exponencial negativa, 394, 402  
     gaussiana, 392  
     normal, 392  
     uniforme, 385, 402  
 divide y vencerás, algoritmos de tipo, 313–322  
     análisis, 315–319  
     definición, 331  
     introducción, 313  
     problema de la suma máxima de una  
         subsecuencia contigua, 313–315  
     tiempos de ejecución, 320–322  
**do**, instrucción, 15, 21  
 doble hash, 788, 792  
 doblemente enlazadas, listas, 618–621, 635  
**double**, tipo primitivo, 6, 7  
**downcast** o especialización, 121

## E

**echo**, comando, 46  
**elementAt**, método, 683  
 elementos  
     clase de equivalencia, 880–881  
     en la interfaz **Collection**, 232  
 encadenamiento circular, 589, 604  
 encadenamiento separado, 788–789, 792  
 encapsulación, 70, 98  
 enlace horizontal, 719, 754  
**enqueue**, 588–589, 591–593, 600–601  
 enteros, tipos, 6, 22  
 entrada estándar, 4, 21  
     a través de terminal, 8  
 entrada y salida (E/S), 51–55  
     archivos secuenciales, 56–59  
     componentes swing, 919–924  
     definición, 59  
     introducción, 51  
     operaciones básicas de flujos, 52  
     patrón decorador, 136–139  
         tipo **Scanner**, 52–55  
**entrySet**, 258

- envoltorios**  
 anidados, 139  
 definición, 168  
 para tipos primitivos, 141–143
- envolvimiento**  
*autoboxing*, 143–144  
 definición, 143–144  
*unboxing*, 143–144
- equals**, método, 78, 98  
 de comparación de cadenas, 34–35  
 definición, 59  
 implementación, 261–264
- equivalencia dinámica y aplicaciones**, 880–889  
 árboles mínimos de recubrimiento, 883–886  
 generación de laberintos, 881–883  
 introducción, 880–881  
 problema del ancestro común más próximo, 886–889
- Error**, excepciones no recuperables, 50, 60
- estadísticas de orden**, 687–692  
 implementación Java, 688–692
- estrategia de autoajuste**, 833–835
- estrategia de particionamiento**, 358, 363–365
- estrategia de rotación hacia la raíz**, 833, 853
- estrategia minimax**, 327  
 definición, 331, 426  
 tres en raya, juego, 326, 418
- estrella fractal**, 303–305
- estructura de datos**  
 definición, 226, 275  
 introducción, 225
- estructura primitiva del lenguaje Java**, 3–26  
 conceptos clave, 20–22  
 ejercicios y proyectos, 23–25  
 en Internet, 23  
 errores comunes, 22–23  
 instrucciones condicionales, 11–18  
 introducción, 3–4  
 métodos, 18–20  
 operadores básicos, 8–10  
 resumen, 20  
 tipos primitivos, 6–8
- ES-UN**, relación, 108, 115, 169
- Euclides**, algoritmo de, 308
- EvalTokenizer**, clase anidada, 451
- evaluación cortocircuitable**, 11, 21
- Evaluator**, clase, 448, 450
- excepciones**, 47–51  
 cláusula `finally`, 48–49  
 cláusulas `throw` y `throws`, 50–51
- comunes, 49–50  
 definición, 60  
 en tiempo de ejecución, 49  
 errores como excepciones no recuperables, 50, 60  
 estándar comprobadas, 50  
 introducción, 47  
 procesamiento, 47
- expansión dinámica de matrices**, 40–41, 60
- exponenciación modular**, 305, 306–307
- expresión postfija**  
 conversión a infix, 445–448  
 definición, 459  
 extraer elemento superior, 453  
 introducción, 445
- expresión infix**  
 conversión a postfija, 445–448  
 definición, 459
- expresiones**, 8
- extends**, cláusula, 112, 168
- F**
- factor de carga**, 770, 792
- factoriales**, 300, 301
- falsos positivos / falsos negativos**, 398, 402
- Fermat**, pequeño teorema de, 399–400, 402
- Fibonacci**, números de, 298–299, 332
- FileReader**, 56–59, 60
- FileWriter**, 57–58, 60
- final**, clase, 117–118, 167  
**final**, métodos  
 definición, 168  
 en la herencia, 117–118, 128
- finally**, cláusula  
 definición, 60  
 tratamiento de excepciones y, 48–49
- find** y **findPrevious**, rutinas, 616–617
- find**, operación clase conjunto disjunto, 880
- find**, operación, y sondeo lineal, 772–774
- findKth**, operación, 500, 690
- findMin**, operación, 678
- float**, tipo primitivo, 6, 7
- FlowLayout**, 924, 937
- flujo de datos predefinidos**, 52
- foco móvil**, 531
- for** avanzado, bucle, 46–47, 59
- for**, instrucción, 14–15, 21
- fuera de línea**, algoritmo, 881, 905
- función de probabilidad**, 503

función global de estilo C, 18  
 función potencial, 839, 853  
 funciones de crecimiento, 200  
 functor, 155–163  
     dases anidadas y, 162  
     dases anónimas y, 161–162  
     dases genéricas y, 162  
     dases locales y, 160–161  
     definición, 168  
     introducción, 155–158

**G**

generador de congruencia lineal, 387, 402  
 generador de congruencia lineal de periodo completo, 387, 402  
 generador de referencias cruzadas, 484–489  
     definición, 484, 489  
     ideas básicas, 484  
     implementación Java, 485–489  
 gestores de disposición, 924  
     BorderLayout, 925–926  
     definición, 937  
     FlowLayout, 924  
     herramientas visuales, 928  
     null, 927  
     sofisticados, 928  
 getNextOpenClose, rutina, 440  
 getToken, rutina, 452  
 getValue, rutina, 454  
 getVertex, rutina, 525  
 grado entrante, 543, 552  
 gráficos, 928–929  
 grafo de nodos de actividad, 548, 552  
 grafo de nodos de sucesos, 548, 552  
 grafo dirigido, 515, 552  
 grafo dirigido acíclico (DAG), 516, 552  
 grafo, definición, 515, 552  
 grafos acíclicos, problemas de caminos en, 543–550  
     análisis del camino crítico, 548–550  
     implementación Java, 545–548  
     ordenación topológica, 543–545  
     teoría del algoritmo del camino más corto, 545  
 grafos densos y dispersos, 517, 552  
 grafos y caminos, 515–557  
     añadir aristas, 525  
     arcos, 515  
     conceptos clave, 551–552  
     definiciones, 515–517

ejercicios y proyectos, 553–557  
 en Internet, 553  
 errores comunes, 552–553  
 grafo denso, 517  
 grafos acíclicos, problemas de caminos en, 543–550  
 nodos, 515  
 problema del camino más corto con ponderaciones negativas y un único origen, 540–543  
 problema del camino más corto con ponderaciones positivas y un único origen, 533–540  
 problema del camino más corto no ponderado con un único origen, 527–533  
 representación, 517–527  
 resumen, 551  
 vértices, 515  
 Graph, clase, 524, 526

**H**

hash, función, 764, 765–768, 792  
     definición, 792  
     hashCode en java.lang.String, 767–768  
     introducción, 764  
 hash, tablas, 763–796  
     aplicaciones, 791  
     conceptos clave, 792–793  
     definición, 793  
     ejercicios y proyectos, 793–795  
     en Internet, 793  
     encadenamiento separado, 788–789, 792  
     errores comunes, 793  
     función hash y, 764, 765–768, 792  
     introducción, 763–764  
     resumen, 791–792  
     sondeo cuadrático, 774–788, 793  
     sondeo lineal, 768–774, 793  
     tabla de transposición, 421  
     y árboles de búsqueda binaria, 789–791  
 hashCode  
     definición, 275  
     implementación, 261–264  
 HashMap, 264, 275  
 HashSet, clase, 260–264  
     definición, 275  
     implementación de equals y hashCode, 261–264  
     introducción, 260

- headSet**, métodos, 273–274  
**heapsort**, ordenación interna y, 813–816, 822  
**herencia**, 107–181
  - clase derivada en la, 112
  - compatibilidad de tipos, 119–122
  - compatibilidad de tipos y, 113–114, 119–122
  - componentes genéricos implementados usando `java`, 4
  - conceptos clave, 167–169
  - constructor y, 116–117
  - creación de nuevas clases y, 108–113
  - definición, 168
  - despacho dinámico y, 114–115, 163–166
  - ejercicios y proyectos, 171–181
  - en Internet, 170–171
  - errores comunes, 169
  - functor y, 155–163
  - genéricos, 147–155
  - implementación de componentes genéricos mediante, 140–147
  - interfaz, 132
  - introducción, 107–108
  - iteradores y factorías, 230–232
  - jerarquías y, 107, 115, 123–129
  - mecanismo de, 70
  - métodos `final` y, 117–118, 128
  - múltiple, 129–131
  - polimorfismo y, 114–115
  - programación orientada a objetos y, 70
  - reglas de visibilidad y, 115–116
  - relaciones, 107
  - resumen, 166
  - reutilización del código y, 107
  - `super` y, 116–117
  - sustitución de métodos y, 119
  - tipos de retorno covariantes y, 122
- herencia fundamental en Java**, 134–139
  - clase `Object` y, 134–135
  - jerarquías de excepciones, 135–136
  - patrón decorador en E/S, 136–139
- hermanos, 642, 672  
**herramientas visuales**, 928  
 hijo, 300, 642, 672  
 hipótesis inductiva, 289, 332  
 hoja, 300, 332, 642, 672  
 hoja, clase, 118, 167  
**Huffman**, algoritmo de, 466–468, 489  
**Huffman**, árbol de codificación de, 649–650
- HuffmanTree**, clase, 470, 474–479
- I**
- identificador**, 7, 21  
**if**, instrucción, 12–13, 21  
**implementación**
  - archivo, 73–74, 98
  - de interfaces, 132
  - mediante matriz circular, 589, 604
- implements**, cláusula, 133, 168  
**import**, directiva, 91–92, 98  
**incremento postfijo**, 10  
**incremento prefijo**, 10  
**índices**, 37–38  
**inducción**, 288, 332
  - demostración por, 288–290
  - matemática, 288–290
- initialización de tipos primitivos**, 7–8  
**initializador estático**, 85, 98  
**immutable**, objeto, 34, 60  
**InputStreamReader**, 137  
**inserción**
  - árboles AA, 719–722
  - arriba-abajo en árboles rojo-negro, 705–707
  - montículo binario, 801–805
  - ordenación por, 343–347
- instanceof**, operador, 82, 98
  - en jerarquías de herencia, 115, 124
  - y miembros estáticos, 82
- instanceof**, pruebas, restricciones a los genéricos y, 153  
**instanciación de tipos genéricos**, 153  
**instrucción nula**, 13, 21  
**instrucciones condicionales**
  - `break`, instrucción, 15–16
  - `continue`, instrucción, 15–16
  - `do`, instrucción, 15
  - `for`, instrucción, 14–15
  - `if`, instrucción, 12–13
  - operador condicional (`? :`), 18
  - operadores lógicos, 11
  - operadores relacionales y de igualdad, 11
  - `switch`, instrucción, 17
  - `while`, instrucción, 13–14
- int**, tipo primitivo, 7  
**IntCell**, clase, 71–72  
**interfaz**
  - `Collection`, 232–236

- interfaz (cont)
- como clase abstracta, 134
  - definición, 168
  - especificación, 132
  - implementación, 132
  - Iterator**, 236–238
  - List**. Véase **List**, interfaz múltiple, 133
  - programación de acuerdo con una, 227, 231, 276
  - tipos utilizados para genéricos, 145–147
- interfaz gráfica de usuario (GUI), 913–940
- Abstract Window Toolkit (AWT)**, 913–915
  - componentes de E/S, 919–924
  - conceptos clave, 937–938
  - contenedores de nivel superior, 917–918
  - definición, 937
  - ejercicios y proyectos, 939–940
  - en Internet, 939
  - errores comunes, 938
  - introducción, 913
  - principios básicos, 924–936
  - resumen, 936
  - Swing**, objetos básicos, 915–916
- inversa multiplicativa, 305, 307–309, 332
- inversión, 345–347, 374
- iterador, objeto, 227–228
- iteradores, 232–238. Véase también contenedores e iteradores
- basados en herencia y factorías, 230–232
  - clases anidadas e, 561–563
  - clases internas e, 563–567
  - definición, 275
  - implementación de **ArrayList** con un iterador, 572–578
  - listas enlazadas y clase iteradoras, 610–612
  - recorrido de árboles y. Véase recorrido de árboles y clases iteradoras
  - relación iterador/contenedor, 564–565
- Iterator**, Interfaz, 236–238, 276
- J**
- jar**, 93
- java**
- definición, 21
  - entorno general, 4
  - programa, 4–5
- java.io**, 51, 60
- java.lang**, 92
- java.math.BigInteger**, 78–80
- java.util.Stack**, clase, 602–603
- javac**, 4, 21
- avadoc**, 73–74, 98
- avadoc**; marcador, 73, 98
- avadoc**; utilidad, comentarios que proporcionan información, 5
- JButton**, 919, 938
- JCheckBox** y **JRadioButton**, 922, 938
- JComboBox**, 921, 938
- JComponent**, 917, 938
- JDialog**, 917, 938
- jerarquías**
- clases y métodos abstractos, 124–128
  - de excepciones en herencia java, 135–136
  - diseño de, 123–129
  - herencia, 107–108, 115
  - Person**, 115
- JFrame**, 917, 938
- JLabel**, 919, 938
- JList**, 921, 938
- JPanel**, 917, 938
- JTextAreas**, 923, 938
- JTextField**, 923, 938
- juegos, 409–429
- ajedrez por computadora, 425–426
  - conceptos clave, 426
  - ejercicios y proyectos, 427–428
  - en Internet, 427
  - errores comunes, 427
  - resumen, 426
  - sopa de letras. Véase sopa de letras
  - tres en raya. Véase tres en raya, juego
- JWindow**, 917, 938
- K**
- keySet**, 258
- Kraft, desigualdad de, 673
- Kruskal, algoritmo de, 884, 905
- L**
- laberintos, generación de, 881–883
- lastToken**, procesamiento, 456
- length**, campo, 37, 60
- length**, método, 36, 60
- lenguaje orientado a objetos, 297
- lenguaje procedimental, 297

- lhs** (lado izquierdo)
    - definición, 60
    - referencias de asignación y, 31
  - lienzo**, 923, 938
  - límites de tipo**, 151, 168
  - LinkedList**
    - clase, 247–248
    - comparación de costes con `ArrayList`, 250–251
    - definición, 276
    - introducción, 249
  - LinkedListIterator**, clase, 612–613
  - List**
    - definición, 244, 276
    - lista enlazada, 247, 276
    - `subList`, método en vistas de la API de Colecciones, 273
  - List**, interfaz, 244–253
    - clase `LinkedList` y, 247–248
    - costes de `ArrayList` y `LinkedList`, 250–251
    - costes de `ArrayList`, 249
    - costes de `LinkedList`, 249
    - eliminación y adición de elementos en mitad de una colección `List`, 251–253
    - introducción, 244
    - `ListIterator`, interfaz, 244–246
    - tiempo de ejecución para listas, 249
  - lista de adyacencia, 518, 522
  - lista enlazada, 607–639
    - borrado en, 609
    - dases iteradoras y, 610–612
    - conceptos clave, 635
    - definición, 276
    - ejercicios y proyectos, 636–638
    - en Internet, 636
    - errores comunes, 635
    - implementaciones. Véase lista enlazada, implementaciones
    - inserción en, 608
    - introducción, 247
    - introducción, 607–609
    - lista simplemente enlazada, 607
    - listas doblemente y circularmente enlazadas, 618–621, 635
    - nodos de cabecera, 609–610
    - ordenadas, 621
    - resumen, 635
  - lista enlazada, implementaciones, 594–601
    - clase `LinkedList` de la API de Colecciones, 621–635
  - colas y, 597–601
  - introducción, 594
  - Java, 612–618
  - pilas y, 594–597
  - ListIterator**, interfaz, 244–246, 276
  - llamadas a métodos, 297
  - local, clase, 160–161, 167
  - logaritmo, 202–204
    - bits requeridos para representar números, 203
    - crecimiento, 202
    - definición, 202, 212
    - N*-ésimo número armónico, 204
    - principio de la división repetida, 204
    - principio de la duplicación repetida, 203
  - lógicos, operadores, 11, 22
  - long**, tipo entero, 6, 7
  - longitud de camino no ponderada, 527, 552
  - longitud externa de camino, 694–695, 754
  - longitud interna de camino, 694, 754
  - longitud ponderada de un camino, 516, 533, 552
  - lote, 817, 823
- ## M
- main**, métodos
    - definición, 21
    - en clases, 78
    - invocación, 5
  - main**, rutina, 443, 481, 483
  - manipulación de cadenas, conceptos básicos, 34
  - Map.Entry**, 258, 276
  - mapas, 264–270
    - de nombres de vértices, 519
    - `HashMap`, 264, 275
    - interfaz de ejemplo, 264–267
    - `Map`, definición, 276
    - `Map.Entry`, 258, 276
    - tablas de transposición, 421
  - TreeMap**, 264, 268–270, 276
  - máquina de estados, 437, 459
  - Máquina Virtual**
    - definición, 21
    - introducción, 4
  - máquinas postfijas, 445, 459
  - Mario*, árbol, 748, 754
  - matrices, 37–47
    - argumentos de la línea de comandos, 46
    - asignación, 37–40
    - bidimensionales irregulares, 45

## matrices (cont.)

- bucle `for` avanzado, 46–47
- como parámetro para un método, 39
- compatibilidad de tipos y, 119–122
- covariantes, 122, 168
- de pequeño tamaño, 367
- de tipos parametrizados, restricciones a los genéricos y, 155
- declaración de, 37–40
- definición, 60
- elemento mínimo en, 189
- expansión dinámica de matrices, 40–41
- introducción, 37
- métodos de, 37
- métodos estáticos que operan sobre, 242
- mezcla en tiempo lineal de matrices ordenadas, 351–352
- multidimensionales, 43–45
- polimorfismo y, 119–122
- tipo matricial covariante, 122
- matrices dinámicas, implementaciones basadas en, 583–594
  - colas y, 588–594
  - introducción, 583
  - pilas y, 583–588
- matriz de adyacencia, 517, 552
- máximo común divisor, 305, 307–309, 332
- mecanismo de cortocircuitado, 612
- `merge`, rutina, 653
- método factoría, 230–232, 276
- métodos, 74–78
  - clases de almacenamiento, 20
  - constructores, 76
  - definición, 21, 98
  - `equals`, 78
  - final y clases, 117–118
  - introducción, 18–19
  - `main`, 78
  - mutadores y accesores, 76–78
  - recursivos, 287, 291–292, 332
  - salida de información y `toString`, 78
  - sobrecarga de nombres de métodos, 19–20
  - `super`, 116–117
  - sustitución de, 119
- métodos abstractos
  - definición, 127, 168
  - en jerarquías de herencia, 124–128
  - parámetros de sustitución y, 124, 126

## resumen, 128

- métodos estáticos, 82–85
  - acoplamiento estático y, 164, 167
  - contextos estáticos y, 153
  - definición, 58, 98
  - despacho dinámico y, 163
  - genéricos, 150–151
  - introducción, 82–85
  - método `main`, 5, 82
  - resumen, 128
- mezcla
  - colas con prioridad. Véase colas con prioridad, mezcla
  - de dos pasadas, 863, 874
  - en tiempo lineal de matrices ordenadas, 351–352
  - modificación del montículo sesgado, 859–860
  - montículo binario y, 857
  - multivía, 818–819, 823
  - polifásica, 819–820, 823
  - simplista de árboles con ordenación de montículo, 858–859
- miembros de clase, 71–72
  - privados, 71–72, 99
  - protegidos, 116, 168
  - públicos, 71, 99
- miembros de instancia, definición, 98
- miembros de instancia y miembros estáticos, 82
- montículo binario, 797–828
  - conceptos clave, 822–823
  - construcción en tiempo lineal, 807–812
  - `decreaseKey` y `merge`, 812–813
  - definición, 271, 276, 823
  - ejercicios y proyectos, 824–827
  - en Internet, 823
  - errores comunes, 823
  - introducción, 797–798
  - operación `buildHeap`, 807–812, 823
  - operaciones básicas, implementación, 801–807
  - operaciones permitidas, 800–801
  - ordenación externa, 816–822
  - ordenación interna y `heapsort`, 813–816
  - propiedad de ordenación, 800
  - propiedad estructural, 798–799
  - resumen, 822
- montículo de emparejamiento, 862–874
  - algoritmo de Dijkstra y, 870–874
  - definición, 874
  - implementación, 865–870

introducción, 862–863  
 operaciones, 863–865  
**montículo izquierdista**, 876  
**montículo maximal**, 800, 823  
**montículo sesgado**, 857–862  
 análisis, 860–862  
 definición, 874  
 mezclado, 857  
 modificación, 859–860  
**multidimensional**, matriz, 43, 60  
**múltiple**, herencia, 129–131, 168  
**mutadores**, 76–78, 98  
**MyContainer**, clase, 562  
**MyContainerIterator**, 561–563

**N**

**negación**, 11  
**Nésimo número armónico**, 204  
**new**, 30, 60  
**nextCall**, método, 508  
**nextLine**, 8  
**nivel de un nodo**, 719, 754  
**nodos**  
 altura, 642, 671  
 de cabecera, 609–610, 635  
 definición, 515  
 nivel de, 719, 754  
 nodo externo del árbol, 694, 754  
 pesados, 860  
 profundidad, 642, 672  
 tamaño, 642, 672  
**null**, gestor de disposición, 927, 937  
**null**, referencia, 27, 60  
**NullPointerException**, mensaje de error, 30, 60  
**números**  
 armónicos, 204, 212  
 en coma flotante, 7  
 pseudoaleatorios, 384, 402  
**números aleatorios**, 383  
 distribución uniforme de, 385, 402  
 generadores de, 384–392  
 necesidad, 383–384  
 no uniformes, 392–394  
 números pseudoaleatorios, 384, 402

**O**

**O mayúscula**, limitaciones del análisis, 211

**O mayúscula**, notación, 198–202  
**O minúscula**, 198, 199, 212  
 $O(N^2)$ , algoritmo, 194  
 $O(N^3)$ , algoritmo, 191–194  
**Object**, clase, 134–135  
**objeto función**  
 Comparator, 238  
 definición, 168  
 introducción, 155  
**objetos**, 29–34  
 como instancia de clase, 71  
 como unidad atómica, 70  
 construcción, 30–31, 59  
 declaración de, 30–31  
 definición, 60, 98  
 en programación orientada a objetos, 70  
**iterador**, 227–228  
 No hay sobrecarga de operadores para, 34  
**operador punto (.)**, 29–30  
 paso de parámetros, 32–33  
 recolección de basura, 31, 60  
**Scanner**, 8  
**objetos matriz genéricos**, 153  
 =, significado de, 31–32, 59  
 ==, significado de, 33–34, 59  
**ocultación de la información**, 70, 76, 98  
**Omega mayúscula**, 198, 213  
**operaciones básicas de flujos**, 52  
**operaciones básicas del montículo binario**,  
 implementación, 801–807  
**deleteMin**, 805–807  
**inserción**, 801–805  
**operador condicional (?)**  
 definición, 21  
 introducción, 18  
**operador de alta precedencia**, 447  
**operador de baja precedencia**, 447  
**operador de indexación de matriz**, 37, 60  
**operadores**  
 aritméticos binarios, 9  
 bit a bit, 941–943  
 conversiones de tipo, 10  
 de asignación, 8–9  
 de igualdad, 11  
 introducción, 911  
 lógicos, 11  
 relacionales, 11  
 unarios, 10

operadores de igualdad  
definición, 22  
introducción, 11  
ordenación con espaciado decreciente, 347  
ordenación externa, 341  
ordenación externa y montículo binario, 816–822  
algoritmo simple, 817–818  
definición, 823  
mezcla multivía, 818–819  
mezcla polifásica, 819–820  
modelo, 816  
necesidad de nuevos algoritmos, 816  
selección de sustitutos, 820–822, 823  
ordenación interna y heapsort, 813–816  
ordenación por mezcla, 350–354  
algoritmo de, 352–354  
definición, 374  
introducción, 350  
mezcla en tiempo lineal de matrices ordenadas, 351–352  
ordenación rápida, 355–370  
algoritmo, 355–358  
análisis del caso mejor del algoritmo de, 358  
análisis del caso peor del algoritmo de, 359  
análisis del caso promedio del algoritmo de, 360–361  
daves iguales al pivote, 365  
definición, 374  
estrategia de particionamiento, 363–365  
introducción, 355  
matrices de pequeño tamaño, 367  
particionamiento basado en la mediana de tres, 366–367, 368–369  
rutina en java, 368–370  
selección del pivote, 362–363  
ordenación topológica, 543–545, 552  
ordenación, algoritmos genéricos, 242–243  
`OutputStreamWriter`, 137

## P

`pack`, 933, 938  
`package`, instrucción, 92–93, 98  
padre, 300, 642, 672  
`paintComponent`, 928, 938  
`PairingHeap`, clase, 865–866  
paquete, 90–94  
  `CLASSPATH`, variable de entorno, 93–94  
  definición, 98

`import`, directiva, 91–92  
introducción, 90–91  
`par`, 95, 98  
parámetros  
  constructores de cero parámetros, 116–117  
  de sustitución, 124, 126  
  formales, 18  
  matrices de tipos parametrizados, 155  
parámetros de tipo  
  comodines como, 148–150, 169  
  declaración de clases y, 148  
  definición, 168  
  matrices de tipos parametrizados, 155  
partición, 356, 374  
particionamiento basado en la mediana de tres, 362, 366–367, 368–369, 374  
paso de parámetros, 32  
  de matrices, 39  
  introducción, 32  
  por referencia, 32, 60  
  por valor, 32  
paso de parámetros por valor  
  definición, 22  
  en métodos, 18–19  
  en objetos y referencias, 32  
patrón de diseño compuesto (`par`), 94–95, 99  
patrón iterador, 227–232  
  diseño, 228–230  
  introducción, 227–228  
patrones  
  adaptadores, 144–145  
  decorador en la entrada/salida, 136–139  
perezoso, borrado, 718, 754, 770, 792  
periodo, 387, 403  
permutación aleatoria, 394–395, 403  
permutaciones, 346, 384, 403  
`Person`, jerarquía, 115  
pila de operadores, 446  
pilas  
  implementaciones basadas en matrices dinámicas y, 583–588  
  implementaciones con lista enlazada y, 594–597  
pilas y colas, 583–606  
  clase `java.util.Stack`, 602–603  
  colas de doble terminación, 604  
  comparación de las implementaciones basadas en matrices dinámicas y con lista enlazada, 601–602

- conceptos clave, 604
- ejercicios y proyectos, 605–606
- en Internet, 605
- errores comunes, 604
- implementaciones basadas en matrices dinámicas, 583–594
- implementaciones con lista enlazada, 594–601
- resumen, 604
- pilas y compiladores, 254–256, 431–461
  - calculadora, 444–458
  - comprobador de equilibrado de los símbolos, 431–443
  - conceptos clave, 458–459
  - definición, 276
  - ejercicios y proyectos, 460–461
  - en Internet, 459–460
  - en la API de Colecciones, 257
  - errores comunes, 459
  - introducción, 254, 431
  - lenguajes informáticos y, 254–256
  - llamadas a métodos y, 297
  - operador, 446
  - resumen, 458
  - secuencias de retornos de métodos y, 297
- pivote
  - claves iguales al, 365
  - definición, 374
  - elección segura, 362
  - función del, 356
  - particionamiento basado en la mediana de tres, 362
  - solución incorrecta, 362
- poda alfa-beta
  - algoritmo minimax y, 329
  - definición, 332, 426
  - en el juego de las tres en raya, 418–425
  - refutación y, 418–419
  - tablas de transposición y, 421–425, 426
- Poisson, distribución de, 392–394, 402
- polimorfismo, 70
  - despacho dinámico y, 114–115
  - matrices y, 119–122
- posición terminal, 418, 426
- positionOf*, 127
- PostOrder*, clase, 662–663
- precedencia, asociatividad para romper empates de, 447
- precedencia, tabla de, 449, 455, 459
- PreOrder*, clase, 664–667, 668
- primer hijo/siguiente hermano, método del, 643, 672
- principio de la división repetida
  - definición, 213
  - en el problema de la búsqueda estática, 204–209
  - en logaritmos, 204
- principio de la duplicación repetida, 203, 213
- println*, método, 6, 8
- printPath*, rutina, 523
- PrintWriter*, 57–58
- problema de Josefo, 497–502
  - algoritmo, 500–502
  - definición, 512
  - solución simple, 498
- problema de la búsqueda estática, 204–209
  - búsqueda binaria y, 205–207
  - búsqueda por interpolación y, 207–209
  - búsqueda secuencial y, 205
  - definición, 204
- problema de la suma máxima de una subsecuencia contigua, 191
- problema del ancestro común más próximo, 886–889, 906
- problema del cambio de moneda, 322–326
- problema del camino más corto con ponderaciones negativas y un único origen, 540–543
  - implementación Java, 541–543
  - introducción, 540
  - teoría, 540–541
- problema del camino más corto con ponderaciones positivas y un único origen, 533–540
  - algoritmo de Dijkstra, 535–538
  - implementación Java, 538–540
  - introducción, 533–534
- problema del camino más corto no ponderado con un único origen, 527–533
  - implementación Java, 533
  - introducción, 527
  - teoría, 527–533
- problema del recuento, 404
- procesamiento de excepciones, 47
- processToken*, rutina, 456
- profundidad de nodo, 642, 672
- programación de acuerdo con una interfaz, 227, 231, 276
- programación dinámica
  - definición, 332
  - en la recursión, 322–326

- programación dinámica (cont.)  
 problema del cambio de moneda, 322–326  
 programación genérica, 140, 169  
 programación orientada a objetos, 69–71, 99  
 programas java  
     comentarios, 5  
     método `main`, 5  
     opciones de entrada, 4  
     salida a través de terminal, 6  
 propagación hacia abajo, 805, 823  
 propagación hacia arriba, 804, 823  
 propiedad de ordenación del montículo, 800, 823  
 prueba de primalidad  
     aleatorización, 398–401  
     en criptosistema RSA, 309  
 puntero, 28  
 punto, operador `(.)`, 29–30, 60  
 puntos colineales en el plano, 189  
 puntos más próximos en el plano, 189
- R**
- rango, 839, 853  
 rangos, 895, 906  
 recolección de basura, 31, 60  
 recorrido  
     en orden, 457, 657, 664, 672  
     en postorden, 646, 659–664, 672  
     en preorden del árbol, 646, 664–667, 672  
     por niveles, 667, 668–671, 672  
     simple, 657  
 recorrido de árboles y clases iteradoras, 657–671  
     por niveles, 667, 668–671, 672  
     recorrido en orden, 657, 664, 672  
     recorrido en postorden, 646, 659–664, 672  
     recorrido en preorden, 646, 664–667, 672  
     recorrido simple, 657  
 recuento de caracteres, clase, 470, 473  
 recursión, 274, 287–340  
     algoritmos de tipo divide y vencerás, 313–322  
     aplicaciones numéricas, 305–312  
     árboles y, 654–656  
     básica, 291–305  
     cómo funciona, 296–297  
     conceptos clave, 331–332  
     ejercicios y proyectos, 334–340  
     en Internet, 333  
     errores comunes, 332  
     impresión de números en cualquier base, 292–294  
     inducción matemática, 288–290  
     introducción, 287–288  
     peligros de demasiada, 297–299  
     por qué funciona, 294–296  
     previsualización de árboles y, 299–300  
     programación dinámica, 322–326  
     reglas de, 292, 299, 332  
     resumen, 331  
     retroceso, 326–330  
     recursión, ejemplos, 300–305  
         búsqueda binaria, 300–302  
         dibujo de una regla, 302–303  
         estrella fractal, 303–305  
         factoriales, 300, 301  
     redirección de archivos, 4  
     refactorización, 129  
     reflexión, 571  
     refutación, 418–419, 426  
     registro de activación, 296, 332  
     regla 90–10, 832, 853  
     regla del interés compuesto, 299  
     regla, dibujo de, 302–303  
     reglas de recursión, 292, 299, 332  
     reglas de visibilidad  
         de paquete, 94  
         herencia y, 115–116  
     rehashing, 777  
     relación TIENE-UN, 107, 169  
     relación, definición, 879, 906  
     relacionales, operadores, 11  
     relaciones de equivalencia, 879, 906  
     relaciones en subclases/superclases, 115, 169  
     remove, operación, 678–679, 688  
     remove, operaciones aleatorias, 695  
     removeMin, método, 686, 691  
     repaint, 929, 938  
     representación implícita, 799, 823  
     reattach, método, 618  
     retroceso, 326–330, 332  
     return, instrucción, 19, 22  
     reutilización de código  
         directa, 112  
         herencia y, 107  
         programación orientada a objetos y, 107  
     rhs (lado derecho)  
         definición, 60

- referencias de asignación y, 31  
 rotación doble, 701–704, 754  
 rotación simple, 699–701, 754  
`runSim`, método, 509, 510  
 rutina de inserción, 618  
 rutina de preparación, 294, 332  
 rutina java de ordenación rápida, 368–370  
 rutina privada, 526  
 rutina recursiva, 526
- S**
- Scanner**, objeto, 8  
**Scanner**, tipo de E/S  
 definición, 60  
 introducción, 52–55  
**secuencia de escape**, 7, 22  
**secuencia de incrementos**, 347  
**secuencias de retornos de métodos**, 297  
**selección de sustitutos**, 820–822, 823  
**selección**, 370–372, 374  
**selección rápida**  
 aleatorizada, 397  
 definición, 374  
 introducción, 370–372  
**semilla**, 387, 403  
**serialización**, 139  
**setLayout**, 924, 938  
**Shell**, ordenación, 347–350  
 definición, 374  
 introducción, 347–348  
 rendimiento, 348–350  
**short**, tipo entero, 6, 7  
**show**, 916, 938  
**signatura**, 19, 22  
**simbolización** 433–434, 459  
**simetría**, 619  
**simulación**, 497–514  
 computadoras usadas en, 497  
 conceptos clave, 512  
 de un servicio de atención telefónica (ejemplo), 504–512  
 definición, 512  
 dirigida por tiempo discreto, 503, 512  
 el problema de Josefo, 497–502  
 introducción, 497  
 resumen, 512  
**simulación dirigida por sucesos**, 501, 503–512  
 definición, 512
- en colas con prioridad, 272  
 ideas básicas, 503–504  
 introducción, 501, 503  
 simulación de un servicio de atención telefónica (ejemplo), 504–512  
**sistemas de archivos**, aplicación  
 árboles y, 644–648  
 implementación Java, 646  
**skew**, 720, 754  
**skipComment**, rutina, 438  
**skipQuote**, rutina, 439  
 sobrecarga de nombres de métodos, 19, 22  
 sobrecarga estática, 163–166, 169  
**sondeo cuadrático**, 774–788  
 análisis del, 783, 788  
 definición, 793  
 implementación Java, 778–788  
 introducción, 774–778  
**sondeo lineal**, 768–810, 793  
 agrupamiento primario y, 771–772  
 análisis simplista de, 770–771  
 definición, 793  
 introducción, 768–770  
 operación `find y`, 772–774  
**sopa de letras**, 409–411  
 definición, 426  
 implementación Java, 411–417  
 teoría, 409–411  
**SortedSet**, 259, 276  
**split**, 720, 754  
**static final**, entidad, 20, 22  
**String**, tipo, 8  
**ArrayList**, 96, 97  
**StringBuilder**, 571–572, 579  
 subclases/superclases, relaciones, 115, 169  
 subcuadrático, algoritmo, 199  
**subList**, método, 273  
**subSet**, métodos, 273–274  
**sucesos**, 929, 938  
 excepcionales, 47  
 suma telescopica, 319, 332  
**super**, método, 116–117  
**super**, objeto, 119, 168  
**sustitución**, 119, 128  
**sustitución parcial**, 119  
**Swing**, componentes E/S, 919–924  
 JButton, 919  
 JCheckBox y JRadioButton, 922

Swing, componentes E/S (cont.)

- JComboBox*, 921
- JLabel*, 919
- JList*, 921
- JTextField* y *JTextAreas*, 923

Swing, objetos básicos, 915–916

- Component*, 916
- Container*, 916
- contenedores de nivel superior, 917–918
- switch*, instrucción, 17, 22
- System.out*, 52, 60
- System.in*, 52, 55, 60
- System.out*, 52, 55, 60

## T

tabla de símbolos, 791

tablas de transposición, 421–425, 426, 791

*tailSet*, método, 273–274

técnica de almacenamiento en caché, 367

terminación de llamada, 509

terminal, entrada y salida a través de, 8

testigo de composición, 400, 403

Theta mayúscula, 198, 213

*this*, abreviatura para constructores, 82

*this*, llamada a constructor, 82, 96

*this*, referencia, 81, 99

*throw* y *throws*, cláusulas, 50–51, 61

*tic*, 503, 513

tiempo de ejecución

- ejemplos, 189–190

- excepciones en, 49, 60

- observado, 201

- para algoritmos aleatorizados, 397

- para algoritmos divide y vencerás, cota superior, 320–322

- para listas, 249

tiempo de servicio, 501

tiempo lineal, algoritmo, 195–198, 212

tiempos entre llegadas, 503

tipo de retorno covariante, 122–123, 169

tipos de referencia, 27–67

- cadenas, 34–37

- conceptos clave, 59–61

- definición, 60

- ejercicios y proyectos, 62–67

- en Internet, 61

- entrada y salida, 51–58

errores comunes, 61

introducción, 27–29

matrices, 37–47

objetos y, 29–34

resumen, 59

tratamiento de excepciones, 47–51

tipos primitivos

- constantes, 6–7

- declaración e inicialización, 7–8

- definición, 22

- entrada y salida a través de terminal, 8

- envoltorios para, 141–143

- introducción, 6

- restricciones sobre los genéricos, 153

*toArray*, 571

tolerancia temporal, 550, 552

*toString*, método, 36, 61, 78, 99

tratamiento de sucesos, 929–935

*TreeMap*, 264, 268–270, 276

- y TreeSet*, implementación de las clases de la API de Colecciones, 726–746

*TreeSet*, 259–260, 276

- y TreeMap*, implementación de las clases de la API de Colecciones, 726–746

tres en raya, juego, 418–425

- estrategia minimax en, 327, 418

- poda alfa-beta y, 418–425, 426

- posición terminal, 418, 426

- tablas de transposición y, 421–425, 426

trie binario, 464, 489

*try*, bloque, 47, 61

## U

unarios, operadores, 11, 22

*unboxing*, 143–144, 169

Unicode

- definición, 22

- estándar para tipos primitivos, 6

unidad atómica, 70, 99

unión por altura, 893, 906

unión por rangos, 895, 906

- algoritmo union-find, análisis de la, 899–904

- y compresión de camino, caso peor, 898–904

unión por tamaño, 892, 906

*union*, operación clase conjunto disjunto, 880

*union/find*, estructura de datos, 880, 905

Unix, sistema de archivos, 644, 646

**utilidades**, 463–495

- compresión de archivos, 463–484
- conceptos clave, 489
- ejercicios y proyectos, 490–494
- en Internet, 490
- errores comunes, 490
- generador de referencias cruzadas, 484–489
- introducción, 463
- resumen, 489

**V****valores**

- en vistas de mapas, 268
- manipulación de tipos primitivos mediante, 29

**variables**, 8

- de referencia (referencias), 27–29
- vértice**
  - adyacente, 515, 531–532, 552
  - definición, 515
  - `getVertex`, rutina, 523
  - nombres, 519
  - `Vertex`, clase, 523

**vistas en la API de Colecciones**, 273–274

- `headSet`, métodos y, 273–274
- introducción, 273
- método `subList` para objetos `List`, 273
- `subSet`, métodos y, 273–274
- `tailSet`, métodos y, 273–274

voraz, algoritmo, 322, 331

**W****`while`, instrucción**, 13–14, 22

- `WindowAdapter`, 933, 938
- `WindowListener`, interfaz, 929, 938

**Z**

- `zig`, 835, 853
- `zig-zag`, 835, 853
- `zig-zig`, 836, 853





El objetivo de esta cuarta edición de *Estructuras de datos en Java*, es proporcionar una introducción práctica a las estructuras de datos y algoritmos, desde el punto de vista de pensamiento abstracto y de las técnicas de resolución de problemas.

Se ha tratado de cubrir todos los detalles importantes concernientes a las estructuras de datos, sus análisis y sus implementaciones Java.

El texto proporciona el necesario rigor matemático para los cursos sobre Estructuras de datos que enfatizan la teoría y para los cursos posteriores que requieren un mayor grado de análisis. Sin embargo, este material destaca del texto principal en forma de teoremas separados y, en algunos casos, secciones o subsecciones separadas.

Se presentan ejercicios de varios tipos; en concreto, cuatro variedades.

- Los ejercicios básicos *En resumen* plantean una pregunta simple o requieren simulaciones a mano de un algoritmo descrito en el texto.
- La sección *En teoría* plantea cuestiones que requieren un análisis matemático o que piden soluciones interesantes, desde el punto vista teórico a los problemas.
- La sección *En la práctica* contiene cuestiones simples de programación, incluyendo cuestiones acerca de la sintaxis o acerca de líneas particularmente complejas de código.
- Finalmente, la sección *Proyectos de programación* contiene ideas para la asignación de trabajos de mayor envergadura.

ISBN: 978-8415552239



**PEARSON**

[www.pearson.es](http://www.pearson.es)

9 788415 552239