

ESTRUCTURAS DE DATOS Y ALGORITMOS

MARK ALLEN WEISS
Florida International University

Versión en español de
Jorge Lozano Moreno
Universidad Autónoma Metropolitana
Unidad Iztapalapa
y
Grupo Telsys-Ingenética
México

Con la colaboración técnica de

Andoni Eguíluz e Inés Jacob
Universidad de Deusto
Bilbao, España



ADDISON-WESLEY IBEROAMERICANA
Argentina • Brasil • Chile • Colombia • Ecuador • España
Estados Unidos • México • Perú • Puerto Rico • Venezuela

Versión en español de la obra titulada *Data Structures and Algorithm Analysis*, de
Mark Allen Weiss, publicada originalmente en inglés por The Benjamin/Cummings
Publishing Company, Inc., Redwood City, California, E.U.A. ©1992 por The
Benjamin/Cummings Publishing Company, Inc.

Esta edición en español es la única autorizada.

A Mona, David y Nina Weiss

ADDISON-WESLEY IBEROAMERICANA
Malabia 2363-2cG, Buenos Aires 1425, Argentina
Cruz 1469-depto. 21 Independencias
Santiago de Chile
Apartado Aéreo 241-943, Santa Fé de Bogotá, Colombia
Espalter 3 bajo, Madrid 28014, España
1 Jacob Way, Reading, Massachusetts 01867, E.U.A.
Apartado Postal 22-012, México, D.F. 14000, México
Apartado Postal 51454, Caracas 1050-A, Venezuela
El monte Mall 2o. Piso Oficina 19-B Ave. Muríoz Rivera
Hato Rey, Puerto Rico 00918

© 1995 por Addison-Wesley Iberoamericana, S.A.
Wilmington, Delaware, E.U.A.

Impreso en Estados Unidos. Printed in U.S.A.

ISBN 0-201-62571-7

2 3 4 5 6 7 8 9 10-CRS-99 98 97 96 95

PREFACIO

Propósito/Objetivos

Este libro describe *estructuras de datos*, métodos de organización de grandes cantidades de datos, y *análisis de algoritmos*, el cálculo del tiempo de ejecución de los algoritmos. Conforme los computadores se vuelven más y más rápidos, se torna más aguda la necesidad de programas que puedan manejar grandes cantidades de entradas. Paradójicamente, esto requiere una mayor atención a la eficiencia, ya que las ineficiencias en los programas son más obvias cuando es grande el tamaño de la entrada. Analizando un algoritmo antes de su codificación, los estudiantes pueden decidir si una solución particular será factible. Por ejemplo, en este libro los estudiantes ven problemas específicos y ven cómo las implantaciones cuidadosas pueden reducir la restricción de tiempo para grandes cantidades de datos, de 16 años a menos de un segundo. Por lo tanto, ningún algoritmo o estructura de datos se presenta sin una explicación de su tiempo de ejecución. En algunos casos, se exploran minuciosos detalles que afectan el tiempo de ejecución de la implantación.

Una vez que se determina un método de solución, todavía hay que escribir un programa. Conforme los computadores se hacen más potentes, los problemas que resuelven se han vuelto mayores y más complejos, requiriéndose así el desarrollo de programas más intrincados para resolverlos. El objetivo de esta obra es enseñar a los estudiantes habilidades para una buena programación y, simultáneamente, análisis de algoritmos, de tal forma que puedan desarrollar programas con el máximo de eficiencia.

Este libro es adecuado para un curso de estructuras de datos avanzadas (CS7) o para un curso de análisis de algoritmos de primer año de posgrado. Los estudiantes deben tener conocimientos previos sobre programación a un nivel intermedio, incluyendo temas como apuntadores y recursión, y alguna educación en matemáticas discretas.

Enfoque

Creo que es importante que los estudiantes aprendan a programar por sí mismos, y no a copiar programas de un libro. Por otro lado, es virtualmente imposible discutir aspectos realistas de programación sin incluir código de ejemplo. Por esta razón, en este libro se ofrece de la mitad a tres cuartos de cada implantación y se invita al estudiante a hacer el resto.

Los algoritmos se presentan en Pascal, debido a que la mayoría de los estudiantes deben de conocerlo bien. En algunos casos se usa seudocódigo cuando una implantación en Pascal podría oscurecer detalles importantes de un algoritmo de ejemplo.

Estructura de la obra

El capítulo 1 contiene material de repaso sobre matemáticas discretas y recursión. Creo que la única forma de familiarizarse con la recursión es estudiar sus buenos usos, una y otra vez. Por lo tanto, la recursión prevalece en este texto, con ejemplos en todos los capítulos, excepto en el capítulo 5.

El capítulo 2 trata sobre el análisis de algoritmos. Este capítulo explica el análisis asintótico y su mayor debilidad. Se presentan varios ejemplos, con una explicación en profundidad del tiempo de ejecución logarítmico. Se analizan programas recursivos sencillos, mediante su conversión intuitiva en programas iterativos. Se introducen programas más complejos con la técnica de "divide y vencerás", pero parte del análisis (la solución de las relaciones de recurrencia) se pospone para el capítulo 7, donde se realiza con todo detalle.

En el capítulo 3 se estudian las listas, pilas y colas. Aquí se da énfasis a la codificación de estas estructuras de datos usando TDA, la implantación rápida de esas estructuras de datos y una exposición de algunos de sus usos. Casi no hay programas (sólo rutinas), pero los ejercicios contienen multitud de ideas para problemas de programación.

El capítulo 4 trata sobre los árboles, en especial sobre los árboles de búsqueda, entre ellos los árboles de búsqueda externos (árboles-B). Como ejemplos se usan el sistema de archivos de UNIX y los árboles de expresión. Se toca el tema de los árboles AVL y de los árboles desplegados (o achaflanados) sin analizarlos. Se escribe el 75% del código, dejando casos similares para que los complete el estudiante. Estudios adicionales sobre los árboles, como la compresión de archivos y los árboles de juego, se posponen hasta el capítulo 10. Las estructuras de datos en un medio externo se consideran como el tema final en varios capítulos.

El capítulo 5 es relativamente corto y se dedica a las tablas de dispersión (*hash tables*), con algo de análisis, y hacia el final del capítulo se cubre la dispersión extensible.

El capítulo 6 trata sobre las colas de prioridad. Se estudian los montículos (*heaps*) binarios, y hay material adicional sobre algunas de las implantaciones de las colas de prioridades más interesantes desde el punto de vista teórico.

El capítulo 7 se dedica a la ordenación o clasificación. Es muy específico con respecto a los detalles de codificación y análisis. Se estudian y comparan todos los algoritmos importantes de ordenación de propósito general. Se analizan en detalle tres algoritmos: la ordenación por inserción, la ordenación de Shell (*Shellsort*) y la ordenación rápida (*quicksort*). Al final del capítulo se estudia la ordenación externa.

El capítulo 8 presenta el algoritmo de conjuntos ajenos (o disjuntos) con una demostración del tiempo de ejecución. Este capítulo es corto y específico, y puede saltarse si no se estudia el algoritmo de Kruskal.

El capítulo 9 trata los algoritmos de grafos. Éstos son interesantes no sólo por su frecuente aparición en la práctica, sino también porque su tiempo de ejecución depende mucho del uso adecuado de las estructuras de datos. Prácticamente todos los algoritmos estándar se presentan junto con sus estructuras de datos adecuadas, seudocódigo y análisis del tiempo de ejecución. Para ubicar estos problemas en un contexto apropiado, se hace un breve análisis de la teoría de la complejidad (incluyendo la complejidad NP y la indecidibilidad).

En el capítulo 10 se estudia el diseño de algoritmos mediante el examen de técnicas comunes de resolución de problemas. Este capítulo está apoyado con ejemplos. En estos últimos capítulos se usa seudocódigo a fin de que la apreciación de los estudiantes no se oscurezca por los detalles de implantación.

El capítulo 11 trata sobre el análisis amortizado. Se estudian tres estructuras de datos de los capítulos 4 y 6, así como el montículo de Fibonacci, presentado en este capítulo.

Los capítulos 1 al 9 proporcionan suficiente material para la mayoría de los cursos de un semestre de estructuras de datos. Si el tiempo lo permite, se puede cubrir el capítulo 10. Un curso de posgrado sobre análisis de algoritmos podría abarcar los capítulos 7 al 11. Las estructuras de datos avanzadas que presentamos en el capítulo 11 podrían ser incluidas con facilidad en los capítulos anteriores. El estudio de la complejidad NP del capítulo 9 es demasiado breve para ser usado en ese curso. El libro de Garey y Johnson sobre la complejidad NP puede servir para consolidar el tratamiento dado en esta obra.

Ejercicios

Los ejercicios presentados al final de cada capítulo vienen en el orden en que se presenta el material. Los últimos ejercicios pueden abarcar el capítulo como un todo y no como una sección específica. Los ejercicios difíciles están marcados con un asterisco y los ejercicios que implican un reto mayor tienen dos asteriscos.

Referencias

Al final de cada capítulo se incluyen las referencias. Por lo general las referencias son históricas, es decir, son las fuentes originales del material, o constituyen extensiones y mejoras a los resultados dados en el libro. Algunas referencias representan soluciones a los ejercicios.

Agradecimientos

Quisiera agradecer a toda la gente que me ayudó en la preparación de este libro. Los profesionales de Benjamin/Cummings hicieron de mi primer libro una experiencia considerablemente menos problemática de lo que esperaba. Quisiera agradecer a mis editores, Alan Apt y John Thompson, y a la asistente de John, Vivian McDougal, por responder todas mis preguntas y por tolerar mis retrasos. Laura Kenney de Benjamin/Cummings y Melissa G. Madsen y Laura Snyder de Publication Services hicieron un maravilloso trabajo de producción. Quisiera agradecer en especial al copieditor, Jerome Colburn, quien hizo un trabajo increíble volviendo a redactar frases oscuras. El índice fue preparado usando *makeindex*, de Pehong Chen y Michael Harrison.

Quisiera agradecer a quienes hicieron una lectura crítica del libro, por ofrecer comentarios valiosos, muchos de los cuales se incorporaron al texto. Por orden alfabetico, son: Vicki Allan (Utah State University), Henry Bauer (University of Wyoming), Alex Biliris (Boston University), Julia Hodges (Mississippi State University), Bill Kraynek (Florida International University), Rayno D. Niemi (Rochester Institute of Technology), Robert O. Pettus (University of South Carolina) y Chris Wilson (University of Oregon). En particular, agradezco a Vicki Allan, por haber leído cuidadosamente cada borrador y dado sugerencias muy detalladas para mejorarlo.

En la Florida International University, mucha gente cooperó en este proyecto. Xinwei Cui y John Tso me ayudaron con sus notas de clase. Quisiera agradecer a Bill Kraynek, Wes Mackey, Jai Navlakha y Wei Sun por usar los borradores en sus clases y a los muchos estudiantes que los sufrieron en sus primeras fases. María Fiorenza, Eduardo González, Ancin Peter, Tim Riley, Jefre Riser y Magaly Sotolongo notaron varios errores, y Mike Hall revisó desde un borrador anterior todos los errores de programación. Gracias especiales a Yuzheng Ding, quien compiló y probó todos los programas del manuscrito final, incluyendo la conversión de seudocódigo a Pascal. No debo olvidar a Carlos Ibarra y Steve Luis, que cuidaron el funcionamiento de las impresoras y el sistema informático y obtuvieron las cintas en cuestión de minutos.

Este libro es un producto del amor a las estructuras de datos y a los algoritmos que sólo se puede obtener de educadores de alto nivel. Quisiera tomar un momento para agradecer a Bob Hopkins, E. C. Horvath y Rich Mendez, quienes me enseñaron en Cooper Union y a Bob Sedgewick, Ken Steiglitz y Bob Tarjan en Princeton.

Por último, quisiera agradecer a todos mis amigos que me apoyaron durante el proyecto. En particular a Michele Dorchack, Arvin Park y Tim Snyder por escuchar mis historias; Bill Kraynek, Alex Pelin y Norman Pestaina por ser gentiles vecinos (en la oficina), incluso cuando yo no lo fui; y al HTMC por la ayuda en el trabajo.

Cualquier error en este libro, por supuesto, es mío. Apreciaría la notificación de los errores que se encuentren; mi correo electrónico es weiss@fiu.edu.

M. A. W.

ÍNDICE GENERAL

1	Introducción	1
1.1.	¿De qué trata este libro?	1
1.2.	Repaso de matemáticas	3
1.2.1.	Exponentes	3
1.2.2.	Logaritmos	3
1.2.3.	Series	4
1.2.4.	Aritmética modular	6
1.2.5.	La palabra con D	6
1.3.	Breve introducción a la recursión	9
	Resumen	13
	Ejercicios	13
	Referencias	15
2	Análisis de algoritmos	17
2.1.	Soporte matemático	17
2.2.	Modelo	20
2.3.	Qué analizar	20
2.4.	Cálculo del tiempo de ejecución	22
2.4.1.	Un ejemplo sencillo	23
2.4.2.	Reglas generales	24
2.4.3.	Soluciones al problema de la suma de la subsecuencia máxima	26
2.4.4.	Logaritmos en el tiempo de ejecución	32
2.4.5.	Verificación del análisis	37
2.4.6.	Un grano de sal	38

- Resumen 39
 Ejercicios 39
 Referencias 44
- 3 Listas, pilas y colas 45**
- 3.1. Tipos de datos abstractos (TDA) 45
 - 3.2. El TDA lista 46
 - 3.2.1. Implantación de listas a base de arreglos sencillos 47
 - 3.2.2. Listas enlazadas 47
 - 3.2.3. Detalles de programación 48
 - 3.2.4. Errores comunes 54
 - 3.2.5. Listas doblemente enlazadas 56
 - 3.2.6. Listas enlazadas circularmente 56
 - 3.2.7. Ejemplos 57
 - 3.2.8. Implantación de listas enlazadas a base de cursos 62
 - 3.3. El TDA pila 66
 - 3.3.1. El modelo pila 66
 - 3.3.2. Implantación de pilas 68
 - 3.3.3. Aplicaciones 74
 - 3.4. El TDA cola 83
 - 3.4.1. El modelo cola 83
 - 3.4.2. Implantación de colas a base de arreglos 83
 - 3.4.3. Aplicaciones de colas 87
 - Resumen 88
 - Ejercicios 88
- 4 Árboles 93**
- 4.1. Preliminares 93
 - 4.1.1. Implantación de árboles 95
 - 4.1.2. Recorridos de árboles con una aplicación 95
 - 4.2. Árboles binarios 100
 - 4.2.1. Implantación 100
 - 4.2.2. Árboles de expresión 101
 - 4.3. El tda árbol de búsqueda: Árboles binarios de búsqueda 105
 - 4.3.1. Crear_vacio 106
 - 4.3.2. Buscar 106
 - 4.3.3. Buscar_mín y buscar_máx 107
 - 4.3.4. Insertar 108

- 4.3.5. Eliminar 109
 4.3.6. Análisis del caso promedio 111
- 4.4. Árboles AVL 114**
- 4.4.1. Rotación sencilla 116
 - 4.4.2. Rotación doble 119
- 4.5. Árboles desplegados 126**
- 4.5.1. Una idea sencilla (que no funciona) 127
 - 4.5.2. Despliege 129
- 4.6. Recorridos de árboles (de nuevo) 137**
- 4.7. Árboles-B 139**
- Resumen 144
 - Ejercicios 145
 - Referencias 152
- 5 Dispersión 155**
- 5.1. Idea general 155
 - 5.2. Función de dispersión 156
 - 5.3. Dispersión abierta (encadenamiento separado) 159
 - 5.4. Dispersión cerrada (direcciónamiento abierto) 162
 - 5.4.1. Exploración lineal 162
 - 5.4.2. Exploración cuadrática 165
 - 5.4.3. Dispersión doble 168
 - 5.5. Redispersión 170
 - 5.6. Dispersión extensible 172
 - Resumen 175
 - Ejercicios 176
 - Referencias 179
- 6 Colas de prioridad (montículos) 181**
- 6.1. Modelo 182
 - 6.2. Implantaciones simples 182
 - 6.3. Montículo binario 183
 - 6.3.1. Propiedad de la estructura 183
 - 6.3.2. Propiedad de orden de montículo 184
 - 6.3.3. Operaciones básicas sobre montículos 185
 - 6.3.4. Otras operaciones sobre montículos 189
 - 6.4. Aplicaciones de las colas de prioridad 194

- 6.4.1. El problema de la selección 194
 6.4.2. Simulación de eventos 196
 6.5. Montículos-*d* 197
 6.6. Montículos a izquierda 198
 6.6.1. Propiedad de montículo a izquierda 198
 6.6.2. Operaciones sobre montículos a izquierda 200
 6.7. Montículos oblicuos 205
 6.8. Colas binomiales 207
 6.8.1. Estructura de cola binomial 208
 6.8.2. Operaciones sobre colas binomiales 209
 6.8.3. Implantación de colas binomiales 213
 Resumen 216
 Ejercicios 216
 Referencias 221
- 7 Ordenación 221**
- 7.1. Preliminares 224
 7.2. Ordenación por inserción 224
 7.2.1. El algoritmo 224
 7.2.2. Análisis de la ordenación por inserción 225
 7.3. Una cota inferior para algoritmos de ordenación simples 225
 7.4. Ordenación de Shell 227
 7.4.1. Análisis del peor caso de la ordenación de Shell 228
 7.5. Ordenación por montículo 231
 7.6. Ordenación por intercalación 233
 7.6.1. Análisis de la ordenación por intercalación 236
 7.7. Ordenación rápida 240
 7.7.1. Selección del pivote 241
 7.7.2. Estrategia de partición 242
 7.7.3. Archivos pequeños 245
 7.7.4. Rutinas reales de ordenación rápida 245
 7.7.5. Análisis de la ordenación rápida 247
 7.7.6. Un algoritmo de selección con un tiempo esperado lineal 251
 7.8. Ordenación de registros grandes 252
 7.9. Una cota inferior general para la ordenación 253
 7.9.1. Árboles de decisión 253
 7.10. Ordenación por cubetas 255
 7.11. Ordenación externa 256
 7.11.1. ¿Por qué necesitamos algoritmos nuevos? 256

- 7.11.2. Modelo para ordenación externa 257
 7.11.3. El algoritmo sencillo 257
 7.11.4. Intercalación de vías múltiples 258
 7.11.5. Intercalación polifásica 260
 7.11.6. Selección de sustitución 261
 Resumen 262
 Ejercicios 263
 Referencias 267
- 8 El TDA conjunto ajeno 271**
- 8.1. Relaciones de equivalencia 271
 8.2. El problema de la equivalencia dinámica 272
 8.3. Estructura de datos básica 274
 8.4. Algoritmos de unión refinados 277
 8.5. Compresión de caminos 280
 8.6. Peor caso de la unión por rangos y compresión de caminos 281
 8.6.1. Análisis del algoritmo unión/búsqueda 282
 8.7. Una aplicación 288
 Resumen 289
 Ejercicios 289
 Referencias 291
- 9 Algoritmos de grafos 293**
- 9.1. Definiciones 293
 9.1.1. Representación de grafos 294
 9.2. Ordenación topológica 296
 9.3. Algoritmos del camino más corto 300
 9.3.1. Caminos más cortos no ponderados 302
 9.3.2. Algoritmo de Dijkstra 308
 9.3.3. Grafos con aristas de costo negativo 315
 9.3.4. Grafos acíclicos 316
 9.3.5. Camino más corto entre todos los pares 320
 9.4. Problemas de flujo en redes 320
 9.4.1. Un algoritmo simple de flujo máximo 321
 9.5. Árbol de extensión mínimo 325
 9.5.1. Algoritmo de Prim 326

- 9.5.2. Algoritmo de Kruskal 330
 - 9.6. Aplicaciones de la búsqueda en profundidad 332
 - 9.6.1. Grafos no dirigidos 333
 - 9.6.2. Biconectividad 334
 - 9.6.3. Circuitos de Euler 338
 - 9.6.4. Grafos dirigidos 342
 - 9.6.5. Localización de componentes fuertes 344
 - 9.7. Introducción a la complejidad NP 346
 - 9.7.1. Fácil vs. difícil 346
 - 9.7.2. La clase NP 347
 - 9.7.3. Problemas NP completos 348
 - Resumen 351
 - Ejercicios 351
 - Referencias 357
-
- 10. Técnicas de diseño de algoritmos 361
 - 10.1. Algoritmos ávidos 361
 - 10.1.1. Un problema simple de planificación 362
 - 10.1.2. Códigos de Huffman 366
 - 10.1.3. Empaqueamiento aproximado en recipientes 372
 - 10.2. "Divide y vencerás" 381
 - 10.2.1. Tiempo de ejecución de algoritmos de "divide y vencerás" 382
 - 10.2.2. El problema de los puntos más cercanos 385
 - 10.2.3. El problema de la selección 389
 - 10.2.4. Mejoras teóricas para problemas de aritmética 393
 - 10.3. Programación dinámica 397
 - 10.3.1. Uso de una tabla en vez de la recursión 397
 - 10.3.2. Ordenación de multiplicaciones de matrices 400
 - 10.3.3. Árbol binario de búsqueda óptimo 402
 - 10.3.4. Camino más corto entre todos los pares 407
 - 10.4. Algoritmos aleatorizados 409
 - 10.4.1. Generadores de números aleatorios 410
 - 10.4.2. Listas con saltos 414
 - 10.4.3. Comprobación de primalidad 417
 - 10.5. Algoritmos con retroceso 418
 - 10.5.1. El problema de la reconstrucción del camino de cuota 420
 - 10.5.2. Juegos 425
 - Resumen 432
 - Ejercicios 432
 - Referencias 440

- 11. Análisis amortizado 445
 - 11.1. Un acertijo no relacionado 446
 - 11.2. Colas binomiales 446
 - 11.3. Montículos oblicuos 452
 - 11.4. Montículos de Fibonacci 454
 - 11.4.1. Corte de nodos en montículo a izquierda 455
 - 11.4.2. Fusión perezosa de colas binomiales 458
 - 11.4.3. Las operaciones del montículo de Fibonacci 461
 - 11.4.4. Demostración de la cota del tiempo 463
 - 11.5. Árboles desplegados 465
 - Resumen 469
 - Ejercicios 470
 - Referencias 471
- Vocabulario técnico bilingüe 473
- Índice de materias 479

Introducción

En este capítulo se analizan las intenciones y objetivos de este texto y se hace una breve revisión de conceptos de programación y matemáticas discretas.

- Se verá que el comportamiento de un programa con entrada razonablemente grande es tan importante como su rendimiento con pocas entradas.
- Se resumirá el soporte matemático básico, necesario para el resto del libro.
- Se revisará brevemente la recursión.

1.1. ¿De qué trata este libro?

Supóngase que se tiene un grupo de n números y se quiere determinar el k -ésimo mayor. Esto se conoce como *problema de selección*. La mayoría de los estudiantes que hayan realizado uno o dos cursos de programación no tendrán ninguna dificultad para escribir un programa que resuelva este problema. Hay algunas soluciones "obvias".

Una forma de resolver este problema podría ser leer los n números de un arreglo, ordenar el arreglo en forma decreciente por medio de algún algoritmo sencillo como el método de la burbuja, y después devolver el elemento de la posición k .

Un algoritmo un poco mejor podría basarse en leer los primeros k elementos de un arreglo y ordenarlos (en orden decreciente). A continuación, se lee uno a uno cada elemento de los restantes. Cuando llega un nuevo elemento, se ignora si es más pequeño que el k -ésimo elemento del arreglo. Si no, se coloca en la posición que le corresponda en el arreglo, eliminando un elemento. Cuando el algoritmo termina, el elemento de la k -ésima posición se devuelve como respuesta..

Ambos algoritmos son fáciles de codificar, y se invita al lector a que lo haga. Entonces las preguntas naturales son ¿qué algoritmo es mejor? y, lo que es más importante, ¿alguno de ellos es suficientemente bueno? Una simulación que utilice un archivo aleatorio de un millón de elementos y $k = 500\,000$ demostrará que ningún

algoritmo termina en una cantidad de tiempo razonable; cada uno requiere varios días de procesamiento en un computador para terminar (aunque finalmente con una respuesta correcta). Un método alternativo, presentado en el capítulo 7, da una solución en cerca de un segundo. Así, aunque los algoritmos propuestos funcionen, no se pueden considerar buenos, porque son completamente imprácticos para tamaños de entradas que un tercer algoritmo puede procesar en un tiempo razonable.

Un segundo problema es resolver el popular juego de la sopa de letras. La entrada consiste en un arreglo bidimensional de letras y una lista de palabras. El objetivo es encontrar las palabras en el tablero. Esas palabras pueden estar en cualquier dirección (horizontal, vertical y diagonal). Como ejemplo, el tablero de la figura 1.1 contiene las palabras *esto*, *ese*, *pato* y *este*. La palabra *esto* comienza en la fila 1, columna 1 (1, 1) y llega hasta (1, 4); *ese* va desde (1, 1) hasta (3, 1); *pato* está entre (4, 1) y (1, 4); y *este* va de (4, 4) a (1, 1).

De nuevo, hay al menos dos algoritmos directos que resuelven el problema. Por cada palabra en la lista, se examina cada tripleta ordenada (*fila*, *columna*, *orientación*) en busca de la palabra. Esto implica varios ciclos *for* anidados; pero es un método muy directo.

Alternativamente, para cada cuadupleta ordenada (*fila*, *columna*, *orientación*, *número de caracteres*) que no rebasa los límites del tablero, se puede probar si la palabra está en la lista. De nuevo, esto implica varios ciclos *for* anidados. Es posible ahorrar algo de tiempo si se conoce el número máximo de caracteres en cualquier palabra.

Es relativamente fácil codificar cualquiera de las dos soluciones y resolver varios casos reales, de los que se publican comúnmente en cualquier diario o revista. Éstos suelen tener 16 filas, 16 columnas y unas 40 palabras. Supóngase, sin embargo, que se considera la variante donde sólo se da el tablero, y la lista de palabras es, en esencia, un diccionario de español. Ambas soluciones propuestas requieren una cantidad considerable de tiempo para resolver el problema y, por lo tanto, no son aceptables. No obstante, es posible, aun con una lista de palabras grande, resolver el problema en cuestión de segundos.

Un concepto importante es que en muchos problemas no basta con escribir programas que funcionen. Si el programa va a utilizar grandes cantidades de datos, entonces el tiempo de ejecución se vuelve un problema. En este libro se verá cómo calcular el tiempo de ejecución de un programa para cantidades grandes de datos y, más importante aún, cómo comparar los tiempos de ejecución de dos programas sin haberlos codificado. Se verán técnicas para mejorar en forma considerable la

Figura 1.1 Ejemplo de un tablero de la sopa de letras

	1	2	3	4
1	e	s	t	o
2	s	t	t	m
3	e	a	s	a
4	p	r	n	e

velocidad de un programa y para detectar los cuellos de botella de los programas. Esas técnicas permitirán encontrar la sección del código en la cual se deben concentrar los esfuerzos de optimización.

1.2. Repaso de matemáticas

En esta sección se incluyen algunas fórmulas básicas que deberán memorizarse o poderse desarrollar; también se revisan técnicas de demostración básicas.

1.2.1. Exponentes

$$x^a x^b = x^{a+b}$$

$$\frac{x^a}{x^b} = x^{a-b}$$

$$(x^a)^b = x^{ab}$$

$$x^n + x^n = 2x^n \neq x^{2n}$$

$$2^n + 2^n = 2^{n+1}$$

1.2.2. Logaritmos

En informática, todos los logaritmos son en base 2 a menos que se especifique otra cosa.

DEFINICIÓN: $x^a = b$ si y sólo si $\log_x b = a$.

Varias igualdades útiles se obtienen de esta definición.

TEOREMA 1.1.

$$\log_a b = \frac{\log_c b}{\log_c a}; \quad c > 0$$

DEMOSTRACIÓN:

Sea $x = \log_c b$, $y = \log_c a$, y $z = \log_c b$. Entonces, por la definición de logaritmo, $c^x = b$, $c^y = a$, y $c^z = b$. Combinando las tres igualdades se obtiene $(c^y)^z = c^x = b$. Por lo tanto, $x = yz$, lo cual implica que $z = \frac{y}{x}$, demostrándose así el teorema.

TEOREMA 1.2.

$$\log ab = \log a + \log b$$

DEMOSTRACIÓN:

Sea $x = \log a$, $y = \log b$, $z = \log ab$. Entonces, tomando 2 como la base por omisión, $2^x = a$, $2^y = b$, $2^z = ab$. Combinando las tres últimas igualdades se tiene $2^x 2^y = 2^z = ab$. Por lo tanto, $x + y = z$, lo cual demuestra el teorema.

A continuación se presentan algunas fórmulas útiles, las cuales se pueden obtener por medios similares.

$$\log ab = \log a + \log b$$

$$\log(a^b) = b \log a$$

$\log x < x$ para toda $x > 0$

$$\log 1 = 0, \quad \log 2 = 1, \quad \log 1024 = 10, \quad \log 1\,048\,576 \approx 20$$

1.2.3. Series

Las fórmulas más fáciles de recordar son

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

y su compañera,

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

En la última fórmula, si $0 < a < 1$, entonces

$$\sum_{i=0}^n a^i \leq \frac{1}{1-a}$$

y conforme n tiende a ∞ , la suma se acerca a $1/(1-a)$. Éstas son las fórmulas de las "series geométricas".

Se puede derivar la última fórmula para $\sum_{i=0}^{\infty} a^i$ ($0 < a < 1$) de la siguiente manera. Sea S la suma. Entonces:

$$S = 1 + a + a^2 + a^3 + a^4 + a^5 + \dots$$

Entonces

$$aS = a + a^2 + a^3 + a^4 + a^5 + \dots$$

Si se restan las dos ecuaciones (lo cual es permisible sólo para series de convergencia), virtualmente todos los términos del lado derecho se cancelan, dejando

$$S - aS = 1$$

lo cual implica que

$$S = \frac{1}{1-a}$$

Esta misma técnica se puede usar para calcular $\sum_{i=1}^{\infty} i/2^i$, una suma que aparece con frecuencia. Escribimos

$$S = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \dots$$

y multiplicando por 2, obtenemos

$$2S = 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \frac{6}{2^5} + \dots$$

Restando las dos ecuaciones, llegamos a

$$S = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \dots$$

Así, $S = 2$.

Otro tipo de series comunes en el análisis es el de las series aritméticas. Cualquiera de estas series se puede evaluar con la fórmula básica:

$$\sum_{i=1}^N i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

Por ejemplo, para calcular la suma $2 + 5 + 8 + \dots + (3k-1)$, se reescribe como $3(1 + 2 + 3 + \dots + k) - (1 + 1 + 1 + \dots + 1)$, lo cual es claramente $3k(k+1)/2 - k$. Otra forma de recordar esto es sumar los términos primero y último (total $3k+1$), los términos segundo y penúltimo (total $3k+1$), etc. Ya que hay $k/2$ de esos pares, la suma total es $k(3k+1)/2$, que es la misma respuesta de antes.

Las dos fórmulas siguientes aparecen sólo de vez en cuando:

$$\sum_{i=1}^N i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

$$\sum_{i=1}^N i^k \approx \frac{N^{k+1}}{|k+1|} \quad k \neq -1$$

Cuando $k = -1$, la segunda fórmula anterior no es válida. Entonces se necesita la fórmula siguiente, que se usa mucho más en informática que en otras disciplinas matemáticas. Los números, H_n , se llaman números armónicos, y la suma se conoce como suma armónica. El error en la siguiente aproximación tiende a $\gamma \approx 0.57721566$ misma que se denomina constante de Euler.

$$H_N = \sum_{i=1}^N \frac{1}{i} \approx \log_e N$$

Estas dos fórmulas son sólo operaciones algebraicas generales:

$$\sum_{i=1}^N f(N) = Nf(N)$$

$$\sum_{i=n_0}^N f(i) = \sum_{i=1}^N f(i) - \sum_{i=1}^{n_0-1} f(i)$$

1.2.4. Aritmética modular

Se dice que a es congruente con b módulo n , que se escribe $a \equiv b \pmod{n}$, si y sólo si n divide a $a - b$. Intuitivamente, esto significa que el residuo es el mismo ya sea que a o b se divida entre n . Así, $81 \equiv 61 \equiv 1 \pmod{10}$. Como con la igualdad, si $a \equiv b \pmod{n}$, entonces $a + c \equiv b + c \pmod{n}$ y $a d \equiv b d \pmod{n}$.

Existen muchos teoremas que se aplican a la aritmética modular, algunos de los cuales requieren demostraciones extraordinarias en teoría de los números. Rara vez se usará aritmética modular, por lo cual bastarán los teoremas anteriores.

1.2.5. La palabra con D

Las dos formas más comunes de demostrar proposiciones en el análisis de estructuras de datos son la inducción y la contradicción o reducción al absurdo (y en ocasiones la demostración por intimidación, sólo por parte de los profesores). La mejor forma de demostrar que un teorema es falso es presentar un contraejemplo.

Demostración por inducción

Una demostración por inducción tiene dos partes estándar. El primer paso es comprobar un *caso base*, esto es, establecer que un teorema es cierto para alguno o algunos valores pequeños (usualmente degenerados); este paso casi siempre es trivial. A continuación se plantea una *hipótesis inductiva*. Por lo general, esto significa que el teorema se supone cierto para todos los casos hasta algún límite k . Con base en esta suposición se demuestra el teorema para el siguiente valor, que suele ser $k+1$. Esto demuestra el teorema (en tanto que k sea finita).

Como ejemplo, se demuestra que los números de Fibonacci, $F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \dots, F_i = F_{i-1} + F_{i-2}$, satisfacen $F_i < (5/3)^i$, para $i \geq 1$. (Algunas definiciones tienen $F_0 = 0$, lo cual desplaza la serie.) Para hacer esto, primero se verifica que el teorema sea cierto en los casos triviales. Es fácil verificar que $F_1 = 1 < 5/3$ y $F_2 = 2 < 25/9$; esto prueba el caso base. Suponemos que el teorema se cumple para $i = 1, 2, \dots, k$; ésta es la hipótesis inductiva. Para demostrar el teorema se necesita demostrar que $F_{k+1} < (5/3)^{k+1}$. Por definición se tiene

$$F_{k+1} = F_k + F_{k-1}$$

y se puede usar la hipótesis inductiva en el lado derecho para obtener

$$\begin{aligned} F_{k+1} &< (5/3)^k + (5/3)^{k-1} \\ &< (3/5)(5/3)^{k+1} + (3/5)^2(5/3)^{k-1} \\ &< (3/5)(5/3)^{k+1} + (9/25)(5/3)^{k-1} \end{aligned}$$

lo cual se reduce a

$$\begin{aligned} F_{k+1} &< (3/5 + 9/25)(5/3)^{k+1} \\ &< (24/25)(5/3)^{k+1} \\ &< (5/3)^{k+1} \end{aligned}$$

quedando demostrado el teorema.

Como segundo ejemplo, se establece el siguiente teorema:

TEOREMA 1.3.

$$\text{Si } n \geq 1, \text{ entonces } \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

DEMOSTRACIÓN:

La demostración es por inducción. Para el caso base, es fácil ver que el teorema se cumple cuando $n = 1$. Para la hipótesis inductiva, se supone que el teorema se cumple para $1 \leq k \leq n$. Con esta suposición, estableceremos que el teorema se cumple para $n+1$. Se tiene

$$\sum_{i=1}^{n+1} i^2 = \sum_{i=1}^n i^2 + (n+1)^2$$

Aplicando la hipótesis inductiva, se obtiene

$$\begin{aligned}\sum_{i=1}^{n+1} i^2 &= \frac{n(n+1)(2n+1)}{6} + (n+1)^2 \\&= (n+1) \left[\frac{n(2n+1)}{6} + (n+1) \right] \\&= (n+1) \frac{2n^2 + 7n + 6}{6} \\&= \frac{(n+1)(n+2)(2n+3)}{6}\end{aligned}$$

Así,

$$\sum_{i=1}^{n+1} i^2 = \frac{(n+1)[(n+1)+1][2(n+1)+1]}{6}$$

quedando demostrado el teorema.

Demostración por medio de un contraejemplo

La proposición $F_k \leq k^2$ es falsa. La forma más fácil de probar esto es calcular $F_{11} = 144 > 11^2$.

Demostración por contradicción

La demostración por contradicción o por reducción al absurdo procede suponiendo que el teorema es falso y demostrando que esta suposición implica que alguna propiedad conocida es falsa, y que con ello la suposición original fue errónea. Un ejemplo clásico es la demostración de que hay una cantidad infinita de números primos. Para probar esto se supone que el teorema es falso, así que existe el primo mayor p_k . Sean p_1, p_2, \dots, p_k todos los primos ordenados y considérese

$$N = p_1 p_2 p_3 \cdots p_k + 1$$

Claramente, N es mayor que p_k , así que por la suposición N no es primo. Sin embargo, ninguno de los números p_1, p_2, \dots, p_k dividen exactamente a N , porque siempre habrá un residuo de valor 1. Esto es una contradicción, porque cualquier número es primo o

un producto de primos. En consecuencia, la suposición original de que p_k es el mayor de los primos resulta falsa, lo cual implica que el teorema se cumple.

1.3. Breve introducción a la recursión

La mayoría de las funciones matemáticas conocidas se describen en términos de una sencilla fórmula. Por ejemplo, se pueden convertir temperaturas del sistema Fahrenheit al Celsius aplicando la fórmula:

$$C = 5(F - 32)/9$$

Dada esta fórmula, es trivial escribir una función en Pascal; eliminando las declaraciones y las proposiciones *begin* y *end*, la fórmula se traduce a una línea de Pascal.

Algunas veces las funciones matemáticas se definen de una manera menos estándar. Como ejemplo, se puede definir una función f , válida para enteros no negativos, que satisface $f(0) = 0$ y $f(x) = 2f(x-1) + x^2$. De esta definición se observa que $f(1) = 1$, $f(2) = 6$, $f(3) = 21$ y $f(4) = 58$. Una función que se define en términos de sí misma se conoce como *recursiva*[†]. Pascal permite que los procedimientos y las funciones sean recursivas. Es importante recordar que lo que Pascal ofrece es sólo un intento por seguir el espíritu recursivo. No todas las funciones matemáticamente recursivas son implantadas con eficiencia (o corrección) por la simulación de recursión de Pascal. La idea es que la función recursiva f sea expresable en sólo unas cuantas líneas, igual que una función no recursiva. La figura 1.2 muestra la implantación recursiva de f .

Las líneas [1] y [2] manejan lo que se denomina *caso base*, esto es, el valor para el cual la función se conoce directamente sin necesidad de la recursión. Así como la sola declaración de $f(x) = 2f(x-1) + x^2$ no tiene significado matemático si no se incluye el hecho de que $f(0) = 0$, la función recursiva en Pascal no tiene sentido sin un caso base. La línea [3] hace la llamada recursiva.

Hay varios puntos importantes acerca de la recursión que pueden llevar a confusión. Una pregunta común es: ¿No se trata sólo de lógica circular? La respuesta es que, si bien definimos una función en términos de sí misma, no definimos un caso particular de la función en términos de sí mismo. En otras palabras, la evaluación

Figura 1.2 Función recursiva

```
function f(x: integer): integer;
begin
[1]  if x = 0 then {caso base}
[2]    f := 0
      else
[3]    f := 2 * f(x-1) + x * x;
```

[†]Por lo general, no es buena idea usar la recursión en cálculos numéricos. Aquí se hace sólo para ilustrar las ideas básicas.

de $f(5)$ calculando $f(5)$ sería circular. La evaluación de $f(5)$ con base en $f(4)$ no es circular, a menos, por supuesto, que $f(4)$ finalmente se evalúe a partir de $f(5)$. Los dos aspectos más importantes quizás sean las preguntas *cómo* y *por qué*. En el capítulo 3 se resuelven formalmente el *cómo* y el *porqué*. Aquí se dará una descripción incompleta.

Resulta entonces que las llamadas recursivas no se manejan en una forma diferente de otras. Si suponemos que f es llamada con el valor 4, entonces la línea [3] requiere el cálculo de $2 * f(3) + 4 * 4$. Así, se hace una llamada para calcular $f(3)$. Esto requiere el cálculo de $2 * f(2) + 3 * 3$. Por lo tanto, se hace otra llamada para calcular $f(2)$. Esto significa que se debe evaluar $2 * f(1) + 2 * 2$. Para hacerlo, $f(1)$ se calcula como $2 * f(0) + 1 * 1$. Ahora se debe calcular $f(0)$. Ya que se trata de un caso base, se sabe *a priori* que $f(0) = 0$. Esto permite terminar el cálculo de $f(1)$, que ahora se ve que es 1. Entonces pueden determinarse $f(2)$, $f(3)$ y finalmente $f(4)$. Todo el trabajo requerido para hacer el seguimiento de las llamadas a funciones pendientes (aquellas que comenzaron, pero quedaron a la espera de que terminase una llamada recursiva), junto con sus variables, lo realiza el computador automáticamente. Sin embargo, un punto importante es que las llamadas recursivas se seguirán haciendo hasta que se alcance un caso base. Por ejemplo, el intento de evaluar $f(-1)$ originará llamadas recursivas a $f(-2)$, $f(-3)$, etc. Como esto nunca llegará a un caso base, el programa no será capaz de obtener una respuesta (que de cualquier manera está indefinida). En ocasiones, se comete un error mucho más sutil, que se muestra en la figura 1.3. El error en el programa de la figura 1.3 es que *malo(1)* está definida, en la línea [3], como *malo(1)*. Es obvio que esto no da ninguna pista de lo que en realidad es *malo(1)*. Así, el computador hará llamadas repetidas a *malo(1)* tratando de hallar su valor. Finalmente, el sistema agotará la memoria, y el programa abortará. Por lo general, se podría decir que esta función no sirve para un caso especial, pero que es correcta para cualquier otro caso. Esto no es cierto aquí, ya que *malo(2)* llama a *malo(1)*. Así, tampoco *malo(2)* puede evaluarse. Además, *malo(3)*, *malo(4)* y *malo(5)* hacen llamadas a *malo(2)*. Y como *malo(2)* no es evaluable, ninguno de esos valores

Con lo es. De hecho, este programa no sirve para ningún valor n , excepto para 0. Con programas recursivos no existen "casos especiales".

Estas consideraciones llevan a las dos primeras reglas fundamentales de la recursión:

1. *Casos base*. Siempre se deben tener algunos casos base, que se puedan resolver sin recursión.

Figura 1.3 Programa recursivo que no termina

```

function malo(n: integer): integer;
begin
  if n = 0 then
    malo := 0
  else
    malo := malo(n div 3 + 1) + n - 1;
end;
  
```

2. *Progreso*. Para los casos que deben resolverse recursivamente, la llamada recursiva siempre debe tender a un caso base.

A lo largo de este libro se usará la recursión para resolver problemas. Para poner un ejemplo no matemático, considérese un diccionario grande. Las palabras de los diccionarios se definen en términos de otras palabras. Cuando se busca una palabra, quizás no siempre se entienda la definición, así que se tendría que consultar algunas palabras de la definición.

De manera similar, podría no entenderse algo de lo que se encontró, y se puede seguir buscando indefinidamente. Como el diccionario es finito, tarde o temprano se llegará a un punto en donde se entenderán todas las palabras de alguna definición (para así entender la definición y desandar el camino hacia las otras definiciones); o se encontrará que las definiciones son circulares, llegando a un callejón sin salida, o que algunas de las palabras que se necesitan para entender una definición no se encuentran en el diccionario.

La estrategia recursiva para entender palabras es como sigue: si se sabe el significado de una palabra, queda resuelto el problema; si no es así, se busca la palabra en el diccionario. Si se entienden todas las palabras de la definición, queda resuelto el problema; si no, se descubre el significado de la definición buscando *recursivamente* las palabras que no se entienden. Este procedimiento terminará si el diccionario está bien definido, pero puede prolongarse indefinidamente si una palabra no está definida, o si está definida circularmente.

Visualización de números

Supóngase que se tiene un entero positivo, n , que se desea visualizar. La rutina tendrá la cabecera *visualizar(n)*. Supóngase que las únicas rutinas de E/S disponibles toman un número de un solo dígito y lo envían a la terminal. Llamaremos a esta rutina *visualizar_dígito*; por ejemplo, *visualizar_dígito(4)* enviará un 4 a la terminal.

La recursión da una solución muy clara a este problema. Para visualizar 76234, se necesita visualizar primero 7623 y después el 4. El segundo paso se logra fácilmente con la proposición *visualizar_dígito(n mod 10)*, pero el primero no parece más sencillo que el problema original. De hecho, es prácticamente el mismo problema, así que se puede resolver recursivamente con la instrucción *visualizar(n div 10)*.

Esto nos dice cómo resolver el problema general, pero aún es necesario asegurar que el programa no itere indefinidamente. Puesto que no se ha definido todavía un caso base, es claro que aún nos falta algo por hacer. El caso base será *visualizar_dígito(n)* si $0 \leq n < 10$. Ahora *visualizar(n)* está definido para cualquier número positivo desde 0 hasta 9, y los números mayores se definen en términos de números positivos menores. Así, no existe ningún ciclo. El procedimiento completo se muestra en la figura 1.4.

No se ha hecho esfuerzo alguno para hacer esto con eficiencia. Se podría haber evitado el uso de la función *mod* (que es muy costosa) porque $n \bmod 10 = n - (n \bmod 10) * 10$.

Recursión e inducción

Demostramos ahora de un forma (casi) rigurosa que el programa de visualización de números funciona. Lo haremos con una demostración por inducción.

TEOREMA 1.4

El algoritmo recursivo de visualización de números es correcto para $n = 0$.

DEMOSTRACIÓN (POR INDUCCIÓN SOBRE EL NÚMERO DE DÍGITOS EN n):

Primero, si n tiene un dígito, entonces el programa es trivialmente correcto, ya que sólo se hace la llamada a `visualizar_dígito`. Supóngase entonces que `visualizar` funciona para todos los números de k dígitos o menos. Un número de $k + 1$ dígitos se expresa con sus primeros k dígitos seguidos de su dígito menos significativo. Sin embargo, el número formado por los primeros k dígitos es exactamente $n \text{ div } 10$, el cual, por la hipótesis indicada, se visualiza correctamente, y el último dígito es $n \text{ mod } 10$, así que el programa visualiza correctamente cualquier número con $k+1$ dígitos. Luego, por inducción, todos los números se visualizan correctamente.

Esta demostración puede parecer un poco extraña, ya que es prácticamente idéntica a la descripción del algoritmo. Esto ilustra que en el diseño de un programa recursivo se puede *suponer* que todos los ejemplos más pequeños del mismo problema (los cuales están en el camino hacia un caso base) funcionan correctamente. El programa recursivo sólo necesita combinar soluciones de problemas más pequeños, las cuales se obtienen "mágicamente" gracias a la recursión, para llegar a una solución del problema actual. La justificación matemática de esto es la demostración por inducción. Esto da la tercera regla de la recursión.

Figura 1.4 Rutina recursiva para visualizar un entero

```
procedure visualizar(n: integer); {visualiza n no negativo}

begin
    if n < 0 then
        error('n es negativo')
    else
        if n < 10 then
            visualizar_dígito(n)
        else
            begin
                visualizar(n div 10);
                visualizar_dígito(n mod 10);
            end;
    end;
end;
```

*En todo el libro los identificadores aparecen castellanizados, con tilde y éñes; la mayoría de los compiladores no admitirán estos símbolos, por lo que será necesario eliminarlos. (N. def T.)

3. Regla de diseño. Supóngase que todas las llamadas recursivas funcionan.

Esta regla es importante porque significa que cuando se diseñan programas recursivos, generalmente no es necesario conocer los detalles del funcionamiento interno ni se tiene que rastrear la gran cantidad de llamadas recursivas. Con frecuencia, es en extremo difícil seguir la secuencia real de llamadas recursivas. Por supuesto, en muchos casos esto indica el buen uso de la recursión, ya que el computador se usa para encargarse de los detalles complicados.

El problema principal con la recursión reside en los costos ocultos de su funcionamiento interno. Aunque estos costos casi siempre son justificables, porque los programas recursivos no sólo simplifican el diseño del algoritmo sino que también tienden a proporcionar un código más claro, la recursión nunca debe usarse para sustituir un simple ciclo `for`. En la sección 3.3 se analizará con más detalle la sobrecarga ocasionada por la recursión.

Cuando se escriben rutinas recursivas, es crucial tener presentes las cuatro reglas básicas de la recursión.

1. *Casos base.* Siempre se deben tener algunos casos base que se puedan resolver sin recursión.
2. *Progreso.* Para los casos que se resuelven recursivamente, la llamada recursiva siempre debe tender a un caso base.
3. *Regla de diseño.* Supóngase que todas las llamadas recursivas funcionan.
4. *Regla del interés compuesto.* El trabajo nunca se debe duplicar resolviendo el mismo ejemplar de un problema en llamadas recursivas separadas.

La cuarta regla, que se justificará (junto con su sobrenombre) en secciones posteriores, es la razón de que por lo general se considere una mala idea usar la recursión para evaluar funciones matemáticas sencillas, como los números de Fibonacci. Con tal de que se tengan presentes estas reglas, la programación recursiva es sencilla.

Resumen

Este capítulo establece las bases para el resto del libro. El tiempo que tarde un algoritmo en manejar grandes cantidades de datos será un criterio importante para decidir si se trata de un buen algoritmo. (Por supuesto, la corrección es lo más importante.) La velocidad es relativa. Lo que es rápido para un problema en un computador puede ser lento para otro problema en un equipo diferente. Esos aspectos se empezarán a abordar en el siguiente capítulo, y se usarán las matemáticas presentadas aquí para establecer un modelo formal.

Ejercicios

- 1.1 Escriba un programa para resolver el problema de selección. Sea $k = n/2$. Dibuje una tabla que muestre el tiempo de ejecución del programa para diferentes valores de n .
- 1.2 Escriba un programa para resolver el problema de la sopa de letras.

1.3 Escriba un procedimiento para visualizar un número real arbitrario (que puede ser negativo) usando sólo *visualizar_dígito* como E/S.

1.4 El lenguaje C permite proposiciones de la forma

```
#include nombre_de_archivo
```

que lee *nombre_de_archivo* e inserta su contenido en el lugar de la proposición *include*. Las proposiciones *include* pueden ser anidadas; en otras palabras, el propio archivo *nombre_de_archivo* puede contener una proposición *include*, pero es obvio que un archivo no se puede incluir a sí mismo en ninguna cadena. Escriba un programa que lea un archivo y obtenga como salida el mismo archivo modificado por las proposiciones *include*.

1.5 Demuestre las siguientes fórmulas:

- $\log x < x$ para todo $x > 0$
- $\log(a^b) = b \log a$

1.6 Evalúe las siguientes sumatorias:

$$\text{a. } \sum_{i=0}^{\infty} \frac{1}{4^i}$$

$$\text{b. } \sum_{i=0}^{\infty} \frac{i}{4^i}$$

$$\text{*c. } \sum_{i=0}^{\infty} \frac{i^2}{4^i}$$

$$\text{**d. } \sum_{i=0}^{\infty} \frac{i^3}{4^i}$$

1.7 Calcule

$$\sum_{i=[n/2]}^n \frac{1}{i}$$

*1.8 ¿Qué es $2^{100} \pmod{5}$?

1.9 Sean F_i los números de Fibonacci definidos en la sección 1.2. Pruebe lo siguiente:

$$\text{a. } \sum_{i=1}^{n-2} F_i = F_n - 2.$$

$$\text{b. } F_n < \phi^n, \text{ con } \phi = (1 + \sqrt{5})/2.$$

**c. Proporcione una expresión precisa de forma cerrada de F_n .

1.10 Demuestre las siguientes fórmulas:

$$\text{a. } \sum_{i=1}^n (2i - 1) = n^2$$

$$\text{b. } \sum_{i=1}^n i^3 = \left(\sum_{i=1}^n i \right)^2$$

Referencias

Hay excelentes libros de texto que se dedican a las matemáticas contempladas en este capítulo. Un pequeño subconjunto es [1], [2], [3], [11], [14] y [15]. La referencia [11] está especialmente orientada al análisis de algoritmos. Éste es el primer volumen de una serie de tres que serán citados en todo el texto. En [6] se cubre material más avanzado.

En todo este libro se supondrá el conocimiento de Pascal [9]. Cuando sea necesario, se agregarán ciertas características en aras de la claridad. También se supone que el lector está familiarizado con apuntadores y recursión (el resumen sobre recursión en este capítulo está concebido como un rápido repaso). A lo largo del libro se intentará dar sugerencias sobre su uso cuando sea apropiado. Los lectores que no estén familiarizados con esto deben consultar [4], [8], [12], [13] o cualquier buen libro intermedio sobre programación.

El estilo general de programación se analiza en varios libros. Algunos de los clásicos son [5], [7] y [10].

1. M. O. Albertson y J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, Nueva York, 1988.
2. Z. Bavel, *Math Companion for Computer Science*, Reston Publishing Company, Reston, Va., 1982.
3. R. A. Brualdi, *Introductory Combinatorics*, North-Holland, Nueva York, 1977.
4. W. H. Burgue, *Recursive Programming Techniques*, Addison-Wesley, Reading, Mass., 1975.
5. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
6. R. L. Graham, D. E. Knuth y O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, Mass., 1989.
7. D. Gries, *The Science of Programming*, Springer-Verlag, Nueva York, 1981.
8. P. Helman y R. Veroff, *Walls and Mirrors: Intermediate Problem Solving and Data Structures*, 2a. ed., Benjamin/Cummings Publishing, Menlo Park, Calif., 1988.
9. K. Jensen y N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, Berlin, 1974.
10. B. W. Kernighan y P. J. Plauger, *The Elements of Programming Style*, 2a. ed., McGraw-Hill, Nueva York, 1978
11. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 2a. ed., Addison-Wesley, Reading, Mass., 1973.
12. E. B. Koffman, *Pascal: Problem Solving and Program Design*, 3a. ed., Addison-Wesley, Reading, Mass., 1988. (Existe versión en español de la primera edición por Addison-Wesley Iberoamericana.)
13. E. Roberts, *Thinking Recursively*, John Wiley & Sons, Nueva York, 1986.
14. F. S. Roberts, *Applied Combinatorics*, Prentice Hall, Englewood Cliffs, N.J., 1984.
15. A. Tucker, *Applied Combinatorics*, 2a. ed., John Wiley & Sons, Nueva York, 1984.

Análisis de algoritmos

Un *algoritmo* es un conjunto de instrucciones sencillas, claramente especificado, que se debe seguir para resolver un problema. Una vez que se da un algoritmo para un problema y se decide (de alguna forma) que es correcto, un paso importante es determinar la cantidad de recursos, como tiempo o espacio, que requerirá. Un algoritmo que resuelve un problema, pero tarda un año en hacerlo, difícilmente será de utilidad. De manera similar, un algoritmo que necesita un gigabyte de memoria principal no es (actualmente) útil.

En este capítulo se estudiará:

- Cómo calcular el tiempo que emplea un programa.
- Cómo reducir el tiempo de ejecución de un programa de días o años a fracciones de segundo.
- Los resultados del uso indiscriminado de la recursión.
- Algoritmos muy eficientes para elevar un número a una potencia y para calcular el máximo común divisor de dos números.

2.1. Soporte matemático

En general, el análisis requerido para estimar el uso de recursos de un algoritmo es una cuestión teórica, y por lo tanto, necesita un marco formal. Comenzaremos con algunas definiciones matemáticas.

Las siguientes cuatro definiciones serán válidas en todo el libro:

DEFINICIÓN: $T(n) = O(f(n))$ si existen constantes c y n_0 tales que $T(n) \leq c f(n)$ cuando $n \geq n_0$.

DEFINICIÓN: $T(n) = \Omega(g(n))$ si existen constantes c y n_0 tales que $T(n) \geq c g(n)$ cuando $n \geq n_0$.

DEFINICIÓN: $T(n) = \Theta(h(n))$ si y sólo si $T(n) = O(h(n))$ y $T(n) = \Omega(h(n))$.

DEFINICIÓN: $T(n) = o(p(n))$ si $T(n) = O(p(n))$ y $T(n) \neq \Theta(p(n))$.

El objetivo de estas definiciones es establecer un orden relativo entre funciones. Dadas dos funciones, por lo general hay puntos donde una función es menor que la otra, de modo que no tiene sentido afirmar que, por ejemplo, $f(n) < g(n)$. Así, se comparan sus *tasas de crecimiento relativos*. Cuando esto se aplica al análisis de algoritmos, se verá por qué ésta es la medida importante.

Aunque $1000n$ es mayor que n^2 para valores pequeños de n , n^2 crece con una tasa mayor, y así, n^2 finalmente será la función mayor. El punto de cambio es, en este caso, $n = 1000$. La primera definición dice que finalmente existe un punto n_0 pasado el cual $c \cdot f(n)$ es siempre al menos tan grande como $T(n)$, de tal modo que si se ignoran los factores constantes, $f(n)$ es al menos tan grande como $T(n)$. En este caso, se tiene $T(n) = 1000n$, $f(n) = n^2$, $n_0 = 1000$ y $c = 1$. Se podría usar $n_0 = 10$ y $c = 100$. Así se puede decir que $1000n = O(n^2)$ (orden n cuadrada). Esta notación se conoce como *O grande* (*Big-Oh*). Con frecuencia, en vez de decir "orden...", suele decirse "*O grande ...*".

Si se usan los operadores tradicionales de desigualdad para comparar las tasas de crecimiento, entonces la primera definición dice que la tasa de crecimiento de $T(n)$ es menor o igual (\leq) que la de $f(n)$. La segunda definición, $T(n) = \Omega(g(n))$ (dígase "omega"), dice que la tasa de crecimiento de $T(n)$ es mayor o igual (\geq) que la de $g(n)$. La tercera definición, $T(n) = \Theta(h(n))$ (dígase "theta"), dice que la tasa de crecimiento de $T(n)$ es igual ($=$) a la de $h(n)$. La última definición, $T(n) = o(p(n))$ (dígase "o pequeña"), dice que la tasa de crecimiento de $T(n)$ es menor ($<$) que la tasa de crecimiento de $p(n)$. Ésta es diferente de la *O grande*, porque *O grande* permite que las tasas de crecimiento se igualen.

Para demostrar que alguna función $T(n) = O(f(n))$, no se suelen aplicar estas definiciones formalmente, sino que se usa un repertorio de resultados conocidos. En general, esto significa que una demostración (o determinación de que la suposición es incorrecta) es un proceso muy sencillo y no debe implicar cálculo, excepto en circunstancias extraordinarias (que no es probable que ocurran en el análisis de un algoritmo).

Cuando se dice que $T(n) = O(f(n))$, se está garantizando que la función $T(n)$ crece a una velocidad no mayor que $f(n)$; así $f(n)$ es una *cota superior* de $T(n)$. Como esto implica que $f(n) = \Omega(T(n))$, se dice que $T(n)$ es una *cota inferior* de $f(n)$.

Por ejemplo, n^3 crece más rápido que n^2 , así se puede decir que $n^2 = O(n^3)$ o $n^3 = \Omega(n^2)$. $f(n) = n^2$ y $g(n) = 2n^2$ crecen a la misma velocidad, así que ambas, $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$, se cumplen. Cuando dos funciones crecen a la misma velocidad, la decisión de representar esto o no con $\Theta()$ puede depender del contexto particular. Intuitivamente, si $g(n) = 2n^2$, entonces $g(n) = O(n^4)$, $g(n) = O(n^3)$ y $g(n) = O(n^2)$ son técnicamente correctas, pero es obvio que la última opción es la mejor respuesta. Escribir $g(n) = \Theta(n^3)$ dice no sólo que $g(n) = O(n^3)$, sino también que el resultado es tan bueno (exacto) como es posible.

Las cosas importantes a saber son:

REGLA 1:

Si $T_1(n) = O(f(n))$ y $T_2(n) = O(g(n))$, entonces

- $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$,
- $T_1(n) * T_2(n) = O(f(n) * g(n))$,

Función	Nombre
c	constante
$\log n$	logarítmica
$\log^2 n$	logarítmica cuadrada
n	lineal
$n \log n$	
n^2	cuadrática
n^3	cúbica
2^n	exponencial

Figura 2.1 Tasas de crecimiento características

REGLA 2:

Si $T(x)$ es un polinomio de grado n , entonces $T(x) = \Theta(x^n)$.

REGLA 3:

$\log^k n = O(n)$ para cualquier k constante. Esto indica que los logaritmos crecen muy lentamente.

Para ver que la regla 1(a) es correcta, nótese que por definición existen cuatro constantes c_1 , c_2 , n_1 y n_2 , tales que $T_1(n) \leq c_1 f(n)$ para $n \geq n_1$ y $T_2(n) \leq c_2 g(n)$ para $n \geq n_2$. Sea $n_0 = \max(n_1, n_2)$. Entonces, para $n \geq n_0$, $T_1(n) \leq c_1 f(n)$ y $T_2(n) \leq c_2 g(n)$, de modo que $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. Sea $c_3 = \max(c_1, c_2)$. Entonces,

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq c_3 (f(n) + g(n)) \\ &\leq 2c_3 \max(f(n), g(n)) \\ &\leq c \max(f(n), g(n)) \end{aligned}$$

para $c = 2c_3$ y $n \geq n_0$.

Se dejan al lector como ejercicio las demostraciones de las otras relaciones dadas anteriormente. Esta información es suficiente para ordenar por tasas de crecimiento la mayoría de las funciones comunes (véase la figura 2.1).

Hay varios puntos que destacar. Primero, es muy mal estilo incluir constantes o términos de orden menor en una *O grande*. No se debe decir $T(n) = O(2n^2)$ o $T(n) = O(n^2 + n)$. En ambos casos, la forma correcta es $T(n) = O(n^2)$. Esto significa que en cualquier análisis que requiera una respuesta *O grande*, todos los tipos de simplificaciones son posibles. Por lo regular pueden ignorarse los términos de orden menor y desecharse las constantes. La precisión que se requiere en estos casos es considerablemente menor.

En segundo lugar, siempre se pueden determinar las tasas de crecimiento relativo de dos funciones $f(n)$ y $g(n)$ mediante el cálculo $\lim_{n \rightarrow \infty} f(n)/g(n)$, usando la regla de L'Hôpital si es necesario.[†]

[†]La regla de L'Hôpital establece que si $\lim_{n \rightarrow \infty} f(n) = \infty$ y $\lim_{n \rightarrow \infty} g(n) = \infty$, entonces $\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} f'(n)/g'(n)$ donde $f'(n)$ y $g'(n)$ son las derivadas de $f(n)$ y $g(n)$, respectivamente

El límite puede tener uno de cuatro valores:

- El límite es 0: esto significa que $f(n) = o(g(n))$.
- El límite es $c \neq 0$: esto significa que $f(n) = \Theta(g(n))$.
- El límite es ∞ : esto significa que $g(n) = o(f(n))$.
- El límite oscila: no hay ninguna relación (esto no sucederá en el contexto de este libro).

Casi nunca es necesaria la utilización de este método. En general la relación entre $f(n)$ y $g(n)$ se puede obtener por simple álgebra. Por ejemplo, si $f(n) = n \log n$ y $g(n) = n^{0.5}$, entonces para decidir cuál de las funciones $f(n)$ y $g(n)$ crece con mayor rapidez, lo que realmente se necesita es determinar qué crece más rápidamente, $\log n$ o $n^{0.5}$. Esto es como determinar qué crece con mayor rapidez, $\log n$ o n . Este problema es sencillo, porque es bien sabido que n crece más rápidamente que cualquier potencia de un logaritmo. Así, $g(n)$ crece con mayor rapidez que $f(n)$.

Una nota de estilo: es incorrecto decir $f(n) \leq O(g(n))$ porque la desigualdad está implícita en la definición. Es erróneo escribir $f(n) \geq O(g(n))$, pues no tiene sentido.

2.2. Modelo

Para analizar algoritmos en un marco formal, se necesita un modelo de computación. Nuestro modelo es básicamente un computador normal, en el cual las instrucciones se ejecutan de modo secuencial. El modelo tiene el repertorio estándar de instrucciones sencillas, como adición, multiplicación, comparación y asignación, pero, a diferencia de los computadores reales, éste tarda exactamente una unidad de tiempo hacer cualquier cosa (sencilla). Para ser razonable, se supondrá que, como un computador moderno, este modelo tiene enteros de tamaño fijo (digamos 32 bits) y que no tiene instrucciones refinadas, como la inversión de matrices o la clasificación, que claramente no se pueden hacer en una unidad de tiempo. También suponemos una memoria infinita.

Es obvio que este modelo tiene algunas debilidades. En la vida real, por supuesto, no todas las operaciones tardan exactamente el mismo tiempo. En particular, en este modelo una lectura de disco cuenta igual que una adición, aun cuando la adición suele ser varios órdenes de magnitud más rápida. También, al suponer memoria infinita, nunca hay que preocuparse por faltas de página, lo cual puede ser un problema real, en especial en algoritmos eficientes. Éste puede ser un grave problema en muchas aplicaciones.

2.3. Qué analizar

En general, el recurso más importante a analizar es el tiempo de ejecución. Varios factores afectan el tiempo de ejecución de un programa. Algunos, como el compilador y el computador usado, están más allá del alcance de cualquier modelo teórico, así que, aunque son importantes, no pueden ser tratados aquí. Los otros factores relevantes son el algoritmo usado y su entrada.

2.3. QUÉ ANALIZAR

El tamaño de la entrada suele ser la consideración principal. Se definen dos funciones, $T_{\text{prom}}(n)$ y $T_{\text{peor}}(n)$, como el tiempo de ejecución promedio y el del peor caso, respectivamente, empleados por un algoritmo para una entrada de tamaño n . Claro está, $T_{\text{prom}}(n) \leq T_{\text{peor}}(n)$. Si hay más de una entrada, esas funciones pueden tener más de un argumento.

Cabe señalar que en general la cantidad requerida es el tiempo del peor caso, a menos que se especifique otra cosa. Una razón para esto es que da una cota para todas las entradas, incluyendo entradas particularmente malas, que un análisis del caso promedio no puede ofrecer. La otra razón es que la cota del caso promedio suele ser mucho más difícil de calcular. En algunos casos, la definición de "promedio" puede afectar el resultado. (Por ejemplo, ¿qué es una entrada promedio para el problema siguiente?)

Como ejemplo, en la siguiente sección, se considerará el siguiente problema:

PROBLEMA DE LA SUMA DE LA SUBSECUENCIA MÁXIMA:

Dados enteros (posiblemente negativos) a_1, a_2, \dots, a_n , encontrar el valor máximo de $\sum_{k=i}^j a_k$. (Por conveniencia, la suma de la subsecuencia máxima es 0 si todos los enteros son negativos.)

Ejemplo:

Para la entrada $-2, 11, -4, 13, -5, -2$, la respuesta es 20 (a_2 hasta a_4).

Este problema es interesante sobre todo porque existen muchos algoritmos para resolverlo, y su rendimiento varía drásticamente. Estudiaremos cuatro algoritmos para resolver el problema. En la figura 2.2 se muestra el tiempo de ejecución en cierto computador (el modelo exacto carece de importancia) para esos algoritmos.

Hay varias cosas importantes que observar en esta tabla. Para una entrada pequeña, todos los algoritmos se ejecutan en un instante, así que si sólo se espera una pequeña cantidad de datos, no valdría la pena esforzarse por tratar de encontrar un algoritmo muy ingenioso. Por otro lado, existe un gran mercado hoy en día de reescritura de programas que fueron creados cinco años atrás, basados en la suposición, inválida hoy, de que la entrada es pequeña. Esos programas son

Figura 2.2 Tiempo de ejecución de varios algoritmos para la suma de la subsecuencia máxima (en segundos)

Algoritmo		1	2	3	4
Tiempo		$O(n^3)$	$O(n^2)$	$O(n \log n)$	$O(n)$
Tamaño de la entrada	$n = 10$	0.00103	0.00045	0.00066	0.00034
	$n = 100$	0.47015	0.01112	0.00486	0.00063
	$n = 1,000$	448.77	1.1233	0.05843	0.0333
	$n = 10,000$	NA	111.13	0.68631	0.03042
	$n = 100,000$	NA	NA	8.0113	0.29832

demasiado lentos ahora porque se usaron algoritmos deficientes. Para entradas grandes, el algoritmo 4 es claramente la mejor opción (aunque el 3 también es utilizable).

Segundo, los tiempos dados no incluyen el tiempo requerido para leer la entrada. Para el algoritmo 4, el tiempo de lectura de la entrada desde un disco es probablemente de un orden de magnitud mayor que el requerido para resolver el problema. Esto es característico en muchos algoritmos eficientes. La lectura de datos suele ser el cuello de botella; una vez que se leen los datos, el problema se puede resolver rápidamente. Para algoritmos ineficientes esto no es cierto, y hay que valerse de recursos de cómputo significativos. Así, es importante que, siempre que sea posible, los algoritmos sean suficientemente eficientes para no volverse el cuello de botella del problema.

La figura 2.3 muestra las tasas de crecimiento de los tiempos de ejecución de los cuatro algoritmos. Aun cuando esta gráfica sólo presenta valores para n entre 10 y 100, las tasas de crecimiento relativas son evidentes. Aunque el gráfico del algoritmo 3 parece lineal, es fácil verificar que no lo es usando una regla (o una hoja de papel). La figura 2.4 muestra el rendimiento para valores mayores. Esto es una ilustración espectacular de cuán inútiles son los algoritmos ineficientes, incluso para cantidades moderadamente grandes de datos.

2.4. Cálculo del tiempo de ejecución

Hay varias formas de calcular el tiempo de ejecución de un programa. La tabla anterior se obtuvo empíricamente. Si se espera que dos programas tomen tiempos parecidos, es probable que la mejor forma de decidir cuál es más rápido sea codificarlos y ejecutarlos.

En general existen varias ideas algorítmicas, y es deseable la eliminación rápida de las ineficaces, por lo que se suele requerir un análisis. Más aún, la habilidad de

Figura 2.3 Gráfica (n vs. milisegundos) de varios algoritmos de la suma de la subsecuencia máxima

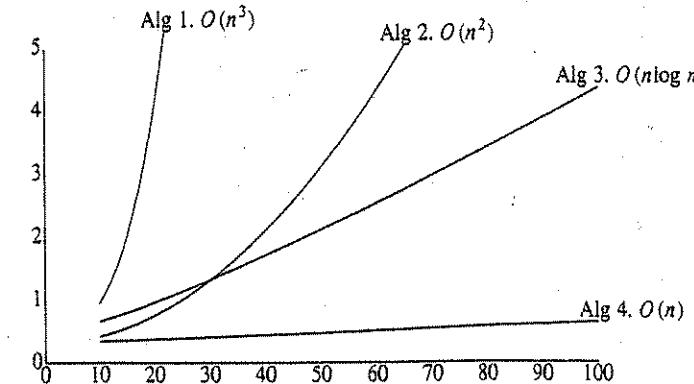
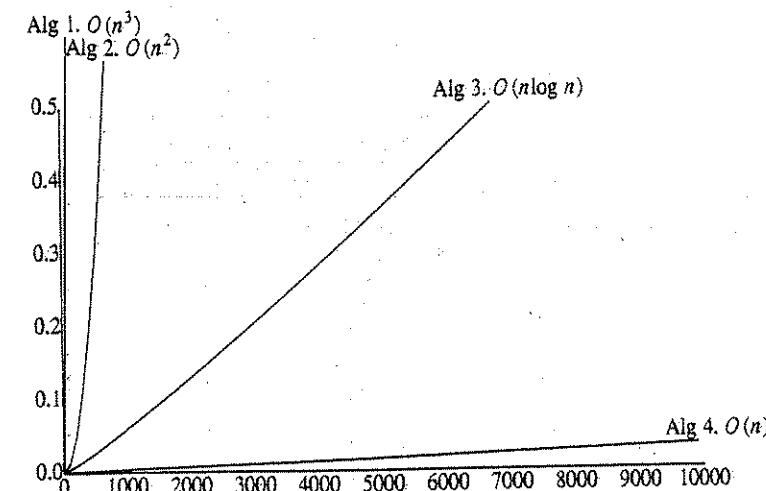


Figura 2.4 Gráfica (n vs. segundos) de varios algoritmos de la suma de la subsecuencia máxima



hacer un análisis permite aprender a diseñar algoritmos eficientes. El análisis también suele detectar los cuellos de botella, que vale la pena codificar con gran cuidado.

Para simplificar el análisis, adoptaremos el convenio de que no hay unidades de tiempo particulares. Así, se desechan las constantes iniciales. Tampoco se toman en cuenta los términos de orden menor, pues lo que en esencia estamos haciendo es calcular el tiempo de ejecución O grande. Como O grande es una cota superior, nunca se debe subestimar el tiempo de ejecución del programa. En efecto, la respuesta obtenida es una garantía de que el programa terminará en un cierto lapso; el programa puede terminar antes de éste, pero nunca después.

2.4.1. Un ejemplo sencillo

Aquí está un fragmento de programa sencillo para calcular $\sum_{i=1}^n i^3$:

```
function suma(n: integer): integer;
  var i, suma_parcial: integer;
begin
  {1}  suma_parcial := 0;
  {2}  for i := 1 to n do
  {3}    suma_parcial := suma_parcial + i * i * i;
  {4}  suma := suma_parcial;
end;
```

El análisis de este programa es sencillo. Las declaraciones no cuentan en el tiempo. Las líneas [1] y [4] cuentan por una unidad de tiempo cada una. La línea [3] cuenta por cuatro unidades cada vez que se ejecuta (dos multiplicaciones, una adición y una asignación) y se ejecuta n veces, para un total de $4n$ unidades. La línea [2] tiene el costo oculto de la iniciación de i , la comprobación de $i \leq n$ y el incremento de i . El costo total de todo ello es 1 para la iniciación, $n + 1$ para todas las comprobaciones y n para todos los incrementos, lo cual da $2n + 2$. Se ignoran los costos de llamar y retornar de la función, para un total de $6n + 4$. Así, se dice que la función es de orden $O(n)$.

Si tuviéramos que efectuar todo este trabajo cada vez que necesitemos analizar un programa, pronto la tarea se haría irrealizable. Por fortuna, como estamos dando la respuesta en términos de O grande, hay muchos medios de abreviar que se pueden seguir sin afectar la respuesta final. Por ejemplo, la línea [3] obviamente es un enunciado $O(1)$ (por ejecución), así que es inútil contar precisamente si lleva dos, tres o cuatro unidades; esto no importa. La línea [1] es obviamente insignificante comparada con el ciclo *for*, así que es inútil consumir tiempo aquí. Esto lleva a varias reglas generales obvias.

2.4.2. Reglas generales

REGLA 1-CICLOS FOR:

El tiempo de ejecución de un ciclo for es a lo más el tiempo de ejecución de las instrucciones que están en el interior del ciclo for (incluyendo las condiciones) por el número de iteraciones.

REGLA 2-CICLOS FOR ANIDADOS:

Analizarlos de adentro hacia afuera. El tiempo de ejecución total de una proposición dentro del grupo de ciclos for anidados es el tiempo de ejecución de la proposición multiplicado por el producto de los tamaños de todos los ciclos for.

Como un ejemplo, el siguiente fragmento de programa es $O(n^2)$:

```
for i := 1 to n do
    for j := 1 to n do
        k := k + 1;
```

REGLA 3-PROPOSICIONES CONSECUTIVAS:

Simplemente se suman (lo cual significa que el máximo es el único que cuenta; véase 1(a) en la página 18).

Como ejemplo, el siguiente fragmento de programa, que tiene trabajo $O(n)$ seguido de trabajo $O(n^2)$, también es $O(n^2)$:

```
for i := 1 to n do
    a[i] := 0;
for i := 1 to n do
    for j := 1 to n do
        a[i] := a[i] + a[j] + i + j;
```

REGLA 4-IF/ELSE:

Para el fragmento

```
if cond then
    S1
else
    S2
```

el tiempo de ejecución de una proposición if/else nunca es más grande que el tiempo de ejecución de la condición más el mayor de los tiempos de ejecución de S1 y S2.

Claramente, esto puede ser un tiempo sobrevalorado en algunos casos, pero nunca es una subestimación.

Otras reglas son obvias, pero una estrategia básica de análisis que funciona es ir del interior (o parte más profunda) hacia afuera. Si hay llamadas a funciones, es obvio que éstas deben ser analizadas primero. Si hay procedimientos recursivos, hay varias opciones. Si la recursión es un ciclo *for* ligeramente disfrazado, el análisis suele ser trivial. Por ejemplo, la siguiente función es en realidad sólo un ciclo sencillo y obviamente es $O(n)$:

```
function factorial(n: integer): integer;
begin
    if (n = 0) or (n = 1) then
        factorial := 1
    else
        factorial := n * factorial(n-1);
end;
```

Este ejemplo es realmente un uso ineficiente de la recursión. Cuando la recursión se usa adecuadamente, es difícil convertirla en una estructura iterativa sencilla. En este caso, el análisis implicará una relación de recurrencia que hay que resolver. Para ver qué puede suceder, considérese el siguiente programa, que resulta ser un uso horrible de la recursión.

{Cálculo de los números de Fibonacci como se describe en el capítulo 1}
{Supóngase $n \geq 0$ }

```
function fib(n: integer): integer;
begin
{1}    if (n = 0) or (n = 1) then
```

```

(2)      fib := 1
        else
(3)          fib := fib(n-1) + fib(n-2);
end;

```

A primera vista, éste parece ser un uso muy inteligente de la recursión. Sin embargo, si el programa se codifica y ejecuta para valores de n alrededor de 30, se hace evidente que el programa es terriblemente ineficiente. El análisis es bastante sencillo. Sea $T(n)$ el tiempo de ejecución de la función $\text{fib}(n)$. Si $n = 0$ o $n = 1$, entonces el tiempo de ejecución es algún valor constante, que es el tiempo requerido para evaluar la condición en la línea [1] y regresar. Se puede decir que $T(0) = T(1) = 1$, ya que las constantes no importan. El tiempo de ejecución para otros valores de n se mide entonces en relación con el tiempo de ejecución del caso base. Para $n > 2$, el tiempo de ejecución de la función es el trabajo constante de la línea [1] más el trabajo de la línea [3]. La línea [3] consta de una adición más dos llamadas a función. Puesto que las llamadas a función no son operaciones simples, se deben analizar por separado. La primera llamada a función es $\text{fib}(n-1)$ y, por tanto, por la definición de T , requiere $T(n-1)$ unidades de tiempo. Un razonamiento similar muestra que la segunda llamada requiere $T(n-2)$ unidades de tiempo. Entonces el tiempo total requerido es $T(n-1) + T(n-2) + 2$, donde 2 cuenta por el trabajo de la línea [1] más la adición de la línea [3]. Así, para $n \geq 2$, se tiene la siguiente fórmula del tiempo de ejecución de $\text{fib}(n)$:

$$T(n) = T(n-1) + T(n-2) + 2$$

Como $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$, es fácil demostrar por inducción que $T(n) \geq \text{fib}(n)$. En la sección 1.2.5, se demostró que $\text{fib}(n) < (5/3)^n$. Un cálculo similar demuestra que $\text{fib}(n) \geq (3/2)^n$, y así el tiempo de ejecución de este programa crece exponencialmente. Esto es lo peor posible. Con un arreglo simple y un ciclo *for*, se obtiene una reducción sustancial del tiempo de ejecución.

Este programa es lento porque se efectúa una cantidad enorme de trabajo redundante, violándose la cuarta regla de la recursión (la del interés compuesto) presentada en la sección 1.3. Nótese que la primera llamada de la línea [3], $\text{fib}(n-1)$, realmente calcula $\text{fib}(n-2)$ en algún momento. Esta información es desechara y recalculada en la segunda llamada de la línea [3]. La cantidad de información desechara se compone recursivamente y hace que el tiempo de ejecución sea enorme. Éste es tal vez el mejor ejemplo de la máxima "no calcular nada más de una vez", y no debe ahuyentar al estudiante del uso de la recursión. En el libro se verán usos sobresalientes de la recursión.

2.4.3. Soluciones al problema de la suma de la subsecuencia máxima

Ahora presentaremos los cuatro algoritmos que resuelven el problema de la suma de la subsecuencia máxima antes planteado. El primer algoritmo se ilustra en la figura 2.5.

```

function suma_subsecuencia_máx(var a: arreglo_entrada; n: integer): integer;
    var esta_suma, suma_máx, mejor_i, mejor_j, i, j, k: integer;
begin
    suma_máx := 0; mejor_i := 0; mejor_j := 0;
    for i := 1 to n do
        for j := i to n do begin
            esta_suma := 0;
            for k := i to j do
                esta_suma := esta_suma + a[k];
            if esta_suma > suma_máx then begin
                (actualiza suma_máx, mejor_i, mejor_j)
                suma_máx := esta_suma;
                mejor_i := i;
                mejor_j := j;
            end; {if}
        end; {for}
        suma_subsecuencia_máx := suma_máx;
    end;

```

Figura 2.5. Algoritmo 1

Convéngase de que este algoritmo funciona (no le debe llevar demasiado tiempo). El tiempo de ejecución es $O(n^3)$ y se debe por completo a las líneas [5] y [6], las cuales consisten en una proposición $O(1)$ inmersa en tres ciclos *for* anidados. El ciclo de la línea [2] es de tamaño n . El segundo ciclo tiene un tamaño $n - i + 1$, el cual podría ser pequeño, pero también puede ser de un tamaño n . Se debe suponer lo peor, con el entendimiento de que esta cota final puede ser un poco alta. El tercer ciclo tiene un tamaño $j - i + 1$, que, de nuevo, se debe suponer de tamaño n . El total es $O(1 \cdot n \cdot n \cdot n) = O(n^3)$. La proposición [1] sólo toma $O(1)$ en total, y las proposiciones [7] a [10] toman sólo $O(n^2)$ en total, puesto que son proposiciones sencillas dentro de dos iteraciones.

Un análisis más preciso, tomando en cuenta el tamaño real de estos ciclos, demuestra que la respuesta es $\Theta(n^3)$, y que el cálculo anterior fue demasiado grande en un factor de 6 (lo cual es correcto, porque las constantes no importan). En general esto es así en esta clase de problemas. El análisis preciso se obtiene de la suma $\sum_{i=1}^n \sum_{j=i}^n \sum_{k=i}^j 1$, que dice cuántas veces se ejecuta la línea [6]. La suma se puede evaluar de adentro hacia afuera, con las fórmulas de la sección 1.2.3. En particular, se usarán las fórmulas de la suma de los n primeros enteros y los n primeros cuadrados. Primero se tiene

$$\sum_{k=i}^j 1 = j - i + 1$$

Después se evalúa

$$\sum_{j=i}^n (j - i + 1) = \frac{(n - i + 1)(n - i + 2)}{2}$$

Esta suma se calcula observando que sólo es la suma de los primeros $n - i + 1$ enteros. Para terminar los cálculos, se evalúa

$$\begin{aligned} \sum_{i=1}^n \frac{(n - i + 1)(n - i + 2)}{2} &= \frac{1}{2} \sum_{i=1}^n i^2 - \left(n + \frac{3}{2}\right) \sum_{i=1}^n i + \frac{1}{2}(n^2 + 3n + 2) \sum_{i=1}^n 1 \\ &= \frac{1}{2} \frac{n(n+1)(2n+1)}{6} - \left(n + \frac{3}{2}\right) \frac{n(n+1)}{2} + \frac{n^2 + 3n + 2}{2} n \\ &= \frac{n^3 + 3n^2 + 2n}{6} \end{aligned}$$

Se puede evitar el tiempo de ejecución cúbico eliminando un ciclo *for*. Es obvio que esto no siempre es posible, pero en este caso hay una gran cantidad de cálculos innecesarios en el algoritmo. La ineficiencia que el algoritmo mejorado corrige se puede apreciar viendo que $\sum_{k=i}^j a_k = a_i + \sum_{k=i+1}^{j-1} a_k$, así que los cálculos de las líneas [5] y [6] del algoritmo 1 son indebidamente costosos. La figura 2.6 muestra un algoritmo mejorado. Claramente, el algoritmo 2 es $O(n^2)$; el análisis es aún más simple que el anterior.

Este problema tiene una solución $O(n \log n)$ recursiva y relativamente compleja, que ahora describiremos. Si no fuera porque existe una solución (lineal) $O(n)$, éste sería un excelente ejemplo del poder de la recursión. El algoritmo se vale de la estrategia de "divide y vencerás". La idea es partir el problema en dos subproblemas más o menos iguales, cada uno de los cuales es de tamaño igual a la mitad del

Figura 2.6 Algoritmo 2

```

function suma_subsecuencia_máx(var a: arreglo_entrada; n: integer): integer;
var esta_suma, suma_máx, mejor_i, mejor_j, i, j: integer;
begin
  suma_máx := 0; mejor_i := 0; mejor_j := 0;
  for i := 1 to n do begin
    esta_suma := 0;
    for j := i to n do begin
      esta_suma := esta_suma + a[j];
      if esta_suma > suma_máx then
        (actualiza suma_máx, mejor_i, mejor_j)
    end; {for j}
  end; {for i}
  suma_subsecuencia_máx := suma_máx;
end;

```

original. Entonces los subproblemas se resuelven recursivamente. Ésta es la parte de "dividir". La etapa de "vencer" consiste en unir las dos soluciones de los subproblemas, y posiblemente en hacer un poco de trabajo adicional, para llegar a una solución del problema global.

En este caso, la suma de la subsecuencia máxima puede estar en uno de tres lugares. O está entera en la primera mitad de la entrada, o en la segunda mitad, o bien pasa por el punto medio y se encuentra en ambas mitades. Los primeros dos casos se pueden resolver recursivamente. El último caso se obtiene encontrando la suma mayor en la primera mitad que incluya al último elemento de esa primera mitad y la suma más grande de la segunda mitad que incluya al primer elemento de la segunda mitad. Estas dos sumas se pueden sumar. Como ejemplo, considérese la siguiente entrada:

Primera mitad	Segunda mitad
4 -3 5 -2 -1 2 6 -2	

La suma de la subsecuencia máxima de la primera mitad es 6 (los elementos entre a_1 y a_3), y para la segunda mitad es 8 (los elementos entre a_6 y a_7).

La suma máxima de la primera mitad que incluye al último elemento de la primera mitad es 4 (elementos entre a_1 y a_4), y la suma máxima de la segunda mitad que incluye al primer elemento de la segunda mitad es 7 (elementos entre a_5 y a_7). Así, la suma máxima que abarca ambas mitades y pasa por el medio es $4 + 7 = 11$ (elementos entre a_1 y a_7).

Se ve, entonces, que de las tres formas de encontrar la subsecuencia de longitud máxima, para el ejemplo, la mejor forma es incluir elementos de ambas mitades. Así, la respuesta es 11. La figura 2.7 muestra una implantación de esta estrategia.

El código del algoritmo 3 merece algún comentario. La forma general de la llamada al procedimiento recursivo es pasar el arreglo de entrada junto con los límites izquierdo y derecho, que delimitan la porción del arreglo sobre la que se va a operar. Un programa manejador de una línea hace esto pasando los bordes 1 y n junto con el arreglo. El arreglo se pasa por referencia (usando la instrucción *var*), para evitar hacer una copia. Si se olvida hacer esto el tiempo de ejecución de la rutina puede variar drásticamente, como se verá después.

Las líneas [1] a [4] manejan el caso base. Si $izq = der$, entonces hay un elemento, y éste es la subsecuencia máxima si el elemento es no negativo. El caso $izq > der$ no es posible a menos que n sea negativo (aunque perturbaciones menores en el código podrían echar esto a perder). Las líneas [6] y [7] hacen dos llamadas recursivas. Se puede ver que las llamadas recursivas son siempre sobre problemas más pequeños que el original, aunque, de nuevo, perturbaciones menores en el código pueden destruir esta propiedad. Las líneas [8] a la [12] y después de la [13] a la [17] calculan las dos sumas máximas que alcanzan el centro. La suma de estos dos valores es la suma máxima que abarca ambas mitades. La subrutina *máx3* devuelve la posibilidad mayor de las tres.

Claro está, el algoritmo 3 requiere más esfuerzo de codificación que cualquiera de los dos anteriores. Sin embargo, la brevedad de un código no siempre implica que éste sea el mejor. Como se vio en la tabla anterior, donde se muestran los

```

function suma_sub_máx(var a: arreglo_entrada; izq, der: integer): integer;
var suma_máx_izq, suma_máx_der,
    suma_máx_izq_borde, suma_máx_der_borde,
    suma_borde_izq, suma_borde_der,
    centro, i: integer;
begin
[1]   if izq = der then {caso base}
[2]     if a[izq] > 0 then
[3]       suma_sub_máx := a[izq]
[4]     else
[5]       suma_sub_máx := 0
[6]     else
[7]       begin
[8]         centro := (izq + der) div 2;
[9]         suma_máx_izq := suma_sub_máx(a, izq, centro);
[10]        suma_máx_der := suma_sub_máx(a, centro+1, der);
[11]        suma_máx_izq_borde := 0; suma_borde_izq := 0;
[12]        for i := centro downto izq do
[13]          begin
[14]            suma_borde_izq := suma_borde_izq + a[i];
[15]            if suma_borde_izq > suma_máx_izq_borde then
[16]              suma_máx_izq_borde := suma_borde_izq;
[17]          end;
[18]          suma_máx_der_borde := 0; suma_borde_der := 0;
[19]          for i := centro + 1 to der do
[20]            begin
[21]              suma_borde_der := suma_borde_der + a[i];
[22]              if suma_borde_der > suma_máx_der_borde then
[23]                suma_máx_der_borde := suma_borde_der;
[24]            end;
[25]            suma_sub_máx := máx3(suma_máx_izq, suma_máx_der,
[26]                           suma_máx_izq_borde + suma_máx_der_borde);
[27]          end;
[28]      end;
[29]      function suma_subsecuencia_máx(var a: arreglo_entrada; n: integer): integer;
[30]      begin
[31]        suma_subsecuencia_máx := suma_sub_máx(a, 1, n);
[32]      end;

```

Figura 2.7 Algoritmo 3

tiempos de ejecución de los algoritmos, este algoritmo es considerablemente más rápido que los otros dos, excepto con entradas de tamaño reducido.

El tiempo de ejecución se analiza casi en la misma forma que el programa que calcula los números de Fibonacci. Sea $T(n)$ el tiempo que lleva resolver un problema de suma de la subsecuencia máxima de tamaño n . Si $n = 1$, entonces el programa usa una cantidad de tiempo constante para ejecutar las líneas [1] a [4], a la cual se tomará como la unidad. Así $T(1) = 1$. De otro modo, el programa debe realizar dos llamadas recursivas, los dos ciclos que están entre las líneas [9] y [17], y alguna cantidad reducida para la gestión interna, como en las líneas [5] y [18]. Los dos ciclos *for* se combinan para alcanzar todo elemento entre a_i y a_n , con un trabajo constante en el interior de los ciclos, así que el tiempo consumido en las líneas [9] a [17] es $O(n)$. El código de las líneas [1] a [5], [8] y [18] es constante en trabajo y se puede ignorar cuando se compara con $O(n)$. El resto del trabajo se realiza en las líneas [6] y [7]. Esas líneas resuelven dos problemas de suma de la subsecuencia máxima con tamaño $n/2$ (suponiendo n par). Así, estas líneas utilizan $T(n/2)$ unidades de tiempo cada una, para un total de $2T(n/2)$. El tiempo total consumido por el algoritmo, por lo tanto, es $2T(n/2) + O(n)$. Esto da las ecuaciones

$$T(1) = 1$$

$$T(n) = 2T(n/2) + O(n)$$

Para simplificar los cálculos, se puede sustituir por n el término $O(n)$ de la ecuación anterior; como de todos modos $T(n)$ se expresará en notación O grande, esto no afectará la respuesta. En el capítulo 7 se verá cómo resolver rigurosamente esta ecuación. Por ahora, si $T(n) = 2T(n/2) + n$, y $T(1) = 1$, entonces $T(2) = 4 = 2 * 2$, $T(4) = 12 = 4 * 3$, $T(8) = 32 = 8 * 4$, $T(16) = 80 = 16 * 5$. El patrón que es evidente, y que se puede obtener, es que si $n = 2^k$, entonces $T(n) = n * (k + 1) = n \log n + n = O(n \log n)$.

Este análisis supone que n es par, ya que de otra forma $n/2$ no está definido. Por la naturaleza recursiva del análisis, esto es realmente válido sólo cuando n es una potencia de 2, pues de otra forma tarde o temprano se llega a un subproblema que no es de tamaño par, y la ecuación es inválida. Cuando n no es una potencia de 2, se requiere un análisis un poco más complejo, pero el resultado O grande permanece sin cambio.

Si el arreglo de entrada no se declara como *var*, entonces este análisis deja de ser válido porque en cada llamada recursiva se tienen que hacer copias del arreglo. El análisis debe considerar este hecho. Para hacerlo, se calcula el número total de llamadas recursivas que efectúa el algoritmo. Sea $R(n)$ el número de llamadas recursivas hechas para resolver un problema de subsecuencia de tamaño n . Entonces $R(1) = 0$, ya que éste es el caso base. El número de llamadas recursivas de la línea [6] es igual al número de llamadas hechas cuando se resuelve el primer subproblema más, por supuesto, la llamada recursiva real. La misma lógica se aplica a la línea [7]. Ésta dice que $R(n) = 2R(n/2) + 2$. Esto se puede resolver (cuando n es una potencia de 2) para llegar a $R(n) = 2n - 2$. Si el arreglo de entrada no se declara como *var*, entonces se puede ver que si el arreglo es de tamaño n , el tiempo de ejecución será $O(n^2)$, que es dominado por el tiempo que tomará hacer las $2n - 2$ copias del arreglo

en el curso del algoritmo. Esta omisión de una palabra podría hacer que el algoritmo 3 fuera tan ineficiente como el algoritmo 2.

En capítulos futuros veremos varias aplicaciones ingeniosas de la recursión. Aquí presentamos un cuarto algoritmo para encontrar la suma de la subsecuencia máxima. Este algoritmo es más simple de implantar que el algoritmo recursivo y también es más eficiente. Se muestra en la figura 2.8.

Debe quedar claro por qué la cota del tiempo es correcta, pero lleva más tiempo entender por qué de hecho funciona. Esto se deja al lector. Una ventaja adicional del algoritmo es que sólo recorre una vez los datos, y una vez que se lee y procesa $a[i]$, no necesita recordarse esto. Así, si el arreglo está en disco o cinta, se puede leer secuencialmente, sin necesidad de almacenar parte alguna en la memoria principal. Más aún, en cualquier momento, el algoritmo puede dar correctamente una respuesta al problema de la subsecuencia para los datos que ya ha leído (los otros algoritmos no comparten esta propiedad). Los algoritmos que pueden hacer esto se conocen como *algoritmos en línea*. Un algoritmo en línea que requiere sólo espacio constante y se ejecuta en tiempo lineal es el mejor posible.

2.4.4. Logaritmos en el tiempo de ejecución

Es posible que el aspecto más confuso del análisis de algoritmos se centre en el logaritmo. Ya se ha visto que algunos algoritmos de "divide y vencerás" se ejecutarán en un tiempo $O(n \log n)$. Además de los algoritmos de "divide y vencerás", la aparición más frecuente de los logaritmos está alrededor de la siguiente regla general: *un algoritmo es $O(\log n)$ si usa un tiempo constante ($O(1)$) en dividir el problema*

Figura 2.8 Algoritmo 4

```

function suma_subsecuencia_máx(var a: arreglo_entrada; n: integer): integer;
var esta_suma, suma_máx, mejor_i, mejor_j, i, j: integer;
begin
  (1) i := 1; esta_suma := 0; suma_máx := 0; mejor_i := 0; mejor_j := 0;
  for j := 1 to n do begin
    esta_suma := esta_suma + a[j];
    if esta_suma > suma_máx then begin
      (actualiza suma_máx, mejor_i, mejor_j)
      suma_máx := esta_suma;
      mejor_i := i;
      mejor_j := j;
    end {if}
    else if esta_suma = 0 then begin
      i := j + 1;
      esta_suma := 0;
    end; {else}
  end; {for}
  suma_subsecuencia_máx := suma_máx;
end;

```

en partes (normalmente en $\frac{1}{2}$). Por otro lado, si se requiere un tiempo constante simplemente para reducir el problema en una cantidad constante (como hacer el problema más pequeño en 1), el algoritmo es $O(n)$.

Algo que debe ser obvio es que sólo esa clase especial de problemas puede ser $O(\log n)$. Por ejemplo, si la entrada es una lista de n números, un algoritmo sólo debe tardar $\Omega(n)$ en leer los datos. Así, cuando se habla de algoritmos $O(\log n)$ para esa clase de problemas, por lo regular se supone que la entrada fue leída antes. A continuación se dan tres ejemplos de comportamiento logarítmico.

Búsqueda binaria

El primer ejemplo se conoce cómo búsqueda binaria:

BÚSQUEDA BINARIA:

Dado un entero x y enteros a_1, a_2, \dots, a_n , los cuales están ordenados y en memoria, encontrar i tal que $a_i = x$, o devolver $i = 0$ si x no está en la entrada.

La solución obvia consiste en rastrear la lista de izquierda a derecha y se ejecuta en tiempo lineal. No obstante, este algoritmo no aprovecha el hecho de que la lista está ordenada, y por tanto, probablemente no sea la mejor solución. La mejor estrategia es probar si x está en la mitad de la lista. De ser así, la respuesta está a la mano. Si x es menor que el elemento del centro, se puede aplicar la misma estrategia al subarreglo ordenado a la izquierda del elemento central; si x es mayor que el elemento del centro, se buscará en la mitad derecha. (También está el caso de cuándo parar.) Para simplificar el código, se define $a_0 = x$: así la prueba es más simple. Esto requiere que el arreglo de entrada esté declarado empezando en 0; si esto no es posible, se requiere un código ligeramente más complicado. La figura 2.9 muestra el código para la búsqueda binaria (la respuesta es *mitad*).

Por supuesto todo el trabajo realizado por iteración dentro del ciclo es $O(1)$, así que el análisis requiere determinar el número de veces que se itera. El ciclo empieza con $alto - bajo = n - 1$ y termina cuando $alto - bajo \geq -1$. Cada vez que se itera el valor $alto - bajo$ debe reducirse al menos a la mitad de su valor original; así, el número de veces que se itera es a lo más $\lceil \log(n-1) + 2 \rceil$. (Por ejemplo, si $alto - bajo = 128$, entonces los valores máximos de $alto - bajo$ después de cada iteración son 64, 32, 16, 8, 4, 2, 1, 0, -1.) Así, el tiempo de ejecución es $O(\log n)$. En forma equivalente, se podría escribir una fórmula recursiva para el tiempo de ejecución, pero esta clase de enfoque por fuerza bruta es innecesario cuando se entiende qué está ocurriendo realmente y por qué.

La búsqueda binaria se puede ver como nuestra primera estructura de datos. Permite la operación *buscar* en tiempo $O(\log n)$, pero todas las demás operaciones (en particular *insertar*) requieren tiempo $O(n)$. En aplicaciones donde los datos son estáticos (esto es, no se permiten inserciones y eliminaciones), puede ser una estructura de datos muy útil. La entrada podría ser ordenada sólo una vez, y después los accesos serían rapidísimos. Un ejemplo podría ser un programa que necesita mantener información acerca de la tabla periódica de los elementos (problema típico en física y química). Esta tabla es relativamente estable, dado que es

```

function búsqueda_binaria(var a: arreglo_entrada; x: tipo_entrada; n: integer);
integer;
var bajo, mitad, alto: integer;
begin
[1]   a[0] := x;
[2]   bajo := 1;
[3]   alto := n;

repeat
[4]   mitad := (bajo + alto) div 2;
[5]   if bajo > alto then
[6]       mitad := 0
[7]   else if a[mitad] < x then
[8]       bajo := mitad + 1
[9]   else
[10]      alto := mitad - 1;
[11] until (a[mitad] = x);

[12]   búsqueda_binaria := mitad;
end;

```

Figura 2.9 Búsqueda binaria

poco frecuente que se agreguen elementos. Los nombres de los elementos se podrían almacenar ordenados. Como sólo hay unos 110 elementos, a lo más se requerirían ocho accesos para encontrar un elemento. Realizar la búsqueda secuencial requeriría muchos más accesos.

Algoritmo de Euclides

Un segundo ejemplo es el algoritmo de Euclides para calcular el máximo común divisor. El máximo común divisor (*mcd*) de dos enteros es el mayor entero que divide a ambos. Así, $mcd(50, 15) = 5$. El algoritmo de la figura 2.10 calcula $mcd(m, n)$, suponiendo que $m \geq n$. (Si $n > m$, la primera iteración del ciclo los intercambia).

El algoritmo funciona a base de calcular continuamente los restos hasta llegar a 0. La respuesta es el último distinto de cero. Así, si $m = 1989$ y $n = 1590$, entonces la secuencia de restos es 399, 393, 6, 3, 0. Por lo tanto, $mcd(1989, 1590) = 3$. Como lo muestra el ejemplo, éste es un algoritmo rápido.

Como antes, el tiempo de ejecución total del algoritmo depende de lo grande que sea la secuencia de residuos. Aunque $\log n$ parece ser una buena respuesta, no es en absoluto obvio que el valor del resto tenga que descender en un factor constante, pues se ve que el resto va de 399 a sólo 393 en el ejemplo. En efecto, el resto no disminuye en un factor constante en una iteración. Sin embargo, se puede demostrar que, después de dos iteraciones, el resto es a lo más la mitad de su valor original. Esto podría demostrar que el número de iteraciones es a lo más $2 \log n =$

```

function mcd(m, n: integer): integer;
var resto: integer;
begin
[1]   while n > 0 do begin
[2]       resto := m mod n;
[3]       m := n;
[4]       n := resto;
[5]   end; {while}
[6]   mcd := m;
end; {mcd}

```

Figura 2.10 Algoritmo de Euclides.

$O(\log n)$ y establecer el tiempo de ejecución. Esta demostración es fácil, por lo que se incluye aquí. Se infiere directamente del siguiente teorema.

TEOREMA 2.1

Si $m > n$, entonces $m \bmod n < m/2$.

DEMOSTRACIÓN:

Hay dos casos. Si $n \leq m/2$, entonces obviamente, como el resto es menor que n , el teorema se cumple para este caso. El otro caso es $n > m/2$. Pero entonces n cabe en m una vez con un resto $m - n < m/2$, demostrando el teorema.

Uno puede preguntarse si ésta es la mejor cota posible, ya que $2 \log n$ es cerca de 20 para este ejemplo, y sólo se realizaron siete operaciones. Resulta que la constante se puede mejorar ligeramente, para llegar a $1.44 \log n$, en el peor caso (el cual es alcanzable si m y n son números de Fibonacci consecutivos). El rendimiento en el caso promedio del algoritmo de Euclides requiere páginas y páginas de análisis matemático altamente complicado, y resulta finalmente que el número medio de iteraciones es de cerca de $(12 \ln 2 \ln n)/\pi^2 + 1.47$.

Exponenciación

El último ejemplo de esta sección trata de la elevación de un entero a una potencia (que también es entera). Los números que resultan de la exponenciación suelen ser bastante grandes, así que un análisis sólo sirve si se supone que se tiene una máquina con capacidad para almacenar tales enteros grandes (o un compilador que los pueda simular). Contaremos el número de multiplicaciones como la medida del tiempo de ejecución.

El algoritmo obvio para calcular x^n usa $n-1$ multiplicaciones. El algoritmo recursivo de la figura 2.11 lo hace mejor. Las líneas [1] a la [4] manejan el caso base de la recursión. De otra forma, si n es par, se tiene $x^n = x^{n/2} \cdot x^{n/2}$, y si n es impar, $x^n = x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x$.

n	Tiempo de CPU (T)	T/n^2	T/n^3	$T/n^2 \log n$
100	.022	.002200	.000022000	.0004777
200	.056	.001400	.000007000	.0002642
300	.118	.001311	.000004370	.0002299
400	.207	.001294	.000003234	.0002159
500	.318	.001272	.000002544	.0002047
600	.466	.001294	.000002157	.0002024
700	.644	.001314	.000001877	.0002006
800	.846	.001322	.000001652	.0001977
900	1.086	.001341	.000001490	.0001971
1000	1.362	.001362	.000001362	.0001972
1500	3.240	.001440	.000000960	.0001969
2000	5.949	.001482	.000000740	.0001947
4000	25.720	.001608	.000000402	.0001938

Figura 2.13 Tiempos de ejecución empíricos para la rutina anterior

El lector debe ser capaz de hacer instantáneamente el análisis de este programa. La figura 2.13 muestra el tiempo de ejecución real observado para esta rutina en un computador real. La tabla muestra que la última columna es más probable, y así el análisis obtenido debió ser correcto. Nótese que no hay gran diferencia entre $O(n^2)$ y $O(n^2 \log n)$, ya que los logaritmos crecen muy lentamente.

2.4.6. Un grano de sal

A veces se demuestra empíricamente que el análisis es una sobreestimación. Si éste es el caso, entonces el análisis necesita ser ajustado (por lo regular por una observación más cuidadosa), o puede suceder que el tiempo de ejecución *medio* sea significativamente menor que el tiempo de ejecución en el peor caso y no hay mejora posible en la cota. Hay muchos algoritmos complicados para los cuales la cota del peor caso se obtiene por alguna entrada mala pero, en general, está sobreestimada en la práctica. Desafortunadamente, para la mayoría de esos problemas, un análisis del caso medio es complejo en extremo (y en algunos casos permanece sin solución),

y una cota del peor caso, aun cuando sea muy pesimista, es el mejor resultado analítico conocido.

Resumen

Este capítulo proporciona varios indicios de cómo analizar la complejidad de los programas. Lamentablemente, no es una guía completa. Los programas simples, por lo regular, tienen análisis sencillos, pero no siempre es éste el caso. Por ejemplo, más adelante en el texto veremos un algoritmo de ordenación (de Shell, capítulo 7) y un algoritmo de mantenimiento de conjuntos ajenos (capítulo 8), cada uno de los cuales requerirá cerca de 20 líneas de código. El análisis de la ordenación de Shell no está completo aún, y el de los conjuntos ajenos tiene un análisis extremadamente difícil y requiere páginas y páginas de intrincados cálculos. La mayoría de los análisis que se encontrarán aquí serán sencillos e implicarán conteo a través de ciclos.

Una clase interesante de análisis, que no se ha tocado aún, es el análisis de la cota inferior. Se verá un ejemplo en el capítulo 7, donde se demuestra que cualquier algoritmo que clasifica por medio de comparaciones únicamente requiere $\Omega(n \log n)$ comparaciones en el peor caso. Las demostraciones de la cota inferior suelen ser las más difíciles, porque no se aplican a un algoritmo sino a una clase de algoritmos que resuelven un problema.

Terminamos con la mención de que algunos algoritmos descritos aquí tienen aplicación real. El algoritmo del *maíz* y el de la exponenciación se usan en criptografía. Específicamente, un número de 200 dígitos se eleva a una potencia grande (por lo regular otro número de 200 dígitos), conservando sólo los primeros 200 dígitos después de cada multiplicación. Como los cálculos requieren tratar con números de 200 dígitos, la eficiencia es obviamente importante. El algoritmo directo de la exponenciación puede requerir cerca de 10^{20} multiplicaciones, mientras que el algoritmo presentado requiere sólo 1200.

Ejercicios

2.1 Ordene las siguientes funciones por tasa de crecimiento: $n, \sqrt{n}, n^{1.5}, n^2, n \log n, n \log \log n, n \log^2 n, n \log(n^2), 2^n, 2^{2^n}, 37, n^2 \log n, n^3$. Indique cuáles crecen con la misma tasa.

2.2 Supongamos 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. $\frac{T_1(n)}{T_2(n)} = O(1)$

```

function potencia(x, n: integer); integer;
begin
{1}    if n = 0 then
{2}        potencia := 1
    else
{3}    if n = 1 then
        potencia := x
    else
{5}    if even(n) then
{6}        potencia := potencia(x * x, n div 2)
    else
{7}        potencia := potencia(x * x, n div 2) * x;
end;

```

Figura 2.11 Exponenciación eficiente

Por ejemplo, para calcular x^{62} el algoritmo hace los siguientes cálculos, en los que intervienen sólo nueve multiplicaciones:

$$x^3 = (x^2)x, \quad x^7 = (x^3)^2x, \quad x^{15} = (x^7)^2x, \quad x^{31} = (x^{15})^2x, \quad x^{62} = (x^{31})^2$$

Claramente, el número de multiplicaciones requeridas es a lo más $2 \log n$, porque a lo más se necesitan dos multiplicaciones (si n es impar) para partir en dos el problema. De nuevo, se puede escribir y resolver una fórmula recursiva. La simple intuición hace obvia la necesidad de un enfoque por la fuerza bruta.

A veces es interesante ver cuánto código se puede abreviar sin afectar la corrección. En la figura 2.11, de hecho, {3} a {4} son innecesarias ya que si $n = 1$, entonces la línea {7} resuelve ese caso. También la línea {7} se puede reescribir como

{7} potencia := potencia(x, n-1) * x;

sin afectar la corrección del programa. En efecto, el programa seguirá ejecutándose en $O(\log n)$, porque la secuencia de multiplicaciones es la misma que antes. Sin embargo, todas las alternativas a la línea {6} que se encuentran a continuación no sirven, aunque parezcan correctas:

- {6a} potencia := potencia(potencia(x, 2), n div 2)
- {6b} potencia := potencia(potencia(x, n div 2), 2)
- {6c} potencia := potencia(x, n div 2) * potencia(x, n div 2)

Las líneas {6a} y {6b} son incorrectas porque cuando $n = 2$, una de las llamadas recursivas a *potencia* tiene 2 como el segundo argumento. Así, no se obtiene progreso, y resulta un ciclo infinito (que con el tiempo se bloqueará).

Usar la línea {6c} afecta la eficiencia porque ahora hay dos llamadas recursivas de tamaño $n / 2$ en vez de una. Un análisis demostrará que ese tiempo de ejecución

ya no será $O(\log n)$. Se deja como ejercicio al lector determinar el nuevo tiempo de ejecución.

2.4.5. Verificación del análisis

Una vez que se ha realizado un análisis, es deseable ver si la respuesta es correcta y es la mejor posible. Una forma de hacer esto es codificar el programa y ver si los tiempos de ejecución observados empíricamente concuerdan con los predichos por el análisis. Cuando n se duplica, el tiempo de ejecución crece en un factor de 2 para programas lineales, 4 para programas cuadráticos, y 8 para programas cúbicos. Los programas que se ejecutan en tiempo logarítmico sólo se incrementan en una cantidad constante cuando n se duplica, y los programas que se ejecutan en $O(n \log n)$ tardan ligeramente más del doble bajo las mismas circunstancias. Estos incrementos pueden ser difíciles de determinar si los términos de orden menor tienen coeficientes relativamente grandes y n no es lo bastante grande. Un ejemplo es el salto de $n = 10$ a $n = 100$ en el tiempo de ejecución de las diferentes implantaciones del problema de la suma de la subsecuencia máxima. También puede ser muy difícil diferenciar los programas lineales de los programas $O(n \log n)$ a partir de una simple comprobación empírica.

Otro truco que se usa mucho para verificar que algún programa es $O(f(n))$ consiste en calcular los valores de $T(n) / f(n)$ para un intervalo de n (por lo regular espaciado en factores de 2), donde $T(n)$ es el tiempo de ejecución empíricamente observado. Si $f(n)$ es una respuesta cercana al tiempo de ejecución, entonces los valores calculados convergen a una constante positiva. Si $f(n)$ es una sobreestimación, los valores convergen a cero. Si $f(n)$ está subestimada y, por lo tanto, es incorrecta, los valores divergen.

Por ejemplo, el fragmento de programa de la figura 2.12 calcula la probabilidad de que dos enteros positivos distintos, menores o iguales que n y escogidos al azar, sean primos relativos. (Cuando n aumenta, la respuesta se acerca a $6/\pi^2$.)

Figura 2.12 Cálculo de la probabilidad de que dos números aleatorios sean primos relativos

```

rel := 0; tot := 0;

for i := 1 to n do
    for j := i + 1 to n do begin
        tot := tot + 1;
        if mcd(i, j) = 1 then
            rel := rel + 1;
    end;
writeln('el porcentaje de pares primos relativos es', rel/tot);

```

d. $T_1 = O(T_2(n))$

2.3 ¿Qué función crece más rápido: $n \log n$ o $n^{1+\epsilon} / \sqrt{\log n}$ $\epsilon > 0$?

2.4 Demuestre que para cualquier constante, k , $\log^k n = o(n)$.

2.5 Encuentre dos funciones $f(n)$ y $g(n)$ tales que ni $f(n) = O(g(n))$ ni $g(n) = O(f(n))$.

2.6 Para cada uno de los seis fragmentos de programas siguientes:

- Haga un análisis del tiempo de ejecución (O grande).
- Implante el código en el lenguaje preferido, y dé el tiempo de ejecución para diferentes valores de n .
- Compare su análisis con los tiempos de ejecución reales.

(1) sum := 0;
for i := 1 to n do
 sum := sum + 1;

(2) sum := 0;
for i := 1 to n do
 for j := 1 to n do
 sum := sum + 1;

(3) sum := 0;
for i := 1 to n do
 for j := 1 to n ** 2 do
 sum := sum + 1;

(4) sum := 0;
for i := 1 to n do
 for j := 1 to i do
 sum := sum + 1;

(5) sum := 0;
for i := 1 to n do
 for j := 1 to i ** 2 do
 for k := 1 to j do
 sum := sum + 1;

(6) sum := 0;
for i := 1 to n do
 for j := 1 to i ** 2 do
 if j mod i = 0 then
 for k := 1 to j do
 sum := sum + 1;

2.7 Supongamos que necesitamos generar una permutación *aleatoria* de los primeros n enteros. Por ejemplo, $\{4, 3, 1, 5, 2\}$ y $\{3, 1, 4, 2, 5\}$ son permutaciones legales, pero $\{5, 4, 1, 2, 1\}$ no lo es, porque un número (1) está duplicado y otro (3) no está. Esta rutina se usa a menudo en simulación de algoritmos. Suponemos la existencia de un generador de números aleatorios, *aleat_ent(i, j)*, el cual genera enteros entre i y j con una probabilidad igual. Aquí se presentan tres algoritmos:

- Llenar el arreglo a desde $a[1]$ hasta $a[n]$ como sigue: para llenar $a[i]$ generar números aleatorios hasta que se tiene uno que no está ya en $a[1], a[2], \dots, a[i-1]$.
- Igual que el algoritmo 1, pero mantener un arreglo adicional llamado *usado*. Cuando un número aleatorio, *alea*, se pone en el arreglo a , se pone $usado[alea] = 1$. Esto significa que cuando se llena $a[i]$ con un número aleatorio, se puede comprobar en un paso si el número ya ha sido utilizado, en vez de los (posiblemente) $i - 1$ pasos del primer algoritmo.
- Llenar el arreglo de forma que $a[i] = i$. Después
 - for i := 2 to n do
 intercambia($a[i]$, $a[\text{aleat_ent}(1,i)]$);
 - Demostrar que los tres algoritmos generan sólo permutaciones legales y que todas las permutaciones son igualmente probables.
 - Dar un análisis (O grande) tan preciso como sea posible del tiempo de ejecución *esperado* de cada algoritmo.
 - Escribir programas (separados) para ejecutar cada algoritmo 10 veces, para obtener un buen promedio. Ejecutar el programa (1) para $n = 250, 500, 1000, 2000$; el programa (2) para $n = 2500, 5000, 10\,000, 20\,000, 40\,000, 80\,000$, y el programa (3) para $n = 10\,000, 20\,000, 40\,000, 80\,000, 160\,000, 320\,000, 640\,000$.
 - Comparar el análisis con los tiempos de ejecución reales.
 - ¿Cuál es el tiempo de ejecución en el peor caso para cada algoritmo?
- Complete la tabla de la figura 2.2 con los cálculos de los tiempos de ejecución que fueron demasiado grandes para ser simulados. Interpolate los tiempos de ejecución de esos algoritmos y calcule el tiempo requerido para obtener la suma de la subsecuencia máxima de un millón de números. ¿Qué suposiciones hizo?
- ¿Cuánto tiempo se requiere para calcular $f(x) = \sum_{i=0}^n a_i x^i$ en los dos siguientes casos?
 - Usando una rutina simple para realizar la exponentiación.
 - Usando la rutina de la sección 2.4.4.
- Considere el siguiente algoritmo (conocido como *regla de Horner*) para evaluar $f(x) = \sum_{i=0}^n a_i x^i$.


```
poli := 0;
for i := n downto 0 do
    poli := x * poli + a_i;
```

- a. Muestre cómo se realizan los pasos para este algoritmo con $x = 3$, $f(x) = 4x^4 + 8x^3 + x + 2$.
- b. Explique por qué funciona el algoritmo.
- c. ¿Cuál es el tiempo de ejecución de este algoritmo?
- 2.11 Proporcione un algoritmo eficiente para determinar si existe un entero i tal que $a_i = i$ en un arreglo de enteros $a_1 < a_2 < a_3 < \dots < a_n$. ¿Cuál es el tiempo de ejecución de su algoritmo?
- 2.12 Proporcione algoritmos eficientes (junto con los análisis del tiempo de ejecución) para:
- encontrar la suma de la subsecuencia mínima;
 - *encontrar la suma de la subsecuencia mínima *positiva*;
 - *encontrar el *producto* de la subsecuencia máxima.
- 2.13 a. Escriba un programa para determinar si un entero positivo, n , es primo.
- b. En función de n , ¿cuál es el tiempo de ejecución del peor caso de su programa? (Debe ser capaz de hacer esto en $O(\sqrt{n})$.)
- c. Sea B igual al número de bits en la representación binaria de n . ¿Cuál es el valor de B ?
- d. En términos de B , ¿cuál es el tiempo de ejecución del peor caso de su programa?
- e. Compare los tiempos de ejecución para determinar si un número de 20 bits y uno de 40 bits son primos.
- f. ¿Es más razonable dar tiempos de ejecución en términos de n o de B ? ¿Por qué?
- *2.14 La criba de Eratóstenes es un método para obtener todos los primos menores que n . Se empieza haciendo una tabla de enteros entre 2 y n . Se encuentra el entero más pequeño, i , que no esté marcado, se imprime i , y se marcan $i, 2i, 3i, \dots$. El algoritmo termina cuando $i > \sqrt{n}$. ¿Cuál es el tiempo de ejecución de este algoritmo?
- 2.15 Demuestre que x^{62} se puede calcular con sólo ocho multiplicaciones.
- 2.16 Escriba la rutina de la exponentiación rápida sin recursión.
- 2.17 Proporcione una cuenta precisa del número de multiplicaciones usadas por la rutina de la exponentiación rápida. (*Sugerencia:* Considere la representación binaria de n .)
- 2.18 Se analizan los programas A y B y se encuentra que tienen un tiempo de ejecución en el peor caso no mayor que $150 n \log_2 n$ y n^2 , respectivamente. Responda las siguientes preguntas, si es posible:
- ¿Qué programa tiene la mejor garantía en el tiempo de ejecución, para valores grandes de n ($n > 10\,000$)?
 - ¿Qué programa tiene la mejor garantía en el tiempo de ejecución, para valores pequeños de n ($n < 100$)?
 - ¿Qué programa se ejecutará más rápidamente en promedio para $n = 1000$?

- d. ¿Es posible que el programa B se ejecute más rápidamente que el programa A para *todas* las entradas posibles?

- 2.19 Un elemento mayoría en un arreglo, A , de tamaño n es un elemento que aparece más de $n/2$ veces (así, existe a lo más uno). Por ejemplo, el arreglo

3, 3, 4, 2, 4, 4, 2, 4, 4

tiene un elemento mayoría (4), mientras que el arreglo

3, 3, 4, 2, 4, 4, 2, 4

no lo tiene. Si no hay elemento mayoría, el programa debe indicarlo. Aquí está el bosquejo de un algoritmo para resolver el problema:

Primero, se encuentra un *candidato* a elemento mayoría (ésta es la parte más difícil). Este candidato es el único elemento que podría ser el elemento mayoría. El segundo paso determina si este candidato es en realidad el mayoría. Esto sólo es una búsqueda secuencial en todo el arreglo. Para encontrar un candidato en el arreglo, A , se forma un segundo arreglo, B . Entonces se comparan A_1 y A_2 . Si son iguales se añade uno de ellos en B ; si no, no se hace nada. Entonces se comparan A_3 y A_4 . De nuevo, si son iguales, se añade uno de ellos en B ; si no, no se hace nada. Se continúa en esta forma hasta leer el arreglo completo. Entonces se encuentra recursivamente un candidato para B ; éste es el candidato de A (¿por qué?).

- ¿Cómo termina la recursión?
- ¿Cómo se maneja el caso cuando n es impar?
- ¿Cuál es el tiempo de ejecución del algoritmo?
- ¿Cómo se puede evitar el uso del arreglo adicional B ?
- Escriba un programa para calcular el elemento mayoría.

- *2.20 ¿Por qué es importante suponer que los enteros en el computador modelo tienen un tamaño fijo?

- 2.21 Consideré el problema de la sopa de letras descrito en el capítulo 1. Suponga que se fija la longitud máxima de palabra en 10 caracteres.

- En términos de r y c , que son el número de filas y columnas del juego, y W , que es el número de palabras, ¿cuál es el tiempo de ejecución de los algoritmos descritos en el capítulo 1?
- Suponga que la lista de palabras está ordenada previamente. Muestre cómo usar búsqueda binaria para obtener un algoritmo con un tiempo de ejecución significativamente mejor.

- 2.22 Supongamos que la línea [8] de la rutina de la búsqueda binaria tiene la proposición $bajo := mitad$ en vez de $bajo := mitad + 1$. ¿Todavía funcionará la rutina?

- 2.23 Suponga que las líneas [6] y [7] del algoritmo 3 (figura 2.7) se sustituyen por

- [6] suma_máx_izq := suma_sub_máx(a, izq, centro-1);
- [7] suma_máx_der := suma_sub_máx(a, centro, der);

¿Funcionará todavía la rutina?

*2.24. El ciclo más interno del algoritmo cúbico del problema de la subsecuencia máxima realiza $n(n+1)(n+2)/6$ iteraciones del código más interno. La versión cuadrática realiza $n(n+1)/2$ iteraciones. La versión lineal realiza n iteraciones. ¿Qué patrón es evidente? ¿Se puede dar una explicación combinatoria del fenómeno?

Referencias

El análisis del tiempo de ejecución de algoritmos fue popularizado por primera vez por Knuth en la serie de tres partes [5], [6] y [7]. El análisis del algoritmo del *mcd* aparece en [6]. Otro de los primeros textos sobre el tema es [1].

Las notaciones O grande, theta grande y o pequeña fueron presentadas por Knuth en [8]. Todavía no hay un acuerdo uniforme en torno a la materia, en especial cuando se habla de usar $\Theta()$. Mucha gente prefiere usar $O()$, aun cuando es menos expresiva. Además, $O()$ se sigue usando en algunas partes para expresar cota inferior, cuando $\Omega()$ está definida para eso.

El problema de la suma de la subsecuencia máxima es de [3]. La serie de libros [2], [3] y [4] demuestra cómo optimizar la velocidad de programas.

1. A. V. Aho, J. E. Hopcroft y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. J. L. Bentley, *Writing Efficient Programs*, Prentice Hall, Englewood Cliffs, N. J., 1982.
3. J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, Mass., 1986.
4. J. L. Bentley, *More Programming Pearls*, Addison-Wesley, Reading, Mass., 1988.
5. D. E. Knuth, *The Art of Computer Programming*, Vol 1: *Fundamental Algorithms*, 2a. ed., Addison-Wesley, Reading, Mass., 1973.
6. D. E. Knuth, *The Art of Computer Programming*, Vol 2: *Seminumerical Algorithms*, 2a. ed., Addison-Wesley, Reading, Mass., 1981.
7. D. E. Knuth, *The Art of Computer Programming*, Vol 3: *Sorting and Searching*, Addison-Wesley, Reading, Mass., 1975.
8. D. E. Knuth, "Big Omicron and Big Omega and Big Theta", *ACM SIGACT News*, 8 (1976), págs. 18-23.

Listas, pilas y colas

En este capítulo se estudian tres de las estructuras de datos más sencillas y básicas. Prácticamente todo programa significativo usará explícitamente al menos una de estas estructuras, y siempre se usa una pila implícitamente, se declare o no. Entre los aspectos importantes del capítulo:

- Introduciremos el concepto de tipo de datos abstracto (TDA).
- Mostraremos cómo realizar con eficiencia operaciones sobre listas.
- Introduciremos el TDA pila y su uso en la implantación de la recursión.
- Presentaremos el TDA cola y su uso en sistemas operativos y en diseño de algoritmos.

Debido a que estas estructuras de datos son tan importantes, uno puede esperar que sean difíciles de implantar. De hecho, es extremadamente fácil codificarlas; la principal dificultad es mantener suficiente disciplina para escribir buen código de propósito general para las rutinas que, en general, sólo son de unas pocas líneas.

3.1. Tipos de datos abstractos (TDA)

Una de las reglas básicas concernientes a la programación es que ninguna rutina debe exceder una página. Esto se logra dividiendo el programa en *módulos*. Cada módulo es una unidad lógica y hace un trabajo específico. Su tamaño se mantiene pequeño llamando a otros módulos. La modularidad tiene varias ventajas. Primera, es más fácil depurar rutinas pequeñas que grandes. Segunda, es más fácil que varias personas trabajen simultáneamente en un programa modular. Tercera, un programa modular bien escrito pone ciertas dependencias en sólo una rutina, haciendo más fáciles los cambios. Por ejemplo, si se necesita escribir la salida en algún formato, ciertamente es importante tener una rutina que lo haga. Si los enunciados de impresión están dispersos por todo el programa, se tardará un tiempo considerable en hacer modificaciones. La idea de que las variables globales y sus efectos

laterales son malos se atribuye directamente a la idea de que la modularidad es buena.

Un tipo de datos abstracto (TDA) es un conjunto de operaciones. Los tipos de datos abstractos son abstracciones matemáticas; en ninguna parte de la definición de un TDA se hace mención alguna a cómo se implanta el conjunto de operaciones. Esto se puede ver como una extensión del diseño modular.

Los objetos tales como listas, conjuntos y grafos, así como sus operaciones, se pueden considerar como tipos de datos abstractos, al igual que los enteros, los reales y los booleanos son tipos de datos. Los enteros, los reales y los booleanos tienen operaciones asociadas, igual que los tipos de datos abstractos. Para el TDA conjunto, podemos tener operaciones como la *unión*, la *intersección*, el *tamaño* y el *complemento*. Alternativamente, quizás sólo deseamos las dos operaciones de *unión* y *buscar*, las cuales podrían definir un TDA diferente sobre conjuntos.

La idea básica es que la implantación de esas operaciones se escribe sólo una vez en el programa, y cualquier otra parte del programa que necesite realizar una operación sobre el TDA puede hacerlo llamando a la función apropiada. Si por alguna razón hay que cambiar detalles de la implantación, debe ser fácil hacerlo cambiando sólo las rutinas que realizan las operaciones del TDA. Este cambio, en un mundo perfecto, podría ser completamente transparente para el resto del programa.

No hay ninguna regla que indique qué operaciones debe manejar cada TDA; ésta es una decisión de diseño. El manejo de errores y de rupturas (cuando se requiera) son aspectos que conciernen también al diseñador del programa. Las tres estructuras de datos que se estudiarán en este capítulo son ejemplos primarios de los TDA. Se verá cómo cada una puede ser implantada en varias formas, pero si se hace con corrección, los programas que las usen no necesitarán saber qué implantación se usó.

3.2. El TDA lista

Estudiaremos ahora una lista general que tiene la forma siguiente: $a_1, a_2, a_3, \dots, a_n$. Decimos que el tamaño de la lista es n . A la lista especial de tamaño cero se le conoce como lista nula (o vacía).

Para cualquier lista, excepto para la nula, se dice que a_{i+1} es sucesor de (\circ sigue a) a_i ($i < n$) y que a_{i-1} es predecesor de (\circ precede a) a_i ($i > 1$). El primer elemento de la lista es a_1 y el último, a_n . No se define ni el predecesor de a_1 ni el sucesor de a_n . La posición del elemento a_i en una lista es i . En este análisis se supondrá, con fines de simplificación, que los elementos de la lista son enteros, pero en general se permiten elementos arbitrariamente complejos.

Asociado con estas "definiciones" hay un conjunto de operaciones que quizás nos gustaría realizar sobre el TDA lista. Algunas operaciones populares son *visualizar_lista* y *anular*, que realizan acciones obvias; *buscar*, que devuelve la posición de la primera ocurrencia de una llave; *insertar* y *eliminar*, que insertan y eliminan alguna llave de alguna posición de la lista; y *buscar_k_ésimo*, que devuelve el elemento que está en alguna posición (especificada como argumento). Si la lista es 34, 12, 52, 16,

12, *buscar*(52) puede devolver 3; *insertar*(x, 3) puede transformar la lista en 34, 12, 52, x, 16, 12 (si se inserta después de la posición dada); y *eliminar*(3) puede dejar la lista como 34, 12, x, 16, 12.

Por supuesto, la interpretación de qué es adecuado para una función es decisión completa del programador, igual que el manejo de los casos especiales (por ejemplo, ¿qué debe devolver *buscar*(1) en el ejemplo anterior?). También podríamos agregar operaciones como *siguiente* y *anterior*, las cuales deben tomar una posición como argumento y devolver la posición del sucesor y predecesor, respectivamente.

3.2.1. Implantación de listas a base de arreglos sencillos

Desde luego, todas esas instrucciones se pueden implantar simplemente con el uso de un arreglo. Por supuesto, en algunos lenguajes tiene que haberse declarado el tamaño del arreglo en tiempo de compilación, y en lenguajes que permiten que un arreglo sea asignado "al vuelo", el tamaño necesita conocerse en ese momento. Así, se requiere algún cálculo del tamaño máximo de la lista. Por lo regular esto exige una sobrevaloración, la cual consume espacio considerable. Y esto podría ser una limitación seria, en especial si hay varias listas de tamaño desconocido.

Una implantación a base de arreglos permite llevar a cabo *visualizar_lista* y *buscar* en un tiempo lineal, que es lo mejor que se puede esperar, y la operación *buscar_k_ésimo* lleva un tiempo constante. No obstante, la inserción y eliminación son costosas. Por ejemplo, la inserción en la posición 0 (lo que equivale a crear un nuevo primer elemento) requiere empujar primero todo el arreglo una posición para hacer espacio, mientras que eliminar el primer elemento requiere desplazar una posición hacia adelante todos los elementos de la lista, así que el peor caso de estas operaciones es $O(n)$. En promedio, la mitad de la lista necesita moverse para cualquier operación, así que sigue requiriéndose tiempo lineal. La mera construcción de una lista con n inserciones sucesivas podría exigir un tiempo cuadrático.

Debido a que el tiempo de ejecución de las inserciones y eliminaciones es tan alto y que el tamaño de la lista debe conocerse por anticipado, no suelen usarse arreglos sencillos para implantar listas.

3.2.2. Listas enlazadas

A fin de evitar el costo lineal de la inserción y la eliminación, necesitamos asegurar que la lista no se almacene contiguamente, ya que de otra forma necesitarán moverse partes completas de la lista. La figura 3.1 muestra la idea general de una lista enlazada.

La lista enlazada consta de una serie de registros que no necesariamente son adyacentes en memoria. Cada registro contiene el elemento y un apuntador a un registro que contiene su sucesor. A éste se le llama el apuntador *siguiente*. El apuntador siguiente de la última celda apunta a *nil*; este valor está definido en Pascal y no puede confundirse con algún otro apuntador. Un valor característico usado para *nil* es 0.

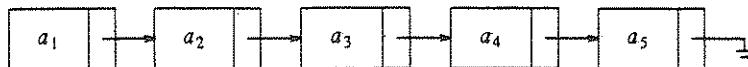


Figura 3.1. Lista enlazada

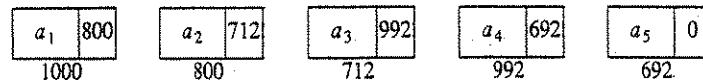


Figura 3.2. Lista enlazada con valores reales de apuntadores.

Recordemos que una variable apuntador sólo es una variable que contiene la dirección donde otro dato está almacenado. Así, si p se declara como apuntador a un registro, el valor almacenado en p se interpretará como el lugar, en memoria principal, donde se puede encontrar un registro. Para tener acceso a un campo del registro se emplea $p^.nombre_campo$, donde *nombre_campo* es el nombre del campo que se desea examinar. La figura 3.2 muestra la representación real de la lista de la figura 3.1. La lista contiene cinco registros, los cuales están en las posiciones de memoria 1000, 800, 712, 992 y 692, respectivamente. El apuntador *siguiente* del primer registro tiene el valor 800, y da la indicación de dónde se encuentra el segundo registro. Los otros registros tienen un apuntador, cada uno, que tiene un propósito similar. Por supuesto, a fin de tener acceso a la lista se necesita conocer dónde se puede encontrar la primera celda de la lista. Una variable apuntador puede servir para este propósito. Es importante recordar que un apuntador sólo es un número. Para el resto de este capítulo dibujaremos los apuntadores con flechas, porque son más ilustrativas.

Para ejecutar *visualizar_lista(L)* o *buscar(L, llave)*, simplemente se pasa el apuntador al primer elemento de la lista y después se recorre la lista siguiendo los apuntadores *siguiente*. Desde luego, esta operación es lineal en tiempo, aunque probablemente la constante sea mayor que en la implantación con arreglos. La operación *buscar_k_ésimo* ya no es tan eficiente como la implantación con arreglos; *buscar_k_ésimo(L, i)* tarda un tiempo $O(i)$ y funciona recorriendo la lista en la forma obvia. En la práctica, esta cota es pesimista, ya que con frecuencia las llamadas a *buscar_k_ésimo* están clasificadas en orden (por i). Por ejemplo, *buscar_k_ésimo(L, 2)*, *buscar_k_ésimo(L, 3)*, *buscar_k_ésimo(L, 4)*, *buscar_k_ésimo(L, 6)* se pueden ejecutar en sólo un recorrido por la lista.

El mandato *eliminar* se puede ejecutar en un cambio de apuntadores. La figura 3.3 muestra el resultado de eliminar el tercer elemento de la lista original.

El mandato *insertar* requiere obtener una celda nueva del sistema mediante una llamada a la función *new* (adelante veremos más sobre esto) para después realizar dos maniobras con apuntadores. La idea general se muestra en la figura 3.4. La línea punteada representa el apuntador antiguo.

3.2.3. Detalles de programación

La descripción anterior es realmente suficiente para hacer que todo funcione, pero hay varios puntos en donde es probable cometer errores. Ante todo, no existe

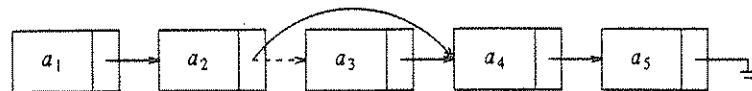


Figura 3.3 Eliminación en una lista enlazada

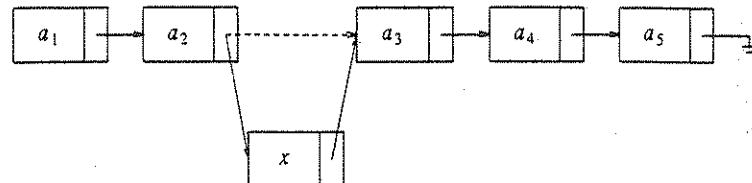


Figura 3.4. Inserción en una lista enlazada

ninguna forma realmente obvia para insertar al frente de la lista partiendo de las definiciones dadas. Segundo, la eliminación en el comienzo de la lista es un caso especial porque cambia el inicio de la lista; una codificación descuidada podría perder la lista. Un tercer problema se refiere a la eliminación en general. Aunque los movimientos de apuntadores son sencillos, el algoritmo de eliminación requiere cuidar la celda que está *antes* de la que se desea eliminar.

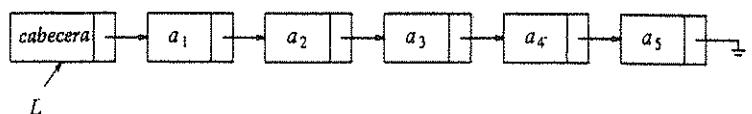
Pero ocurre que un sencillo cambio resuelve los tres problemas. Mantendremos un nodo centinela, al que a veces se le denomina *nodo cabecera* o *falso*. Ésta es una práctica común que veremos en repetidas ocasiones en el futuro. El convenio aquí será que la cabecera esté en la posición 0. La figura 3.5 muestra una lista enlazada con una cabecera que representa la lista a_1, a_2, \dots, a_5 .

Para evitar los problemas asociados con las eliminaciones, hay que escribir una rutina *buscar_previo*, la cual devuelve la posición del predecesor de la celda que se desea eliminar. Si usamos una cabecera, entonces, si se desea eliminar el primer elemento de la lista, *buscar_previo* devolverá la posición de la cabecera. El uso de un nodo cabecera es un tanto polémico. Hay quienes argumentan que evitar los casos especiales no es una justificación suficiente para agregar celdas ficticias; consideran que usar nodos cabecera es sólo un truco de programación. Aún así, las usaremos aquí precisamente porque nos permiten mostrar manipulaciones básicas de apuntadores sin oscurecer el código con casos especiales. De otra forma, si una cabecera se debe usar o no una cabecera es una cuestión de preferencia personal.

Como ejemplos, escribiremos cerca de la mitad de las rutinas del TDA lista. Primero necesitamos las declaraciones, que se encuentran en la figura 3.6.

La primera función que escribiremos verifica si una lista está vacía. Cuando escribimos código para cualquier estructura de datos en que intervienen apuntadores, siempre es mejor dibujar primero un esquema. La figura 3.7 muestra una lista vacía; y a partir de ella es fácil escribir la función de la figura 3.8.

Figura 3.5. Lista enlazada con cabecera



```

type
  ap_nodo = ^nodo;
  nodo = record
    elemento : tipo_elemento;
    siguiente : ap_nodo;
  end;
  LISTA = ap_nodo;
  posición = ap_nodo;

```

Figura 3.6 Declaraciones de tipos para listas enlazadas

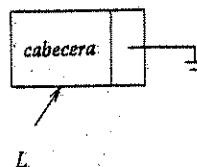


Figura 3.7 Lista vacía con cabecera

```

function está_vacía(L : LISTA) : boolean; {usando nodo cabecera}
begin
  está_vacía := (L^.siguiente = nil);
end;

```

Figura 3.8 Función para comprobar si la lista enlazada está vacía

La siguiente función (figura 3.9), comprueba si el elemento actual, que existe por hipótesis, es el último de la lista.

La siguiente rutina que se escribirá es *buscar*, mostrada en la figura 3.10, devuelve la posición de algún elemento en la lista. Las líneas [2] y [3] son necesarias y no pueden ser combinadas implícitamente porque no se puede hacer $p^.elemento \neq x$ a menos que p no sea *nil*. Por desgracia, esto complica la ruptura del ciclo. Por ello, algunos compiladores de Pascal implantan una operación *and* en cortocircuito: si la primera mitad de la *and* es falsa, entonces el resultado es falso automáticamente y la segunda mitad no se evalúa. Esto hace el código muy limpio, como se muestra en la figura 3.11. El problema, por supuesto, es que esto no es Pascal estándar, y este código sólo funciona si el compilador permite esta mejora. Debido a la importancia de la transportabilidad, no se debe suponer la forma en cortocircuito si se escribe código en Pascal.

En muchos lenguajes los operadores booleanos en cortocircuito están garantizados y deben usarse. En Pascal, podemos resolver el problema con un *goto*. En

```

function es_último(p: posición; L: LISTA); boolean; {se supone que p no es nil}
begin
  es_último := (p^.siguiente = nil);
end;

```

Figura 3.9 Función para comprobar si la posición actual es la última en la lista enlazada

{Devuelve la posición de x en L ; *nil* si no lo encuentra}

```

function buscar(x: tipo_elemento; L: LISTA); posición;
var p: posición;
label 999;
begin
  (1) p := L^.siguiente;
  (2) while p <> nil do
  (3)   if p^.elemento = x then
  (4)     goto 999
  (5)   else
  (6)     p := p^.siguiente;
999:
  buscar := p;
end;

```

Figura 3.10 Rutina *buscar* con *goto*

{Devuelve la posición de x en L ; *nil* si no lo encuentra}

```

function buscar(x: tipo_elemento; L: LISTA); posición;
var p: posición;
begin
  (1) p := L^.siguiente;
  (2) while (p <> nil) and (p^.elemento <> x) do
  (3)   p := p^.siguiente;
  (4) buscar := p;
end;

```

Figura 3.11 Rutina *buscar* con *and* en cortocircuito. Esta no es transportable.

general los *goto* son inaceptables porque hacen el código más ilegible y difícil de seguir. Éste no es el caso aquí, aunque para muchos programadores no se trata de una solución satisfactoria. La alternativa *while not encontrado* que vemos en la figura 3.12 es también un modismo común.

El uso de los *goto* sigue siendo más bien polémico, y muchos programadores tienen opiniones sorprendentemente fuertes al respecto. En este texto usaremos los *goto* sólo para romper ciclos que de otra forma requerirían varias líneas de código. Como el lector puede ver, es difícil defender un estilo contra otro. En realidad, esto es una cuestión de elección personal. No obstante, la diferencia en claridad entre las alternativas de Pascal mostradas en la figuras 3.10 y 3.12 y el código de la figura 3.11, que es representativo de otros lenguajes, indica una deficiencia importante de Pascal.

Algunos programadores encuentran tentador codificar recursivamente la rutina *buscar*, posiblemente porque evita la condición de terminación rebuscada. Véremos después qué ésta es una muy mala idea y se debe evitar a cualquier costo.

Nuestra cuarta rutina eliminará algún elemento *x* de *L*. Necesitamos decidir qué hacer si *x* aparece más de una vez o ninguna. La rutina elimina la primera aparición de *x* y no hace nada si *x* no está en la lista. Para hacer esto, encontramos *p*, que es la celda anterior a la que contiene *x*, vía una llamada a *buscar_previo*. El código para implantar esto se muestra en la figura 3.13. La rutina *buscar_previo* es similar a *buscar* y está en la figura 3.14.

La última rutina que escribiremos es una de inserción. Pasaremos un elemento a insertar además de la lista *L* y la posición *p*. Nuestra rutina de inserción particular insertará el elemento *después* de la posición indicada por *p*. Esta decisión es arbitraria y viene a mostrar que no hay reglas establecidas para lo que hace la inserción. Es posible insertar el elemento nuevo en la posición *p* (lo que significa antes del elemento que está actualmente en la posición *p*), pero hacerlo requiere conocer el elemento antes de *p*. Esto se obtiene mediante una llamada a *buscar_previo*.

Figura 3.12 Rutina *buscar* sin *goto*

(Devuelve la posición de *x* en *L*; *nil* si no lo encuentra)

```
function buscar(x: tipo_elemento; L: LISTA): posición;
  var p: posición;
  var encontrado: boolean;
begin
  [0]  encontrado := false;
  [1]  p := L^.siguiente;
  [2]  while (p <> nil) and (not encontrado) do
  [3]    if p^.elemento = x then
  [4]      encontrado := true
  [5]    else
  [6]      p := p^.siguiente;
  buscar := p;
end;
```

[Elimina de una lista. La celda apuntada por *p*^.siguiente es desechada]
[Supone que la posición es legal. Supone el uso de un nodo cabecera]

```
procedure eliminar(x: tipo_elemento; L: LISTA);
  var p, celda_temp : posición;
begin
  p := buscar_previo(x, L);
  if p^.siguiente nil then {suposición implícita del uso de la cabecera}
  begin
    {x encontrado: eliminarse}
    celda_temp := p^.siguiente;
    p^.siguiente := celda_temp^.siguiente; {se desenlaza la celda a eliminar}
    dispose(celda_temp); {libera el espacio}
  end; {if}
end;
```

Figura 3.13 Rutina de eliminación para listas enlazadas

[Usa una cabecera. Si el elemento no se encuentra, entonces el campo siguiente] {del valor devuelto es *nil*}

```
function buscar_previo(x: tipo_elemento; L: LISTA): posición;
  var p: posición;
  label 999;
begin
  [1]  p := L;
  [2]  while p^.siguiente <> nil do
  [3]    if p^.siguiente^.elemento = x then
  [4]      goto 999
  [5]    else
  [6]      p := p^.siguiente;
  999:
  [6]  buscar_previo := p;
end;
```

Figura 3.14 *Buscar_previo*: la rutina *buscar* para uso con *eliminar*

Así, es importante comentar lo que se está haciendo. Esto se ha hecho en la figura 3.15:

Nótese que pasamos la lista a las rutinas *insertar* y *es_último*, aun cuando nunca la usan. Lo hicimos porque otra implantación podría requerir esta información, y no hacerlo podría ir contra la idea del uso de los TDA.[†]

Con excepción de las rutinas *buscar* y *buscar_previo*, todas las operaciones codificadas tardan un tiempo *O(1)*. Esto es porque en todos los casos sólo se realiza

[†] Esto es legal, pero algunos compiladores generarán una advertencia.

```

{Inserta (después de la posición legal p).}
{Supone una implantación con cabecera}

procedure insertar(x : tipo_elemento; L : LISTA; p : posición);
var celda_temp : posición;

begin
(1) new(celda_temp);
(2) if celda_temp = nil then
    error_fatal("¡¡Espacio agotado!!!")
else
begin
(4) celda_temp^.elemento := x;
(5) celda_temp^.siguiente := p^.siguiente;
(6) p^.siguiente := celda_temp;
end;
end;

```

Figura 3.15 Rutina de inserción para listas enlazadas

un número fijo de instrucciones, independientemente de lo grande que sea la lista. Para las rutinas *buscar* y *buscar_previo*, el tiempo de ejecución es $O(n)$ en el peor caso, porque podría hacer falta recorrer la lista completa si el elemento no se encuentra o es el último de la lista. En promedio, el tiempo de ejecución es $O(n)$ ya que, en promedio, hay que recorrer la mitad de la lista.

Podríamos escribir rutinas adicionales para visualizar la lista y para realizar la función *siguiente*. Son casi directas. También se podría escribir una rutina para implantar *previo*. Lo dejamos como ejercicio.

3.2.4. Errores comunes

El error más común es que el programa falle con un mensaje de error del sistema, como "violación de acceso a memoria" o "violación en la segmentación". Este mensaje significa, normalmente, que una variable apuntador contenía una dirección falsa. Una razón común es la falta de valores iniciales en las variables. Por ejemplo, si se omite la línea [1] de la figura 3.16, entonces *p* queda indefinida y no es probable que apunte a una parte de memoria válida. Otro error común estaría en la línea [6] de la figura 3.15. Si *p* es *nil*, la dirección es ilegal. Esta función sabe que *p* no es *nil*, así que la rutina está bien. Por supuesto, se debe comentar esto, de modo que la rutina que llama a *insertar* se asegure de ello. *Siempre que se hace una indirección, se debe estar seguro de que el apuntador no es nil.* Algunos compiladores de Pascal lo comprueban implícitamente, pero esto no es parte de Pascal estándar. Cuando se transporta un programa de un compilador a otro, se puede encontrar que no funciona. Ésta es una de las razones comunes.

El segundo error se refiere a cuándo usar o no la instrucción *new* para crear una celda nueva. Hay que recordar que la declaración del apuntador no crea el registro

```

procedure eliminar_lista(var L: LISTA);
var p : posición;
begin
(1) p := L^.siguiente;           {suponiendo el uso de cabecera}
(2) L^.siguiente := nil;
(3) while p <> nil do
begin
(4) dispose(p);
(5) p := p^.siguiente;
end;
end;

```

Figura 3.16 Forma incorrecta de eliminar una lista

sino sólo da suficiente espacio para almacenar la dirección en donde puede estar algún registro. La única forma de crear un registro que no está declarado es usar la orden *new*. La orden *new(p)* que tiene el sistema crea, mágicamente, un registro nuevo del tipo al que *p* puede apuntar; entonces se hace que *p* apunte al registro nuevo. Por el contrario, si se quiere usar una variable apuntador para tener acceso a registros existentes, como para recorrer una lista, no hay ninguna necesidad de crear un registro nuevo; en ese caso la orden *new* es inadecuada.

Cuando las cosas dejan de necesitarse, podemos emplear una orden *dispose* para informar al sistema que puede reclamar el espacio. Una consecuencia de la orden *dispose(p)* es que la dirección que contiene *p* no cambia, pero los datos que residen en esa dirección ya no están definidos.

Si nunca se elimina de una lista enlazada, el número de llamadas a *new* debe ser igual al tamaño de la lista, más 1 si se usa cabecera. Si es menor, probablemente no se tenga un programa que funcione correctamente. Si es mayor, se estará desperdi-ciando espacio y probablemente tiempo. En ocasiones, si el programa ocupa mucho espacio, es posible que el sistema no sea capaz de satisfacer la petición de una celda nueva. Pascal estándar no tiene forma de manejar esto, y el programa fallará. En este código supusimos una extensión a la instrucción *new* que pone el apuntador a *nil* si por alguna razón *new* no puede dar espacio. Es posible usar esto en algún sistema particular (puede ser necesario dar valor inicial *nil* al apuntador antes de la llamada a *new*).

Después de una eliminación en una lista enlazada suele ser buena idea liberar la celda, sobre todo si hay muchas inserciones y eliminaciones entremezcladas, por lo que la memoria se podría volver un problema. Hay que mantener una variable temporal que apunte a la celda que se va a liberar, porque después de terminar los movimientos de apuntadores no se tendrá referencia a ella. Por ejemplo, el código de la figura 3.16 no es la forma correcta de eliminar una lista completa (aunque en algunos sistemas puede funcionar).

La figura 3.17 muestra la forma correcta de hacerlo. La liberación no es necesariamente rápida, así que quizás se quiera comprobar si la rutina de liberación está causando un rendimiento lento y comentarlo si tal es el caso. El autor ha escrito un programa (véanse los ejercicios) que incrementó su velocidad 25 veces cuando se

```

procedure eliminar_lista(var L: LISTA);
  var p, temp: posición;
begin
(1)  p := L^.siguiente;           {cabecera supuesta}
(2)  L^.siguiente := nil;
(3)  while p <> nil do
begin
(4)    temp := p^.siguiente;
(5)    dispose(p);
(6)    p := temp;
end;
end;

```

Figura 3.17 Forma correcta de eliminar una lista

omitió (convirtiendo en comentario) la liberación (de 10 000 nodos). Resulta que las celdas se liberaron en un orden más bien peculiar y al parecer causó que un programa normalmente lineal consumiera tiempo $O(n \log n)$ al liberar n celdas.

3.2.5. Listas doblemente enlazadas

A veces es conveniente recorrer las listas hacia atrás. La implantación estándar no ayuda en esto pero la solución es simple. Sólo se añade un campo adicional a la estructura de datos, que contiene un apuntador a la celda anterior. El costo de esto es un enlace más que se suma al espacio requerido y duplica el costo de las inserciones y eliminaciones porque hay que manejar más apuntadores. Por otro lado, se simplifican las eliminaciones porque ya no hay que referirse a una llave por medio del apuntador a la celda anterior; esta información ahora está a la mano. La figura 3.18 muestra una lista doblemente enlazada.

3.2.6. Listas enlazadas circularmente

Un convenio popular consiste en tener la última celda con un apuntador de regreso a la primera. Esto se puede hacer con cabecera o sin ella (si la cabecera está presente, la última celda apunta a ella), y también es posible con listas doblemente enlazadas (el apuntador anterior de la primera celda apunta a la última). Claro está, ello afecta algunas de las condiciones, pero la estructura es popular en algunas aplicaciones. La figura 3.19 muestra una lista enlazada circularmente sin cabecera.

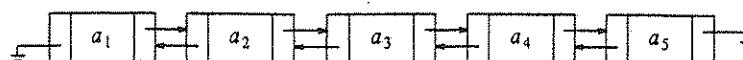


Figura 3.18 Lista doblemente enlazada

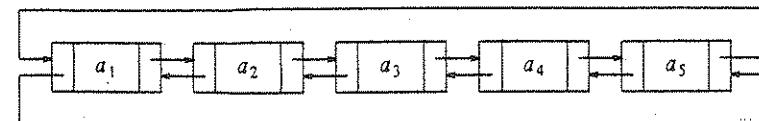


Figura 3.19 Lista enlazada circularmente

3.2.7. Ejemplos

Proporcionamos tres ejemplos con listas enlazadas. El primero es una forma sencilla de representar polinomios de una variable. El segundo es un método de ordenar en tiempo lineal, para algunos casos especiales. Por último, mostramos un ejemplo complicado de cómo las listas enlazadas pueden servir para llevar el registro de cursos en una universidad.

El polinomio TDA

Podemos definir un tipo de datos abstracto para polinomios de una variable (con exponentes no negativos) mediante una lista. Sea $f(x) = \sum_{i=0}^n a_i x^i$. Si la mayoría de los coeficientes a_i son diferentes de cero, se puede usar un simple arreglo para almacenar los coeficientes. Entonces podríamos escribir rutinas para efectuar adición, sustracción, multiplicación, diferenciación y otras operaciones con esos polinomios. En este caso sirven las declaraciones de tipos dadas en la figura 3.20. Con ello podríamos escribir rutinas para realizar varias operaciones. Dos posibilidades son la adición y la multiplicación, las cuales se muestran en las figuras 3.21 a 3.23.

```

type
  POLINOMIO = record
    arreglo_coef: array [0..GRADO_MÁX] of integer;
    potencia_mayor: integer;
  end;

```

Figura 3.20 Declaraciones de tipos para la implantación con arreglos del TDA polinomio

```

procedure polinomio_cero(var poli: POLINOMIO);
  var i: integer;
begin
  for i := 0 to GRADO_MÁX do
    poli.arreglo_coef[i] := 0;
  poli.potencia_mayor := 0;
end;

```

Figura 3.21 Procedimiento para iniciar un polinomio a cero

```

procedure suma_polinomios(var poli1, poli2, suma_poli: POLINOMIO);
var i: integer;

begin
    polinomio_cero(suma_poli);
    suma_poli.potencia_mayor := máx(poli1.potencia_mayor, poli2.potencia_mayor);

    for i := suma_poli.potencia_mayor downto 0 do
        suma_poli.arreglo_coeff[i] := poli1.arreglo_coeff[i] + poli2.arreglo_coeff[i]
end;

```

Figura 3.22 Procedimiento para sumar dos polinomios.

```

procedure mult_polinomios(var poli1, poli2, prod_poli : POLINOMIO);
var i, j: integer;

begin
    polinomio_cero(prod_poli);
    prod_poli.potencia_mayor := poli1.potencia_mayor + poli2.potencia_mayor;
    if prod_poli.potencia_mayor > GRADO_MÁX then
        error("Tamaño de arreglo excedido")
    else
        for i := 0 to poli1.potencia_mayor do
            for j := 0 to poli2.potencia_mayor do
                prod_poli.arreglo_coeff[i+j] := prod_poli.arreglo_coeff[i+j] +
                    poli1.arreglo_coeff[i]*poli2.arreglo_coeff[j];
    end;

```

Figura 3.23 Procedimiento para multiplicar dos polinomios

De ignorarse el tiempo para iniciar con cero los polinomios de salida, el tiempo de ejecución de la rutina de multiplicación es proporcional al producto del grado de los dos polinomios de entrada. Esto es adecuado para polinomios densos, donde están presentes la mayoría de los términos, pero si $p_1(x) = 10x^{1000} + 5x^{14} + 1$ y $p_2(x) = 3x^{1990} - 2x^{1492} + 11x + 5$, entonces es probable que el tiempo de ejecución sea inaceptable. Se puede ver que la mayoría del tiempo se consume en multiplicar ceros y pasar a través de lo que viene a ser partes no existentes en los polinomios de entrada. Esto siempre es indeseable.

Una alternativa es usar una lista de enlace simple. Cada término del polinomio está en una celda, y las celdas se clasifican en orden decreciente de sus exponentes. Por ejemplo, las listas enlazadas de la figura 3.24 representan $p_1(x)$ y $p_2(x)$. Entonces podríamos usar las declaraciones de la figura 3.25.

De hacerlo así sería directa la implantación de las operaciones. La única dificultad potencial es que cuando dos polinomios se multiplican, en el polinomio resul-

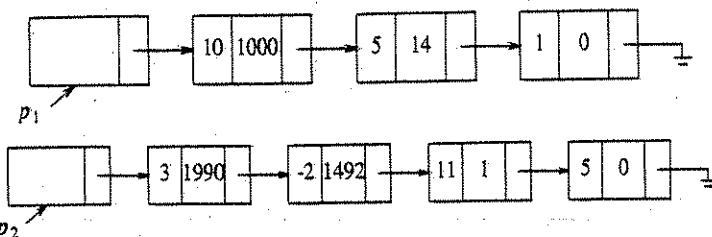


Figura 3.24 Representaciones de lista enlazada de dos polinomios

```

type
    ap_nodo = ^nodo;
    POLINOMIO = ^nodo; {mantiene los nodos ordenados por exponente}

    nodo = record
        coeficiente : integer;
        exponente : integer;
        siguiente : ap_nodo;
    end;

```

Figura 3.25 Declaración de tipos para la implantación con listas enlazadas del polinomio TDA

tante tendrán que combinarse los términos semejantes combinados. Hay varias formas de hacer esto, pero se dejará como ejercicio.

Ordenación por bases

Un segundo ejemplo donde se usan las listas enlazadas es la llamada *ordenación por bases*. A este método se le llama a veces *ordenación de tarjetas*, porque se usó, hasta que surgieron los computadores modernos, para ordenar las antiguas tarjetas perforadas.

Si tenemos n enteros en el intervalo desde 1 a m (o desde 0 hasta $m-1$), a partir de esta información podemos obtener una ordenación rápida conocida como *ordenación por cubetas*. Mantenemos un arreglo llamado *cuenta*, de tamaño m , que se inicia a cero. Así, *cuenta* tiene m celdas (o cubetas) que al inicio están vacías. Cuando se lee a_i , se incrementa en uno *cuenta*[a_i]. Despues de leída toda la entrada, se recorre el arreglo *cuenta*, para visualizar una representación de la lista ya ordenada. Este algoritmo tarda $O(m + n)$; la demostración se deja como ejercicio. Si $m = \Theta(n)$, entonces la ordenación por cubetas es $O(n)$.

La ordenación por bases es una generalización de lo anterior. La forma más fácil de ver qué ocurre es con un ejemplo. Supongamos que tenemos 10 números, en el intervalo 0 a 999, que se quiere ordenar. En general, hay n números en el intervalo de 0 a n^{p-1} para alguna constante p . Es obvio que no se puede usar la ordenación

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

Figura 3.26 Cubetas después del primer paso de la ordenación por bases

8		729							
1	216	27							
0	512	125	343	64					
0	1	2	3	4	5	6	7	8	9

Figura 3.27 Cubetas después del segundo paso de la ordenación por bases

por cubetas; habría demasiadas cubetas. El truco es usar varios pasos de la ordenación por cubetas. El algoritmo natural sería ordenar por cubetas por el "dígito" (en base n) más significativo, después el siguiente más significativo, etc. Este algoritmo no funciona, pero si las ordenaciones por cubetas se realizan primero con el "dígito" menos significativo, entonces sí funciona. Por supuesto, es posible que más de un número caiga en la misma cubeta, y, a diferencia de la ordenación por cubetas original, esos números podrían ser diferentes, así que los guardamos en una lista. Nótese que todos los números podrían tener algún dígito en común, de modo que si se usó un simple arreglo para las listas, cada arreglo tendría que ser de tamaño n , para un espacio total requerido de $\Theta(n^2)$.

El siguiente ejemplo muestra la acción de la ordenación por bases de 10 números. La entrada es 64, 8, 216, 512, 27, 729, 0, 1, 343, 125 (los primeros cubos en orden aleatorio). El primer paso ordena las cubetas con el dígito menos significativo. En este caso las operaciones se hacen en base 10 (por sencillez), pero esto no es una suposición general. Las cubetas se muestran en la figura 3.26, así que la lista, ordenada por el dígito menos significativo, es 0, 1, 512, 343, 64, 125, 216, 27, 8, 729. Después, éstos se ordenan según el siguiente dígito menos significativo, es decir, los dígitos de las decenas en este caso (véase la figura 3.27). El paso 2 da la salida 0, 1, 8, 512, 216, 125, 27, 729, 343, 64. Ahora la lista está ordenada con respecto a los dos dígitos menos significativos. El paso final (figura 3.28), ordena por cubetas el dígito más significativo. La lista final es 0, 1, 8, 27, 64, 125, 216, 343, 512, 729.

Para ver que el algoritmo funciona, observemos que el único fallo posible ocurriría si salen dos números de la misma cubeta en el orden incorrecto. Pero los pasos previos aseguran que cuando varios números entran a una cubeta, llegan en orden. El tiempo de ejecución es $O(p(n+b))$ donde p es el número de pasos, n es el número de elementos a ordenar y b es el número de cubetas. En este caso $b = n$.

64									
27									
8									
1									
0	125	216	343	512	729				
0	1	2	3	4	5	6	7	8	9

Figura 3.28 Cubetas después del último paso de la ordenación por bases

Por ejemplo, podríamos ordenar todos los enteros que son representables en un computador (de 32 bits) por medio de la ordenación por bases, si se hacen tres pasos sobre una cubeta de tamaño 2^{11} . Este algoritmo siempre sería $O(n)$ en este computador, pero probablemente no tan eficiente como alguno de los algoritmos que veremos en el capítulo 7, debido a la elevada constante que interviene (recordemos que un factor $\log n$ no es tan alto, y este algoritmo padecería además el costo de mantener listas enlazadas).

Listas múltiples

El último ejemplo muestra un uso más complicado de las listas enlazadas. Una universidad con 40 000 estudiantes y 2500 cursos necesita generar dos tipos de informes. El primero lista el número de matriculados en cada clase, y el segundo las clases en que está inscrito cada uno de los alumnos.

La implantación obvia puede ser usar un arreglo bidimensional. Dicho arreglo tendría 100 millones de entradas. Pero el estudiante medio sólo se inscribe en alrededor de tres cursos, así que sólo 120 000 de esas entradas, más o menos el 0.1%, tendría datos significativos.

Lo que se necesita es una lista por cada clase que contenga los estudiantes de la clase. También necesitarnos una lista por cada estudiante, que registre en qué clases está inscrito el estudiante. La figura 3.29 muestra nuestra implantación.

Como se muestra en la figura, tenemos dos listas combinadas en una. Todas las listas usan una cabecera y son circulares. Para listar todos los estudiantes de la clase C3, empezamos en C3 y recorremos su lista (yendo hacia la derecha). La primera celda pertenece al estudiante E1. Aunque no hay información explícita a este efecto, se puede determinar siguiendo la lista enlazada del estudiante hasta alcanzar la cabecera. Una vez hecho esto, regresamos a la lista de C3 (almacenamos la posición donde estábamos en la lista del curso antes de recorrer la lista del estudiante) y encontramos la siguiente celda, que se determina que pertenece a E3. Se puede continuar y encontrar que E4 y E5 también están en esta clase. En una forma similar se determinan, para cualquier estudiante, todas las clases en que está matriculado.

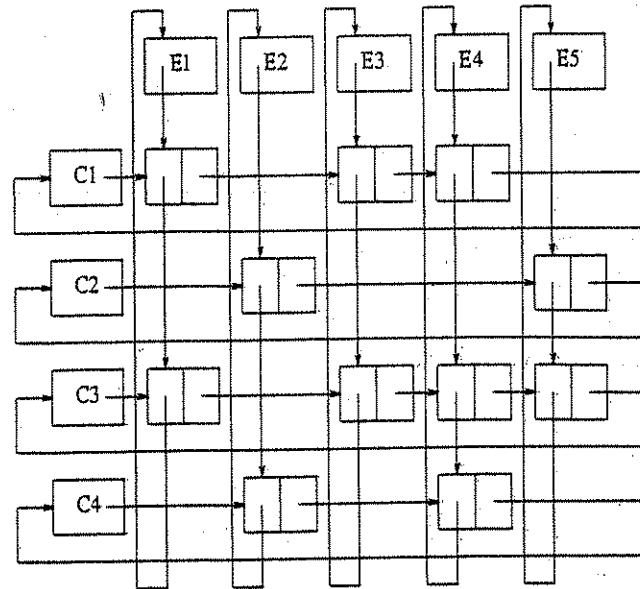


Figura 3.29: Implementación de listas múltiples para el problema de las inscripciones

El uso de una lista circular ahorra espacio, pero a expensas del tiempo. En el peor caso, si el primer estudiante se inscribió a todos los cursos, entonces se tendría que examinar cada entrada a fin de determinar todos los cursos para ese estudiante. Como en esta aplicación hay relativamente pocos cursos por estudiante y pocos estudiantes por curso, es probable que tal cosa no suceda. Si se sospechara que esto podría causar un problema, entonces cada una de las celdas (no cabecera) podría tener apuntadores de regreso directo a la cabecera de clase y estudiante. Esto duplicaría el requerimiento de espacio, pero lograría simplificar y acelerar la implantación.

3.2.8. Implementación de listas enlazadas a base de cursores

Muchos lenguajes, como BASIC o FORTRAN, no manejan los apuntadores. Si se requieren listas enlazadas y no se cuenta con apuntadores, hay que usar una implantación alternativa. El método alterno que describiremos se conoce como implantación con *cursos*.

Los dos aspectos importantes en una implantación con apuntadores de listas enlazadas son:

1. Los datos se almacenan en una colección de registros. Cada registro contiene los datos y un apuntador al siguiente registro.
2. Se puede obtener un registro nuevo de la memoria global del sistema por medio de una llamada a *new* y liberar con una llamada a *dispose*.

La implantación con cursores debe ser capaz de simular esto. La forma lógica de satisfacer la condición 1 es tener un arreglo global de registros. Para cualquier celda del arreglo, se puede usar su índice en lugar de una dirección. La figura 3.30 da las declaraciones de tipos para una implantación de listas enlazadas a base de cursores.

Ahora debemos simular la condición 2 permitiendo los equivalentes a *new* y *dispose* para celdas del arreglo *ESPACIO_CURSOR*. Para ello, se tiene una lista de las celdas (*lista_libre*) que no están en ninguna lista. La lista usará la celda 0 como cabecera. La configuración inicial se muestra en la figura 3.31.

Un valor de 0 para *siguiente* es el equivalente a un apuntador *nil*. La iniciación de *ESPACIO_CURSOR* es un ciclo directo, el cual se deja como ejercicio al lector. Para realizar un *new*, el primer elemento (después de la cabecera) se retira de la lista libre. Para efectuar un *dispose* se coloca la celda al frente de la lista libre. La figura 3.32 muestra la implantación con cursores de *new* y *dispose*. Hay que ver que si no hay espacio disponible, las rutinas hacen lo correcto poniendo *p* = 0. Esto indica que no quedan celdas libres, y también hace que la segunda línea de *new_cursor* sea una operación (no-op).

Dado lo anterior, la implantación de listas enlazadas a base de cursores es directa. Por consistencia, implantaremos las listas con un nodo cabecera. Por ejemplo, en la figura 3.33, si el valor de *L* es 5 y el valor de *M* es 3, entonces *L* representa la lista *a, b, e*, y *M* representa la lista *c, d, f*.

Para escribir las funciones de una implantación por cursores de listas enlazadas hay que pasar y devolver los mismos parámetros que la implantación con apuntadores. Las rutinas son directas. La figura 3.34 implanta una función para comprobar si una lista está vacía. La figura 3.35 implanta la comprobación de si la posición actual es la última de la lista enlazada.

Figura 3.30 Declaraciones para la implantación de listas enlazadas a base de cursores

```

type
  ap_nodo = integer;

nodo = record
  elemento : tipo_elemento;
  siguiente : ap_nodo;
end;

LISTA = ap_nodo;
posición = ap_nodo;

var ESPACIO_CURSOR : array [0..TAMAÑO_ESPACIO] of nodo;
  
```

Ranura	Elemento	Siguiente
0		1
1	1	2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

Figura 3.31 ESPACIO_CURSOR con valores iniciales

La función *buscar* devuelve la posición de x en L . La figura 3.36 es lógicamente equivalente a la implantación con apuntadores y , de hecho, es más simple, porque ahora 0 es una dirección legal. En la implantación con apuntadores, *nil* no era una dirección legal.

El código para implantar la eliminación se muestra en la figura 3.37. Una vez más la interfaz de la implantación con cursores es idéntica a la implantación con apuntadores. Por último, la figura 3.38 muestra una implantación de *inserta* a base de cursores.

El resto de las rutinas se codifican de manera semejante. El punto crucial es que esas rutinas siguen la especificación del TDA. Toman argumentos específicos y realizan operaciones específicas. La implantación es transparente para el usuario. La implantación con cursores puede usarse en vez de la implantación con listas enlazadas prácticamente sin ningún cambio en el resto del código.

Figura 3.32: Rutinas *new* y *dispose* para cursores

```

procedure new_cursor(var p: posición);
begin
  p := ESPACIO_CURSOR[0].siguiente;
  ESPACIO_CURSOR[0].siguiente := ESPACIO_CURSOR[p].siguiente;
end;
procedure dispose_cursor(p: posición);
begin
  ESPACIO_CURSOR[p].siguiente := ESPACIO_CURSOR[0].siguiente;
  ESPACIO_CURSOR[0].siguiente := p;
end;

```

Ranura	Elemento	Siguiente
0	—	6
1	b	9
2	f	0
3	cabecera	7
4	—	0
5	cabecera	10
6	—	4
7	c	8
8	d	2
9	e	0
10	a	1

Figura 3.33 Ejemplo de una implantación de listas enlazadas a base de cursores

```

function está_vacía(L: LISTA): boolean; {con uso del nodo cabecera}
begin
  está_vacía := (ESPACIO_CURSOR[L].siguiente = 0);
end;

```

Figura 3.34: Función para comprobar si una lista enlazada está vacía: Implantación con cursores

```

function es_último(p: posición; L: LISTA): boolean;
begin
  es_último := (ESPACIO_CURSOR[p].siguiente = 0);
end;

```

Figura 3.35 Función para comprobar si p es último en una lista enlazada: Implantación con cursores

La lista de espacio libre representa una estructura de datos interesante por derecho propio. La celda retirada de la lista de espacio libre es la que más recientemente ha sido colocada allí en virtud de un *dispose*. Así, la última celda colocada en

```

function buscar(x: tipo_elemento; L: LISTA): posición;
    var p: posición;

begin
    (1)   p := ESPACIO_CURSOR[L].siguiente;
    (2)   while (p <> 0) and (ESPACIO_CURSOR[p].elemento <> x) do
    (3)       p := ESPACIO_CURSOR[p].siguiente;
    (4)   buscar := p;
end;

```

Figura 3.36 Rutina *buscar*: Implantación con cursores

{Elimina de una lista:}

```

procedure eliminar(x: tipo_elemento; L: LISTA);
    var p, celda_temp : posición;

begin
    p := buscar_previo(x, L);

    if ESPACIO_CURSOR[p].siguiente <> 0 then
begin
    celda_temp := ESPACIO_CURSOR[p].siguiente;
    ESPACIO_CURSOR[p].siguiente :=
        ESPACIO_CURSOR[celda_temp].siguiente;
    dispose_cursor(celda_temp);  {libera el espacio}
end;
end;

```

Figura 3.37 Rutina de eliminación para listas enlazadas:
Implantación con cursores

la lista de espacio libre es la primera celda que se toma. La estructura de datos que también tiene esta propiedad se conoce como *pila*, y es el tema de la siguiente sección.

3.3. El TDA pila

3.3.1. El modelo *pila*

Una *pila* es una lista con la restricción de que las *insertiones* y las *eliminaciones* se pueden realizar sólo en una posición, a saber, al final de la lista, llamado *cima*. Las operaciones fundamentales en una pila son *meter* (en la pila), que es equivalente a una inserción, y *sacar* (de la pila), que elimina el elemento insertado más reciente-

{Inserta (después de la posición *p*). Implantación con cabecera}

```

procedure insertar(x: tipo_elemento; L: LISTA; p: posición);
    var celda_temp := posición;

begin
    (1)   new_cursor(celda_temp);
    (2)   if celda_temp = 0 then
            error_fatal("!!!Espacio agotado!!!")
        else
            begin
                (4)   ESPACIO_CURSOR[celda_temp].elemento := x;
                (5)   ESPACIO_CURSOR[celda_temp].siguiente :=
                    ESPACIO_CURSOR[p].siguiente;
                (6)   ESPACIO_CURSOR[p].siguiente := celda_temp;
            end;
end;

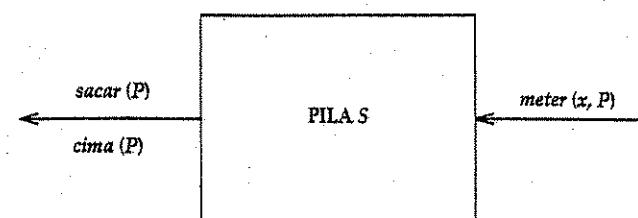
```

Figura 3.38 La rutina de inserción de listas enlazadas:
Implantación con cursores

mente. El elemento insertado más recientemente se puede examinar antes de realizar un *sacar* mediante la rutina *cima*. Un *sacar* o *cima* en una pila vacía suele considerarse como un error en el TDA pila. Por otro lado, quedarse sin espacio al realizar un *meter* es un error de implantación y no del TDA.

En ocasiones a las pilas se les denomina listas LIFO (*last in, first out*; el último en llegar es el primero en salir). El modelo bosquejado en la figura 3.39 significa sólo que las *meter* son operaciones de entrada y las *sacar* y *cima* son salidas. Las operaciones comunes para crear pilas vacías y comprobar que están vacías son parte del repertorio, pero en esencia todo lo que se puede hacer con una pila es *meter* y *sacar*.

La figura 3.40 muestra una pila abstracta después de varias operaciones. El modelo general es que hay algún elemento que está en la cima de la pila, y es el único visible.

Figura 3.39 Modelo pila: Se entra a una pila con
un meter, se sale con *sacar*

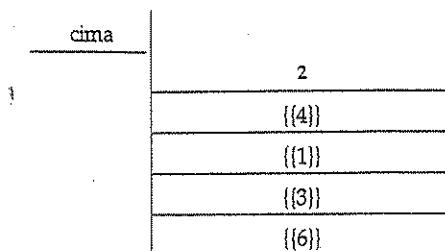


Figura 3.40 Modelo pila: Sólo el elemento de la cima es accesible

3.3.2. *Implantación de pilas*

Por supuesto, como una pila es una lista, cualquier implantación de listas funciona. Presentaremos dos implantaciones bien conocidas. Una se vale de apuntadores y la otra usa un arreglo, pero, como vimos en la sección previa, si empleamos buenos principios de programación las rutinas que las usan no necesitan conocer cuál es el método que se está usando.

Implantación de pilas a base de listas enlazadas

La primera implantación de una pila usa una lista de enlace simple. Se realiza un *meter* insertando al frente de la lista. Se efectúa un *sacar* eliminando el elemento que está al frente de la lista. Una operación *cima* sólo examina el elemento del frente de la lista, devolviendo su valor. A veces las operaciones *sacar* y *cima* se combinan en una. Podríamos usar llamadas a las rutinas sobre listas enlazadas de la sección anterior, pero las reescribiremos para evitar la pérdida de claridad.

Primero, en la figura 3.41 se dan las definiciones. Implantamos la pila con cabecera. Después, como en la figura 3.42, se examina una pila vacía en la misma manera que una lista vacía.

La creación de una pila vacía también es sencilla. Basta con crear una cabecera con un apuntador *siguiente* a *nil* (véase la figura 3.43). El *meter* se implanta como inserción al frente de la lista enlazada, donde el frente de la lista sirve como cima de la pila (véase la figura 3.44). La *cima* se efectúa examinando el elemento de la primera posición de la lista (véase la figura 3.45). Por último, implantaremos *sacar* eliminando del frente de la lista (véase la figura 3.46).

Debe quedar claro que todas las operaciones tarden un tiempo constante, porque en ninguna de las rutinas hay la menor referencia al tamaño de la pila (excepto en la comprobación de si está vacía); mucho menos se encuentra un ciclo que dependa del tamaño. La desventaja de esta implantación es que las llamadas a *new* y *dispose* son costosas, en especial en comparación con las rutinas de manipulación de apuntadores. Algo de esto se puede evitar mediante una segunda pila, que al principio está vacía. Cuando se libera una celda de la primera pila, sólo se

```

type
  ap_nodo = ^nodo;
  nodo = record
    elemento: tipo_elemento;
    siguiente: ap_nodo;
  end;

  PILA = ^nodo;
  {la implantación con pilas usará cabecera}
  
```

Figura 3.41 Declaración de tipos para la implantación del TDA pila con listas enlazadas

```

function está_vacía(P: PILA): boolean;
begin
  está_vacía := (P^.siguiente = nil); {se usa cabecera}
end;
  
```

Figura 3.42 Rutina para comprobar si la pila está vacía:
Implantación con listas enlazadas

```

procedure crear_vacía(var P: PILA);
begin
  new(P);           {hace la cabecera}
  if P = nil then
    error_fatal('¡¡¡Espacio agotado!!!')
  else
    P^.siguiente := nil;
end;
  
```

Figura 3.43 Rutina para crear una pila vacía:
Implantación con listas enlazadas

```

procedure meter(x: tipo_elemento; P: PILA);
var celda_temp: ^nodo;

begin
  new(celda_temp);
  if celda_temp = nil then
    error_fatal('!!Espacio agotado!!!')
  else
    begin
      celda_temp^.elemento := x;
      celda_temp^.siguiente := P^.siguiente;
      P^.siguiente := celda_temp;
    end;
end;

```

Figura 3.44 Rutina para meter en una pila:
Implantación con listas enlazadas

```

function cima(P: PILA): tipo_elemento;
begin
  if está_vacia(P) then
    error('Pila vacía')
  else
    cima := P^.siguiente^.elemento;
end;

```

Figura 3.45 Rutina para devolver el elemento de la cima de la pila:
Implantación con listas enlazadas

```

procedure sacar(P: PILA);
var primera_celda: ^nodo;

begin
  if está_vacia(P) then
    error('Pila vacía')
  else
    begin
      primera_celda := P^.siguiente;
      P^.siguiente := P^.siguiente^.siguiente;
      dispose(primera_celda);
    end;
end;

```

Figura 3.46 Rutina para sacar de la pila: Implantación con listas enlazadas

coloca en la segunda pila. Entonces, cuando se necesitan nuevas celdas para la primera pila, primero se buscan en la segunda pila.

Implantación de pilas a base de arreglos

Una implantación alternativa evita los apuntadores y probablemente es la solución más utilizada. El único inconveniente potencial de esta estrategia es que se necesita declarar el tamaño de un arreglo con anticipación. Por lo general, esto no es problema porque en aplicaciones típicas, aun si se tienen muy pocas operaciones con pilas, el número real de elementos en la pila nunca crece demasiado. Casi siempre es fácil declarar el arreglo suficientemente grande sin desperdiciar demasiado espacio. Si esto no es posible, un camino seguro podría ser usar la implantación con listas enlazadas.

Si utilizamos una implantación con arreglos, la implantación es trivial. Asociada a cada pila se tiene la cima de la pila, *cdp*, la cual es 0 para una pila vacía (así es como se da valor inicial a una pila vacía). Para meter algún elemento *x* en la pila, se incrementa *cdp* y después se hace *PILA[cdp] := x*; donde *PILA* es el arreglo que representa la pila real. Para sacar, se devuelve el valor de *PILA[cdp]* y después se decremente *cdp*. Por supuesto, ya que puede haber varias pilas, el arreglo *PILA* y *cdp* son parte de un registro que representa una pila. Casi siempre es mala idea usar variables globales y nombres fijos para representar esta (o cualquiera) estructura de datos, porque en la mayoría de las situaciones reales habrá más de una pila. Cuando se escribe el código real hay que intentar seguir el modelo tan exactamente como sea posible, de modo que ninguna parte del código, excepto las rutinas sobre pilas, pueda tratar de tener acceso al arreglo o a la variable *cima-de-pila* implicados en cada pila. Esto es verdadero para *todas* las operaciones de TDA. Lenguajes modernos como Ada y C++, de hecho, pueden imponer esta regla.

Cabe señalar que estas operaciones se realizan no sólo en tiempo constante, sino en un tiempo muy corto. En algunas máquinas, *meter* y *sacar* (de enteros) se pueden escribir en una sola instrucción de máquina, y operan en un registro de direccionamiento con autoincremento y autodecreto. El hecho de que la mayoría de las máquinas modernas tengan operaciones sobre pilas como parte del conjunto de instrucciones refuerza la idea de que la pila es probablemente la estructura de datos más importante en informática, después del arreglo.

Un problema que afecta la eficiencia de la implantación de pilas es la comprobación de errores. Nuestra implantación con listas enlazadas comprueba cuidadosamente los errores. Como se describió antes, un *sacar* de una pila vacía o un *meter* en una pila llena excederá los límites del arreglo y causará una ruptura. Esto es obviamente indeseable, pero si se incluyen estas comprobaciones en la implantación de arreglos podrían tomar tanto tiempo como la manipulación de la pila. Por esta razón, se ha vuelto práctica común omitir la comprobación de errores en las rutinas de pilas, excepto cuando el manejo de errores es crucial (como en los sistemas operativos). Aunque en la mayoría de los casos uno pueda arreglárselas con este problema declarando que el tamaño de la pila sea suficiente para no excederlo y asegurándose de que las rutinas que usan *sacar* y *meter* nunca intenten *sacar* de una pila vacía, esto hace que el código apenas funcione, en el mejor de los casos, en

```

type
PILA = record
    cima_de_pila: integer;
    arreglo_pila: array [1..tamaño_pila] of tipo_elemento;
end;

```

Figura 3.47 Declaración de tipos para el TDA pila:
Implantación con arreglos

especial cuando los programas son grandes y son escritos por más de una persona o en más de una ocasión. Debido a que las operaciones sobre pilas tardan un tiempo constante breve, es raro que una parte significativa del tiempo de ejecución de un programa se consuma en estas rutinas. Esto significa que en general no es justificable omitir comprobaciones de errores. Siempre hay que escribir las comprobaciones de error; si son redundantes, siempre podemos ponerlas como comentarios si son realmente costosas en tiempo. Dicho todo lo anterior, ya podemos escribir rutinas para implantar una pila general usando arreglos.

Para una implantación con arreglos, definiremos una pila como en la figura 3.47. Se ha supuesto que todas las pilas tendrán el mismo tamaño máximo. Podríamos haber hecho esta parte de la estructura de la pila, pero en Pascal no hay manera conveniente de manejar arreglos de longitud variable. Los lenguajes que tienen tal mecanismo pueden estar más cercanos al espíritu del TDA.

También hemos supuesto que todas las pilas tratan con el mismo tipo de elemento. En muchos lenguajes, si hay diferentes tipos de pilas, se necesita reescribir una nueva versión de las rutinas de pilas para cada tipo diferente, dando a cada versión un nombre distinto. Ada ofrece una alternativa más clara, que permite escribir un conjunto de rutinas genéricas para pilas y pasar el tipo como un argumento.[†] Ada también permite que pilas de diferentes tipos mantengan los mismos nombres para los procedimientos y las funciones (como *meter* y *sacar*): el compilador decide qué rutinas están implicadas al revisar el tipo de rutina que hace la llamada.

Dicho lo anterior, ahora volvemos a escribir las cuatro rutinas para pilas. En un verdadero espíritu de TDA, se hará que la cabecera de los procedimientos y funciones se vea idéntica a la de la implantación con listas enlazadas. Las rutinas mismas son muy simples y siguen exactamente la descripción escrita (véanse las figuras 3.48 a 3.52).

Declaramos las pilas como *var* en todas las rutinas, aun cuando la pila no se modifique (en *está_vacia*), porque no hacerlo así en Pascal significaría que se haría una copia de la pila para asegurar que las rutinas no alteren su contenido. Esto puede ser costoso si el tamaño máximo de pila es grande. Un compilador inteligente podría darse cuenta de que la rutina *está_vacia* no intenta alterar la pila y no molestarse en hacer la copia, pero es mejor tener prevenciones al respecto. En ocasiones, *sacar* se escribe como una función que devuelve el elemento extraído (y altera la pila). Sin embargo, hoy se tiende a pensar que las funciones no deben

[†] Esto es de cierta forma una sobreimplementación.

```

function está_vacia(var P: PILA): boolean;
begin
    está_vacia := (P.cima_de_pila = 0);
end;

```

Figura 3.48 Rutina para comprobar si una pila está vacía:
Implantación con arreglos

```

procedure crear_vacia(var P: PILA);
begin
    P.cima_de_pila := 0;
end;

```

Figura 3.49 Rutina para crear una pila vacía:
Implantación con arreglos

```

procedure meter(x: tipo_elemento; var P: PILA);
begin
    if P.cima_de_pila = tamaño_pila then
        error('Pila llena')
    else
        begin
            P.cima_de_pila := P.cima_de_pila + 1;
            P.arreglo_pila[P.cima_de_pila] := x;
        end;
end;

```

Figura 3.50 Rutina para meter en una pila:
Implantación con arreglos

```

function cima(var P: PILA): tipo_elemento;
begin
    if está_vacia(P) then
        error('Pila vacía')
    else
        cima := P.arreglo_pila[P.cima_de_pila];
end;

```

Figura 3.51 Rutina para devolver el elemento de la cima de la pila:
Implantación con arreglos

```

procedure sacar(var P: PILA);
begin
  if está_vacia(P) then
    error('Pila vacía')
  else
    P.cima_de_pila := P.cima_de_pila - 1;
end;

```

Figura 3.52 Rutina para sacar de la pila:
Implantación con arreglos

```

procedure sacar(var P: PILA; var elemento_cima: tipo_elemento);
begin
  if está_vacia(P) then
    error('Pila vacía')
  else
    begin
      elemento_cima := P.arreglo_pila[P.cima_de_pila];
      P.cima_de_pila := P.cima_de_pila - 1;
    end;
end;

```

Figura 3.53 Rutina para devolver el elemento de la cima y sacarlo de la pila:
Implantación con arreglos

cambiar sus variables de entrada. Para cumplir con esta regla, algunas veces *sacar* se escribe como procedimiento, como en la figura 3.53.

3.3.3. Aplicaciones

No debe sorprendernos que si restringimos las operaciones permitidas sobre una lista, esas operaciones puedan realizarse con gran rapidez. La gran sorpresa, no obstante, es que el pequeño número de operaciones que queda sea tan potente e importante. Presentaremos tres de las muchas aplicaciones de las pilas. La tercera aplicación ofrece una profunda percepción de cómo se organizan los programas.

Equilibrio de símbolos

Los compiladores verifican los errores de sintaxis en los programas, pero con frecuencia la ausencia de un símbolo (como un *begin* o principio de comentario) provocará que el compilador dé cientos de líneas de diagnósticos sin identificar el error real.

Una herramienta útil en esta situación es un programa que verifique que todo esté equilibrado. Así, todo *end* debe corresponder a un *begin*, y corchetes, llaves y

paréntesis derechos deben corresponder a su contraparte izquierda. La secuencia `(())` es legal, pero `(()` es errónea. Obviamente, no vale la pena escribir un programa enorme para esto, pero es fácil comprobar estas cosas. Por simplicidad, sólo revisaremos el equilibrio de paréntesis, llaves y corchetes, ignorando cualquier otro carácter que aparezca.

El algoritmo simple usa una pila y es como sigue:

Hacer una pila vacía. Leer los caracteres hasta el fin del archivo. Si el carácter es de apertura, se mete en la pila. Si es de clausura y la pila está vacía se anuncia un error. Si no, se saca un elemento de la pila. Si el elemento sacado no es el correspondiente símbolo de apertura, se anuncia un error. Al final del archivo, si la pila no está vacía se anuncia un error.

El lector debe ser capaz de convencerse de que el algoritmo funciona. Desde luego es lineal, y de hecho sólo pasa una vez por la entrada. Por lo tanto, es en línea y muy rápido. Se puede hacer trabajo adicional para intentar decidir qué hacer cuando se anuncia un error, por ejemplo identificar la causa probable.

Expresiones posfijas

Supongamos que tenemos una calculadora de bolsillo y queremos obtener el costo de un recorrido de compras. Para hacerlo, sumamos una lista de números y multiplicamos el resultado por 1.06; esto da el precio de compra de algunos artículos más el impuesto local por ventas. Si los precios son 4.99, 5.99 y 6.99, la forma natural de introducirlos sería la secuencia

$$4.99 + 5.99 + 6.99 * 1.06 =$$

Dependiendo de la calculadora, esto produce ya sea la respuesta deseada 19.05, o la respuesta científica 18.39. Las calculadoras más simples de 4 funciones darán la primera respuesta, pero otras mejores saben que la multiplicación tiene mayor prioridad que la adición.

Por otro lado, algunos elementos son gravables y otros no, así que si sólo el primer elemento y el último fueran gravables, entonces la secuencia

$$4.99 * 1.06 + 5.99 + 6.99 * 1.06 =$$

daría la respuesta correcta (18.69) en una calculadora científica y la respuesta errónea (19.37) en una sencilla. Una calculadora científica por lo general tiene paréntesis, por lo que al aplicarlos siempre podemos obtener la respuesta correcta, pero con una calculadora sencilla hay que recordar los resultados parciales.

Una secuencia característica de evaluación de este ejemplo podría ser multiplicar 4.99 por 1.06, guardando esta respuesta como a_1 . Luego se suma 5.99 y a_1 , guardando el resultado en a_2 . Se multiplica 6.99 por 1.06, guardando el resultado en a_2 , y se termina sumando a_1 y a_2 , dejando el resultado en a_3 . Podemos escribir esta secuencia de operaciones como sigue:

$$4.99 \ 1.06 * 5.99 + 6.99 \ 1.06 * +$$

Esta notación se conoce como *posfija* o *polaca inversa*, y se evalúa exactamente como se describió antes. La forma más fácil de hacerlo es usar una pila. Cuando se ve un número, se mete en la pila; cuando aparece un operador, se aplica sobre los dos números (símbolos) que se sacan de la pila y el resultado se mete en la pila. Por ejemplo, la expresión posfija

$$6 \ 5 \ 2 \ 3 + 8 * + 3 + *$$

se evalúa como sigue: los primeros cuatro símbolos se colocan en la pila. La pila resultante es

cdp →	3 2 5 6
-------	------------------

Después se lee un '+', luego se saca el 2 y 3 de la pila y su suma, 5, se mete en la pila.

cdp →	5 5 6
-------	-------------

Después se mete un 8.

cdp →	8 5 5 6
-------	------------------

Ahora aparece un '*', así que se saca el 8 y 5 de la pila para meter $8 * 5 = 40$.

cdp →	40 5 6
-------	--------------

Después aparece un '+', así que se sacan de la pila 40 y 5 para meter $40 + 5 = 45$.

cdp →	45 6
-------	---------

Ahora se mete el 3.

cdp →	3 45 6
-------	--------------

A continuación '+' saca 3 y 45 de la pila y mete $45 + 3 = 48$.

cdp →	48 6
-------	---------

Por último, aparece un '*' y se sacan 6 y 48 y se mete el resultado $6 * 48 = 288$ en la pila.

cdp →	288
-------	-----

El tiempo para evaluar una expresión en posfija es $O(n)$, porque el procesamiento de cada elemento a la entrada consiste en operaciones sobre pilas, por lo que lleva un tiempo constante. El algoritmo es muy sencillo. Cabe indicar que cuando se da una expresión en notación posfija, no hay necesidad de saber ninguna regla de prioridad; ésta es una ventaja obvia.

Conversión infija a posfija

No sólo se puede usar una pila para evaluar expresiones en notación posfija, sino que una pila también puede servir para convertir una expresión en forma estándar

(es decir en notación *infija*) en posfija. Nos concentraremos en una pequeña versión del problema general, sólo con los operadores $+$, $*$, $()$, e insistiendo en las reglas de prioridad usuales. Además, supondremos que la expresión es legal. Supongamos que se quiere convertir la expresión en notación infija

$$a + b * c + (d * e + f) * g$$

a posfija. Una respuesta correcta es $a b c * + d e * f + g * +$.

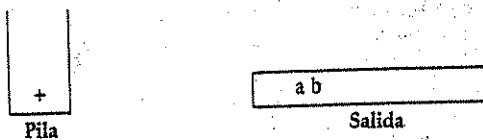
Cuando se lee un operando, inmediatamente se coloca en la salida. Los operadores no se sacan de inmediato, así que se deben almacenar en algún lugar. Lo correcto es colocar los operadores que ya se han leído, pero no colocado en la salida, en una pila. También se apilarán los paréntesis izquierdos cuando se encuentren. Se empieza con una pila vacía.

Si vemos un paréntesis derecho, sacamos elementos de la pila, y escribimos símbolos hasta que se encuentra el paréntesis izquierdo (correspondiente), el cual se saca de la pila pero no se envía a la salida.

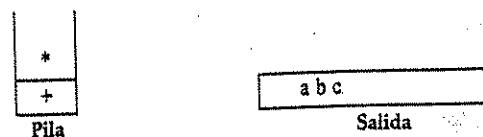
Si vemos cualquier otro símbolo ($+$, $*$, $()$), entonces sacamos los elementos de la pila hasta encontrar uno de menor prioridad. Una excepción es que nunca se saca un $()$ de la pila más que cuando se procesa el $'+'$. Para los propósitos de esta operación, $'+'$ tiene la menor prioridad y $()$ la más alta. Cuando se termina de sacar de la pila, se mete el operando en ella.

Por último, si se llega al fin de la entrada, se saca todo lo que está en la pila, enviando cada elemento a la salida.

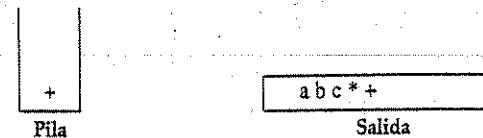
Para ver cómo funciona el algoritmo, convertiremos la expresión anterior a su forma posfija. Primero, se lee el símbolo a , pasándolo directamente a la salida. Entonces se lee $'+'$ y se mete en la pila. Después se lee b y se pasa a la salida. El estado de cosas en esta coyuntura va como sigue:



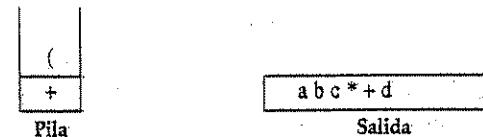
Después se lee $*$. La entrada de la cima de la pila de operadores tiene menor prioridad que $*$, así que nada se envía a la salida y el $*$ se mete en la pila. Después, se lee c y se envía a la salida. Hasta aquí tenemos



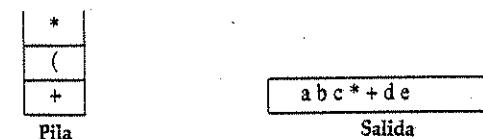
El siguiente símbolo es un $'+'$. Revisando la pila, se encuentra que se sacará el $*$ y se enviará a la salida, se sacará el otro $'+'$, que no es de menor sino de igual prioridad, y después se meterá el $'+'$.



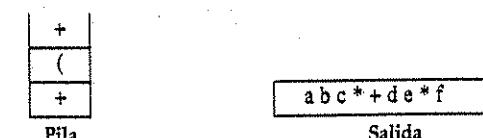
El siguiente símbolo leído es un $($, el cual, siendo de la más alta prioridad, se coloca en la pila. Después se lee d y se envía a la salida



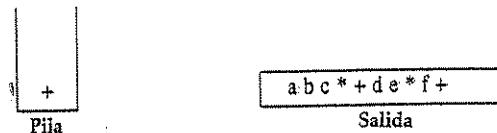
Después se lee un $*$. Como los paréntesis izquierdos no se sacan, excepto cuando se procesa un paréntesis derecho, no hay salida. Después, se lee e y se envía a la salida.



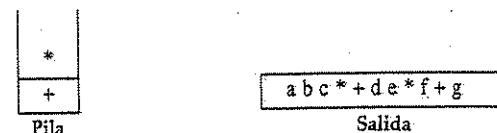
El siguiente símbolo en leerse es un $'+'$. Sacamos de la pila el $*$ y lo enviamos a la salida, y después el $'+'$. En seguida leemos f y lo enviamos a la salida.



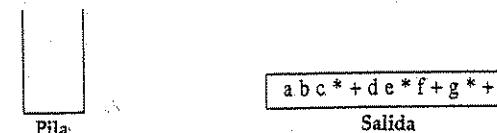
Ahora leemos un $)$, así que la pila se vacía hasta encontrar el $($. Se envía a la salida un $'+'$.



Se lee a continuación un '*', éste se mete a la pila. Después se lee g y se manda a la salida.



La entrada ahora está vacía, así que sacamos los símbolos que quedan en la pila y los ponemos en la salida hasta que la pila quede vacía.



Como antes, esta conversión requiere sólo un tiempo $O(n)$ y funciona en una pasada por la entrada. Podemos agregar la sustracción y la división a este repertorio, asignando igual prioridad a la sustracción y a la adición, e igual prioridad a la división y a la multiplicación. Un punto sutil es que la expresión $a - b - c$ se convertirá en $a b - c$ y no $a b c - -$. El algoritmo funciona correctamente porque estos operadores se asocian de izquierda a derecha. Esto no es necesariamente el caso general, ya que la exponentiación se asocia de derecha a izquierda: $2^2 = 2^3 = 256$, no $4^3 = 64$. Se deja como ejercicio el problema de agregar la exponentiación al repertorio de asignaciones.

Llamadas a procedimientos y funciones

El algoritmo para revisar el equilibrio de símbolos sugiere una forma de implantar llamadas a funciones y procedimientos. El problema aquí es que cuando se hace una llamada a un procedimiento nuevo, el sistema tiene que guardar todas las variables locales a la rutina que llama, ya que de otra forma el procedimiento nuevo sobreescribirá en ellas. Más aún, se debe guardar la posición actual en la rutina para que el procedimiento nuevo sepa a dónde volver cuando termine. En general, el compilador asigna las variables a registros de la máquina, y existen ciertos conflictos

(por lo regular todos los procedimientos tienen alguna variable asignada al registro #1), sobre todo si interviene la recursión. La razón de que este problema sea semejante al equilibrio de símbolos es que una llamada a un procedimiento y un retorno de procedimiento son en esencia lo mismo que un paréntesis que abre y uno que cierra, así que la misma idea debe funcionar.

Cuando hay una llamada a función, toda la información importante que se necesita guardar, como los valores de los registros del procesador (correspondientes a nombres de variables) y la dirección de regreso (que se puede obtener del contador del programa, el cual suele estar en un registro), se guarda en un "trozo de papel" en una forma abstracta y se pone en la cima de una pila. Entonces se transfiere el control a la función nueva, que es libre para sustituir los registros con sus propios valores. Si ésta hace otras llamadas a funciones, se sigue el mismo procedimiento. Cuando la función quiere regresar, observa en el "papel" de la cima de la pila y recupera todos los registros. Despues hace el salto de regreso.

Claro está, todo este trabajo se puede hacer usando una pila, y eso es exactamente lo que ocurre en prácticamente cualquier lenguaje de programación que implanta la recursión. La información guardada se conoce como *registro de activación* o *marco de la pila*. La pila de los computadores reales crece de la parte alta de la partición de la memoria hacia abajo, y en muchos sistemas no se comprueba el rebasamiento. Siempre existe la posibilidad de que se agote el espacio de la pila, teniendo muchas funciones activas al mismo tiempo. Es innecesario decir que el agotamiento del espacio de la pila es siempre un error fatal.

En lenguajes y sistemas que no revisan el rebasamiento, los programas fracasan sin explicación explícita. En tales sistemas pueden ocurrir cosas extrañas cuando la pila crece demasiado, porque ésta puede llegar a alguna parte del código. Puede ser el programa principal o parte de los datos, en especial si se tienen arreglos muy grandes. Si invade un programa, éste se estropeará; se tendrán instrucciones sin sentido que harán que el programa aborte tan pronto como se ejecuten. Si la pila invade los datos, lo que es probable que ocurra es que al escribir algo en ellos se destruya la información —quizás la dirección de regreso— de la pila y el programa intente regresar a alguna dirección errónea, abortando de inmediato.

En casos normales, no se debe agotar el espacio asignado a la pila; hacerlo es una indicación usual de recursión fuera de control (olvidando el caso base). Por otro

Figura 3.54 Uso erróneo de la recursión:
Visualización de una lista enlazada.

```
procedure visualizar_lista(L: LISTA); {sin uso de cabecera}
begin
  (1) if L <> nil then begin
  (2)   writeln(L^.elemento);
  (3)   visualizar_lista(L^.siguiente);
  end; {if}
end; {visualizar_lista}
```

lado, algún programa perfectamente legal y aparentemente inocuo puede desbordar el espacio de la pila. La rutina de la figura 3.54, que visualiza el contenido de una lista enlazada, es perfectamente legal y de hecho correcta. Da un manejo adecuado del caso base de la lista vacía, y la recursión está bien. Se puede demostrar que el programa es correcto. Por desgracia, si la lista contiene 20 000 elementos, habrá una pila de 20 000 registros de activación representando las llamadas anidadas de la línea [3]. Los registros de activación suelen ser grandes, por toda la información que contienen, así que es probable que este programa exceda el espacio de la pila. (Si 20 000 elementos no son suficientes para hacer que el programa aborte, reemplace el número con uno mayor.)

Este programa es un ejemplo de un uso extremadamente malo de la recursión, denominado *recursión por la cola*. La recursión por la cola se refiere a una llamada recursiva en la línea final. La recursión por la cola se puede eliminar mecánicamente, cambiando la llamada recursiva por un goto precedido de una asignación por cada argumento de la función. Esto simula la llamada recursiva porque no se necesita guardar nada: cuando termina la llamada recursiva, no hay ninguna necesidad real de conocer los valores guardados. Por ello se puede ir directamente al inicio de la función con los valores que se habrían usado en la llamada recursiva. El programa de la figura 3.55 muestra la versión mejorada. Tenga en mente que debe usar una construcción de ciclo *while* más natural. El goto sirve aquí para mostrar cómo un compilador podría eliminar automáticamente la recursión.

La eliminación de la recursión por la cola es tan simple que algunos compiladores lo hacen automáticamente. Aun así, lo mejor es no averiguar que el suyo no lo hace.

La recursión siempre se puede eliminar por completo (obviamente, el compilador lo hace al convertir a lenguaje ensamblador), pero hacerlo puede ser muy tedioso. La estrategia general requiere el uso de una pila y, desde luego, vale la pena sólo si se puede conseguir poner la información mínima en la pila. No abundaremos en esto, excepto para señalar que, aunque los programas no recursivos en general son más rápidos que los programas recursivos, la ventaja de la velocidad difícilmente justifica la pérdida de la claridad que resulta de eliminar la recursión.

Figura 3.55 Visualización de una lista sin recursión.
Un compilador podría hacer esto (el programador no debería)

```
procedure visualizar_lista(L: LISTA); {sin cabecera}
  label 0;
begin
  0:
    if L <> nil then begin
      writeln(L^.elemento);
      L := L^.siguiente;
      goto 0;
    end; {if}
end; {visualizar_lista}
```

3.4 El TDA cola

Al igual que las pilas, las *colas* son listas. Con una cola, no obstante, la inserción se hace por un extremo, mientras que la eliminación se realiza por el otro extremo.

3.4.1. El modelo cola

Las operaciones básicas sobre una cola son *encolar*, que inserta un elemento al final de la lista (llamado final), y *desencolar*, que elimina (y devuelve) el elemento del inicio de la lista (conocido como frente). La figura 3.56 muestra el modelo abstracto de una cola.

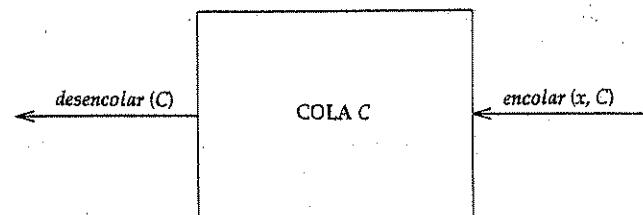


Figura 3.56 Modelo de una cola

3.4.2. Implementación de colas a base de arreglos

Como con las pilas, cualquier implementación de listas es legal para las colas. Como las pilas, tanto las implementaciones con listas enlazadas como con arreglos dan rápidos tiempos de ejecución $O(1)$ para toda operación. La implementación con listas enlazadas es directa y la dejamos como ejercicio. Ahora estudiaremos una implementación a base de arreglos.

Para cada estructura de datos cola, se tiene un arreglo, *COLA[]*, y las posiciones *c_frente* y *c_final*, las cuales representan los extremos de la cola. También se conserva el número de elementos que de hecho están en la cola, *c_tamaño*. Toda esta información es parte de una estructura, y como es usual, exceptuando las rutinas sobre colas, ninguna rutina debe tener acceso directo a ellas. La siguiente figura muestra una cola en un estado intermedio. A propósito, las celdas en blanco tienen valores indefinidos. En particular, las dos primeras celdas tienen elementos que fueron parte de la cola.

		5	2	7	1		

c_frente *c_final*

Las operaciones deben estar claras. Para *encolar* un elemento *x*, incrementamos *c_tamaño* y *c_final*, y luego hacemos *COLA[c_final] = x*. Para *desencolar* un elemento,

se pone en el valor a devolver $COLA[c_frente]$, se decrementa $c_tamaño$ y se incrementa c_frente . Hay otras estrategias posibles (esto se estudia después). De momento haremos más comentarios sobre la revisión de errores.

Hay un problema potencial con esta implantación. Después de 10 encoladas, la cola parece estar llena, ya que c_frente es 10 ahora, y el siguiente *encolar* sería en una posición inexistente. No obstante, puede haber sólo unos cuantos elementos en la cola, porque varios elementos pueden haber sido ya desencolados. Las colas, como las pilas, con frecuencia se mantienen pequeñas aun en presencia de una gran cantidad de operaciones.

La solución simple es que, siempre que c_frente y c_final lleguen al final del arreglo, regresen al inicio. La figura siguiente muestra la cola durante algunas operaciones. Esto se conoce como implantación de *arreglo circular*.

Estado inicial

					2	4
					^	^
					c_frente	c_final

Después de *encolar*(1)

1					2	4
					^	^
					c_final	c_frente

Después de *encolar*(3)

1	3				2	4
					^	^
					c_final	c_frente

Después de *desencolar*, que devuelve 2

1	3				2	4
					^	^
					c_final	c_frente

Después de *desencolar*, que devuelve 4

1	3				2	4
					^	^
					c_frente	c_final

Después de *desencolar*, que devuelve 1

1	3					2	4
						^	
						c_final	c_frente

Después de *desencolar*, que devuelve 3 y vacía la cola

1	3					2	4
						^	^
						c_final	c_frente

El código adicional requerido para implantar el movimiento circular es mínimo (aunque probablemente duplique el tiempo de ejecución). Si al incrementar c_final o c_frente se causa que salga del arreglo, el valor se reinicia a la primera posición del arreglo.

Hay dos advertencias acerca de la implantación de colas con arreglos circulares. Primera, es importante verificar si la cola está vacía porque un *desencolar* cuando la cola está vacía devolverá un valor indefinido sin indicarlo.

En segundo lugar, algunos programadores representan el frente y el final de la cola de diferentes formas. Por ejemplo, algunos no usan una variable para hacer un seguimiento del tamaño, porque confían en el caso base de que cuando la cola está vacía $c_final = c_frente - 1$. El tamaño se calcula implícitamente comparando c_final y c_frente . Ésta es una forma muy trampa de proceder, porque hay algunos casos especiales, que nos exigen ser muy cuidadosos si necesitamos modificar el código escrito de esta forma. Si el tamaño no es parte de la estructura, entonces si el tamaño del arreglo es $A_TAMAÑO$, la cola está llena cuando hay $A_TAMAÑO - 1$ elementos, ya que sólo se pueden distinguir $A_TAMAÑO$ diferentes tamaños, y uno de esos es 0. Seleccione cualquier estilo que le guste y asegúrese de que todas las rutinas sean consistentes. Debido a que hay unas cuantas opciones de implantación, es probable

Figura 3.57 Declaraciones de tipos para colas:
Implantación a base de arreglos

```
type
  COLA = record
    c_frente: integer;
    c_final: integer;
    c_tamaño: integer;
    c_arreglo: array [1..C_TAM_MÁX] of tipo_elemento;
  end;
```

```
function está_vacía(var C: COLA): boolean;
begin
    está_vacía := (C.c_tamaño = 0);
end;
```

Figura 3.58 Rutina para comprobar si una cola está vacía:
Implantación a base de arreglos

```
procedure crear_nula(var C: COLA);
begin
    C.c_tamaño := 0;
    C.c_frente := 1;
    C.c_final := C.TAM_MÁX;
end;
```

Figura 3.59 Rutina para crear una cola vacía:
Implantación a base de arreglos

```
procedure incrementar(var valor: integer);
begin
    if valor = C.TAM_MÁX then
        valor := 1;
    else
        valor := valor + 1;
end;

procedure encolar(x : tipo_elemento; var C: COLA);
begin
    if está_llena(C) then
        error('Cola llena')
    else
        begin
            C.c_tamaño := C.c_tamaño + 1;
            incrementar(C.c_final);
            C.c_arreglo[C.c_final] := x;
        end;
end;
```

Figura 3.60 Rutinas para encolar:
Implantación a base de arreglos

que valga la pena hacer uno o dos comentarios en el código, si usted no usa el campo para el tamaño.

Es obvio que no es necesario el movimiento circular en aplicaciones donde se está seguro de que el número de *encolar* no es mayor que el tamaño de la cola. Como con las pilas, *desencolar* se realiza muy pocas veces, a menos que las rutinas que llaman tengan la seguridad de que la cola no está vacía. Así, con frecuencia se saltan las llamadas a errores en esta operación, excepto en código crítico. En general esto no es justificable, porque es probable que el ahorro de tiempo esperado sea mínimo.

Termina esta sección escribiendo algunas rutinas de colas. Dejamos las demás como ejercicio al lector. Primero, damos la definición de tipos en la figura 3.57. También damos las rutinas para comprobar si una cola está vacía y para crear una cola vacía (figuras. 3.58 y 3.59). El lector puede escribir la función *está_llena*, la cual realiza la comprobación que su nombre indica. Nótese que *c_final* es preiniciada a 1 antes que *c_frente*. La última operación que se escribirá es la rutina *encolar*. Siguiendo la exacta descripción anterior, se llega a la implantación de la figura 3.60.

Cabe añadir que las colas se pasan como parámetros *var* en todas las rutinas (incluso *crear_nula* y *está_vacía*), a fin de evitar que se haga una copia. De nuevo recuerde que esto es una anomalía de Pascal.

3.4.3. Aplicaciones de colas

Hay varios algoritmos que se valen de colas para dar tiempos de ejecución eficientes. Varios de ellos se encuentran en la teoría de grafos, y los estudiaremos después en el capítulo 9. Por ahora, daremos algunos ejemplos sencillos sobre el uso de colas.

Cuando se envían trabajos a una impresora, se quedan en orden de llegada. Así, en esencia, los trabajos enviados a una impresora se ponen en una cola.[†]

Prácticamente toda fila real es (supuestamente) una cola. Por ejemplo, las colas en las taquillas son colas porque se atiende primero a quien llega primero.

Otro ejemplo se refiere a las redes de computadores. Hay muchas redes de computadores personales en las que el disco está conectado a una máquina, conocida como *servidor de archivos*. Los usuarios en otras máquinas obtienen acceso a los archivos sobre la base de que el primero en llegar es el primero atendido, así que la estructura de datos es una cola.

Entre otros ejemplos podemos mencionar:

- Por lo general, las llamadas telefónicas a compañías grandes se colocan en una cola, cuando todas las operadoras están ocupadas.
- En universidades grandes, cuando los recursos son limitados, los estudiantes deben firmar una lista de espera si todas las terminales están ocupadas. Aquel estudiante que haya estado trabajando más tiempo en una terminal es el primero que debe desocuparla, y el que haya estado esperando más tiempo será el que tenga acceso primero.

[†] Se dice en *esencia* una cola porque los trabajos pueden ser cancelados incluso en la mitad de la cola, lo cual es una violación a la definición estricta.

Una rama completa de las matemáticas, denominada *teoría de colas*, se ocupa de hacer cálculos probabilistas; de cuánto tiempo deben esperar los usuarios en una fila, cuán larga es la fila y otras cuestiones similares. La respuesta depende de la frecuencia con que llegan los usuarios a la fila y cuánto le lleva procesar a un usuario una vez que ha sido atendido. Ambos parámetros se dan como funciones de distribución de probabilidad. En casos sencillos, se puede calcular una respuesta analíticamente. Un ejemplo de un caso fácil sería una línea telefónica con un operador. Si el operador está ocupado, los usuarios se colocan en una fila de espera (hasta llegar a un límite máximo). Este problema es importante para los negocios porque hay estudios que han demostrado que la gente cuelga rápido el teléfono.

Si hay k operadores, entonces es mucho más difícil resolver este problema. Los problemas cuya solución analítica es difícil de obtener a menudo se resuelven con simulaciones. En este caso, podríamos necesitar una cola para efectuar la simulación. Si k es grande, también necesitaremos otras estructuras de datos para hacer esto con eficiencia. En el capítulo 6 se verá cómo hacer esta simulación. Entonces podríamos ejecutar la simulación con diferentes valores de k y escoger la k mínima que dé un tiempo razonable de espera.

Son abundantes otros usos de las colas, y como con las pilas, es sorprendente que sea tan importante una estructura de datos tan sencilla.

Resumen

Este capítulo describe el concepto de TDA que se ilustra con tres de los tipos de datos abstractos más comunes. El objetivo primario es distinguir la implantación de los tipos de datos abstractos de su función. El programa debe saber qué hacen las operaciones, pero es mejor no saber cómo lo hacen.

Las listas, pilas y colas son tal vez las tres estructuras de datos fundamentales en toda la informática, y su uso está documentado en una multitud de ejemplos. En particular, vimos cómo se usan las pilas para hacer el seguimiento de las llamadas a procedimientos y funciones, y cómo se implanta realmente la recursión. Es importante entender esto, no sólo porque hace posibles los lenguajes de procedimientos, sino porque al saber cómo se implanta la recursión se elimina una buena parte del misterio que rodea su uso. Aunque la recursión es muy potente, no se trata de una operación completamente libre; el uso inadecuado y el abuso de ella puede provocar que los programas aborten o no funcionen.

Ejercicios

- 3.1 Escriba un programa que visualice los elementos de una lista de un solo enlace.
- 3.2 Supongamos que una lista enlazada, L , y otra lista enlazada, P , contienen enteros, y están clasificadas en orden ascendente. La operación $\text{visualizar_lotes}(L, P)$ imprimirá los elementos de L que estén en posiciones dadas por P .

Por ejemplo, si $P = 1, 3, 4, 6$, se imprimen los elementos primero, tercero, cuarto y sexto de L . Escriba el procedimiento $\text{visualizar_lotes}(L, P)$, usando sólo las operaciones básicas sobre listas. ¿Cuál es el tiempo de ejecución del procedimiento?

- 3.3 Intercambie dos elementos adyacentes, ajustando sólo los apuntadores (y no los datos) usando
 - a. listas de un solo enlace,
 - b. listas doblemente enlazadas.
- 3.4 Dadas dos listas ordenadas, L_1 y L_2 , escriba un procedimiento para calcular $L_1 \cap L_2$ usando sólo las operaciones básicas sobre listas.
- 3.5 Dadas dos listas clasificadas, L_1 y L_2 , escriba un procedimiento para calcular $L_1 \cup L_2$ usando sólo las operaciones básicas sobre listas.
- 3.6 Escriba una función para sumar dos polinomios, sin destruir la entrada. Use una implantación con listas enlazadas. Si los polinomios tienen m y n términos, respectivamente, ¿cuál es la complejidad en tiempo de su programa?
- 3.7 Escriba una función para multiplicar dos polinomios, usando una implantación de listas enlazadas. Asegúrese de que el polinomio de salida esté ordenado según los exponentes y tenga a lo más un término de cada potencia.
 - a. Proponga un algoritmo para resolver este problema en tiempo $O(m^2n^2)$.
 - *b. Escriba un programa para efectuar la multiplicación en tiempo $O(m^2n)$, donde m es el número de términos del polinomio con menos términos.
 - *c. Escriba un programa para efectuar la multiplicación en tiempo $O(mn \log(mn))$.
 - d. ¿Cuál de las anteriores cotas de tiempo es la mejor?
- 3.8 Escriba un programa que tome un polinomio, $f(x)$, y calcule $(f(x))^p$. ¿Cuál es la complejidad del programa? Proponga al menos una solución alternativa que sea competitiva para algunas elecciones plausibles de $f(x)$ y p .
- 3.9 Escriba un paquete de aritmética entera con precisión arbitraria. Debe usar una estrategia semejante a la de la aritmética polinómica. Calcule la distribución de los dígitos 0 a 9 en 2^{1000} .
- 3.10 El *problema de Josefo* es el siguiente “juego” de suicidios en masa: n personas, numeradas de 1 a n , están sentadas en círculo. Empezando por la persona 1, se pasa un revólver. Después de m pasadas, la persona que tiene el arma se suicida, se quita el cuerpo, se cierra el círculo y el juego continúa con la persona que estaba sentada después del muerto. El último sobreviviente queda después de $n - 1$ disparos. Así, si $m = 0$ y $n = 5$, los jugadores son muertos en orden y el jugador 5 sobrevive. Si $m = 1$ y $n = 5$, el orden de las muertes es 2, 4, 1, 5.
 - a. Escriba un programa para resolver el problema de Josefo para valores generales de m y n . Intente hacer su programa tan eficiente como sea posible. Asegúrese de liberar las celdas.
 - b. ¿Cuál es el tiempo de ejecución del programa?

- c. Si $m = 1$, ¿cuál es el tiempo de ejecución de su programa? ¿Cómo es afectada la velocidad real con la rutina *dispose* para valores grandes de n ($n > 10\,000$)?
- 3.11 Escriba un programa para buscar un elemento particular en una lista de enlace sencillo. Hágalo recursivo y no recursivo, y compare los tiempos de ejecución. ¿Qué tan grande tiene que ser la lista antes de que aborde la versión recursiva?
- 3.12 a. Escriba un procedimiento no recursivo para invertir una lista de enlace sencillo en tiempo $O(n)$.
 b. Escriba un procedimiento para invertir una lista de enlace sencillo en un tiempo $O(n)$ usando espacio extra constante.
- 3.13 Se tiene que ordenar un arreglo de registros de estudiantes según su número de la seguridad social. Escriba un programa para ello, usando ordenación por bases con 1000 cubetas y tres pasos.
- 3.14 Escriba un programa para leer un grafo en listas de adyacencia usando
 a. listas enlazadas;
 b. cursorres.
- 3.15 a. Escriba una implantación de arreglos de listas autoajustables. Una lista *autoajustable* es como una lista normal, excepto en que todas las inserciones se efectúan en el frente, y cuando se tiene acceso a un elemento con un *buscar* se mueve al frente de la lista sin cambiar el orden relativo de los demás elementos.
 b. Escriba una implantación con listas enlazadas de listas autoajustables.
 *c. Supongamos que se tiene una probabilidad fija, p_i , de tener acceso a cada elemento. Demuestre que se espera que los elementos con mayor probabilidad de acceso estén cerca del frente.
- 3.16 Supongamos que se tiene una lista basada en arreglos $a[1..n]$ y queremos eliminar todos los duplicados. Última posición inicialmente es n , pero disminuye conforme se eliminan elementos. Considere el fragmento de pseudocódigo de la figura 3.61. El procedimiento *ELIMINAR* elimina el elemento de la posición j y se contrae la lista.

Figura 3.61 Rutina para eliminar los elementos duplicados de una lista:
 Implementación a base de arreglos

```
(1) for i := 1 to última_posición do begin
(2)   j := i + 1;
(3)   while j < última_posición do begin
(4)     if a[i] = a[j] then
(5)       ELIMINAR(j)
(6)     else
(7)       j := j + 1;
(8)   end; {while}
(9) end; {for}
```

- a. Explique cómo funciona el procedimiento.
 b. Reescriba este procedimiento usando operaciones generales sobre listas.
 *c. Usando la implantación estándar con arreglos, ¿cuál es el tiempo de ejecución de este procedimiento?
 d. ¿Cuál es el tiempo de ejecución usando una implantación con listas enlazadas?
 *e. Proporcione un algoritmo para resolver este problema en tiempo $O(n \log n)$.
 **f. Demuestre que cualquier algoritmo que resuelva este problema requiere $\Omega(n \log n)$ comparaciones si y sólo si se usan comparaciones. Sugerencia: Véase el capítulo 7.
 *g. Demuestre que si se permiten operaciones además de comparaciones y las llaves son números reales, podemos resolver el problema sin usar comparaciones entre elementos
- 3.17 Una alternativa a la estrategia de eliminación que hemos dado es la *eliminación perezosa*. Para borrar un elemento, sólo se marca como eliminado (usando un campo adicional de un bit). El número de elementos eliminados y no eliminados se mantiene como parte de la estructura de datos. Si hay tantos elementos eliminados como no eliminados, se recorre la lista completa, realizando el algoritmo estándar de eliminación sobre todos los nodos marcados.
 a. Indique las ventajas y desventajas de la eliminación perezosa.
 b. Escriba rutinas para implantar las operaciones estándar sobre listas enlazadas, usando eliminación perezosa.
- 3.18 Escriba un programa para comprobar el equilibrio de símbolos en los siguientes lenguajes:
 a. Pascal (*begin/end*, *()*, *[]*, *{ }*).
 b. C (*/* */*, *()*, *[]*, *{ }*).
 *c. Explique cómo visualizar un mensaje de error que intente reflejar la causa probable.
- 3.19 Escriba un programa para evaluar una expresión posfija.
- 3.20 a. Escriba un programa para convertir una expresión infija, que incluye *'('*, *')'*, *+*, *-*, *** y */*, en posfija.
 b. Agregue el operador de exponentiación.
 c. Escriba un programa para convertir una expresión posfija en infija.
- 3.21 Escriba rutinas para implantar dos pilas usando sólo un arreglo. Las rutinas no deben declarar un desbordamiento a menos que se hayan usado todas las posiciones del arreglo.
- 3.22 *a. Proponga una estructura de datos que permita las operaciones *meter* y *sacar* y una tercera operación *buscar_mín*, que devuelva el elemento menor de la estructura de datos, todo en tiempo $O(1)$ para el peor caso.

*b. Demuestre que si se agrega la cuarta operación *elimina_mín*, la cual busca y elimina el menor elemento, entonces al menos una de las operaciones deberá tomar un tiempo $\Omega(\log n)$. (Esto requiere la lectura del capítulo 7.)

3.23 *Muestre cómo se implantan tres pilas en un arreglo.

3.24 Si la rutina recursiva de la sección 2.4 usada para obtener los números de Fibonacci se ejecuta con $n = 50$, ¿es probable que el espacio de la pila se agote? ¿Por qué sí o por qué no?

3.25 Escriba las rutinas para implantar colas usando

- listas enlazadas,
- arreglos

3.26 Una *doble cola* es una estructura de datos consistente en una lista de elementos, sobre la cual son posibles las siguientes operaciones:

meter(x; d): Inserta un elemento x en el extremo frontal de la doble cola d .

sacar(d): Elimina y devuelve el elemento que está al frente de d .

injectar(x; d): Inserta un elemento x en el extremo posterior de la doble cola d .

eyectar(d): Elimina y devuelve el elemento que está en la parte posterior de d .

Escriba rutinas para permitir que la doble cola tome un tiempo $O(1)$ por operación.

Árboles

Cuando las entradas son muy grandes, el tiempo de acceso lineal de las listas enlazadas se hace prohibitivo. En este capítulo estudiaremos una estructura de datos sencilla para la cual, en promedio, el tiempo de ejecución de la mayoría de las operaciones es $O(\log n)$. También esbozamos una modificación conceptualmente sencilla de esta estructura de datos que garantiza el límite de tiempo mencionado en el peor caso, y planteamos una segunda modificación que en esencia da un tiempo de ejecución por operación de $O(\log n)$ para una secuencia grande de instrucciones.

La estructura de datos de que hablamos es el *árbol binario de búsqueda*. En general, los *árboles* son abstracciones muy útiles en informática, por lo que se tratará su uso en otras aplicaciones más generales. En este capítulo:

- Se verá cómo usar los árboles para implantar el sistema de archivos de varios sistemas operativos populares.
- Se verá cómo con los árboles se evalúan expresiones aritméticas.
- Se demostrará cómo los árboles sirven para manejar operaciones de búsqueda en un tiempo medio $O(\log n)$, y cómo refinar esas ideas para obtener cotas $O(\log n)$ en el peor caso. También veremos cómo implantar esas operaciones cuando los datos están almacenados en disco.

4.1. Preliminares

Podemos definir un *árbol* de varios modos. Una forma natural de hacerlo es recursivamente. Un árbol es una colección o conjunto de nodos. La colección puede estar vacía, a lo que se denota a veces con Λ . Si no es así, un árbol consiste en un nodo distinguido r , conocido como *raíz*, y cero o más (sub)árboles A_1, A_2, \dots, A_k , cada uno de los cuales tiene su raíz conectada a r por medio de una *arista* (o *rama*) dirigida.

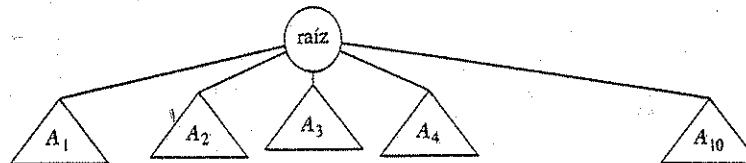


Figura 4.1 Árbol genérico

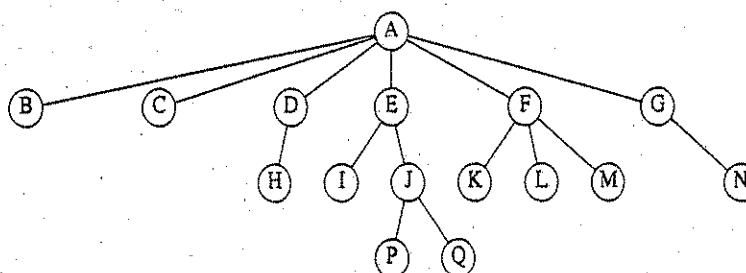


Figura 4.2 Árbol

Se dice que la raíz de cada subárbol es un *hijo* de r, y que r es el *padre* de cada raíz de los subárboles. La figura 4.1 muestra un árbol representativo según la definición recursiva.

Por la definición recursiva, encontramos que un árbol es un conjunto de n nodos, uno de los cuales es la raíz, y $n-1$ aristas. Que hay $n-1$ aristas se sigue del hecho de que cada arista conecta algún nodo con su padre, y todo nodo tiene un padre, excepto la raíz (véase la figura 4.2).

En el árbol de la figura 4.2, la raíz es A. El nodo F tiene a A como parente y a K, L y M como hijos. Cada nodo puede tener un número arbitrario de hijos, incluyendo cero. Los nodos sin hijos se conocen como *hojas*; las hojas en el árbol anterior son B, C, H, I, P, Q, K, L, M y N. Los nodos con el mismo parente son *hermanos*; así, K, L, y M son hermanos. Las relaciones *abuelo* y *nieta* se pueden definir de un modo semejante.

Un *camino* de un nodo n_1 a otro n_k se define como la secuencia de nodos n_1, n_2, \dots, n_k tal que n_i es el parente de n_{i+1} para $1 \leq i < k$. La *longitud* de este camino es el número de aristas que lo forman, $k - 1$. Existe un camino de longitud cero entre cada nodo y él mismo. Nótese que en un árbol hay exactamente un camino de la raíz a cada nodo.

Para cualquier nodo n_i , la *profundidad* de n_i es la longitud del camino único entre la raíz y n_i . Así, la raíz tiene una profundidad cero. La *altura* de n_i es el camino más largo de n_i a una hoja. Por lo tanto, todas las hojas son de altura cero. La altura de un árbol es igual a la altura de la raíz. Para el árbol de la figura 4.2, E está a una profundidad de 1 y a una altura de 2; F está a una profundidad de 1 y a una altura de 1; la altura del árbol es 3. La profundidad de un árbol es igual a la profundidad de la hoja más profunda; esto es siempre igual a la altura del árbol.

Si hay un camino de n_1 a n_2 , entonces n_1 es el *antecesor* de n_2 y n_2 es un *descendiente* de n_1 . Si $n_1 \neq n_2$, entonces n_1 es un *antecesor propio* de n_2 y n_2 es un *descendiente propio* de n_1 .

```
type
  ap_arbol = ^nodo_arbol;
  nodo_arbol = record
    elemento : tipo_elemento;
    primer_hijo : ap_arbol;
    sig Hermano : ap_arbol;
  end;
```

Figura 4.3 Declaraciones de nodos para árboles

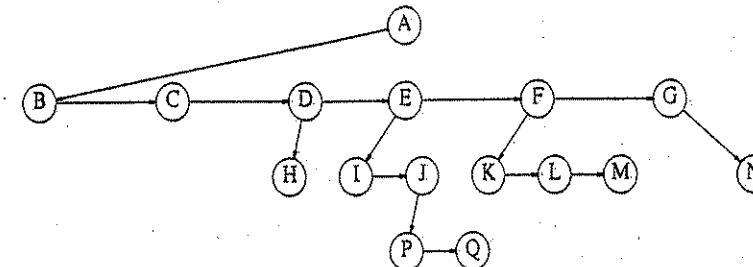


Figura 4.4 Representación primer hijo/siguiente hermano del árbol mostrado en la figura 4.2

4.1.1. *Implantación de árboles*

Una forma de implantar árboles podría ser poniendo en cada nodo, además de sus datos, un apuntador a cada hijo del nodo. No obstante, como el número de hijos por nodo puede variar bastante y no se conoce de antemano, parece imposible hacer directos los enlaces con los hijos en la estructura de datos, porque se malgastaría demasiado espacio. La solución es sencilla: guardar los hijos de cada nodo en una lista enlazada de nodos de árbol. La declaración de la figura 4.3 es representativa.

La figura 4.4 muestra cómo representar un árbol en esta implantación. Las flechas que apuntan hacia abajo son apuntadores *primer_hijo*. Las flechas que van de izquierda a derecha son apuntadores *sig_hermano*. Por ser demasiados, los apuntadores nulos no están dibujados.

En el árbol de la figura 4.4, el nodo E tiene tanto un apuntador a un hermano (F) como un apuntador a un hijo (I), mientras que algunos nodos no tienen ninguno.

4.1.2. *Recorridos de árboles con una aplicación*

Los árboles tienen muchas aplicaciones. Uno de los usos populares es la estructura de directorios en muchos sistemas operativos comunes, como UNIX, VAX/VMS y DOS. La figura 4.5 es un típico directorio del sistema de archivos de UNIX.

La raíz de este directorio es `/usr`. (El asterisco que sigue a los nombres indica que `/usr` es por sí mismo un directorio.) `/usr` tiene tres hijos, `marcos`, `alex` y `guillermo`, que son a su vez directorios. Así, `/usr` contiene tres directorios y no contiene archivos normales. El nombre de archivo `/usr/marcos/libro/cap1.r` se obtiene siguiendo tres veces los hijos del extremo izquierdo. Cada `/` después del primero indica una arista; el resultado es el *nombre del camino* completo. Este sistema de archivos jerárquico es muy popular porque permite que los usuarios organicen lógicamente sus datos. Más aún, dos archivos en directorios diferentes pueden compartir el mismo nombre porque deben tener caminos diferentes desde la raíz, y así tener nombre de camino diferente. Un directorio en el sistema de archivos de UNIX es sólo un archivo con una lista de todos sus hijos, así que los directorios son estructuras en concordancia casi total con la declaración de tipos anterior.[†] En efecto, si el mandato normal para imprimir un archivo se aplica a un directorio, entonces los

Figura 4.5 Directorio de Unix

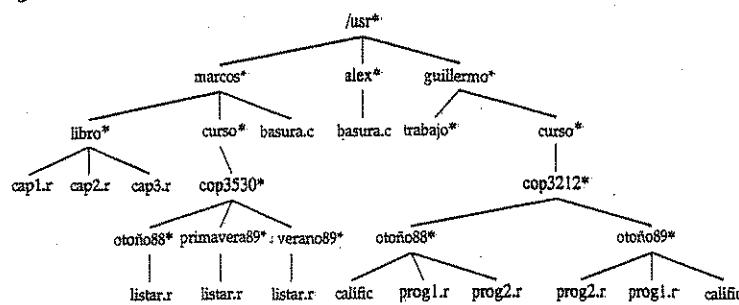


Figura 4.6 Rutina para listar un directorio en un sistema de archivos jerárquico

```

procedure listar_dir(D: directorio_o_archivo; profundidad: integer);
begin
  (1) if D es una entrada legítima then
      begin
        visualizar_nombre(profundidad, D);
        (2) if (D es un directorio) then
            for cada hijo, h, de D do
              (3) listar_dir(h, profundidad + 1);
        end;
      end;
  procedure listar_directorio(D: directorio_o_archivo);
  begin
    listar_dir(D, 0);
  end;

```

[†] Cada directorio en el sistema de archivos de UNIX tiene también una entrada que apunta a sí mismo y otra que apunta al padre del directorio. Así, técnicamente, el sistema de archivos de UNIX no es un árbol, pero lo aparenta.

nombres de los archivos del directorio pueden verse en la salida (junto con otra información no ASCII).

Supongamos que queremos listar los nombres de todos los archivos del directorio. Nuestro formato de salida mostrará los nombres de los archivos con profundidad *d* sangrados con *d* tabuladores. El algoritmo se da en la figura 4.6..

El corazón del algoritmo es el procedimiento recursivo *listar_dir*. Esta rutina necesita ser llamada inicialmente con el nombre del directorio y una profundidad de 0, para señalar que la raíz no se sangra. Esta profundidad es una variable de manipulación interna, y difícilmente es un parámetro que deba conocer la rutina que la llame. Así, la rutina controladora *listar_directorio* se usa como interfaz entre la rutina recursiva y el mundo exterior.

La lógica del algoritmo es simple de seguir. El argumento de *listar_dir* es algún tipo de apuntador dentro del árbol. Mientras el apuntador sea válido, el nombre implicado por el apuntador se visualiza con el número correcto de tabuladores. Si la entrada es un directorio, todos sus hijos se procesan recursivamente, uno por uno. Esos hijos son un nivel más profundo, por lo que necesitan estar sangrados un espacio adicional. La salida está en la figura 4.7.

Figura 4.7 Listado del directorio (en orden previo)

```

/usr
  marcos
    libro:
      cap1.r
      cap2.r
      cap3.r
    curso
      cop3530
        otoño88
          lista.r
        primavera89
          lista.r
        verano89
          lista.r
        basura.h
      alex
        basura.h
      guillermo
        trabajo
      curso
        cop3112
          otoño88
            califics
            prog1.r
            prog2.r
          otoño89
            prog2.r
            prog1.r
            califics

```

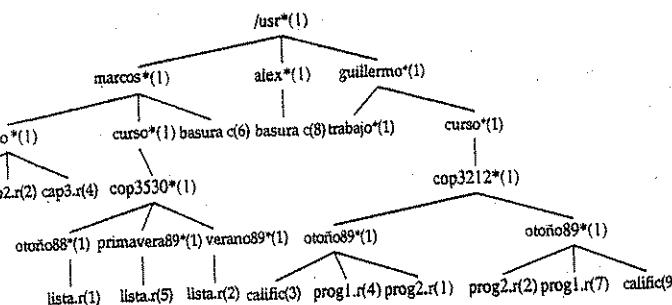
Esta estrategia se conoce como recorrido en *orden previo* (o *preorden*). En un recorrido en orden previo se trabaja en cada nodo antes (*previo*) de procesar sus hijos. Es claro que al ejecutar este programa la línea [2] se ejecuta exactamente una vez por nodo, ya que cada nodo se manda a la salida una vez. Como la línea [2] se ejecuta a lo más una vez por nodo, la línea [3] también se debe ejecutar una vez por nodo. Más aún, la línea [5] se puede ejecutar como máximo una vez por cada hijo de cada nodo. Pero el número de hijos es exactamente uno menos que el número de nodos. Para terminar, el ciclo *for* itera una vez por cada ejecución de la línea [5], más una vez al término de cada ciclo. Cada ciclo *for* termina en un apuntador *nil*, pero a lo más existe uno de estos por nodo. Así, la cantidad total de trabajo es constante por nodo. Si hay n archivos a visualizar, el tiempo de ejecución es $O(n)$.

Otro método común de recorrer un árbol es el *orden posterior* (o *postorden*). En un recorrido en orden posterior, el trabajo en cada nodo se realiza después (*posteriormente*) de evaluar sus hijos. Como ejemplo, la figura 4.8 representa la misma estructura de directorio de antes, pero aquí los números entre paréntesis representan el número de bloques de disco ocupados por cada archivo.

Los directorios son archivos también, así que tienen tamaño. Supongamos que se quiere calcular el número total de bloques usados por todos los archivos del árbol. La forma más natural de hacerlo consistiría en encontrar el número de bloques contenidos en los subdirectorios */usr/marcos* (30), */usr/alex* (9) y */usr/guillermo* (32). El número total de bloques es, entonces, el total en los subdirectorios (71) más uno, que corresponde al bloque usado por */usr*, para un total de 72. La función *tamaño_dir* de la figura 4.9 implanta esta estrategia.

Si D no es un directorio, entonces *tamaño_dir* sólo devuelve el número de bloques ocupados por D . En caso contrario, el número de bloques usados por D se suma al número de bloques encontrados (recursivamente) en todos los hijos. Para ver la diferencia entre la estrategia de recorrido en orden posterior y la estrategia de recorrido en orden previo, la figura 4.10 muestra cómo el algoritmo obtiene el tamaño de cada directorio o archivo.

Figura 4.8 Directorio de Unix con los tamaños de los archivos obtenidos mediante un recorrido en orden posterior



```

function tamaño_dir(D: directorio_o_archivo): integer;
var tam_total: integer;
begin
  [1] tam_total := 0;
  [2] if D es una entrada legítima then
      begin
        [3] tam_total := tamaño_archivo(D);
        [4] if (D es un directorio) then
            for cada hijo, h, en D do
              tam_total := tam_total + tamaño_directorio(h);
        [6] end;
        [7] tam_directorio := tam_total;
      end;
  end;
  
```

Figura 4.9 Rutina para calcular el tamaño de un directorio

cap1.r	3
cap2.r	2
cap3.r	4
libro	10
listar.r	1
otoño88	2
listar.r	5
primavera89	6
listar.r	2
verano89	3
cop3530	12
curso	13
basura.h	6
marcos	30
basura.h	8
alex	9
trabajo	1
califcs	3
prog1.r	4
prog2.r	1
otoño88	9
prog2.r	2
prog1.r	7
califcs	9
otoño89	19
cop3212	29
curso	30
guillermo	32
/usr	72

Figura 4.10 Seguimiento de la función *tamaño*

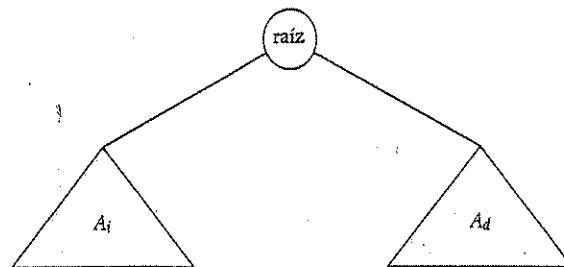


Figura 4.11 Árbol binario genérico

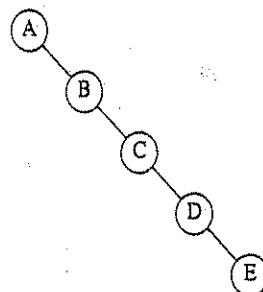


Figura 4.12 Árbol binario en el peor caso

4.2. Árboles binarios

Un árbol binario es un árbol en el cual ningún nodo tiene más de dos hijos.

La figura 4.11 muestra que un árbol binario consta de una raíz y dos subárboles, A_l y A_d , pudiendo ambos estar vacíos.

Una propiedad de los árboles binarios que a veces es importante es que la profundidad de un árbol binario promedio es considerablemente menor que n . Un análisis muestra que la profundidad media es $O(\sqrt{n})$, y que para un tipo especial de árbol binario, denominado *árbol binario de búsqueda*, el valor medio de la profundidad es $O(\log n)$. Desafortunadamente, la profundidad puede ser tan grande como $n - 1$, como lo muestra el ejemplo de la figura 4.12.

4.2.1. Implementación

Puesto que un árbol binario tiene a lo más dos hijos, podemos tener apuntadores directos a ellos. La declaración de nodos de árbol es similar en estructura a la de las listas doblemente enlazadas, donde un nodo es un registro consistente en la información *llave* más dos apuntadores (*izquierdo* y *derecho*) a otros nodos (véase la figura 4.13).

```

type
  ap_arbol = ^nodo_arbol;

  nodo_arbol = record
    elemento : tipo_elemento;
    izquierdo : ap_arbol;
    derecho : ap_arbol;
  end;

  ARBOL = ap_arbol;
  
```

Figura 4.13 Declaraciones del nodo de árbol binario

Muchas de las reglas que se aplican a las listas enlazadas también se aplicarán a árboles. En particular, cuando se realiza una inserción se tendrá que crear un nodo por medio de una llamada a *new*. Los nodos pueden ser liberados después del borrado, llamando a *dispose*.

Es posible dibujar árboles binarios usando recuadros rectangulares igual que en las listas enlazadas, pero en general los árboles se dibujan con círculos conectados por líneas, porque en realidad son grafos. Tampoco dibujaremos explícitamente los apuntadores nulos en los árboles porque cada árbol binario con n nodos requiere $n + 1$ apuntadores *nil*.

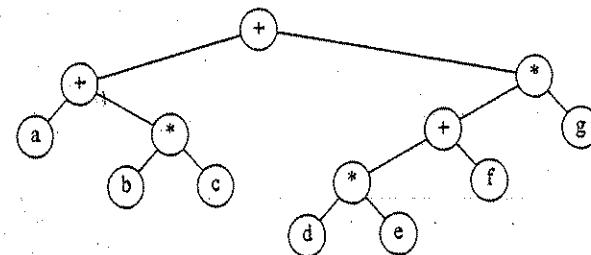
Los árboles binarios tienen muchas aplicaciones importantes no relacionadas con la búsqueda. Uno de los usos principales se da en el área del diseño de compiladores, que estudiaremos en seguida.

4.2.2. Árboles de expresión

La figura 4.14 muestra el ejemplo de un *árbol de expresión*.

Las hojas de un árbol de expresión son *operandos*, como constantes o nombres de variables, y los otros nodos contienen *operadores*. Este árbol particular tiene que ser binario, porque todas sus operaciones son binarias, y aunque es el caso más sencillo, es posible que algunos nodos tengan más de dos hijos. Un nodo también puede tener sólo un hijo, como es el caso con el operador *menos unario*. Podemos evaluar un árbol de expresión, A , aplicando el operador en la raíz a los valores obtenidos evaluando recursivamente los subárboles izquierdo y derecho. En nuestro ejemplo, el subárbol izquierdo se evalúa en $a + (b * c)$ y el subárbol derecho se evalúa en $((d * e) + f) * g$. El árbol completo representa $(a + (b * c)) + (((d * e) + f) * g)$.

Podemos obtener una expresión infija (con paréntesis de más) produciendo recursivamente una expresión izquierda con paréntesis, visualizando después el operador de la raíz y, por último, produciendo, recursivamente, una expresión derecha con paréntesis. Esta estrategia general (izquierda, nodo, derecha) se denomina *recorrido simétrico*; es fácil de recordar por el tipo de expresiones que produce.

Figura 4.14 Árbol de expresión para $(a + b * c) + ((d * e + f) * g)$

Una estrategia de recorrido alternativa consiste en visualizar recursivamente el subárbol izquierdo, el subárbol derecho y después el operador. Si aplicamos esta estrategia al árbol anterior, la salida es $a \ b \ c * + d \ e * f + g * +$, que se identifica fácilmente como la representación posfija vista en la sección 3.3.3. Por lo regular, a esta estrategia de recorrido se le conoce como recorrido en orden *postorden* (o *postorden*), y la vimos en la sección 4.1.

Una tercera estrategia es visualizar primero el operador y visualizar recursivamente los subárboles izquierdo y derecho. La expresión resultante, $+ \ + \ a \ * \ b \ c \ * \ + \ * \ d \ e \ f \ g$, es la notación *prefija* menos útil y la estrategia de recorrido es en orden *previo* (o *preorden*), la cual también vimos en la sección 4.1. Más adelante en el capítulo se volverá a esas estrategias.

Construcción de un árbol de expresión

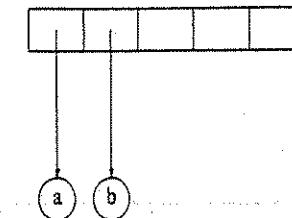
Ahora presentaremos un algoritmo para convertir una expresión posfija en un árbol de expresión. Puesto que ya se tiene un algoritmo para convertir de infija a posfija, podemos generar árboles de expresión a partir de los dos tipos comunes de entrada. El método que describimos guarda un fuerte parecido con el algoritmo de evaluación de posfija de la sección 3.2.3. Leemos la expresión, símbolo por símbolo. Si el símbolo es un operando, creamos un árbol de un nodo y metemos un apuntador a él en una pila. Si el símbolo es un operador, sacamos los apuntadores a dos árboles A_1 y A_2 de la pila (primero A_1) y se forma un nuevo árbol cuya raíz es el operador, con hijos izquierdo y derecho apuntando a A_2 y A_1 , respectivamente. Un apuntador a este nuevo árbol se mete entonces a la pila.

Por ejemplo, supongamos la entrada

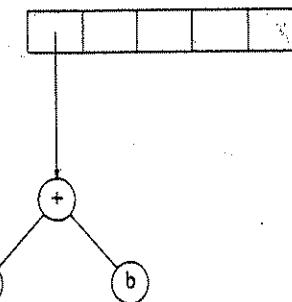
$a \ b \ + \ c \ d \ e \ + \ ^\star$

Los primeros dos símbolos son operandos, así que creamos árboles de un nodo y metemos apuntadores a ellos en una pila.[†]

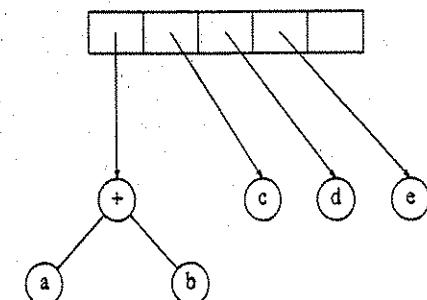
[†] Por conveniencia, se tendrá que la pila crece de izquierda a derecha en los diagramas.



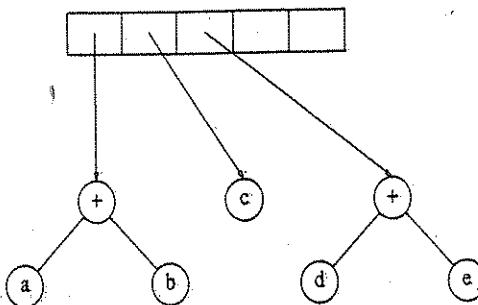
Después, se lee un '+', así que se sacan dos apuntadores a árbol de la pila, se forma un árbol nuevo y se mete a la pila un apuntador a éste.



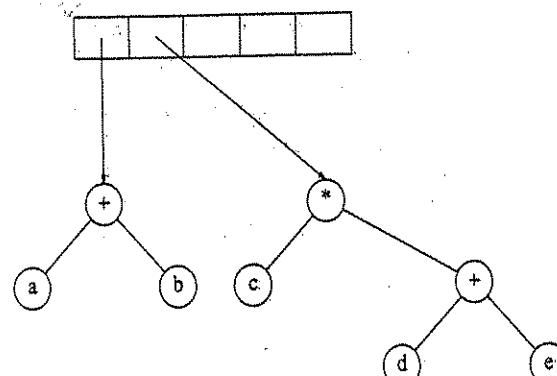
A continuación, se leen c , d y e , y para cada uno se forma un árbol de un nodo y el apuntador al árbol correspondiente se mete en la pila.



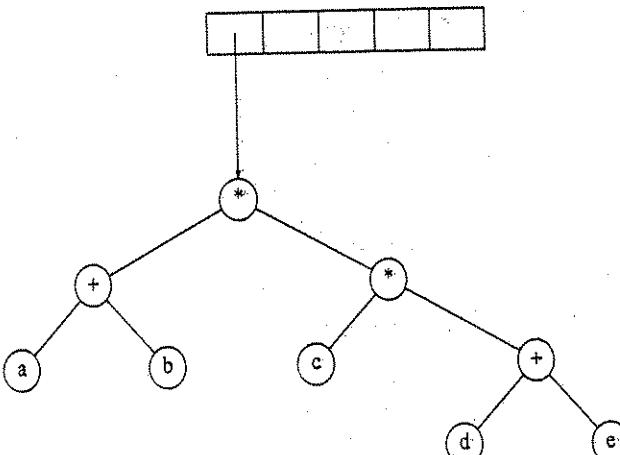
Ahora se lee un '+', lo cual hace que se combinen dos árboles.



Continuando, se lee un '*', así que se sacan dos apuntadores y se forma un árbol nuevo con '*' como raíz.



Para terminar, se lee el último símbolo, se combinan dos árboles y se deja en la pila un apuntador al árbol final.



4.3. El TDA árbol de búsqueda: Árboles binarios de búsqueda

Una aplicación importante de los árboles binarios es su uso en la búsqueda. Suponemos que cada nodo del árbol tiene asignado un valor de llave. En estos ejemplos, por simplicidad supondremos que son enteros, aunque se permiten llaves arbitrariamente complejas. Supondremos también que todas las llaves son diferentes, dejando para después el tratamiento de duplicados.

La propiedad que convierte a un árbol binario en un árbol binario de búsqueda es que para cada nodo, X , en el árbol, los valores de todas las llaves del subárbol izquierdo son menores que el valor de la llave de X , y los valores de todas las llaves del subárbol derecho son mayores que el valor de la llave de X . Nótese que ésto implica que todos los elementos del árbol deben estar ordenados en una forma consistente. En la figura 4.15 el árbol de la izquierda es un árbol binario de búsqueda, pero el de la derecha no lo es. El árbol de la derecha tiene un nodo con llave 7 en el subárbol izquierdo de un nodo con llave 6 (el cual es la raíz).

Ahora daremos unas breves descripciones de las operaciones que suelen realizarse con árboles binarios de búsqueda. Nótese que a causa de la definición recursiva de los árboles, es común escribir esas rutinas como recursivas. Debido a

Figura 4.15 Dos árboles binarios (sólo el de la izquierda es de búsqueda)

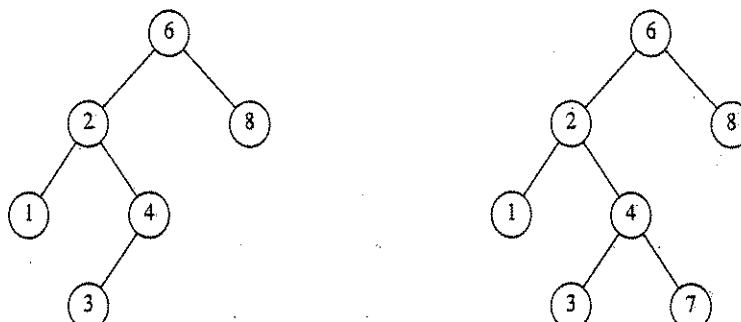


Figura 4.16 Declaraciones del árbol binario de búsqueda

```

type
  ap_arbol = ^nodo_arbol;
  nodo_arbol = record
    elemento: tipo_elemento;
    izquierdo: ap_arbol;
    derecho: ap_arbol;
  end;
  ARBOL_DE_BUSQUEDA = ap_arbol;
  
```

```

procedimiento crear_vacio(var A: ÁRBOL_DE_BÚSQUEDA);
begin
    A := nil;
end;

```

Figura 4.17 Rutina para hacer un árbol vacío

general no es necesario preocuparse del desbordamiento de la pila. Repetimos la definición de tipos de la figura 4.16. Puesto que todos los elementos se pueden ordenar, supondremos que los operadores $<$, $>$ e $=$ se pueden aplicar sobre ellos, aun cuando esto parezca sintácticamente erróneo para algunos tipos.

4.3.1. Crear_vacio

Esta operación es para poner valores iniciales, principalmente. Algunos programadores prefieren poner como valor inicial un árbol de un nodo, pero esta implantación sigue más de cerca la definición recursiva de los árboles. También es una rutina sencilla, como queda de manifiesto en la figura 4.17.

4.3.2. Buscar

Por lo regular, esta operación requiere devolver un apuntador al nodo del árbol A que tiene la llave x , o nil si no existe tal nodo. La estructura del árbol simplifica esto. Si A es nil , se puede devolver sólo nil . Si no, si la llave almacenada en A es x , se puede devolver A . Si no es así, hacemos una llamada recursiva con el subárbol de A , bien el izquierdo o el derecho, dependiendo de la relación entre x y la llave almacenada en A . El código de la figura 4.18 es una implantación de esta estrategia.

Cabe atender al orden de las condiciones. Es crucial que la comprobación del árbol vacío se realice al inicio, ya que de otro modo las indirecciones se aplicarían

Figura 4.18 Operación buscar para árboles binarios de búsqueda

```

function buscar(x: integer; A: ÁRBOL_DE_BÚSQUEDA): ap_árbol;
begin
    if A = nil then
        buscar := nil
    else
        if x < A^.elemento then
            buscar := buscar(x, A^.izquierdo)
        else
            if x > A^.elemento then
                buscar := buscar(x, A^.derecho)
            else
                buscar := A;
    end; [buscar]

```

4.3. EL TDA ÁRBOL DE BÚSQUEDA: ÁRBOLES BINARIOS DE BÚSQUEDA

Cabe atender al orden de las condiciones. Es crucial que la comprobación del árbol vacío se realice al inicio, ya que de otro modo las indirecciones se aplicarían sobre un apuntador nil . Las condiciones restantes se disponen con el caso menos probable al final. También debe notarse que en ambas llamadas recursivas hay recursión por la cola y se puede eliminar fácilmente con una asignación y un *goto*. El uso de la recursión por la cola se justifica aquí porque la simplicidad en la expresión algorítmica compensa el descenso de la velocidad, y se espera que la cantidad de espacio usado en la pila sea de sólo $O(\log n)$.

4.3.3. Buscar_mín y buscar_máx

Estas rutinas devuelven la posición de los elementos menor y mayor en el árbol, respectivamente. Aunque la devolución de los valores exactos de esos elementos podría parecer más razonable, esto sería inconsistente con la operación *buscar*. Es importante que las operaciones que parecen semejantes hagan cosas semejantes. Para realizar un *buscar_mín*, se empieza en la raíz y se va a la izquierda tantas veces como hijos haya en el lado izquierdo. El punto final es el elemento mínimo. La rutina *buscar_máx* es lo mismo, excepto que el salto se hace a la derecha.

Figura 4.19 Implantación recursiva de buscar_mín para árboles binarios de búsqueda

```

function buscar_mín(A: ÁRBOL_DE_BÚSQUEDA): ap_árbol;
begin
    if A = nil then
        buscar_mín := nil
    else
        if A^.izquierdo = nil then
            buscar_mín := A
        else
            buscar_mín := buscar_mín(A^.izquierdo);
    end;

```

Figura 4.20 Implantación no recursiva de buscar_máx para árboles binarios de búsqueda

```

function buscar_máx(A: ÁRBOL_DE_BÚSQUEDA): ap_árbol;
begin
    if A <> nil then
        while A^.derecho <> nil do
            A := A^.derecho;
        buscar_máx := A;
    end;

```

Nótese el cuidado que se tiene con el caso degenerado de un árbol vacío. Aunque esto siempre es importante, es especialmente crucial en programas recursivos. También cabe observar que el programa no recursivo no funciona si A se pasa como *var*. Pasando una copia podemos cambiar A con impunidad. De todos modos hay que tener un extremo cuidado, porque una proposición como $A^{\wedge}.\text{derecho} := A^{\wedge}.\text{derecho}^{\wedge}.\text{derecho}$ provocará cambios en la mayoría de los lenguajes, aun cuando A no se haya declarado como *var*.

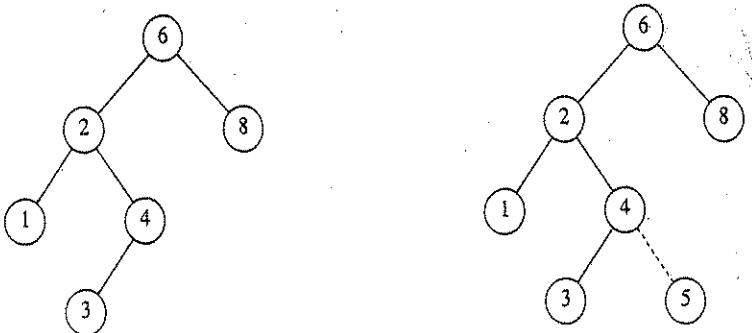
4.3.4. Insertar

Desde el punto de vista conceptual, la rutina de inserción es sencilla. Para insertar x en el árbol A , se procede hacia abajo en el árbol igual que en *buscar*. Si se encuentra x , no se hace nada (o se "actualiza" algo). Si no es así, se inserta x en el último espacio del camino recorrido. La figura 4.21 muestra qué sucede. Para insertar 5, se recorre el árbol como si se ejecutara un *buscar*. En el nodo con la llave 4 se necesita ir a la derecha, pero no hay ningún subárbol, así que 5 no está en el árbol, y éste es su lugar correcto.

Los duplicados se pueden manejar con un campo adicional en el registro del nodo, que indique la frecuencia de aparición. Esto agrega un poco de espacio adicional al árbol completo, pero es mejor que poner duplicados en el árbol (lo cual tiende a hacer el árbol muy profundo). Por supuesto esta estrategia no funciona si la llave sólo es parte de un registro más grande. Si ése es el caso, es posible tener todos los registros con la misma llave en una estructura de datos auxiliar, como una lista u otro árbol de búsqueda.

La figura 4.22 muestra el código para la rutina de inserción.

Figura 4.21 Árboles binarios de búsqueda antes y después de la inserción del 5



4.3. EL TDA ÁRBOL DE BÚSQUEDA: ÁRBOLES BINARIOS DE BÚSQUEDA

```

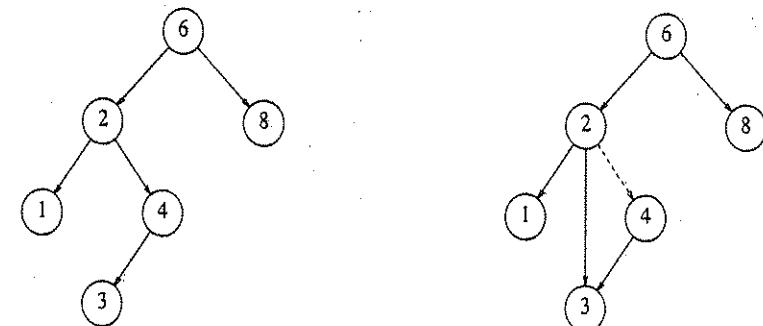
procedure insertar(x: tipo_elemento; var A: ÁRBOL_DE_BÚSQUEDA);
begin
  if A = nil then
    begin
      [crear y devolver un nodo de árbol]
      new(A);
      if A = nil then
        error_fatal("¡¡Memoria agotada!!!");
      else
        begin
          A^.elemento := x;
          A^.izquierdo := nil;
          A^.derecho := nil;
        end;
    end; [if A = nil]
  else
    begin
      if x < A^.elemento then
        insertar(x, A^.izquierdo)
      else:
        if x > A^.elemento then
          insertar(x, A^.derecho);
        [si no, x ya está en el árbol, y no se hace nada]
    end; [insertar]
end;
```

Figura 4.22 Inserción en un árbol binario de búsqueda

4.3.5. Eliminar

Como es común con muchas estructuras de datos, la operación más complicada es la eliminación. Una vez encontrado el nodo a eliminar, es necesario considerar varias posibilidades.

Figura 4.23 Eliminación de un nodo (4) con un hijo, antes y después



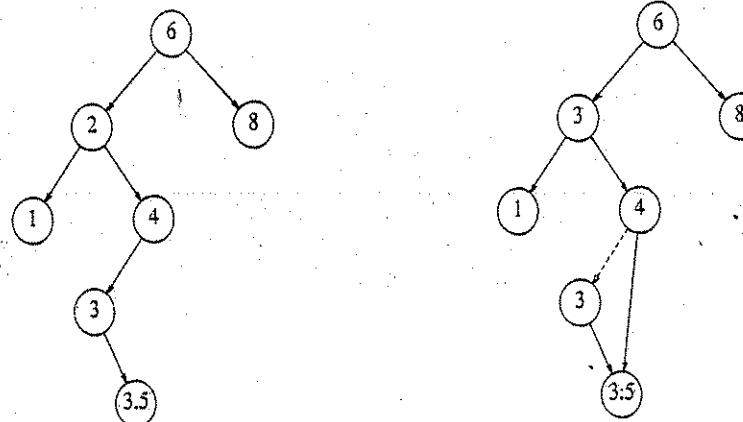


Figura 4.24 Eliminación de un nodo (2) con dos hijos, antes y después

Si el nodo es una hoja, se puede eliminar de inmediato. Si el nodo tiene un hijo, el nodo se puede eliminar después de ajustar un apuntador del padre, para saltar el nodo (por claridad se dibujarán las direcciones de los apuntadores explícitamente). Véase la figura 4.23. Nótese que el nodo eliminado queda sin referencia, y puede ser liberado sólo si se conservó un apuntador a él.

El caso complicado se da cuando el nodo tiene dos hijos. La estrategia general consiste en sustituir la llave de este nodo por la llave más pequeña del subárbol derecho (que es fácil encontrar) y eliminar recursivamente ese nodo (que ahora está vacío). Como el nodo menor del subárbol derecho no puede tener un hijo izquierdo, el segundo *eliminar* es fácil. La figura 4.24 muestra un árbol inicial y el resultado de la eliminación. El nodo por eliminar es el hijo izquierdo de la raíz; la llave es 2. Se sustituye por la llave menor de su subárbol derecho (3), y luego ese nodo se elimina como antes.

El código de la figura 4.25 realiza la eliminación. Es ineficiente, porque hace dos descensos en el árbol para encontrar y eliminar el menor nodo del subárbol derecho cuando esto es apropiado. Es fácil quitar esta ineficiencia escribiendo una función especial *eliminar_min*, la cual se dejó por simplicidad únicamente.

Si se espera que el número de eliminaciones sea pequeño, entonces una estrategia popular es usar la *eliminación perezosa*: cuando se ha de eliminar un elemento, se deja en el árbol y sólo se *marca* como eliminado. Esto es especialmente popular si hay llaves duplicadas, ya que entonces es posible disminuir el campo que lleva la cuenta de la frecuencia de apariciones. Si el número de nodos reales en el árbol es igual al número de nodos "eliminados", se espera que la profundidad del árbol sólo crezca en 1 (por qué?), así que hay una penalización de tiempo muy pequeña asociada con la eliminación perezosa. También, si una llave eliminada se vuelve a insertar, se evita la sobrecarga de asignar una celda nueva.

```

procedure eliminar(x: tipo_elemento; var A: ÁRBOL_DE_BÚSQUEDA);
  var celda_temp: ap_árbol;

begin
  if A = nil then
    error('Elemento no encontrado')
  else
    begin
      if x < A^.elemento then {ir a la izquierda}
        eliminar(x, A^.izquierdo)
      else
        if x > A^.elemento then {ir a la derecha}
          eliminar(x, A^.derecho)
        else {buscar el elemento a eliminar}
          begin
            if A^.izquierdo = nil then {sólo un hijo derecho}
              begin
                celda_temp := A;
                A := A^.derecho;
                dispose(celda_temp);
              end
            else
              if A^.derecho = nil then {sólo un hijo izquierdo}
                begin
                  celda_temp := A;
                  A := A^.izquierdo;
                  dispose(celda_temp);
                end
              else {dos hijos. Se reemplaza con el menor del subárbol derecho}
                begin
                  celda_temp := buscar_mín(A^.derecho);
                  A^.elemento := celda_temp^.elemento;
                  eliminar(A^.elemento, A^.derecho),
                end,
              end; {procesamiento de los dos hijos}
            end; {if A=nil else}
          end; {eliminar}
    end;
end;
  
```

Figura 4.25 Rutina de eliminación para árboles binarios de búsqueda

4.3.6. Análisis del caso promedio

Intuitivamente, esperamos que todas las operaciones de la sección anterior, excepto *crear_nulo*, deben tardar un tiempo $O(\log n)$, porque en tiempo constante se desciende de un nivel en el árbol, operando así sobre un árbol que tiene aproximadamente la mitad de longitud. En efecto, el tiempo de ejecución de todas las operaciones, excepto *crear_nulo*, es $O(d)$, donde d es la profundidad del nodo que contiene la llave de acceso.

En esta sección demostraremos que la profundidad media de todos los nodos en un árbol es $O(\log n)$, suponiendo que todos los árboles son igualmente probables.

La suma de las profundidades de todos los nodos en el árbol se denomina *longitud del camino interno*. Calcularemos la longitud media del camino interno de un árbol binario de búsqueda, donde el promedio se toma de todos los árboles binarios de búsqueda posibles.

Sea $D(n)$ la longitud media del camino interno de un árbol de n nodos. $D(1) = 0$. Un árbol de n nodos consta de un subárbol izquierdo de i nodos y un subárbol derecho de $(n - i - 1)$ nodos, más la raíz a la profundidad cero para $0 \leq i < n$. $D(i)$ es la longitud media del camino interno del subárbol izquierdo con respecto a su raíz. En el árbol principal, todos esos nodos son un nodo más profundos. Lo mismo es válido para el subárbol derecho. Así, obtenemos la recurrencia

$$D(n) = D(i) + D(n - i - 1) + n - 1$$

Si todos los tamaños de subárboles son igualmente probables, lo cual es cierto para árboles binarios de búsqueda, pero no para árboles binarios, entonces el valor promedio de $D(i)$ y $D(n - i - 1)$ es $(1/n) \sum_{j=0}^{n-1} D(j)$. Esto da:

$$D(n) = \frac{2}{n} \left[\sum_{j=0}^{n-1} D(j) \right] + n - 1$$

Esta recurrencia se encontrará y resolverá en el capítulo 7, obteniendo $D(n) = O(n \log n)$. Así, la profundidad esperada de cualquier nodo es $O(\log n)$. Por ejemplo, el árbol de 500 nodos generado aleatoriamente que se muestra en la figura 4.26 tiene nodos en una profundidad esperada de 9.98.

Es tentador afirmar de inmediato que este resultado implica que el tiempo medio de ejecución de todas las operaciones estudiadas en la sección previa es $O(\log n)$, pero esto no es del todo cierto. La razón de ello es que, debido a las eliminaciones, no está claro que todos los árboles binarios de búsqueda sean igualmente probables. En particular, el algoritmo de eliminación antes descrito favorece la creación de subárboles izquierdos más profundos que los derechos, porque siempre se sustituye un nodo eliminado por un nodo del subárbol derecho. El efecto exacto de esta estrategia es aún desconocido, pero parece sólo una novedad teórica. Se ha demostrado que si alternamos las inserciones y eliminaciones $\Theta(n^2)$ veces, los árboles tendrán una profundidad esperada de $\Theta(\sqrt{n})$. Después de 250 mil pares *insertar/eliminar* aleatorios, el árbol que estaba cargado a la derecha en la figura 4.26 parece decididamente desequilibrado (profundidad media = 12.51). Véase la figura 4.27.

Podríamos intentar eliminar el problema escogiendo aleatoriamente entre el menor elemento del subárbol derecho y el mayor elemento del izquierdo cuando se reemplaza el elemento eliminado. Al parecer, esto elimina la tendencia y debe conservar los árboles equilibrados, pero de hecho nadie ha probado realmente esto. En cualquier caso, este fenómeno parece ser más una novedad teórica porque el

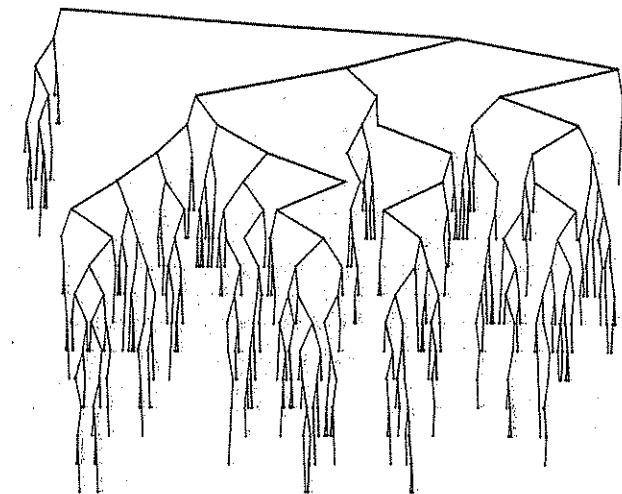


Figura 4.26 Árbol binario de búsqueda generado aleatoriamente

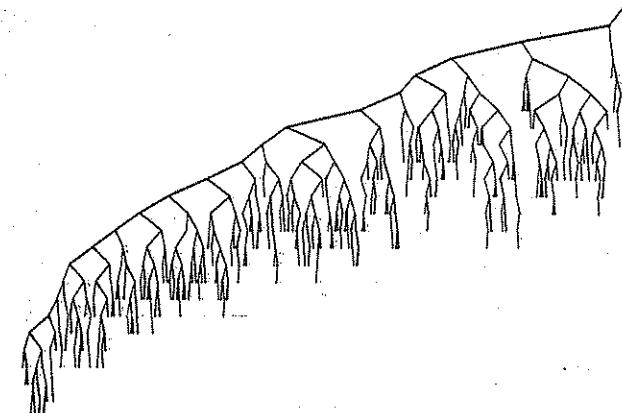


Figura 4.27 Árbol binario de búsqueda después de $O(n^2)$ pares *insertar/eliminar*

efecto no aparece en los árboles pequeños, y lo que es más extraño aún, si se usan $O(n^2)$ pares *insertar/eliminar*, parece que el árbol logra el equilibrio!

El punto principal de este análisis es que la decisión respecto al significado de "promedio" suele ser difícil en extremo y puede requerir suposiciones que pueden ser o no válidas. En ausencia de eliminaciones, o cuando se usa la eliminación perezosa, se puede demostrar que todos los árboles binarios de búsqueda son igualmente probables y podemos concluir que los tiempos medios de ejecución de

las operaciones anteriores son $O(\log n)$. Salvo en casos extraños como el antes mencionado, este resultado es muy consistente con el comportamiento observado.

Si la entrada viene en un árbol ordenado previamente, entonces una serie de operaciones *insertar* tardarán un tiempo cuadrático y darán una implantación muy costosa de lista enlazada, ya que el árbol consistirá sólo en nodos sin hijos izquierdos. Una solución al problema es insistir en una condición estructural adicional, denominada *equilibrio*: no se permite que ningún nodo alcance una profundidad excesiva.

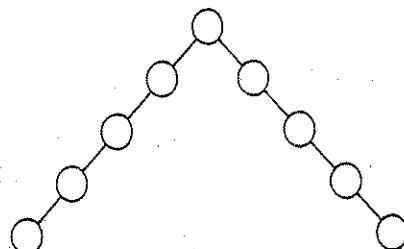
Hay unos cuantos algoritmos generales para implantar árboles equilibrados. La mayoría son más complicados que en los árboles binarios de búsqueda estándar, y tardarán más tiempo en promedio. No obstante, ofrecen una protección contra los casos desconcertantemente simples. Más adelante esbozaremos una de las formas más antiguas de árboles equilibrados, el árbol AVL.

Un segundo método, más novedoso, es abstenerse de la condición de equilibrio y permitir que el árbol sea arbitrariamente profundo, pero aplicando después de cada operación una regla de reestructuración que tiende a hacer eficientes las operaciones futuras. Estos tipos de estructuras de datos se suelen clasificar como *autoajustables*. En el caso de un árbol binario de búsqueda, ya no podemos garantizar una cota $O(\log n)$ en cualquier operación sencilla, pero podemos demostrar que cualquier secuencia de m operaciones tarda un tiempo total de $O(m \log n)$ en el peor caso. En general, esta protección es suficiente contra el peor caso. La estructura de datos que estudiaremos se conoce como *árbol desplegado*; su análisis es algo intrincado y se tratará en el capítulo 11.

4.4. Árboles AVL

Un árbol AVL (Adelson-Velskii y Landis) es un árbol binario de búsqueda con una condición de *equilibrio*. Debe ser fácil mantener la condición de equilibrio, y ésta asegura que la profundidad del árbol es $O(\log n)$. La idea más simple es exigir que los subárboles izquierdo y derecho tengan la misma altura. Como se muestra en la figura 4.28, esta idea no obliga a que el árbol sea poco profundo.

Figura 4.28 Un mal árbol binario. La condición de equilibrio en la raíz no es suficiente

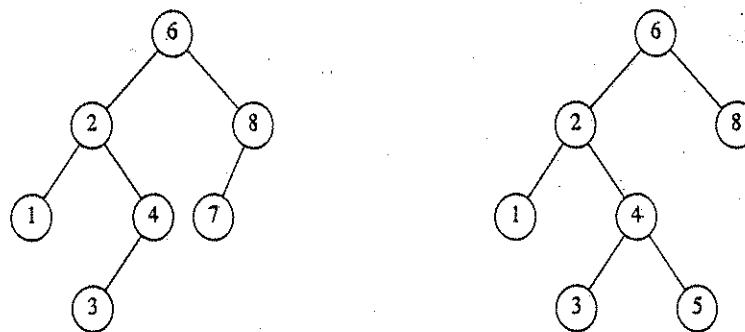


Otra condición de equilibrio insistiría en que todo nodo debe tener subárboles izquierdo y derecho a la misma altura. Si la altura de un subárbol vacío se define como -1 (como suele hacerse), sólo los árboles perfectamente equilibrados de $2^k - 1$ nodos satisfarían este criterio. Así, aunque esto garantiza árboles de profundidad pequeña, la condición de equilibrio es demasiado rígida para ser útil y es necesario que la moderemos.

Un árbol AVL es idéntico a un árbol binario de búsqueda, excepto en que, para todo nodo en el árbol, la altura de los subárboles izquierdo y derecho puede diferir a lo más en 1. (La altura de un árbol vacío se define como -1.) En la figura 4.29 el árbol de la izquierda es AVL, pero el de la derecha no. Hay que mantener la información de altura de cada nodo (en el registro nodo). Es fácil mostrar que la altura de un árbol AVL es a lo más, aproximadamente, $1.44 \log(n + 2) - 0.328$, pero en la práctica es cercana a $\log(n + 1) + 0.25$ (aunque lo último no ha sido demostrado). Por ejemplo, el árbol AVL de altura 9 con la menor cantidad posible de nodos (143) se muestra en la figura 4.30. Este árbol tiene como subárbol izquierdo un árbol AVL de altura 7 de tamaño mínimo. El subárbol derecho es un árbol AVL de altura 8 de tamaño mínimo. Esto nos dice que el número mínimo de nodos, $N(h)$, en un árbol AVL de altura h está dado por $N(h) = N(h - 1) + N(h - 2) + 1$. Para $h = 0$, $N(h) = 1$. Para $h = 1$, $N(h) = 2$. La función $N(h)$ está muy relacionada con los números de Fibonacci, de lo cual se infiere la cota indicada antes sobre la altura de un árbol AVL.

Así, todas las operaciones sobre árboles se pueden resolver en tiempo $O(\log n)$, con la posible excepción de la inserción (suponiendo eliminación perezosa). Cuando se hace una inserción es necesario actualizar toda la información de equilibrio para los nodos en el camino a la raíz, pero la razón de que la inserción sea potencialmente difícil es que insertar un nodo puede violar la propiedad de árbol AVL. (Por ejemplo, insertar $6\frac{1}{2}$ en el árbol AVL de la figura 4.29 destruiría la condición de equilibrio en el nodo con llave 8.) Si éste es el caso, se tiene que restaurar la propiedad antes de considerar terminado el paso de inserción. Resulta que esto se puede hacer siempre

Figura 4.29 Dos árboles binarios de búsqueda. Sólo el de la izquierda es AVL



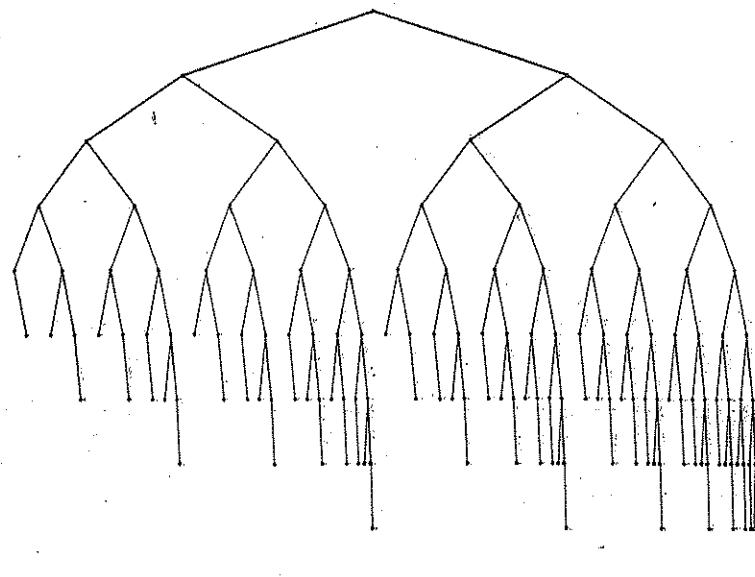


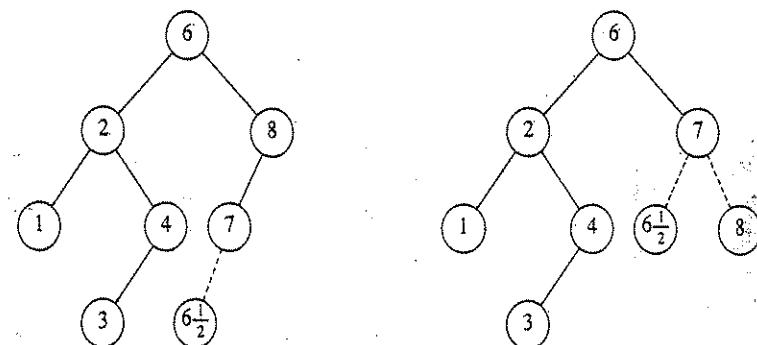
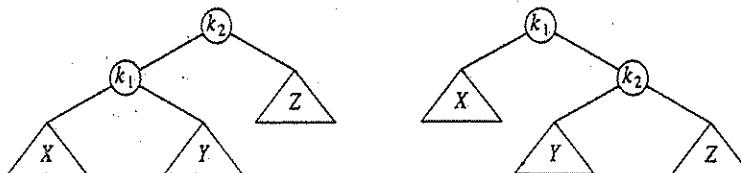
Figura 4.30 El árbol AVL de altura 9 más pequeño

con una sencilla modificación al árbol, conocida como rotación. A continuación se describe esto.

4.4.1. Rotación sencilla

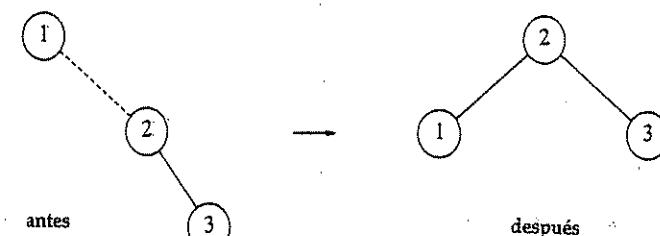
Los dos árboles de la figura 4.31 contienen los mismos elementos y ambos son árboles binarios de búsqueda. Ante todo, en ambos árboles $k_1 < k_2$. Segundo, todos los elementos del subárbol X son menores que k_1 en ambos árboles. Tercero, todos los elementos del subárbol Z son mayores que k_2 . Por último, todos los elementos del subárbol Y están entre k_1 y k_2 . La conversión de uno de los árboles mencionados al otro se denomina *rotación*. En una rotación sólo intervienen unos cuantos cambios de apuntadores (más adelante veremos cuántos exactamente), y cambia la estructura del árbol pero preserva la propiedad de árbol de búsqueda.

Figura 4.31: Rotación simple

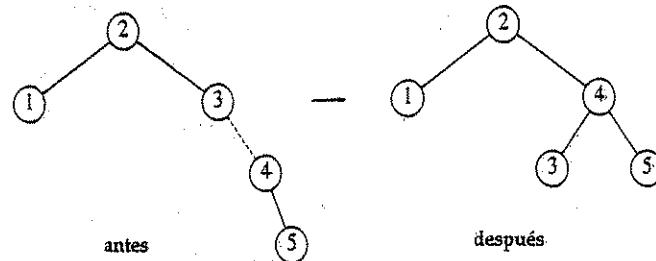
Figura 4.32 Propiedad AVL destruida con la inserción de $6\frac{1}{2}$, después recomposta con una rotación

No es obligatorio que la rotación se haga en la raíz del árbol; se puede hacer en cualquier nodo del árbol, ya que cualquier nodo es la raíz de algún subárbol, y puede transformar cualquier árbol en el otro. Esto da un método sencillo para arreglar un árbol AVL si la inserción causa que algún nodo pierda la condición de equilibrio: se hace una rotación en ese nodo. El algoritmo básico consiste en iniciar en el nodo insertado y subir en el árbol, actualizando la información del equilibrio en cada nodo del camino. Acabamos si se llega a la raíz sin haber encontrado ningún nodo desequilibrado. Si no, se aplica una rotación al primer nodo incorrecto que se encuentre, se ajusta su equilibrio, y ya está (no necesitamos continuar hasta llegar a la raíz). En muchos casos, esto basta para reequilibrar el árbol. Por ejemplo, en la figura 4.32, después de la inserción de $6\frac{1}{2}$ en el árbol AVL original a la izquierda, el nodo 8 se desequilibra. Así, hacemos una rotación simple entre 7 y 8, obteniendo el árbol de la derecha.

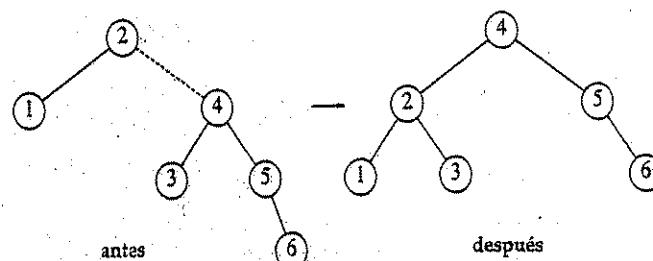
Ahora veamos un ejemplo más grande. Supongamos que se comienza con un árbol AVL vacío inicialmente, y se insertan las llaves 1 a 7 en orden secuencial. El primer problema surge en el momento de insertar la llave 3, porque la propiedad AVL se viola en la raíz. Efectuamos una rotación simple entre la raíz y su hijo derecho para reparar el problema. El árbol se muestra en la figura siguiente, antes y después de la rotación:



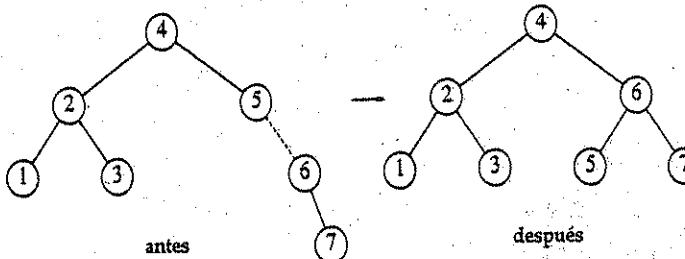
Para aclarar las cosas, una línea punteada indica los dos nodos que se someten a la rotación. Después, insertamos la llave 4, lo que no ocasiona problemas, pero la inserción de 5 crea una violación en el nodo 3, la cual se repara con una rotación simple. Además del cambio local causado por la rotación, el programador debe recordar que el resto del árbol debe estar enterado del cambio. Esto significa que se debe re establecer el hijo derecho de 2 para apuntar a 4 en vez de a 3. Es fácil olvidarse de hacer esto y podría destruir el árbol (quedando inaccesible el 4).



Después, se inserta el 6. Esto ocasiona un problema de equilibrio para la raíz, ya que su subárbol izquierdo es de altura 0, y su subárbol derecho sería de altura 2. Por lo tanto, efectuamos una rotación simple en la raíz, entre 2 y 4.

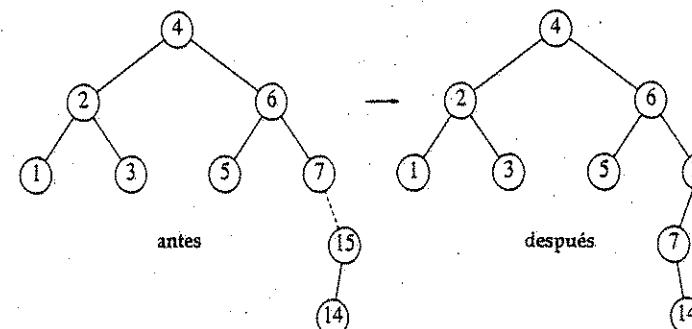


La rotación se realiza haciendo de 2 un hijo de 4 y haciendo del subárbol izquierdo original de 4 el nuevo subárbol derecho de 2. Toda llave de este subárbol debe estar entre 2 y 4, para que esta transformación tenga sentido. La siguiente llave que insertamos es 7, lo cual causa otra rotación.



4.4.2. Rotación doble

El algoritmo descrito en los párrafos precedentes tiene un problema. Existe un caso en que la rotación no recomponen el árbol. Continuando con nuestro ejemplo, supongamos que se insertan las llaves 8 a 15 en orden inverso. La inserción del 15 es fácil, ya que no destruye la propiedad de equilibrio, pero al insertar el 14 se ocasiona un desequilibrio de altura en el nodo 7.



Como muestra el diagrama, la rotación simple no ha corregido el desequilibrio de altura. El problema es que el desequilibrio de altura fue ocasionado por un nodo insertado en el árbol que contiene los elementos medios (árbol B) a la vez que los otros árboles tenían alturas idénticas. El caso es fácil de revisar, y la solución se conoce como *rotación doble*, que es semejante a la simple pero abarca cuatro subárboles.

Figura 4.33 Rotación doble (derecha-izquierda)

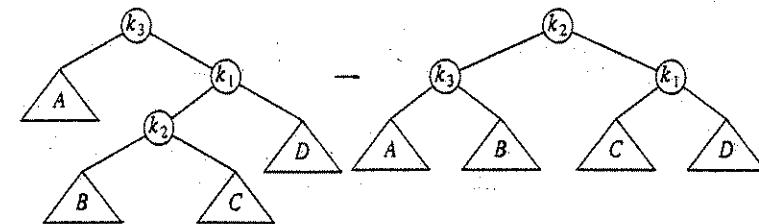
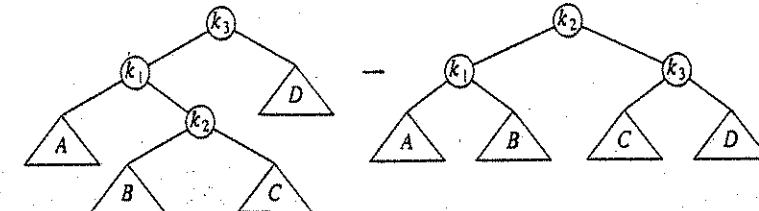
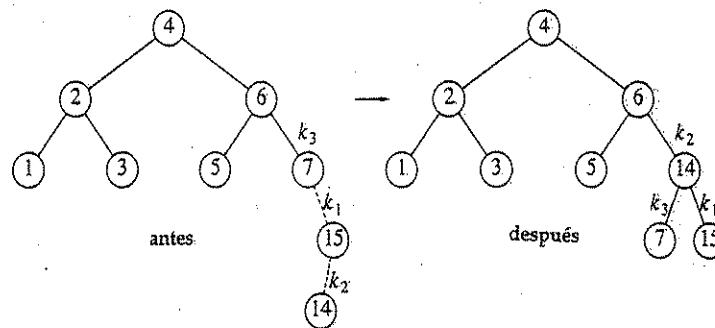


Figura 4.34 Rotación doble (izquierda-derecha)

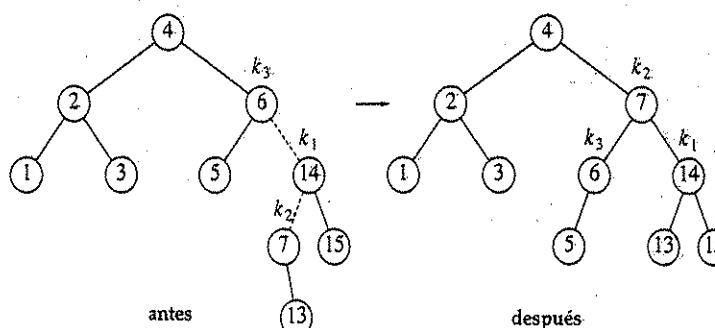


boles en vez de tres. En la figura 4.33, el árbol de la izquierda se convierte en el árbol de la derecha. A propósito, el efecto es el mismo que si se rotara entre k_1 y k_2 y después entre k_2 y k_3 . Hay un caso simétrico, que también se muestra (figura 4.34).

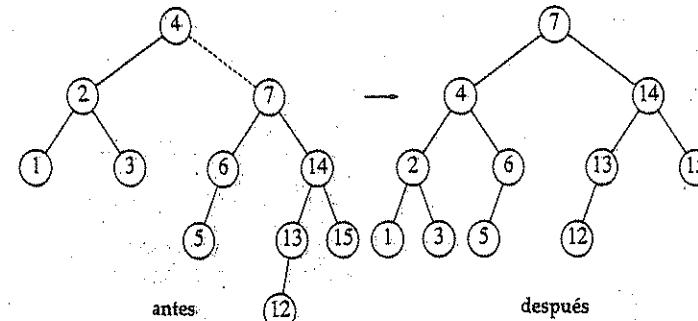
En este ejemplo, la rotación doble es una rotación derecha-izquierda y en ella intervienen los nodos 7, 15 y 14. Aquí, k_3 es el nodo con la llave 7, k_1 es el nodo con la llave 15 y k_2 es el nodo con la llave 14. Los subárboles A, B, C y D están todos vacíos.



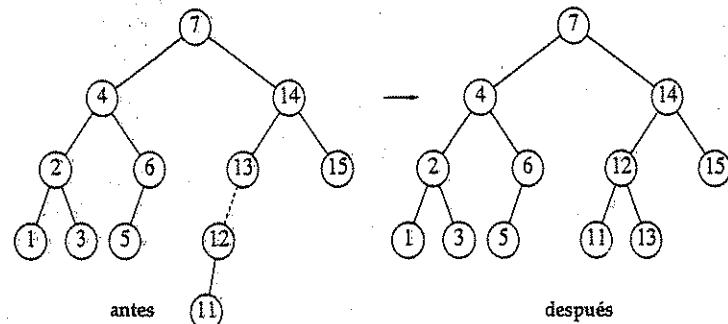
Después se inserta el 13, lo cual requiere una rotación doble. De nuevo, aquí la rotación es una rotación doble derecha-izquierda en que intervienen 6, 14 y 7 y que restaurará el árbol. En este caso, k_3 es el nodo con la llave 6, k_1 es el nodo con la llave 14 y k_2 es el nodo con la llave 7. El subárbol A es el árbol cuya raíz está en el nodo con la llave 5, el subárbol B es el subárbol vacío que originalmente era el hijo izquierdo del nodo con la llave 7, el subárbol C es el árbol cuya raíz está en el nodo con la llave 13 y, finalmente, el subárbol D es el árbol con raíz en el nodo con la llave 15.



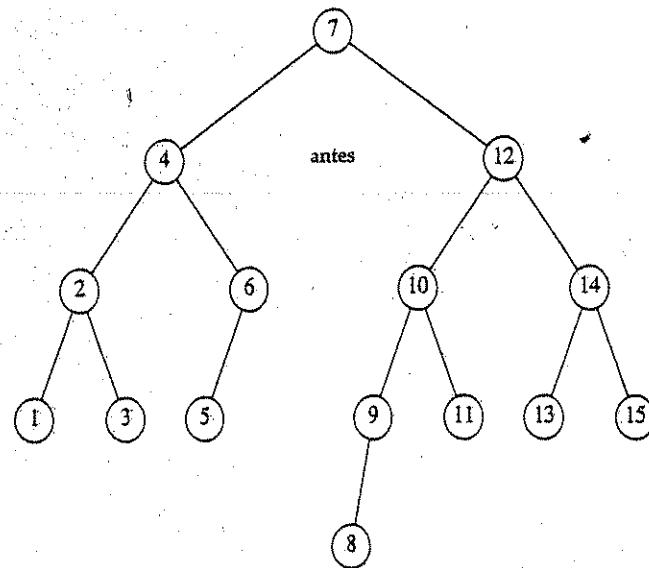
Si ahora se inserta el 12, hay un desequilibrio en la raíz. Como el 12 no está entre 4 y 7, sabemos que la rotación simple funcionará.



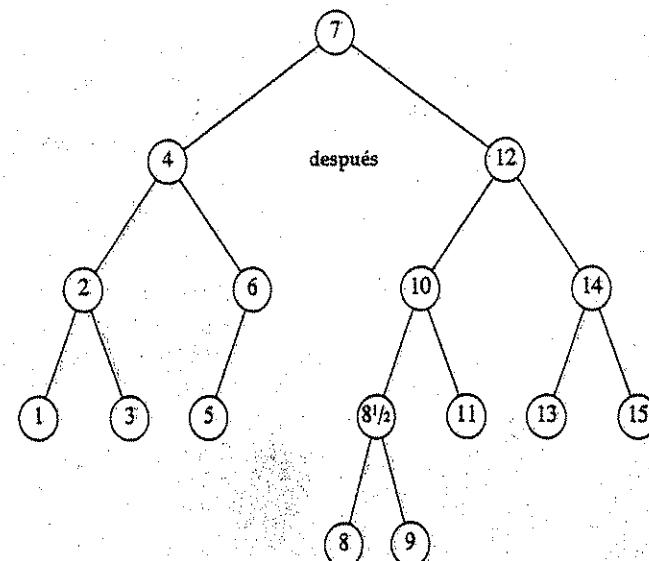
La inserción de 11 requerirá una rotación simple:



Para insertar 10 se necesita una rotación simple, y lo mismo se cumple para la inserción subsecuente del 9. Insertamos el 8 sin rotación, creando así el siguiente árbol casi perfectamente equilibrado.



Finalmente, se inserta $8\frac{1}{2}$ para mostrar el caso simétrico de la rotación doble. Observemos que $8\frac{1}{2}$ causa que se desequilibre el nodo con la llave 9. Puesto que $8\frac{1}{2}$ está entre 9 y 8 (que es hijo de 9 en el camino a $8\frac{1}{2}$), hay que efectuar una rotación doble para obtener el siguiente árbol.



El lector puede verificar que cualquier desequilibrio causado por una inserción en un árbol AVL siempre se puede solucionar con una rotación, simple o doble. Los detalles de programación son casi directos, excepto que haya varios casos. Para insertar un nodo nuevo con llave x en un árbol AVL, A , insertamos recursivamente x en el subárbol adecuado de A (al que llamaremos A_{id}). Si la altura de A_{id} no cambia, ya está. Si no es así, si aparece un desequilibrio de altura en A , efectuamos la rotación adecuada, simple o doble, dependiendo de x y las llaves en A y A_{id} , actualizamos las alturas (haciendo la conexión del resto del árbol anterior), y queda resuelto. Ya que siempre es suficiente una rotación, una bien cuidada versión codificada no recursiva resulta ser significativamente más rápida que la recursiva. No obstante, es bastante difícil hacer codificaciones correctas de las versiones no recursivas, por lo que muchos programadores implantan recursivamente los árboles AVL.

Otro aspecto de eficiencia se refiere al almacenamiento de la información sobre la altura. Como en realidad lo que se requiere es la diferencia en la altura, la cual se garantiza que es pequeña, podría hacerse con dos bits (para representar +1, 0, -1) si realmente se intenta. Hacerlo así evitará el cálculo repetitivo de factores de equilibrio pero a costa de una pérdida de claridad. El código resultante es un poco más complicado que si se almacenara la altura en cada nodo. Si se escribe una rutina recursiva, es probable que la velocidad no sea la consideración principal. En este caso, la ligera ventaja en velocidad que se obtiene al almacenar los factores de equilibrio difícilmente compensa la pérdida de claridad y la relativa simplicidad. Más aún, como de cualquier modo la mayoría de las máquinas alinearán esto a una frontera de 8 bits al menos, es probable que no haya ninguna diferencia en la cantidad de espacio usado. Ocho bits nos permitirán almacenar alturas absolutas de hasta 255. Como el árbol está equilibrado, es inconcebible que esto sea insuficiente (véanse los ejercicios).

Con todo lo anterior estamos listos para escribir las rutinas AVL. Sólo haremos parte del trabajo y el resto se dejará como ejercicio. Primero, necesitamos las declaraciones. Éstas se dan en la figura 4.35. También hace falta una función para devolver la altura de un nodo. Esta función es necesaria para manejar el molesto caso de un apuntador *nil*. Esto se muestra en la figura 4.36. Es fácil escribir la rutina de inserción básica, ya que consiste casi totalmente en llamadas a funciones (véase la figura 4.37).

Figura 4.35 Declaración de nodos para árboles AVL

```

type
  ap_avl = ^nodo_avl;

  nodo_avl = record
    elemento: tipo_elemento;
    izq : ap_avl;
    der : ap_avl;
    altura: integer;
  end;

  ARBOL_DE_BUSQUEDA = ^nodo_avl;

```

```

function altura(p: ap_avl): integer;
begin
  if p = nil then
    altura := -1
  else
    altura := p^.altura;
end;

```

Figura 4.36 Función para calcular la altura de un nodo AVL

```

procedure insertar(x: tipo_elemento; var A: ÁRBOL_DE_BÚSQUEDA);
begin
  if A = nil then
    begin
      [Crear un árbol de un nodo]

      new(A);
      if A = nil then
        error_fatal('¡¡¡Memoria agotada!!!');
      else
        begin
          A^.elemento := x;
          A^.izq := nil;
          A^.der := nil;
          A^.altura := 0;
        end;
    end [if A = nil]
  else
    begin
      if x < A^.elemento then
        begin
          insertar(x, A^.izq);
          if altura(A^.izq) - altura(A^.der) = 2 then
            if x < A^.izq^.elemento then
              rotar_s_izq(A)
            else
              rotar_d_izq(A)
          else
            A^.altura := máx(altura(A^.izq), altura(A^.der)) + 1;
        end
      else
        begin
          [caso simétrico para el subárbol derecho]
        end;
      [Si no, x ya está en el árbol. No se hace nada]
    end; [if A <> nil]
  end; [insertar]

```

Figura 4.37 Inserción en un árbol AVL

Para los árboles de la figura 4.38, *rotar_s_izq* convierte el árbol de la izquierda en el de la derecha; *rotar_s_der* es simétrica. El código se muestra en la figura 4.39.

La última función que escribiremos efectuará la rotación doble ilustrada en la figura 4.40, cuyo código se muestra en la figura 4.41.

La eliminación en árboles AVL es un poco más compleja que la inserción. Es probable que la eliminación perezosa sea la mejor estrategia si las eliminaciones son relativamente poco frecuentes.

Figura 4.38

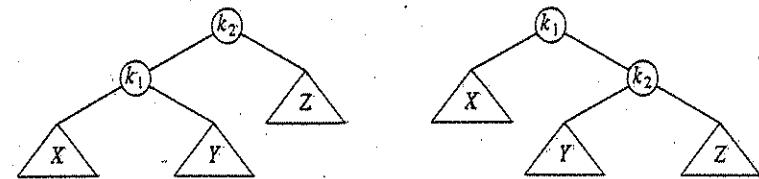


Figura 4.39 Rutina para efectuar la rotación simple

{Este procedimiento sólo puede ser llamado si k2 tiene un hijo izquierdo}
{Realiza una rotación entre un nodo (k2) y su hijo izquierdo}
{Actualiza las alturas}
{Después asigna la nueva raíz a k2}

```

procedure rotar_s_izq(var k2: ap_avl);
  var k1: ap_avl;

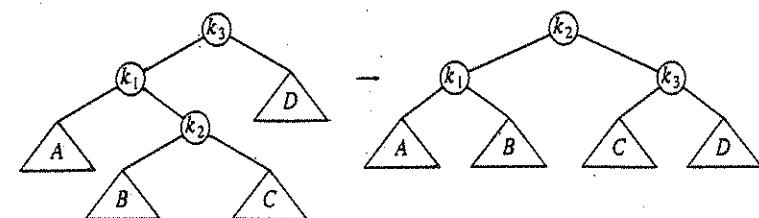
begin
  k1 := k2^.izq;
  k2^.izq := k1^.der;
  k1^.der := k2;

  k2^.altura := máx(altura(k2^.izq), altura(k2^.der)) + 1;
  k1^.altura := máx(altura(k1^.izq), k2^.altura) + 1;

  k2 := k1; {asigna una raíz nueva}
end;

```

Figura 4.40



{Esta función sólo se puede llamar si k3 tiene un hijo izquierdo}
 {y el hijo izquierdo de k3 tiene un hijo derecho}
 {Hace la rotación doble izquierda-derecha. Actualiza las alturas}

```
procedure rotar_d_izq(var k3: ap_avl);
begin
  rotar_s_der(k3^.izq); {rotar entre k1 y k2}
  rotar_s_izq(k3); {rotar entre k3 y k2}
end;
```

Figura 4.41 Rutina para efectuar la rotación doble

4.5. Árboles desplegados

Ahora describiremos una estructura de datos relativamente sencilla, denominada *árbol desplegado (splay tree)*,[†] que garantiza que cualesquiera m operaciones consecutivas sobre árboles tardan a lo más un tiempo $O(m \log n)$. Aunque esta garantía no evita la posibilidad de que cualquier operación *individual* pueda tardar un tiempo $O(n)$, y así la cota no será tan fuerte como una cota por operación de $O(\log n)$ para el peor caso, el efecto neto es el mismo: no hay secuencias de entrada malas. En general, cuando una secuencia de m operaciones tiene un tiempo de ejecución total en el peor caso de $O(m f(n))$, decimos que el tiempo de ejecución *amortizado* es de $O(f(n))$. Así, un árbol desplegado tiene costo amortizado $O(\log n)$ por operación. Con una secuencia larga de operaciones, algunas pueden tardar más, algunas menos.

Los árboles desplegados se basan en el hecho de que un tiempo de $O(n)$ por operación en el peor caso para árboles binarios de búsqueda no es malo, en tanto que esto ocurría con poca frecuencia. Cualquier acceso, aun si toma $O(n)$, todavía puede ser sumamente rápido. El problema con los árboles binarios de búsqueda es que es posible, y no extraño, que se den secuencias completas de malos accesos. Entonces el tiempo de ejecución acumulado se vuelve notorio. Una estructura de datos de árbol de búsqueda con tiempo $O(n)$ en el peor caso, pero con una garantía de a lo más $O(m \log n)$ para cualesquiera m operaciones consecutivas, es verdaderamente satisfactoria porque no hay malas secuencias.

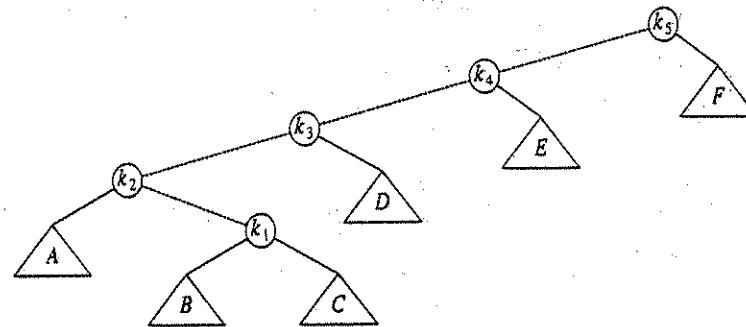
Si se permite que cualquier operación particular tenga como cota un tiempo $O(n)$ para el peor caso, y aún queremos una cota amortizada de $O(\log n)$, es obvio que siempre que se tenga acceso a un nodo, éste debe ser movido. Si no, una vez que encontramos un nodo profundo, podríamos seguir realizando operaciones *buscar* en él. Si el nodo no cambia de posición, y cada acceso cuesta $O(n)$, una secuencia de m accesos costará $O(m \cdot n)$.

[†] El término *splay tree* aún no tiene una traducción bien aceptada. Árbol desdoblado, extendido, biselado, achafanado son otras posibles traducciones. (N. del T.)

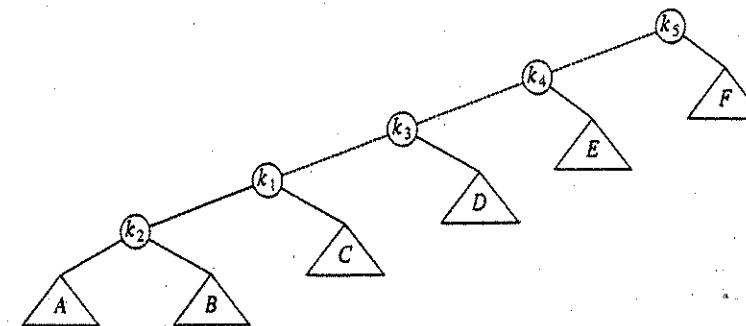
La idea básica de los árboles desplegados es que, después de tener acceso a un nodo, se lleva hasta la raíz mediante una serie de rotaciones de árbol AVL. Obsérvese que si un nodo es profundo hay muchos nodos en el camino que también son relativamente profundos, y con la reestructuración se pueden hacer más rápidos los accesos futuros sobre todos esos nodos. Así, si el nodo es indebidamente profundo, queremos que esta reestructuración tenga el efecto colateral de equilibrar el árbol (en algún grado). Además de dar una buena cota de tiempo en teoría, este método puede ser de utilidad práctica porque en muchas aplicaciones, cuando se tiene acceso a un nodo, es probable que muy pronto se tenga acceso a él otra vez. Algunos estudios han demostrado que esto ocurre con mucha más frecuencia de lo que se podría esperar. Los árboles desplegados tampoco requieren el mantenimiento de la información sobre la altura o el equilibrio, con lo que, en cierta medida, se ahorra espacio y se simplifica el código (en especial cuando se escriben implantaciones bien cuidadas).

4.5.1. Una idea sencilla (que no funciona)

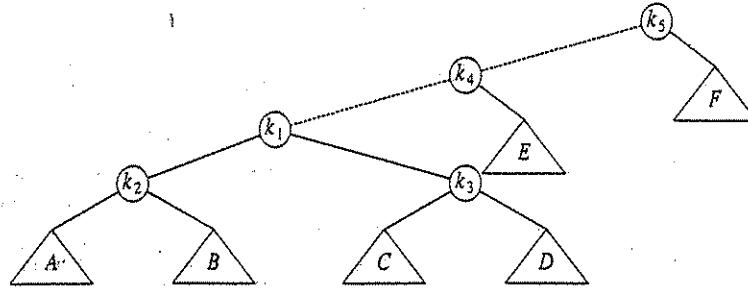
Una forma de efectuar la reestructuración antes descrita es realizar rotaciones simples, de abajo hacia arriba. Esto significa que todos los nodos encontrados en el camino de acceso se rotan con su padre. Por ejemplo, consideremos qué sucede después de un acceso (un *encuentra*) sobre k_1 en el árbol siguiente.



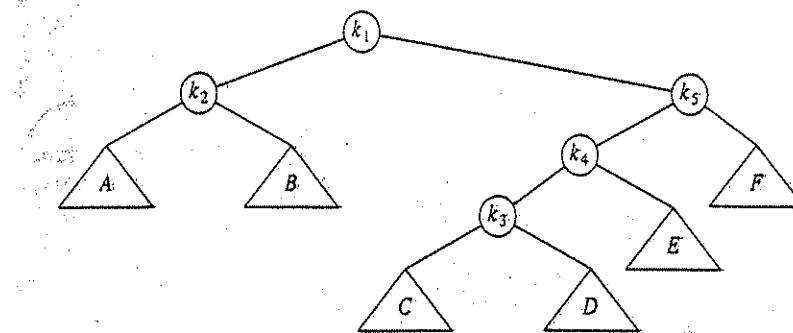
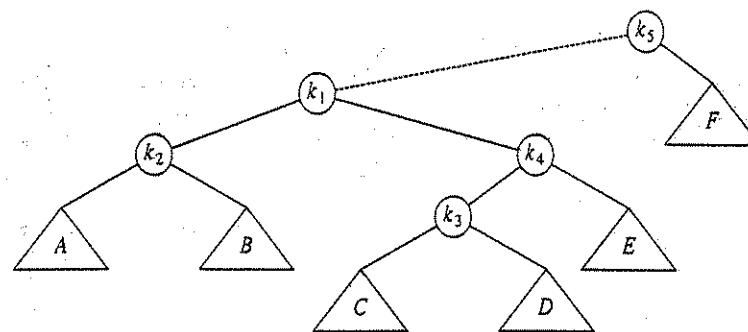
El camino de acceso está punteado. Primero, efectuaríamos una rotación simple entre k_1 y su padre, obteniendo el árbol siguiente.



Después, rotamos entre k_1 y k_3 , obteniendo el árbol siguiente.



Después de dos rotaciones más llegamos a la raíz.



Estas rotaciones tienen el efecto de empujar k_1 todo el camino hasta la raíz, así que los accesos futuros sobre k_1 son fáciles (de momento). Desafortunadamente, esto ha empujado otro nodo (k_3) al menos tan profundamente como estaba k_1 . Un acceso sobre ese nodo llevará otro nodo a esa profundidad, y así sucesivamente. Aunque esta estrategia hace más económicos los accesos futuros a k_1 , no mejora significati-

vamente la situación para los otros nodos del camino (original) de acceso. Es posible probar que con esta estrategia hay una secuencia de m operaciones que requieren un tiempo $\Omega(m \cdot n)$, por lo que esta idea no es lo bastante buena. La forma más sencilla de demostrar esto consiste en considerar el árbol formado por la inserción de las llaves $1, 2, 3, \dots, n$ en un árbol inicialmente vacío (resuelva este ejemplo). Esto da un árbol consistente en sólo hijos izquierdos, lo cual no es necesariamente malo, ya que el tiempo para construir este árbol es un total de $O(n)$. La parte mala es que tener acceso al nodo con la llave 1 lleva $n - 1$ unidades de tiempo. Después de completar las rotaciones, un acceso al nodo con la llave 2 lleva $n - 2$ unidades de tiempo. El total para tener acceso a las llaves en orden es $\sum_{i=1}^{n-1} i = \Omega(n^2)$. Después del acceso a las llaves, el árbol vuelve a su estado original, y podemos repetir la secuencia.

4.5.2. Despliegue

La estrategia de despliegue es semejante a la idea anterior sobre la rotación, excepto que es un poco más selectiva sobre cómo se efectúan las rotaciones. De nuevo, rotaremos de abajo hacia arriba en el camino de acceso. Sea x un nodo (no raíz) en el camino de acceso en el cual estamos rotando. Si el padre de x es la raíz del árbol, simplemente rotamos x y la raíz. Ésta es la última rotación a lo largo del camino de acceso. Si no es así, x tiene un padre (p) y un abuelo (a), y hay dos casos, más los simétricos, que considerar. El primer caso es el zig-zag (véase la figura 4.42). Aquí x es un hijo derecho y p es un hijo izquierdo (o viceversa). Si éste es el caso, efectuamos una rotación doble, exactamente como una rotación doble AVL. Si no, tenemos un caso zig-zig: x y p son ambos hijos izquierdos o ambos hijos derechos. En tal caso, transformamos el árbol de la izquierda en la figura 4.43 en el de la derecha.

Figura 4.42 Zig-zag

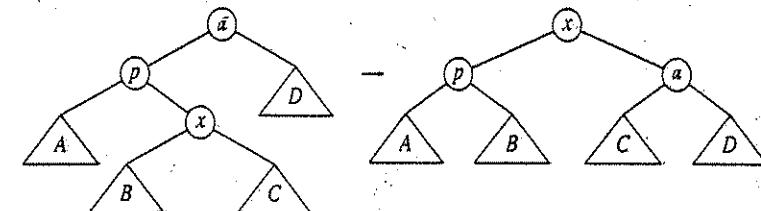
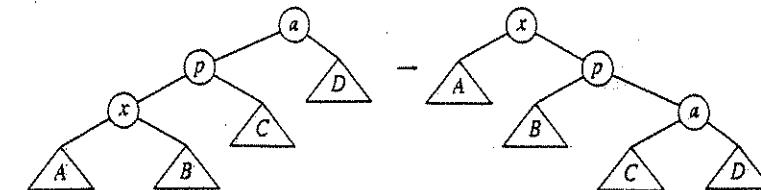
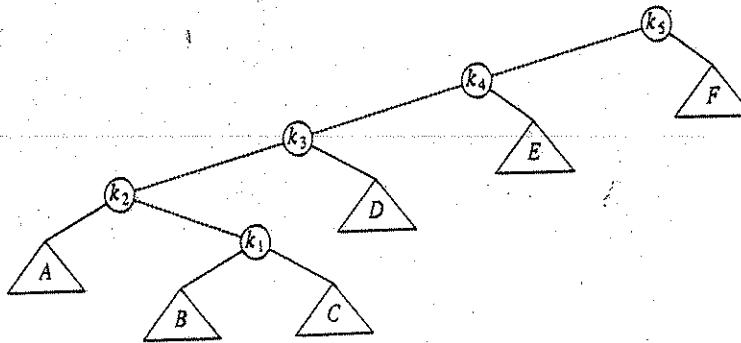


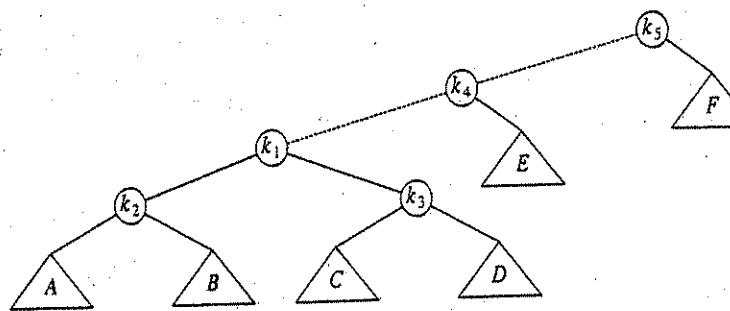
Figura 4.43 Zig-zig



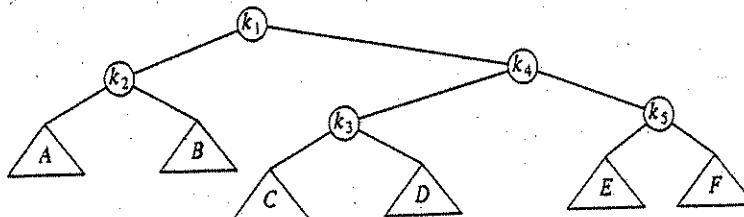
Por ejemplo, consideremos el árbol del último ejemplo, con un *buscar* sobre k_1 :



El primer paso del despliegue es en k_1 , claramente se trata de un zig-zag, así que efectuamos una rotación doble AVL estándar con k_1 , k_2 y k_3 . En seguida está el árbol resultante.



El siguiente paso del despliegue en k_1 es un zig-zig, así que se hace la rotación zig-zig sobre k_1 , k_4 y k_5 , obteniendo el árbol final.



Aunque es difícil verlo en ejemplos pequeños, el despliegue no sólo mueve a la raíz el nodo de acceso, sino que también tiene el efecto de reducir a la mitad, más o menos, la profundidad de la mayoría de los nodos en el camino de acceso (algunos nodos superficiales son bajados a lo más dos niveles).

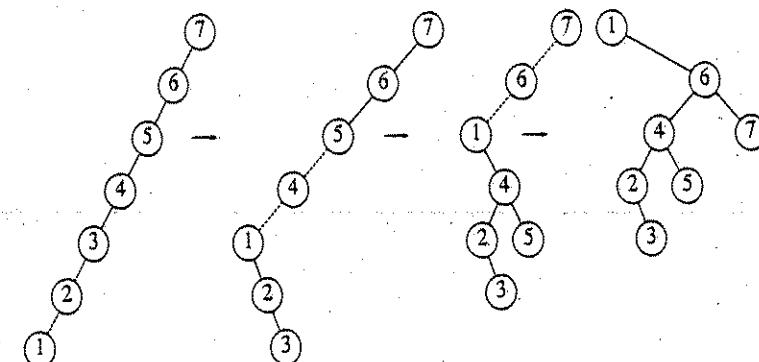
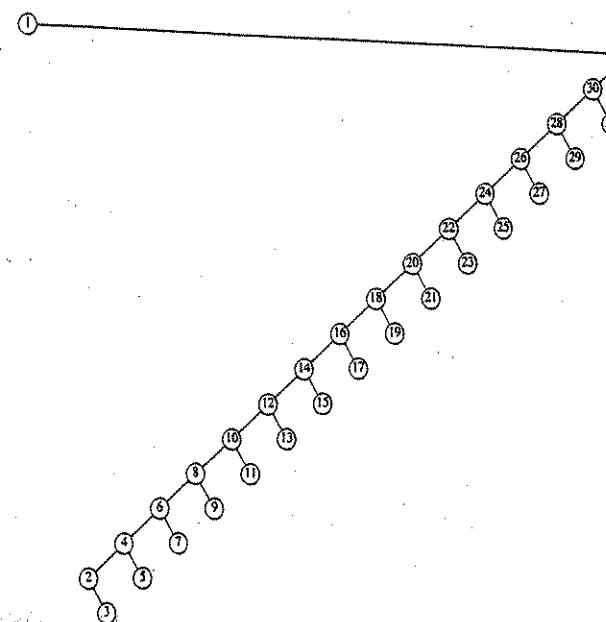


Figura 4.44 Resultado del despliegue en el nodo 1

Para ver la diferencia que provoca el despliegue sobre la rotación simple, consideremos otra vez el efecto de insertar las llaves $1, 2, 3, \dots, n$ en un árbol inicialmente vacío. Esto tarda un total de $O(n)$, como antes, y produce el mismo árbol que con las rotaciones simples. La figura 4.44 muestra el resultado del despliegue en el nodo con la llave 1. La diferencia es que después de un acceso al nodo con la llave 1, el cual tarda $n - 1$ unidades, el acceso al nodo con la llave 2 sólo llevará cerca de $n/2$ unidades en vez de $n - 2$; no hay nodos tan profundos como antes.

Figura 4.45 Resultado del despliegue sobre el nodo 1 en un árbol con todos los hijos izquierdos



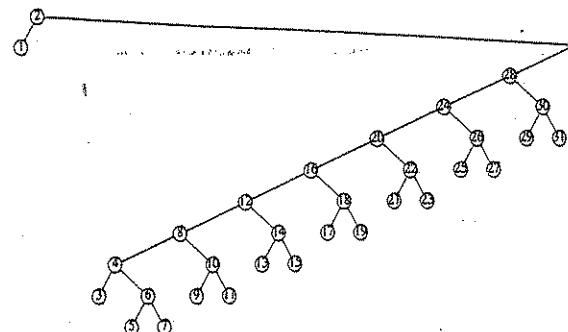


Figura 4.46 Resultado del despliegue del árbol anterior sobre el nodo 2

Un acceso sobre el nodo con la llave 2 llevará los nodos a $n/4$ de la raíz, y esto se repite hasta que la profundidad se acerca a $\log n$ (un ejemplo con $n = 7$ es demasiado pequeño para apreciar bien el efecto). Las figuras 4.45 a 4.53 muestran el resultado del acceso a las llaves 1 a 9 en un árbol de 32 nodos que originalmente contenía sólo hijos izquierdos. Así, no se obtiene con los árboles desplegados el mismo comportamiento que prevalece en la estrategia de la rotación simple. (De hecho, esto tiende a ser un muy buen caso. Una demostración más bien complicada enseña que, para este ejemplo, los n accesos tardan un tiempo total $O(n)$).

Esas figuras muestran la propiedad fundamental y crucial de los árboles desplegados. Cuando los caminos de acceso son largos llevando por ende a un tiempo de búsqueda mayor que lo normal, las rotaciones tienden a ser buenas para operaciones futuras. Cuando no es costoso hacer los accesos, las rotaciones no son tan buenas y pueden ser malas. El caso extremo es el árbol inicial formado por las inserciones. Todas las inserciones fueron operaciones constantes en tiempo, lo que da lugar a un árbol inicial deficiente. En ese momento, teníamos un árbol muy deficiente, pero estábamos ejecutando sin reorganizar y tuvimos la compensación de menos tiempo de ejecución total. Entonces una pareja de accesos realmente horribles preparaban un árbol casi equilibrado, pero el costo fue que tuvimos que restituir parte del tiempo que se había ahorrado. El teorema principal, que demostraremos en el capítulo 11, es que nunca se llegará más allá de $O(\log n)$ por operación: siempre estamos en lo previsto, aun cuando ocurran operaciones deficientes de vez en cuando.

Figura 4.47 Resultado del despliegue sobre el árbol anterior, en el nodo 3

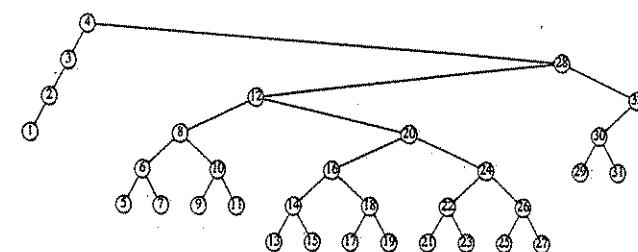
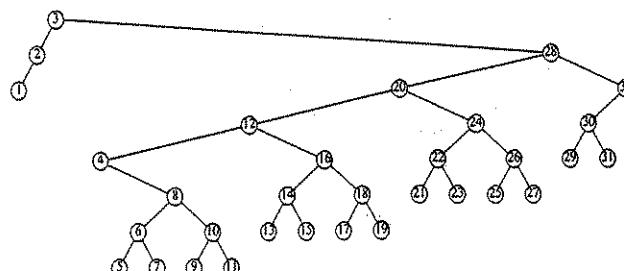


Figura 4.48 Resultado del despliegue sobre el árbol anterior, en el nodo 4

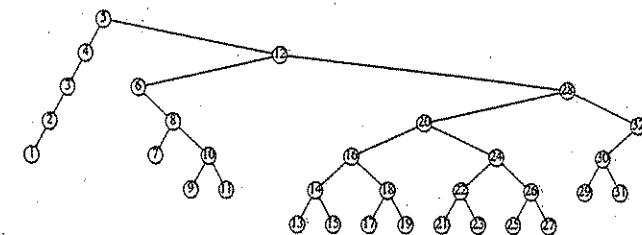


Figura 4.49 Resultado del despliegue sobre el árbol anterior, en el nodo 5

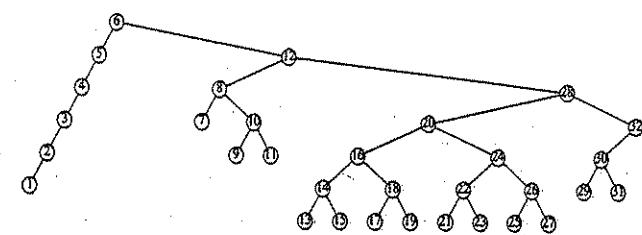


Figura 4.50 Resultado del despliegue sobre el árbol anterior, en el nodo 6

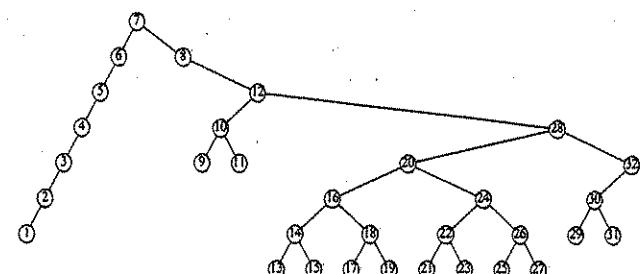


Figura 4.51 Resultado del despliegue sobre el árbol anterior, en el nodo 7

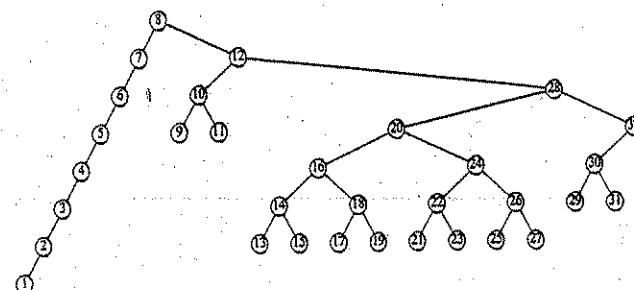


Figura 4.52 Resultado del despliegue sobre el árbol anterior, en el nodo 8

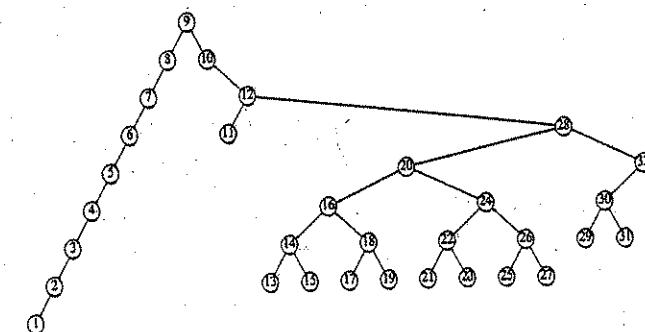


Figura 4.53 Resultado del despliegue sobre el árbol anterior, en el nodo 9

Como las rotaciones para árboles desplegados se efectúan ascendenteamente en parejas, una implantación recursiva no funciona (aunque se pueden hacer modificaciones a los pasos del despliegue para permitir una implantación recursiva). Los pares de nodos a considerar serán desconocidos hasta que se determine si la longitud del camino es par o impar. Así, los árboles desplegados se codifican no recursivamente y funcionan en dos pasadas. La primera pasada descende por el árbol y la segunda vuelve a ascender efectuando rotaciones. Para ello se tiene que guardar el camino. Esto puede hacerse usando una pila (que podría necesitar almacenar n apuntadores) o agregando un campo adicional al registro del nodo que apuntará a su padre. Ni uno ni otro método es particularmente difícil de implantar. Ofreceremos el código para la rutina de despliegue, con base en la suposición de que cada nodo almacena a su padre.

Es fácil comprender las declaraciones de tipos (figura 4.54). La rutina de despliegue (figura 4.55) toma como argumento el último nodo del camino de acceso y lo convierte en la nueva raíz. Las rutinas *rotación_simple* y *rotación_doble* escogen el tipo correcto de rotación. En la figura 4.56 se da el código para *rotación_simple*.

Las rutinas de rotación son semejantes a las AVL, excepto en que se deben mantener los apuntadores a los padres. Algunas rutinas de ejemplo están en las figuras que siguen. Como las rotaciones zig siempre hacen de x la nueva raíz,

```
type
  ap_despl = ^nodo_despl;
  nodo_despl = record
    elemento: tipo_elemento;
    izq : ap_despl;
    der : ap_despl;
    padre: ap_despl;
  end;
```

ÁRBOL_DE_BÚSQUEDA = ^nodo_despl;

Figura 4.54 Declaraciones de tipos para árboles desplegados

```
procedure desplegar(actual: ap_despl);
  var padre: ap_despl;

begin
  padre := actual^.padre;
  while padre <> nil do
    begin
      if padre^.padre = nil then
        rotación_simple(actual)
      else
        rotación_doble(actual);

      padre := actual^.padre;
    end;
end;
```

Figura 4.55 Rutina básica de despliegue

```
procedure rotación_simple(x: ap_despl);
begin
  if x^.padre^.izq = x then
    zig_izq(x)
  else
    zig_der(x);
end;
```

Figura 4.56 Rotación simple

sabemos que x no tendrá ningún padre después de la operación. El código para esto aparece en la figura 4.57.

Los Zig-zig y Zig-zag son semejantes. Escribiremos una rutina para efectuar el despliegue zig-zig cuando tanto x como p son hijos izquierdos. Una forma de hacerlo es escribir una rutina *rotación_simple* que incluya cambios a los apuntadores del

```

procedure zig_izq(x: ap_despl);
var p, B: ap_despl;

begin
  p := x^.padre;
  B := x^.der;

  x^.der := p;           {el nuevo hijo derecho de x es p}
  x^.padre := nil;

  if B <> nil then
    B^.padre := p;
    p^.izq := B;
    p^.padre := x;
end;

```

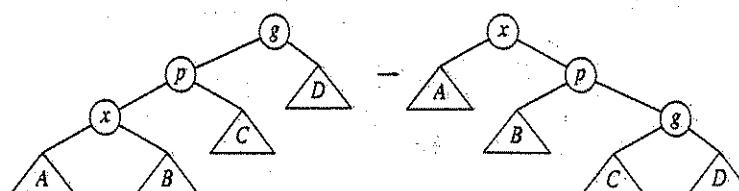
Figura 4.57 Rotación simple entre la raíz y su hijo izquierdo

padre, y después implantar las rotaciones complejas con dos operaciones simples. Esta es la forma en que codificamos las rutinas AVL. Se ha tomado un enfoque diferente en la figura 4.58 para mostrar la diversidad de estilos disponibles. Véase la figura 4.59. El lector debe intentar codificar los otros casos; será un ejercicio excelente de manipulación de apuntadores.

Podemos efectuar la eliminación por medio del acceso al nodo a eliminar. Esta pone al nodo como raíz. Si se elimina, obtenemos dos subárboles A_l y A_r (izquierdo y derecho). Si encontramos el elemento más grande en A_l (lo cual es fácil), este elemento se rotará a la raíz de A_l , y A_l tendrá una raíz sin hijo derecho. Se puede terminar la eliminación haciendo a A_r su hijo derecho.

El análisis de los árboles desplegados es difícil porque debe considerarse la estructura siempre cambiante del árbol. Por otro lado, es mucho más sencillo programar árboles desplegados que árboles AVL, ya que hay menos casos que considerar y no se requiere mantener la información de equilibrio. El código de árboles desplegados puede parecer complejo, pero como ya se indicó, se puede simplificar; probablemente es mucho más sencillo que una implantación no recursiva de árboles AVL. Algunos estudios empíricos indican que esto produce código más rápido en la práctica, aunque su validez aún está lejos de ser completa. Para concluir, debemos señalar que hay muchas variantes de árboles desplegados que pueden funcionar aún mejor en la práctica.

Figura 4.58



```

procedure zig_zig_izq(x: ap_despl);
var p, a, B, C, ba: ap_despl;

begin
  p := x^.padre;
  a := p^.padre;
  B := x^.der;
  C := p^.der;
  ba := a^.padre;

  x^.der := p;           {el nuevo hijo derecho de x es p}
  p^.padre := x;

  p^.der := a;           {el nuevo hijo derecho de p es a}
  a^.padre := p;

  if B <> nil then      {el nuevo hijo izquierdo de p es el subárbol B}
    B^.padre := p;
    p^.izq := B;

  if C <> nil then      {el nuevo hijo izquierdo de a es el subárbol C}
    C^.padre := a;
    a^.izq := C;

  x^.padre := ba;        {conecta al resto del árbol}
  if ba <> nil then
    if ba^.izq = a then
      ba^.izq := x
    else
      ba^.der := x
  end;

```

Figura 4.59 Rutina para efectuar un zig-zig cuando ambos hijos son izquierdos inicialmente

4.6. Recorridos de árboles (de nuevo)

Debido a la información de orden de un árbol binario de búsqueda, es sencillo listar todas las llaves en orden. El procedimiento recursivo de la figura 4.60 lo hace.

Convénzase de que este procedimiento funciona. Como hemos visto antes, cuando se aplica esta clase de rutinas a árboles se habla de recorridos en *orden simétrico*, o *enorden* (lo cual tiene sentido, ya que las llaves son listadas en orden). La estrategia general de un recorrido enorden consiste en procesar primero el subárbol izquierdo, después el nodo actual y por último el subárbol derecho. Lo interesante de este algoritmo, además de su simplicidad, es que el tiempo de ejecución total es $O(n)$. Esto se debe a que hay un trabajo constante en cada nodo del árbol. Cada nodo

```

procedure visualizar_arbol(A: ÁRBOL);
begin
  if A <> nil then begin
    visualizar_arbol(A^.izq);
    writeln(A^.elemento);
    visualizar_arbol(A^.der);
  end; {if}
end;

```

Figura 4.60 Rutina para visualizar un árbol binario de búsqueda en orden

se visita una vez, y el trabajo realizado en cada nodo es comparar con *nil*, preparar dos llamadas recursivas y ejecutar un *writeln*. Como hay un trabajo constante por nodo y son n nodos, el tiempo de ejecución es $O(n)$.

En ocasiones necesitamos procesar ambos subárboles primero y después el nodo. Por ejemplo, para calcular la altura de un nodo, se necesita conocer primero la altura de los dos subárboles. El código de la figura 4.61 calcula esto. Puesto que siempre es buena idea revisar los casos especiales (y es crucial cuando interviene la recursión), cabe indicar que la rutina declarará la altura de una hoja como cero, lo cual es correcto. Este orden general de recorrido, que ya hemos visto antes, se llama recorrido en *orden posterior* (*o postorden*). De nuevo, el tiempo de ejecución total es $O(n)$ porque se efectúa trabajo constante en cada nodo.

El tercer esquema popular de recorrido que hemos visto es el recorrido en *orden previo* (*o preorden*). Aquí, el nodo es procesado antes que los hijos. Esto puede ser útil, por ejemplo, si se desea etiquetar cada nodo con su profundidad.

La idea común a todas estas rutinas es que primero se maneja el caso *nil*, y después el resto. Obsérvese la ausencia de variables extrañas. Esas rutinas pasan sólo el árbol, y no declaran o pasan variables adicionales. Cuanto más compacto sea el código menos probable será que aparezca un error estúpido. Un cuarto recorrido, de menor uso (que hasta ahora no hemos contemplado), es el de *orden de nivel*. En un recorrido de orden de nivel, todos los nodos con profundidad d se procesan antes que cualquier nodo con profundidad $d + 1$. El recorrido por orden de nivel difiere

Figura 4.61 Rutina para calcular la altura de un árbol usando recorrido en orden posterior

```

function altura(A: ÁRBOL); integer;
begin
  if A = nil then
    altura := -1
  else
    altura := 1 + máx(altura(A^.izq), altura(A^.der));
end;

```

de los otros en que no se hace recursivamente; se usa una cola, en vez de la pila implícita en la recursión.

4.7. Árboles-B

Aunque todos los árboles de búsqueda vistos son binarios, hay un popular árbol de búsqueda que no es binario. Éste es el llamado *árbol-B*.

Un árbol-B de orden m es un árbol con las siguientes propiedades estructurales:

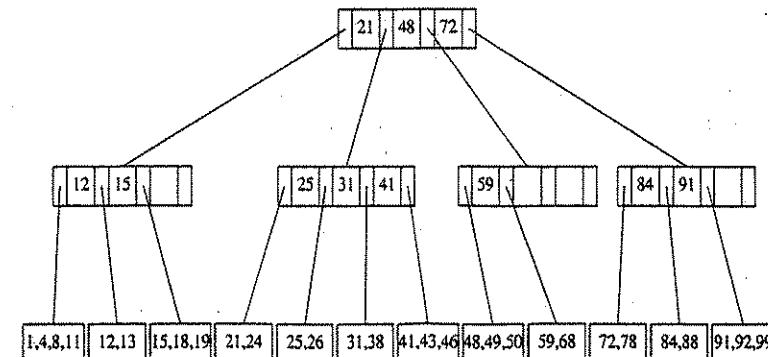
- La raíz es una hoja o tiene entre 2 y m hijos.
- Todos los nodos que no son hojas (excepto la raíz) tienen entre $\lceil m/2 \rceil$ y m hijos.
- Todas las hojas están a la misma profundidad.

Todos los datos se almacenan en las hojas. Contenidos en cada nodo interior están los apuntadores p_1, p_2, \dots, p_m a los hijos, y los valores k_1, k_2, \dots, k_{m-1} , que representan la llave más pequeña encontrada en los subárboles p_2, p_3, \dots, p_m , respectivamente. Por supuesto, algunos de estos apuntadores pueden ser *nil*, y la k_i correspondiente estaría entonces indefinida. Para cada nodo, todas las llaves en el subárbol p_i son menores que las llaves del subárbol p_{i+1} y así sucesivamente. Las hojas contienen todos los datos reales, que son las propias o bien apuntadores a registros que contienen las llaves. Supondremos lo primero para que nuestros ejemplos sean sencillos. Hay varias definiciones de árboles-B que cambian esta estructura en aspectos casi siempre menores, pero esta definición es una de las más conocidas. También insistiremos (por ahora) en que el número de llaves en una hoja está también entre $\lceil m/2 \rceil$ y m .

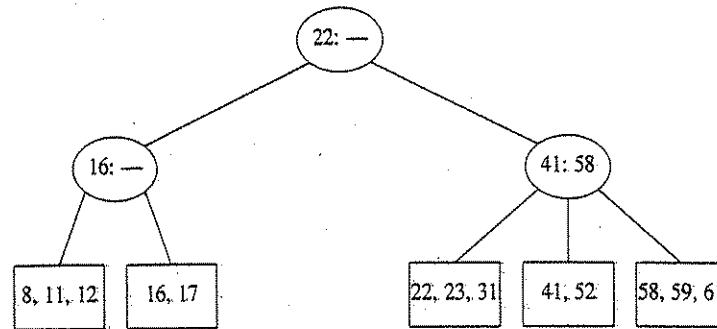
El árbol de la figura 4.62 es un ejemplo de árbol-B de orden 4.

Los árboles-B de orden 4 se conocen más popularmente como árboles 2-3-4, y un árbol de orden 3 se llama árbol 2-3. Describiremos la operación de los

Figura 4.62 Árbol-B de orden 4

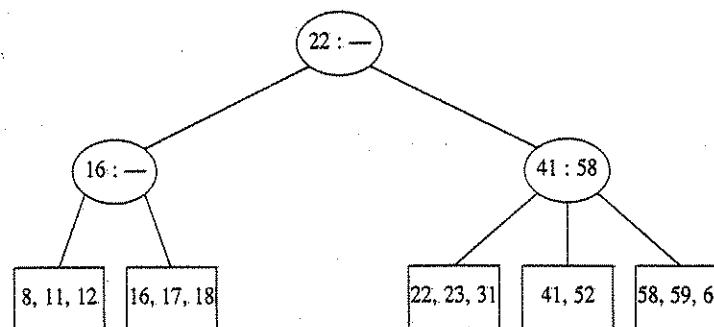


árboles-B en el caso especial de los árboles 2-3. El punto de inicio es el árbol 2-3 siguiente.

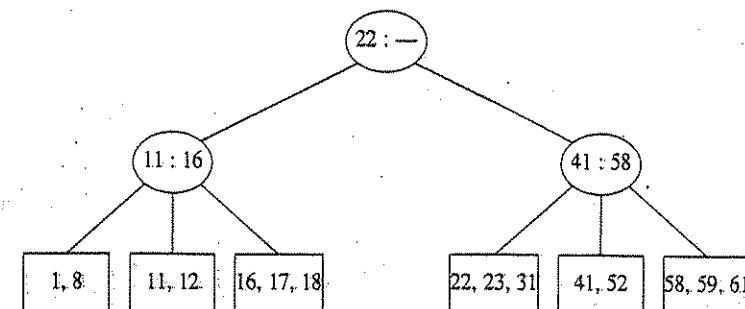


Hemos dibujado los nodos interiores (no hojas) como elipses, las cuales contienen las dos partes de datos de cada nodo. La raya que es la segunda parte de la información en cada nodo interno indica que el nodo tiene sólo dos hijos. Las hojas se dibujan en recuadros, los cuales contienen las llaves. Las llaves en las hojas están ordenadas. Para ejecutar un *buscar*, empezamos en la raíz y seleccionamos una de (a lo más) tres direcciones, dependiendo de la relación de la llave que se busca con los dos (posiblemente uno) valores almacenados en el nodo.

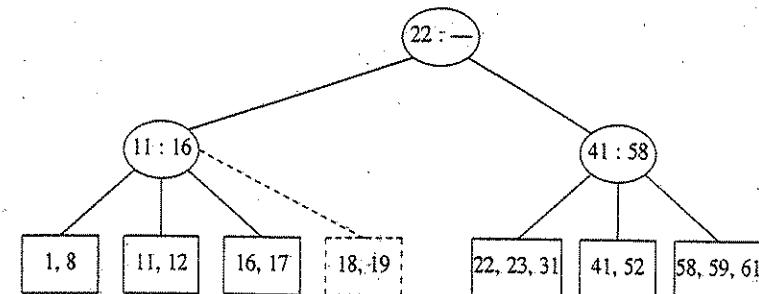
Para efectuar un *insertar* de una llave antes no vista, x , se sigue el camino igual que si fuera un *buscar*. Cuando se llega a una hoja, hemos encontrado el lugar correcto para poner x . Así, para insertar un nodo con la llave 18, basta con agregarlo a una hoja sin causar violación alguna a las propiedades del árbol 2-3. El resultado se muestra en la figura siguiente.



Desafortunadamente, como una hoja sólo puede contener dos o tres llaves, esto no siempre es posible. Si ahora intentamos insertar 1 en el árbol, encontramos que el nodo al que pertenece ya está lleno. Si colocamos esta llave nueva en este nodo obtendríamos un cuarto elemento, lo que no está permitido. Esto se puede resolver haciendo dos nodos de dos llaves cada uno y ajustando la información del padre.

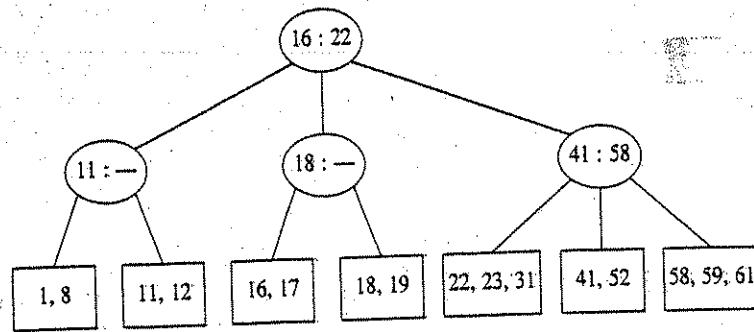


Desafortunadamente, esta idea no siempre funciona, como se puede observar si intentamos insertar 19 en el árbol actual. Si hacemos dos nodos de dos llaves cada uno, se obtiene el siguiente árbol.

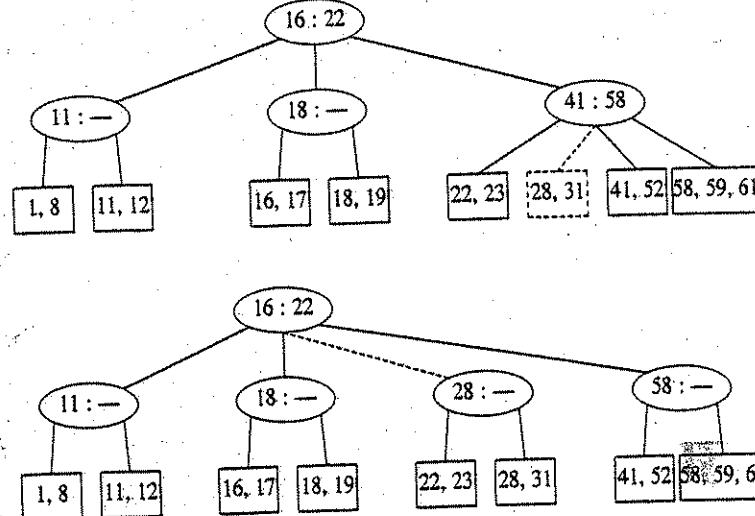


Este árbol tiene un nodo interno con cuatro hijos, pero sólo se permiten tres por nodo. La solución es sencilla. Simplemente partimos este nodo en dos nodos con dos hijos. Por supuesto, este nodo puede ser a su vez uno de tres hijos, y así partirllo crearía un problema para su padre (que tendría cuatro hijos), pero podemos seguir partiendo nodos, ascendiendo a la raíz hasta encontrar ya sea la raíz misma o un

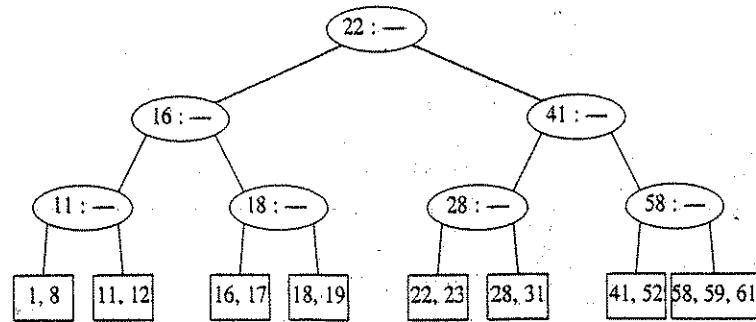
nodo con sólo dos hijos. En nuestro caso, sólo partimos el primer nodo interno que encontramos, obteniéndose el siguiente árbol.



Si ahora se inserta un elemento con la llave 28, creamos una hoja con cuatro hijos, a la cual partimos en dos hojas de dos hijos:



Esto crea un nodo interno con cuatro hijos, el cual se parte entonces en dos hijos. Lo que aquí hemos hecho es partir la raíz en dos nodos. Cuando se hace esto, tenemos un caso especial, que concluimos creando una raíz nueva. Ésta es la (única) forma en que un árbol 2-3 gana altura.



Cabe observar también que cuando se inserta una llave, los únicos cambios a nodos internos ocurren en el camino de acceso. Esos cambios se pueden hacer en un tiempo proporcional a la longitud del camino, pero se anticipa que hay unos pocos casos por manejar, y es fácil hacer esto mal.

Hay otras formas de manejar el caso donde un nodo se sobrecarga de hijos, pero el método que hemos descrito es probablemente el más simple. Cuando se intenta agregar una cuarta llave a una hoja, en vez de partir el nodo en dos, podemos intentar primero encontrar un hermano con sólo dos llaves. Por ejemplo, para insertar 70 en al árbol anterior, podríamos mover 58 a la hoja que contiene 41 y 52, colocar 70 con 59 y 61, y ajustar las entradas en los nodos internos. Esta estrategia también se puede aplicar a nodos internos y tiende a dejar más nodos llenos. El costo de esto radica en que las rutinas son ligeramente más complejas, pero se tiende a malgastar menos espacio.

Podemos efectuar la eliminación encontrando la llave y retirándola. Si ésta llave era una de las únicas dos en un nodo, la operación deja una sola llave. Esto se puede corregir combinando el nodo con un hermano. Si el hermano tiene tres llaves, le quitamos una para dejar ambos con dos llaves. Si el hermano sólo tiene dos llaves, combinamos ambos nodos en uno solo con tres llaves. El padre de este nodo pierde un hijo, así que podemos continuar con esta estrategia todo el camino hacia arriba. Si la raíz pierde su segundo hijo, la raíz también se elimina y el árbol adquiere una altura de un nivel menos. Al combinar nodos debemos recordar actualizar la información guardada en los nodos internos.

Con árboles-B generales de orden m , cuando se inserta una llave, la única dificultad surge cuando el nodo que va a aceptar la llave ya tiene m llaves. Esta llave le da al nodo $m+1$ llaves, que podemos partir en dos nodos con $\lceil(m+1)/2\rceil$ y $\lfloor(m+1)/2\rfloor$ llaves, respectivamente. Como esto da al padre un nodo extra, se debe revisar si este nodo puede ser aceptado por el padre y partir el padre cuando ya tiene m hijos. Repetimos esto hasta encontrar un padre con menos de m hijos. Si partimos la raíz, creamos una nueva, con dos hijos.

La profundidad de un árbol-B es a lo más de $\lceil \log_{m/2} n \rceil$. En cada nodo del camino, se realiza un trabajo $O(\log m)$ para determinar cuál rama seguir (usando una búsqueda binaria), pero un *insertar* o *eliminar* puede requerir trabajo $O(m)$ para recomponer toda la información en el nodo. El tiempo de ejecución en el peor caso

para cada operación *insertar* y *eliminar* es $O(m \log_m n) = O((m/\log m) \log n)$, pero un *buscar* tarda sólo $O(\log n)$. Se ha mostrado empíricamente que la mejor opción (legal) de m para consideraciones del tiempo de ejecución es $m = 3$ o $m = 4$; esto concuerda con las cotas anteriores, que demuestran que conforme m crece, los tiempos de inserción y eliminación se incrementan. Si sólo se atiende a la velocidad en memoria principal, los árboles-B de mayor orden, como los 5-9, no son una ventaja.

El uso real de los árboles-B se da en los sistemas de bases de datos, en donde los árboles se conservan en un disco físico y no en la memoria principal. El acceso al disco suele ser varios órdenes de magnitud más lento que cualquier operación en memoria principal. Si se usa un árbol B de orden m , el número de accesos a disco es $O(\log_m n)$. Aunque cada acceso a disco acarrea la sobrecarga de $O(\log m)$ para determinar la dirección a seguir, por lo regular el tiempo para efectuar este cálculo es mucho menor que el tiempo para leer un bloque de memoria y, por ello, puede considerarse que no tiene consecuencias (en tanto m se seleccione razonablemente). Aun si se realizan actualizaciones y se requiere un tiempo de computación de $O(m)$ en cada nodo, esto generalmente no es significativo. Entonces el valor de m se escoge como el valor más grande que todavía permite a un nodo interior caber en un solo bloque de disco, y por lo regular está en el rango $32 \leq m \leq 256$. El número máximo de elementos que se almacenan en una hoja se escoge de modo que, si la hoja está llena, ésta quepa en un bloque. Esto significa que un registro puede encontrarse siempre con muy pocos accesos a disco, ya que un árbol-B típico tendrá una profundidad de sólo 2 o 3, y la raíz (y posiblemente el primer nivel) pueden mantenerse en la memoria principal.

El análisis sugiere que un árbol-B estará lleno en un porcentaje de $\ln 2 = 69$. La mejor utilización del espacio se puede obtener si, en vez de partir siempre un nodo cuando el árbol obtiene su $m + 1$ -ésima entrada, la rutina busca un hermano que pueda recibir un hijo adicional. Los detalles pueden encontrarse en las referencias.

Resumen

Hemos visto usos de los árboles en sistemas operativos, diseño de compiladores y búsqueda. Los árboles de expresión son un pequeño ejemplo de una estructura más general conocida como *árbol sintáctico*, que es una estructura de datos central en el diseño de compiladores. Los árboles sintácticos no son binarios, pero son extensiones relativamente sencillas de los árboles de expresión (aunque los algoritmos para construirlos no son tan simples).

Los árboles de búsqueda son de gran importancia en el diseño de algoritmos. Permiten casi todas las operaciones útiles, y el costo medio logarítmico es muy bajo. Las implantaciones no recursivas de los árboles de búsqueda son algo más rápidas, pero las versiones recursivas son más directas, elegantes y fáciles de entender y depurar. El problema con los árboles de búsqueda es que su rendimiento depende decisivamente de que la entrada sea aleatoria. Si éste no es el caso, el tiempo de ejecución se incrementa significativamente, hasta el punto en que los árboles de búsqueda se vuelven más costosos que las listas enlazadas.

Vimos varias formas de tratar este problema. Los árboles AVL operan a base de insistir en que los subárboles izquierdo y derecho de todos los nodos difieran en altura a lo más en uno. Esto asegura que el árbol no sea demasiado profundo. Todas las operaciones que no cambian el árbol como hace la inserción, pueden usar el código estándar de árboles binarios de búsqueda. Las operaciones que cambian el árbol deben restaurarla, lo que puede ser algo complicado, en especial en el caso de la eliminación. Mostramos también cómo restaurar el árbol después de inserciones en un tiempo $O(\log n)$.

Así mismo examinamos los árboles desplegados. Los nodos en un árbol desplegado pueden ser de profundidad arbitraria, pero después de cada acceso el árbol se ajusta en una forma un tanto misteriosa. El efecto neto es que cualquier secuencia de m operaciones tarda un tiempo $O(m \log n)$, que es el mismo que tardaría un árbol equilibrado.

Los árboles-B son árboles equilibrados de m caminos (en oposición a los binarios o de 2 caminos), los cuales son adecuados para su uso en discos; un caso especial es el árbol 2-3, que es otro método común de implantación de árboles de búsqueda equilibrada.

En la práctica, el tiempo de ejecución de todos los esquemas de árboles equilibrados es peor (en un factor constante) que el simple árbol binario de búsqueda, pero en general esto es aceptable en vista de la protección dada contra la entrada en el peor caso, fácil de obtener.

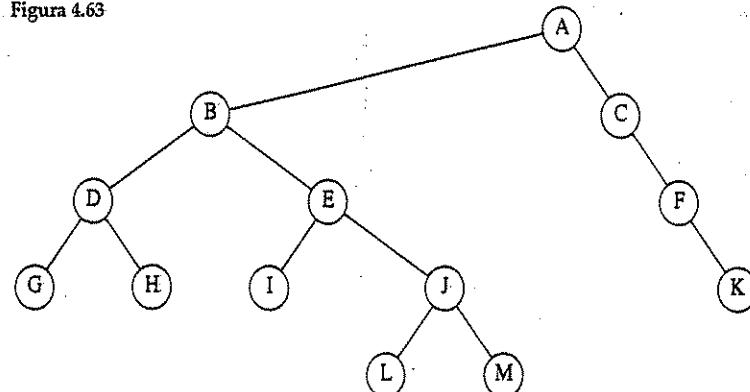
Una nota final: al insertar elementos en un árbol de búsqueda y efectuar después un recorrido en orden simétrico, obtenemos los elementos ordenados. Esto da un algoritmo de ordenación $O(n \log n)$, que es una cota del peor caso si se usa cualquier árbol de búsqueda elaborado. En el capítulo 7 veremos mejores formas, pero ninguna que tenga una cota de tiempo menor.

Ejercicios

Las preguntas 4.1 a la 4.3 se refieren al árbol de la figura 4.63

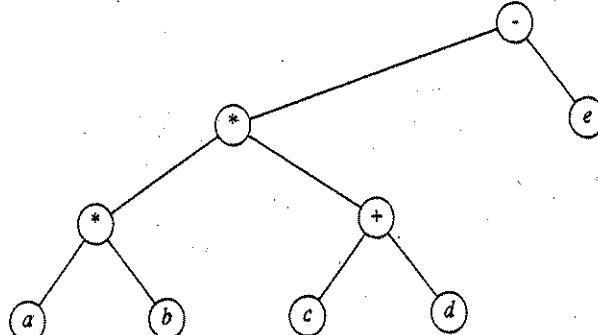
4.1 Para el árbol de la figura 4.63:

Figura 4.63



- a. ¿Qué nodo es la raíz?
- b. ¿Qué nodos son hojas?
- 4.2 Para cada nodo del árbol de la figura 4.63:
 - a. Indique el nombre del nodo padre.
 - b. Liste los hijos.
 - c. Liste los hermanos.
 - d. Calcule la profundidad.
 - e. Calcule la altura.
- 4.3 ¿Cuál es la profundidad del árbol de la figura 4.63?
- 4.4 Demuestre que en un árbol binario de n nodos hay $n + 1$ apuntadores *nil* que representan hijos.
- 4.5 Demuestre que el número máximo de nodos en un árbol binario de altura h es $2^{h+1} - 1$.
- 4.6 Un *nodo lleno* es un nodo con dos hijos. Demuestre que el número de nodos llenos más uno es igual al número de hojas de un árbol binario.
- 4.7 Suponga que un árbol binario tiene hojas l_1, l_2, \dots, l_m a la profundidad 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.
- 4.8 Proporcione las expresiones prefija, infix y posfix correspondientes al árbol de la figura 4.64.
- 4.9 a. Muestre el resultado de insertar 3, 1, 4, 6, 9, 2, 5, 7 en un árbol binario de búsqueda inicialmente vacío.
b. Muestre el resultado de eliminar la raíz.
- 4.10 Escriba rutinas para implantar las operaciones básicas sobre árboles binarios de búsqueda.

Figura 4.64 Árbol para el ejercicio 4.8

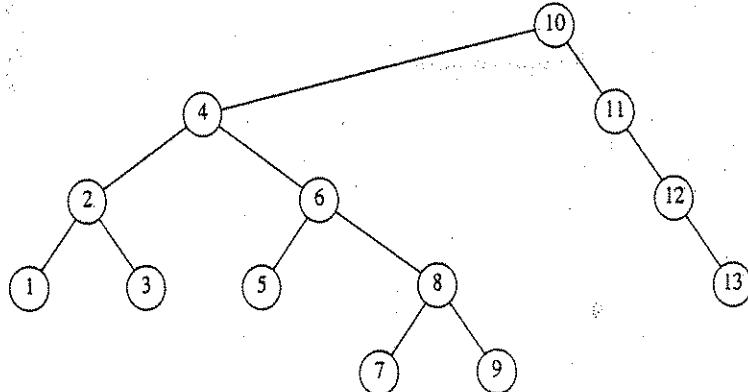


- 4.11 Los árboles binarios de búsqueda se pueden implantar con cursos, usando una estrategia semejante a la implantación de listas enlazadas con cursos. Escriba las rutinas básicas sobre árboles binarios de búsqueda usando la implantación con cursos.
- 4.12 Suponga que desea efectuar un experimento para verificar los problemas que se pueden ocasionar con pares aleatorios *insertar/eliminar*. Aquí se da una estrategia que no es perfectamente aleatoria, pero se acerca lo suficiente. Construya un árbol con n elementos insertando n elementos escogidos al azar en el intervalo de 1 a $m = cn$. Entonces efectúe n^2 pares de inserciones seguidas de eliminaciones. Suponga la existencia de una rutina, *ent_aleatorio(a, b)*, que devuelve un entero aleatorio uniforme entre a y b inclusive.
 - a. Explique cómo generar un entero aleatorio entre 1 y m que no esté ya en el árbol (para así poder efectuar inserción aleatoria). En términos de n y c , ¿cuál es el tiempo de ejecución de esta operación?
 - b. Explique cómo generar un entero aleatorio entre 1 y m que ya esté en el árbol (para así poder efectuar una eliminación aleatoria). ¿Cuál es el tiempo de ejecución de esta operación?
 - c. ¿Cuál es una buena elección de c ? ¿Por qué?
- 4.13 Escriba un programa para evaluar empíricamente las siguientes estrategias para eliminar nodos con dos hijos:
 - a. Reemplace con el nodo mayor, X , en A_l y elimine X recursivamente.
 - b. Alternativamente, reemplace con el nodo mayor en A_l y el nodo menor en A_r , y elimine recursivamente el nodo apropiado.
 - c. Reemplace con el nodo más grande en A_l o con el nodo más pequeño en A_r (eliminando recursivamente el nodo apropiado), haciendo una elección aleatoria.

¿Qué estrategia parece dar el mayor equilibrio? ¿Cuál toma el menor tiempo de UCP para procesar la secuencia completa?
- 4.14 ** Demuestre que la profundidad media de un árbol binario aleatorio (la del nodo más profundo) es $O(\log n)$.
- 4.15 a. Proporcione una expresión precisa para el número mínimo de nodos en un árbol AVL de altura h .
b. ¿Cuál es el número mínimo de nodos en un árbol AVL de altura 15?
- 4.16 Muestre el resultado de insertar 2, 1, 4, 5, 9, 3, 6, 7 en un árbol AVL vacío inicialmente.
- 4.17 * Las llaves 1, 2, ..., $2^k - 1$ se insertan en orden en un árbol AVL inicialmente vacío. Demuestre que el árbol resultante está perfectamente equilibrado.
- 4.18 Escriba los procedimientos pendientes para implantar las rotaciones AVL simples y dobles.
- 4.19 Escriba una función no recursiva para insertar en un árbol AVL.
- 4.20 * ¿Cómo se puede implantar la eliminación (no perezosa) en árboles AVL?

- 4.21 a. ¿Cuántos bits por nodo se requieren para almacenar la altura de un nodo en un árbol AVL con n nodos?
 b. ¿Cuál es el árbol AVL más pequeño que provoca un desbordamiento con un contador de altura de 8 bits?
- 4.22 Escriba las funciones para efectuar la rotación doble sin la ineficiencia de hacer dos rotaciones simples.
- 4.23 Muestre el resultado de tener acceso a las llaves 3, 9, 1 y 5 en ese orden en el árbol desplegado de la figura 4.65.
- 4.24 Muestre el resultado de eliminar el elemento con la llave 6 en el árbol desplegado resultante del ejercicio previo.
- 4.25 Los nodos 1 a $n = 1024$ forman un árbol desplegado de hijos izquierdos.
 a. ¿Cuál es la longitud (exacta) del camino interno del árbol?
 b. Calcule la longitud del camino interno en cada caso después de *buscar(1)*, *buscar(2)*, *buscar(3)*, *buscar(4)*, *buscar(5)*, *buscar(6)*.
 c. Si la secuencia de *buscar* sucesivos se continúa, ¿cuándo se minimiza la longitud del camino interno?
- 4.26 a. Demuestre que si se tiene acceso a todos los nodos de un árbol desplegado en orden secuencial, el árbol resultante consiste en una cadena de hijos izquierdos.
 **b. Demuestre que si se tiene acceso a todos los nodos de un árbol desplegado en orden secuencial, el tiempo de acceso total es $O(n)$, independientemente del árbol inicial.
- 4.27 Escriba un programa para analizar operaciones aleatorias sobre árboles desplegados. Cunte el número total de rotaciones realizadas sobre la secuencia. ¿Cómo es el tiempo de ejecución de los árboles AVL comparado con el de los árboles binarios de búsqueda no equilibrada?
- 4.28 Escriba funciones eficientes que sólo tomen un apuntador a un árbol binario, *A*, y calcule:

Figura 4.65



- a. el número de nodos en *A*;
 b. el número de hojas en *A*, y
 c. el número de nodos llenos en *A*.
- ¿Cuál es el tiempo de ejecución de las rutinas?
- 4.29 Escriba una función para generar un árbol binario de búsqueda aleatorio de n nodos con llaves distintas entre 1 y n . ¿Cuál es el tiempo de ejecución de su rutina?
- 4.30 Escriba una función para generar el árbol AVL de altura h con el menor número de nodos. ¿Cuál es el tiempo de ejecución de su función?
- 4.31 Escriba una función para generar un árbol binario de búsqueda perfectamente equilibrado, de altura h con llaves entre 1 y $2^{h+1} - 1$. ¿Cuál es el tiempo de ejecución de su función?
- 4.32 Escriba una función que tome como entrada un árbol binario de búsqueda, *A*, y dos llaves k_1 y k_2 , cuyo orden es $k_1 \leq k_2$, y visualice todos los elementos x en el árbol, tales que $k_1 \leq \text{llave}(x) \leq k_2$. No suponga ninguna información acerca del tipo de llaves, excepto que pueden ordenarse (consistentemente). Su programa debe ejecutarse en un tiempo medio $O(K + \log n)$, donde K es el número de llaves visualizadas. Acote el tiempo de ejecución del algoritmo.
- 4.33 Los árboles binarios más grandes de este capítulo fueron generados automáticamente por medio de un programa. Esto se hizo asignando una coordenada (x, y) a cada nodo de árbol, dibujando un círculo alrededor de cada coordenada (esto es difícil de ver en algunos dibujos), y conectando cada nodo con su padre. Suponga que tiene un árbol binario de búsqueda almacenado en memoria (tal vez generado por alguna de las rutinas anteriores) y que cada nodo tiene dos campos adicionales para almacenar las coordenadas.
- a. La coordenada x se puede calcular asignando el número de recorrido en orden simétrico. Escriba una rutina para hacer esto para cada nodo del árbol.
 b. La coordenada y puede calcularse usando el negativo de la profundidad del nodo. Escriba una rutina para hacer esto para cada nodo del árbol.
 c. En términos de alguna unidad imaginaria, ¿cuáles serán las dimensiones del dibujo? ¿Cómo se pueden ajustar las unidades para que la altura del árbol siempre sea aproximadamente dos terceras partes de su anchura?
 d. Demuestre que con este sistema no se cruza ninguna línea, y que para cualquier nodo, *X*, todos los elementos en el subárbol izquierdo de *X* aparecen a la izquierda de *X* y todos los elementos del subárbol derecho de *X* aparecen a la derecha de *X*.
- 4.34 Escriba un programa de propósito general para dibujar árboles que convierta un árbol en las siguientes instrucciones de ensamblado de grafos:
- a. *círculo(x, y)*
 b. *dibujar_línea(i, j)*

La primera instrucción dibuja un círculo en (x, y) , y la segunda conecta el i -ésimo círculo con el j -ésimo (los círculos se numeran en el orden en que se dibujan). Usted debe hacer este programa y definir algún tipo de lenguaje de entrada o bien hacer que esta función se pueda llamar desde cualquier programa. ¿Cuál es el tiempo de ejecución de la rutina?

- 4.35 Escriba una rutina para listar los nodos de un árbol binario en *orden de nivel*. Liste la raíz, después los nodos a la profundidad 1, seguidos de los nodos a la profundidad 2, etc. Hágalo en tiempo lineal. Demuestre su cota de tiempo.

- 4.36 a. Muestre el resultado de insertar las siguientes llaves en un árbol 2-3 inicialmente vacío: 3, 1, 4, 5, 9, 2, 6, 8, 7, 0.
 b. Muestre el resultado de eliminar 0 y después 9 del árbol 2-3 creado en la parte (a).

- 4.37 *a. Escriba una rutina para efectuar la inserción en un árbol-B.
 *b. Escriba una rutina para realizar eliminaciones en un árbol B. Al eliminar una llave, ¿es necesario actualizar la información en los nodos internos?
 *c. Modifique la rutina de inserción para que, si se intenta agregar en un nodo que ya tiene m entradas, se efectúe una búsqueda de un hermano con menos de m hijos antes de partir el nodo.

- 4.38 Un *árbol-B** de orden m es un árbol-B en el cual cada nodo interior tiene entre $2m/3$ y m hijos. Describa un método para efectuar la inserción en un árbol-B*.

- 4.39 Muestre cómo se representa el árbol de la figura 4.66 usando una implantación hijo/hermano.

- 4.40 Escriba un procedimiento para recorrer un árbol almacenado con enlaces hijo/hermano.

- 4.41 Dos árboles binarios son semejantes si ambos son vacíos o no vacíos y tienen subárboles izquierdo y derecho semejantes. Escriba una función para decidir si dos árboles binarios son semejantes. ¿Cuál es el tiempo de ejecución del programa?

Figura 4.66 Árbol del ejercicio 4.39

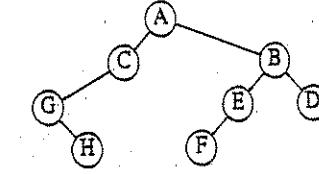
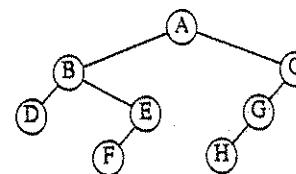
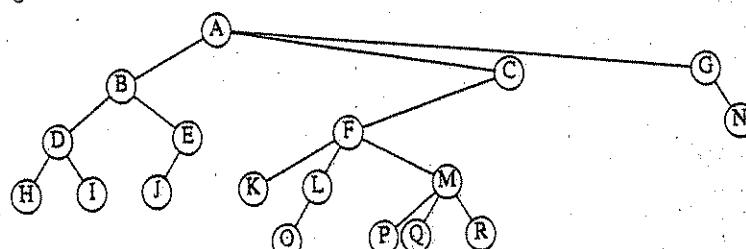


Figura 4.67 Dos árboles isomorfos

- 4.42 Dos árboles, A_1 y A_2 , son isomorfos si A_1 puede ser transformado en A_2 intercambiando los hijos izquierdo y derecho (de algunos) de los nodos en A_1 . Por ejemplo, los dos árboles de la figura 4.67 son isomorfos porque son iguales si se intercambian los hijos de A, B y G, pero no de los otros nodos.

- a. Proporcione un algoritmo con tiempo polinómico para decidir si dos árboles son isomorfos.
 *b. ¿Cuál es el tiempo de ejecución del programa (hay una solución lineal)?
 4.43 *a. Demuestre que mediante rotaciones AVL sencillas, cualquier árbol binario de búsqueda A_1 se puede transformar en otro árbol binario de búsqueda A_2 (con las mismas llaves).
 *b. Proporcione un algoritmo para efectuar esta transformación usando $O(n \log n)$ rotaciones en promedio.
 **c. Demuestre que esta transformación se puede hacer con $O(n)$ rotaciones, en el peor de los casos.

- 4.44 Suponga que quiere agregar la operación *buscar_k_ésimo* a su repertorio. La operación *buscar_k_ésimo(A, i)* devuelve el elemento del árbol A con la i -ésima llave más pequeña. Suponga que todos los elementos tienen llaves diferentes. Explique cómo modificar el árbol binario de búsqueda para permitir esta operación en $O(\log n)$, sin sacrificar las cotas de tiempo de cualquier otra operación.

- 4.45 Puesto que un árbol binario de búsqueda con n nodos tiene $n + 1$ apuntadores *nil*, se malgasta la mitad del espacio asignado a la información de apuntadores. Suponga que si un nodo tiene un hijo izquierdo *nil*, hacemos que su hijo izquierdo apunte a su predecesor en orden simétrico, y si un nodo tiene un hijo derecho *nil*, hacemos que su hijo derecho apunte a su sucesor en orden simétrico. Estos se denominan *árboles enhebrados* y los apuntadores adicionales se llaman *hebras*.
 a. ¿Cómo se distinguen las hebras de los apuntadores a hijo reales?
 b. Escriba rutinas para efectuar la inserción y la eliminación en un árbol enhebrado en la forma antes descrita.
 c. ¿Cuál es la ventaja de usar árboles enhebrados?

- 4.46 Un árbol binario de búsqueda presupone que la búsqueda se basa sólo en una llave por registro. Suponga que queremos ser capaces de efectuar la búsqueda basándonos en una de dos llaves, *llave₁* o *llave₂*.

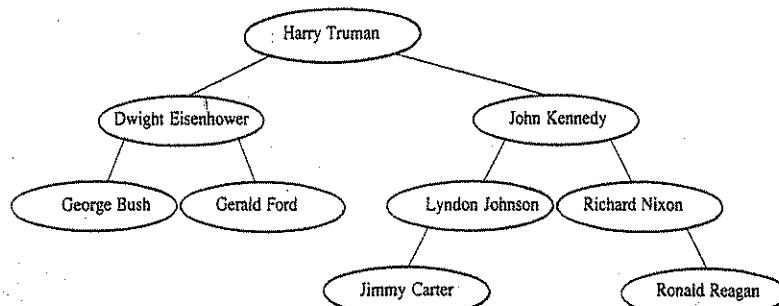


Figura 4.68 Árbol 2-d

- Un método consiste en construir dos árboles binarios de búsqueda separados. ¿Cuántos apuntadores adicionales se requieren?
- Un método alternativo es un árbol 2-d. Un árbol 2-d es semejante a un árbol binario de búsqueda, excepto que la ramificación en los niveles pares se hace con respecto a la $llave_1$, y la ramificación en los niveles impares se hace con respecto a la $llave_2$. La figura 4.68 muestra un árbol 2-d, con los nombres y apellidos como llaves de los ex presidentes de Estados Unidos después de la Segunda Guerra Mundial. Los nombres de los ex presidentes se insertaron en orden cronológico (Truman, Eisenhower, Kennedy, Johnson, Nixon, Ford, Carter, Reagan, Bush). Escriba una rutina para efectuar la inserción en un árbol 2-d.
- Escriba un procedimiento eficiente que visualice todos los registros del árbol que simultáneamente satisfagan las restricciones $menor_1 \leq llave_1 \leq mayor_1$ y $menor_2 \leq llave_2 \leq mayor_2$.
- Muestre cómo extender el árbol 2-d para manejar más de dos llaves de búsqueda. La estrategia resultante se conoce como árbol k-d.

Referencias

Más información sobre árboles binarios de búsqueda, y en particular sobre las propiedades matemáticas de los árboles se puede encontrar en los dos libros de Knuth [23] y [24].

Diversos artículos se ocupan de la ausencia de equilibrio ocasionada por los algoritmos sesgados de eliminación en los árboles binarios de búsqueda. Hibbard [20] propuso el algoritmo original de eliminación y estableció que una eliminación preserva la aleatoriedad de los árboles. Un análisis completo se ha realizado sólo con árboles de tres nodos [21] y de cuatro nodos [5]. El artículo de Eppinger [15] es uno de los primeros estudios empíricos sobre la no aleatoriedad, y los artículos de Culberson y Munro, [11] y [12], presentan algunas pruebas analíticas (pero no

una demostración completa para el caso general de inserciones y eliminaciones intercaladas).

Los árboles AVL fueron propuestos por Adelson-Velskii y Landis [1]. En [22] se presentan resultados de simulación de árboles AVL, y variantes en las cuales se permite que el desequilibrio máximo en la altura sea k para diferentes valores de k . Un algoritmo de eliminación para árboles AVL se puede encontrar en [24]. El análisis de la profundidad media de árboles AVL es aún incompleto, pero disponemos de algunos resultados en [25].

En [3] y [9] se estudian los árboles autoajustables como los de la sección 4.5.1. Los árboles desplegados se describen en [29].

Los árboles-B aparecieron por primera vez en [6]. La implantación descrita en el artículo original permite almacenar los datos en nodos internos así como en hojas. A la estructura de datos que hemos descrito en ocasiones se le llama árbol B*. Una revisión de los diferentes tipos de árbol-B se presenta en [10]. En [18] se proporciona un informe de resultados empíricos de los diferentes esquemas. El análisis de los árboles 2-3 y B se puede encontrar en [4], [14] y [33].

El ejercicio 4.14 es engañosamente difícil; una solución está en [16]. El ejercicio 4.26 proviene de [32]. La información sobre árboles-B*, descrita en el ejercicio 4.38, se puede encontrar en [13]. El ejercicio 4.42 fue proporcionado en [2]. Una solución al ejercicio 4.43 usando $2n - 6$ rotaciones se da en [30]. El uso de las hebras, como en el ejercicio 4.45, se propuso originalmente en [28]. Los árboles k-d se propusieron por primera vez en [7]. Su principal desventaja es que la eliminación y el equilibrio son difíciles de lograr. [8] presenta los árboles k-d y otros métodos usados en la búsqueda multidimensional.

Hay otros árboles de búsqueda equilibrados más conocidos que son los árboles rojo-negro [19] y los árboles equilibrados por peso [27]. En los libros [17], [26] y [31] se pueden encontrar más esquemas de equilibrio de árboles.

- G. M. Adelson-Velskii y E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Math. Doklady*, 3 (1962), págs. 1259-1263.
- A. V. Aho, J. E. Hopcroft y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Reading, MA, 1974.
- B. Allen y J. I. Munro, "Self Organizing Search Trees", *Journal of the ACM*, 25 (1978), págs. 526-535.
- R. A. Baeza-Yates, "Expected Behaviour of B^+ -trees under Random Insertions", *Acta Informática*, 26 (1989), págs. 439-471.
- R. A. Baeza-Yates, "A Trivial Algorithm Whose Analysis Isn't A Continuation", *BIT*, 29 (1989), págs. 88-113.
- R. Bayer y E. M. McCreight, "Organization and Maintenance of Large Ordered Indices", *Acta Informática*, 1 (1972), págs. 173-189.
- J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM*, 18 (1975), págs. 509-517.
- J. L. Bentley y J. H. Friedman, "Data Structures for Range Searching", *Computing Surveys*, 11 (1979), págs. 397-409.
- J. R. Bitner, "Heuristics that Dynamically Organize Data Structures", *SIAM Journal on Computing*, 8 (1979), págs. 82-110.
- D. Comer, "The Ubiquitous B-tree", *Computing Surveys* 11 (1979), págs. 121-137.

11. J. Culberson y J. I. Munro, "Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations", *Computer Journal*, 32 (1989), págs. 68–75.
12. J. Culberson y J. I. Munro, "Analysis of the Standard Deletion Algorithms' in Exact Fit Domain Binary Search Trees", *Algorithmica*, 5 (1990), págs. 295–311.
13. K. Culik, T. Ottman y D. Wood, "Dense Multiway Trees", *ACM Transactions on Database Systems*, 6 (1981), págs. 486–512.
14. B. Eisenbath, N. Ziviana, G.H. Gonnet, K. Melhorn y D. Wood, "The Theory of Fringe Analysis and its Applications to 2-3 Trees and B-trees", *Information and Control*, 55 (1982), págs. 125–174.
15. J. L. Eppinger, "An Empirical Study of Insertion and Deletion in Binary Search Trees", *Communications of the ACM*, 26 (1983), págs. 663–669.
16. P. Flajolet y A. Odlyzko, "The Average Height of Binary Trees and Other Simple Trees", *Journal of Computer and System Sciences*, 25 (1982), págs. 171–213.
17. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2a. ed., Addison-Wesley, Reading MA, 1991.
18. E. Gudes y S. Tsur, "Experiments with B-tree Reorganization", *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), págs. 200–206.
19. 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), págs. 8–21.
20. T. H. Hubbard, "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting", *Journal of the ACM*, 9 (1962), págs. 13–28.
21. A. T. Jonassen y D. E. Knuth, "A Trivial Algorithm Whose Analysis Isn't", *Journal of Computer and System Sciences*, 16 (1978), págs. 301–322.
22. P. L. Karlton, S. H. Fuller, R. E. Scroggs y E. B. Kaehler, "Performance of Height Balanced Trees", *Communications of the ACM*, 19 (1976), págs. 23–28.
23. D. E. Knuth, *The Art of Computer Programming: Volume 1: Fundamental Algorithms*, 2a. ed., Addison-Wesley, Reading, MA, 1973.
24. D. E. Knuth, *The Art of Computer Programming: Volume 3: Sorting and Searching*, 2a. imp., Addison-Wesley, Reading, MA, 1975.
25. K. Melhorn, "A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions", *SIAM Journal of Computing*, 11 (1982), págs. 748–760.
26. K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
27. J. Nievergelt y E. M. Reingold, "Binary Search Trees of Bounded Balance", *SIAM Journal on Computing*, 2 (1973), págs. 33–43.
28. A. J. Perlis y C. Thornton, "Symbol Manipulation in Threaded Lists", *Communications of the ACM*, 3 (1960), págs. 195–204.
29. D. D. Sleator y R. E. Tarjan, "Self-adjusting Binary Search Trees", *Journal of ACM*, 32 (1985), págs. 652–686.
30. D. D. Sleator, R. E. Tarjan y W. P. Thurston, "Rotation Distance, Triangulations, and Hyperbolic Geometry", *Journal of AMS* (1988), págs. 647–682.
31. H. F. Smith, *Data Structures-Form and Function*, Harcourt Brace Jovanovich, 1987.
32. R. E. Tarjan, "Sequential Access in Splay Trees Takes Linear Time", *Combinatorica*, 5 (1985), págs. 367–378.
33. A. C. Yao, "On Random 2-3 trees", *Acta Informatica*, 9 (1978), págs. 159–170.

Dispersión

En el capítulo 4 estudiamos el TDA árbol de búsqueda, que permite varias operaciones sobre un conjunto de elementos. En este capítulo se presenta el TDA *tabla de dispersión (hash table)*, con el que sólo se puede efectuar un subconjunto de las operaciones permitidas en los árboles binarios de búsqueda.

A la implantación de tablas de dispersión con frecuencia se le llama simplemente *dispersión (hashing)*. La dispersión es una técnica empleada para realizar inserciones, eliminaciones y búsquedas en un tiempo promedio constante. Aquí no son eficientes las operaciones sobre árboles que requieren cualquier información de ordenamiento entre datos. Por ello no se cuenta con operaciones como *buscar_mín*, *buscar_máx* ni la visualización de la tabla completa en orden, en un tiempo lineal.

La estructura de datos central de este capítulo es la *tabla de dispersión*. En él

- Examinaremos varios métodos de implantación de tablas de dispersión.
- Haremos una comparación de estos métodos.
- Mostraremos diversas aplicaciones de la dispersión.
- Compararemos las tablas de dispersión con los árboles binarios de búsqueda.

5.1. Idea general

La estructura de datos ideal para la tabla de dispersión es simplemente un arreglo de tamaño fijo que contiene las llaves. Por lo regular, la llave es una cadena con un valor asociado (por ejemplo, nóminas). Nos referiremos al tamaño de la tabla como *TAMAÑO_D*, con el entendimiento de que éste es parte de la estructura de datos de dispersión y no sólo alguna variable global flotante. Hay un convenio común que consiste en que la tabla se declare entre 0 y *TAMAÑO_D*-1; en breve veremos por qué.

Cada llave se hace corresponder con algún número en el intervalo entre 0 y *TAMAÑO_D*-1 y se coloca en la celda correcta. A la correspondencia se le denomina *función de dispersión*, la cual, idealmente, debe ser simple de calcular y debe asegurar

0	
1	
2	
3	Juan 25000
4	Felipe 31250
5	
6	David 27500
7	Maria 28200
8	
9	

Figura 5.1 Tabla de dispersión ideal

que dos llaves distintas cualesquiera caigan en celdas diferentes. Puesto que existe un número finito de celdas y una fuente prácticamente interminable de llaves, esto es, a todas luces, imposible, así que se busca una función de dispersión que distribuya homogéneamente las llaves entre las celdas. La figura 5.1 es representativa de una situación perfecta. En este ejemplo, Juan cae en 3, Felipe en 4, David en 6 y María en 7.

Esta es la idea básica de la dispersión. Los únicos problemas que restan consisten en elegir una función, decidir qué hacer cuando dos llaves caen en el mismo valor (a lo que llamamos *colisión*) y decidir el tamaño de la tabla.

5.2. Función de dispersión

Si las llaves de entrada son enteros, se acepta como una buena estrategia la función *llave mod TAMANO_D*, a menos que *llave* tenga algunas propiedades indeseables. En este caso, la elección de la función de dispersión amerita mayor cuidado. Por ejemplo, si el tamaño de la tabla es 10 y todas las llaves terminan en cero, es obvio que la función de dispersión estándar es una mala opción. Por razones que veremos más adelante y para evitar situaciones como la anterior, es buena idea asegurarse de que el tamaño de la tabla sea un número primo. Cuando las llaves de entrada son enteros aleatorios, esta función no sólo es muy simple de calcular sino que además distribuye las llaves con uniformidad.

Por lo regular, las llaves son cadenas de caracteres, en cuyo caso hay que escoger con cuidado la función de dispersión.

Una opción es sumar los valores ASCII de los caracteres de la cadena. En la figura 5.2 se declara el tipo *ÍNDICE*, que es devuelto por la función de dispersión. La rutina de la figura 5.3 implanta esta estrategia.

```
type
  ÍNDICE = 0..TAMAÑO_D-1;
```

Figura 5.2 Tipo devuelto por la función *dispersión*

La función de dispersión ilustrada en la figura 5.3 se implanta con facilidad y permite calcular una respuesta con rapidez. No obstante, la función no distribuye bien las llaves si el tamaño de la tabla es grande. Por ejemplo, supongamos que *TAMAÑO_D* = 10 007 (un número primo), y que todas las llaves tienen una longitud de ocho o menos caracteres. Puesto que *ord* devuelve un valor entero que siempre es 127 como máximo, la función de dispersión sólo adopta valores entre 0 y 1016, que es 127×8 . Por supuesto, ésta no es una distribución equitativa!

En la figura 5.4 se muestra otra función de dispersión, en la que se supone que *llave* (*key*) tiene al menos tres posiciones. Si *tamaño_llave* < 3, entonces la *llave* se rellena con blancos. 27 representa el número de letras del alfabeto inglés, más el blanco, y 729 es 27^2 . Esta función examina sólo los primeros tres caracteres, pero si son aleatorios y el tamaño de la tabla es 10 007, como antes, entonces esperaríamos una distribución razonablemente homogénea. Desafortunadamente, el inglés no es aleatorio. Aunque hay $26^3 = 17\ 576$ combinaciones posibles de tres caracteres (ignorando blancos), una revisión de un diccionario en línea razonablemente grande revela que el número de combinaciones diferentes es, en realidad, de sólo 2 851. Aun si ninguna de *estas* combinaciones participa en una colisión, sólo el 28% de la tabla puede ser aprovechada en la dispersión. Así, aunque la función se calcula fácilmente, tampoco es apropiada si la tabla de dispersión es razonablemente grande.

La figura 5.5 muestra un tercer intento de la función de dispersión. En ella intervienen todos los caracteres en la llave y en general se puede esperar una buena distribución (la función $\sum_{i=0}^{\text{tamaño_llave}-1} \text{llave}[\text{tamaño_llave} - i] \cdot 32^i$, y lleva el resultado al intervalo apropiado). El código calcula una función polinómica (de 32) con base en la regla de Horner. Por ejemplo, otra forma de calcular es $h_k = k_1 + 27k_2 + 27^2k_3$ con la fórmula $h_k = ((k_3) * 27 + k_2) * 27 + k_1$. La regla de Horner extiende esto a un polinomio de grado *n*.

Hemos usado 32 en vez de 27, porque la multiplicación por 32 no es realmente una multiplicación, sino el desplazamiento de cinco bits. Desafortunadamente, para cuidarse del desbordamiento, que puede ser un error, se efectúa una operación

Figura 5.3 Función de dispersión sencilla

```
function dispersión(llave: tipo_cadena; tamaño_llave: integer): ÍNDICE;
  var val_dispersión, j: integer;
begin
  (1)  val_dispersión := ord(llave[1]);
  (2)  for j := 2 to tamaño_llave do
  (3)    val_dispersión := val_dispersión + ord(llave[j]);
  (4)  dispersión := val_dispersión mod TAMAÑO_D;
end;
```

```

function dispersión(llave: tipo_cadena; tamaño_llave: integer): ÍNDICE;
begin
  dispersión := (ord(llave[1]) + 27 * ord(llave[2]) + 729 * ord(llave[3]))
    mod TAMANO_D;
end;

```

Figura 5.4 Otra posible función de dispersión — no muy buena

mod en cada iteración. Esto hace del cálculo de una función de dispersión un paso algo costoso (y con hacer *TAMANO_D* una potencia de dos obtendríamos una función de dispersión deficiente) y da un buen ejemplo de dónde el desbordamiento es bueno. Para lenguajes que permiten el desbordamiento, la línea 3 debería escribirse sin la operación *mod*; se aplicaría *mod* justo antes de devolver. Si disponemos de un *or* exclusivo, es más rápido que la suma y vale la pena probarlo en la función de dispersión.

La función de dispersión descrita en la figura 5.5 no es necesariamente la mejor con respecto a la distribución de la tabla, pero tiene el mérito de una extrema simplicidad (y de velocidad si se permiten desbordamientos). Si las llaves son muy grandes, la función de dispersión tardará mucho en el cálculo. Más aún, si se efectúa *mod* se realiza sólo una vez porque se permite el desbordamiento, los primeros caracteres saldrán, desplazados por la izquierda, de la respuesta final. En este caso se acostumbra no usar todos los caracteres. La longitud y las propiedades de las llaves influirían en la elección. Por ejemplo, las llaves podrían ser una dirección de domicilio completa. La función de dispersión puede incluir un par de caracteres del domicilio y tal vez un par de caracteres de la ciudad y del código postal. Algunos programadores implantan su función de dispersión usando sólo los caracteres de los espacios impares, con la idea de que el tiempo ahorrado en calcular la función de dispersión compensaría una función de distribución ligeramente menos homogénea.

El detalle principal de programación que queda es la resolución de colisiones. Si al insertar un elemento se dispersa en el mismo valor que un elemento ya insertado, tenemos una *colisión* y es necesario resolverla. Hay varios métodos para ello. Estudiaremos dos de los más simples: la dispersión abierta y la dispersión cerrada.[†]

Figura 5.5 Una función de dispersión correcta

```

function dispersión(llave: tipo_cadena; tamaño_llave: integer): ÍNDICE;
var      val_dispersión, j: integer;
begin
  begin
    [1]  val_dispersión := ord(llave[1]);
    [2]  for j:= 2 to tamaño_llave do
    [3]    val_dispersión := (val_dispersión* 32 + ord(llave[j]))
      mod TAMANO_D;
    [4]  dispersión := val_dispersión;
  end;

```

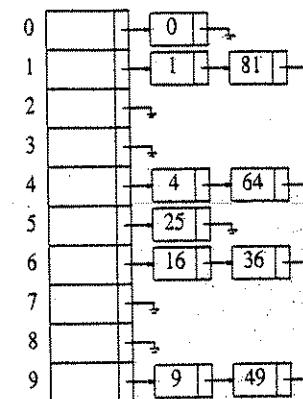


Figura 5.6 Tabla de dispersión abierta

5.3. Dispersión abierta (encadenamiento separado)

La primera estrategia, denominada *dispersión abierta* o *encadenamiento separado*, consiste en tener una lista de todos los elementos que se dispersan en el mismo valor. Por conveniencia, nuestras listas tienen cabeceras. Esto hace que la implantación de las listas sea igual que en el capítulo 3. Si hay poco espacio podría ser preferible evitar su uso. En esta sección suponemos que las llaves son los diez primeros cuadrados perfectos y que la función de dispersión es simplemente $dispersión(x) = x \bmod 10$. (El tamaño de la tabla no es primo, pero se usa así por simplicidad.) La figura 5.6 debe aclarar esto:

Para efectuar *buscar*, usamos la función de dispersión para determinar qué lista recorrer. Entonces recorremos esta lista en la forma usual, devolviendo la posición donde se encuentra el elemento. Para efectuar un *insertar*, recorremos la lista adecuada para revisar si el elemento ya está en la lista (si se espera que haya duplicados, lo normal es conservar un campo adicional que se incremente en el caso de una coincidencia). Si el elemento resulta ser nuevo, se inserta al frente o al final de la lista, lo que sea más fácil. Éste es un aspecto que se puede manejar con la mayor

Figura 5.7 Declaración de tipos para la tabla de dispersión abierta

```

type
  posición = ^nodo;
  nodo = record
    elemento: tipo_elemento;
    sig: posición;
  end;

```

TABLA_DE_DISPERSIÓN = arreglo [ÍNDICE] de posición;
(arreglo de cabeceras)

[†] También se les conoce como encadenamiento separado y direccionamiento abierto, respectivamente.

```

procedure iniciar_tabla(var D: TABLA_DE_DISPERSIÓN);
var i: integer;
begin
  for i := TAMAÑO_D - 1 downto 0 do
  begin
    new(D[i]);
    if D[i] = nil then
      error_fatal('¡¡Memoria agotada!!!')
    else
      D[i]^ .sig := nil;
  end;
end;

```

Figura 5.8 Rutina de iniciación de la tabla de dispersión abierta

facilidad en el momento de escribir el código. Algunas veces se insertan los elementos nuevos al frente, porque resulta conveniente y también porque ocurre con frecuencia que los elementos recién insertados son a los que con mayor probabilidad se tendrá acceso en el futuro cercano.

Las declaraciones de tipos, necesarias para implantar la dispersión abierta están en la figura 5.7.

Para iniciar la tabla de dispersión, asignamos las celdas cabecera y hacemos que todos sus apuntadores *sig* sean *nil*. El campo *elemento* de cada celda cabecera está indefinido. Esto se muestra en la figura 5.8. Aunque es obvio que la puesta de valores inicial debe efectuarse antes de que puedan hacerse las operaciones sobre la tabla de dispersión, es sorprendente cuán fácil es olvidar cosas tan triviales.

La llamada *buscar*(*llave*, *D*) devolverá un apuntador a la celda que contiene *llave*. El código para implantar esto se muestra en la figura 5.9. Observe que las líneas 2

Figura 5.9 Rutina *buscar* para una tabla de dispersión abierta

```

function buscar(llave: tipo_elemento;
               var D: TABLA_DE_DISPERSIÓN): posición;
label 999;
var p, lista: posición;

begin
  lista := D[dispersión(llave)];
  p := lista^.sig;
  while p < > nil do
    if p^.elemento = llave then
      goto 999
    else
      p := p^.sig;
  999: buscar := p;
end;

```

a la 7 son idénticas al código para efectuar un *buscar* que se da en el capítulo 3, lo cual permite usar aquí la implantación del TDA lista de ese capítulo.

La rutina de la figura 5.9 emplea el temido *goto* como una alternativa a una construcción *while not encontrado*. Esto es cosa de preferencia personal. Escoja el estilo que le guste. También pasamos la tabla de dispersión como *var* a fin de garantizar que el compilador de Pascal no trate de hacer una copia. Como hemos dicho varias veces, un compilador inteligente descubrirá que la rutina no intenta modificar la tabla y la copia no se necesita realmente. Hemos elegido el camino más seguro: no suponer un compilador inteligente.

En seguida viene la rutina de inserción: Si el elemento por insertar ya está en la lista, no hacemos nada; si no es así, lo colocamos al frente de la lista (véase la figura 5.10).[†] El elemento puede colocarse en cualquier parte de la lista: esto es lo más conveniente en nuestro caso. Observe que, en esencia, el código para insertar al frente de la lista es idéntico al código del capítulo 3 que implanta *meter* con listas enlazadas. De nuevo, si ya se han implantado con cuidado los TDA del capítulo 3, pueden usarse aquí.

La rutina de inserción de la figura 5.10 está implantada con un código algo deficiente porque calcula dos veces la función de dispersión. Hay que evitar siempre los cálculos redundantes, así que este código debe ser reescrito si resulta que las rutinas de dispersión contribuyen con una porción significativa al tiempo de ejecución del programa.

La rutina de eliminación es una implantación directa de la rutina de eliminación en una lista enlazada, así que no hablaremos de ella. Si el repertorio de rutinas de

Figura 5.10 Rutina *insertar* para una tabla de dispersión abierta

```

procedure insertar(llave: tipo_elemento; var D:TABLA_DE_DISPERSIÓN);
var pos, lista: posición;
celda_nueva: posición;

begin
  {1} pos := buscar(llave, D);
  {2} if pos = nil then {llave no encontrada}
  begin
    {3} new(celda_nueva);
    {4} if celda_nueva = nil then
      error_fatal('¡¡Memoria agotada!!!')
    else
      begin
        {5} lista := D[dispersión(llave)];
        {6} celda_nueva^.sig := lista^.sig;
        {7} celda_nueva^.elemento := llave;
        {8} lista^.sig := celda_nueva;
      end;
  end;
end;

```

[†] Como la tabla de la figura 5.6 se creó insertando al final de la lista, el código de la figura 5.10 producirá una tabla con las listas de la figura 5.6 en orden inverso.

dispersión no incluye las eliminaciones, probablemente es mejor no usar cabeceras, ya que su uso no ofrecería ninguna simplificación y malgastaría una cantidad considerable de espacio. Se deja esto como ejercicio también.

Además de las listas enlazadas, se podría usar cualquier esquema para resolver colisiones: un árbol binario de búsqueda o incluso otra tabla de dispersión funcionarían, pero esperamos que si la tabla es grande y la función de dispersión correcta, todas las listas deben ser cortas, así que no vale la pena intentar nada complicado.

Definimos el factor de carga, λ , de una tabla de dispersión como la razón del número de elementos en la tabla de dispersión al tamaño de la tabla. En el ejemplo anterior, $\lambda = 1.0$. La longitud media de una lista es λ . El esfuerzo requerido para efectuar una búsqueda es el tiempo constante que hace falta para evaluar la función de dispersión más el tiempo necesario para recorrer la lista.

En una búsqueda infructuosa, el número promedio de enlaces por recorrer es λ (excluyendo el enlace final *nil*). Una búsqueda con éxito requiere que se visiten casi $1 + (\lambda/2)$ enlaces, puesto que se garantiza que un enlace sea recorrido (pues la búsqueda es exitosa), y también esperamos ir hasta la mitad de la lista para encontrar el elemento correspondiente. Este análisis demuestra que el tamaño de la tabla no es realmente importante, pero el factor de carga sí lo es. La regla general de una dispersión abierta es hacer el tamaño de la tabla casi tan grande como el número de elementos esperados (en otras palabras, $\lambda \approx 1$). También es buena idea, como se mencionó antes, conservar primo el tamaño de la tabla para asegurar una buena distribución.

5.4. Dispersión cerrada (direcciónamiento abierto)

La dispersión abierta tiene la desventaja de que requiere apuntadores. Esto tiende a hacer un poco lento el algoritmo, debido al tiempo necesario para asignar celdas nuevas, y también requiere en esencia la implantación de una segunda estructura de datos. La *dispersión cerrada*, también conocida como *direcciónamiento abierto*, es una alternativa para la resolución de las colisiones con listas enlazadas. En un sistema de dispersión cerrada, si ocurre una colisión, se intenta buscar celdas alternativas hasta encontrar una vacía. Más formalmente, se busca en sucesión en las celdas $d_0(x), d_1(x), d_2(x), \dots$, donde $d_i(x) = (\text{dispersión}(x) + f(i)) \bmod \text{TAMAÑO_D}$, con $f(0) = 0$. La función f es la estrategia de resolución de las colisiones. Como todos los datos se meten en la tabla, se necesita una tabla más grande para la dispersión cerrada que para la abierta. En general, el factor de carga debe estar por debajo de $\lambda = 0.5$ para la dispersión cerrada. Ahora examinaremos tres estrategias comunes de resolución de colisiones.

5.4.1. Exploración lineal

En el sondeo o exploración lineal, f es una función lineal de i , por lo regular $f(i) = i$. Ésta equivale a recorrer las celdas en secuencia (con vuelta al principio) en busca de una celda vacía. La figura 5.11 muestra el resultado de insertar las llaves {89, 18, 49,

58, 69} en una tabla cerrada usando la misma función de dispersión que antes y la estrategia de resolución de colisiones, $f(i) = i$.

La primera colisión ocurre cuando se inserta el 49; éste se pone en el siguiente espacio disponible, a saber el punto que está abierto. El 58 entra en colisión con 18, 89 y 49 antes de encontrar una celda vacía a tres celdas de distancia. La colisión del 69 se maneja de una forma semejante. En tanto que la lista sea suficientemente grande, siempre se puede encontrar una celda vacía, pero ello puede tomar demasiado tiempo. Lo peor: aun si la tabla está relativamente vacía, es que se empiezan a formar bloques de celdas ocupadas. Este efecto, denominado *agrupamiento primario*, significa que cualquier llave que se disperse en el agrupamiento requerirá varios intentos para resolver la colisión, y después se agregará al agrupamiento.

Aunque no realizamos los cálculos aquí, se puede demostrar que el número esperado de intentos con la exploración lineal es más o menos $\frac{1}{2}(1 + 1/(1 - \lambda)^2)$ para inserciones y búsquedas no exitosas, y $\frac{1}{2}(1 + 1/(1 - \lambda))$ para búsquedas exitosas. Los cálculos son un tanto enredados. Es fácil ver en el código que las inserciones y las búsquedas no exitosas requieren el mismo número de pruebas. Un razonamiento rápido sugiere que, en promedio, las búsquedas exitosas deben tardar menos tiempo que las búsquedas no exitosas.

Las fórmulas correspondientes, si el agrupamiento no fuera un problema, son bastante fáciles de obtener. Supondremos una tabla muy grande y que cada exploración es independiente de las anteriores. Esas suposiciones se satisfacen con una estrategia de resolución de colisiones *aleatoria* y son razonables a menos que λ sea muy cercano a 1. Primero, obtenemos el número esperado de exploraciones en una búsqueda no satisfactoria. Éste sólo es el número esperado de intentos hasta encontrar una celda vacía. Ya que la fracción de celdas vacías es $1 - \lambda$, el número de

Figura 5.11 Tabla de dispersión cerrada con exploración lineal, después de cada inserción

	Tabla vacía	Después de 89	Después de 18	Después de 49	Después de 58	Después de 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

celdas en que esperamos indagar es $1/(1 - \lambda)$. El número de intentos de una búsqueda satisfactoria es igual al número de intentos requeridos cuando se insertó el elemento particular. Al insertar un elemento, se hace como resultado de una búsqueda no exitosa. Así, podemos usar el costo de una búsqueda no exitosa para calcular el costo medio de una búsqueda exitosa.

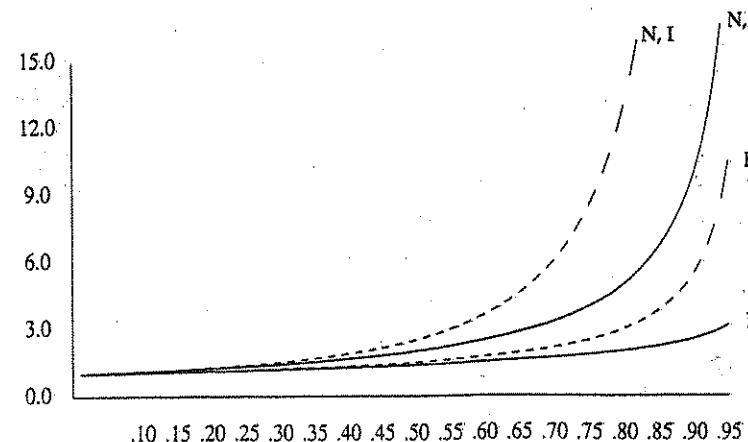
La advertencia que hay que hacer es que λ cambia de 0 a su valor actual, así que las primeras inserciones son menos costosas y deben bajar el promedio. Por ejemplo, en la tabla anterior, $\lambda = 0.5$, pero el costo de tener acceso a 18 se determina cuando se inserta el 18. En este punto, $\lambda = 0.2$. Como 18 se insertó en una tabla relativamente vacía, tener acceso a él debe ser más fácil que a un elemento insertado más recientemente, como el 69. Podemos calcular el promedio mediante una integral para encontrar el valor medio del tiempo de inserción, y obtenemos

$$I(\lambda) = \frac{1}{\lambda} \int_0^\lambda \frac{1}{1-x} dx = \frac{1}{\lambda} \ln \frac{1}{1-\lambda}$$

Claro está, estas fórmulas son mejores que las correspondientes a la exploración lineal. El agrupamiento no sólo es un problema teórico sino que de hecho ocurre en implantaciones reales. La figura 5.12 compara el rendimiento de la exploración lineal (curvas punteadas) con el que se esperaría de una resolución de colisiones más aleatoria. Las búsquedas exitosas se indican con una E, y las búsquedas no exitosas e inserciones se indican con N e I, respectivamente.

Si $\lambda = 0.75$, la fórmula anterior indica que en una exploración lineal se esperan 8.5 intentos por cada inserción. Si $\lambda = 0.9$, entonces se esperan 50 intentos, lo cual es poco razonable. Esto se compara con 4 y 10 intentos para los factores de carga respectivos si el agrupamiento no fuera un problema. En las fórmulas se ve que la exploración lineal puede ser una mala idea si se espera que más de la mitad

Figura 5.12 Gráfico del número de intentos por cada factor de carga de la exploración lineal (punteado) y una estrategia aleatoria. E = búsqueda exitosa, N = búsqueda no exitosa e I = inserción



de la tabla esté ocupada. Si $\lambda = 0.5$, no obstante, sólo se requieren 2.5 intentos en promedio para la inserción y sólo 1.5, en promedio, para una búsqueda exitosa.

5.4.2. Exploración cuadrática

La exploración (o sondeo) cuadrática es un método de resolución de colisiones que elimina el problema del agrupamiento primario que padece la exploración lineal. El sondeo cuadrático es lo que usted esperaría: la función de colisiones es cuadrática. La elección común es $f(i) = i^2$. La figura 5.13 muestra la tabla de dispersión cerrada resultante con esta función de colisiones sobre la misma entrada del ejemplo de la exploración lineal.

Cuando 49 entra en colisión con 89, la siguiente posición a probar está una celda más allá. Esta celda está vacía, así que 49 entra allí. Despues 58 entra en colisión en la posición 8. Luego se intenta con la celda siguiente, pero está ocupada y ocurre otra colisión. Una celda vacante se encuentra en la siguiente celda intentada, la cual está $2^2 = 4$ más allá. Entonces 58 se coloca en la celda 2. Lo mismo ocurre con 69.

En la exploración lineal es mala idea dejar que la tabla de dispersión esté casi llena, porque se degrada el rendimiento. Para el sondeo cuadrático, la situación es aún más drástica. No hay garantía de encontrar una celda vacía una vez que la tabla se llena a más de la mitad, o aun antes, si es que el tamaño de la tabla no es primo. Esto se debe a que a lo más la mitad de la tabla se puede usar como posiciones alternativas al resolver las colisiones. En efecto, a continuación demostramos que si la tabla está medio vacía y su tamaño es primo, podemos contar siempre con la seguridad de poder insertar un elemento nuevo.

Figura 5.13 Tabla de dispersión cerrada con exploración cuadrática, después de cada inserción

	Tabla vacía	Después de 89	Después de 18	Después de 49	Después de 58	Después de 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9	89	89	89	89	89	89

TEOREMA 5.1.

Si se usa la exploración cuadrática, y el tamaño de la tabla es primo, entonces siempre se puede insertar un elemento nuevo si la tabla está, al menos, medio vacía.

DEMOSTRACIÓN:

Sea el tamaño de la tabla, $TAMAÑO_D$, un primo (impar) mayor que 3. Mostremos que las primeras $\lfloor TAMAÑO_D/2 \rfloor$ posiciones alternativas son todas diferentes. Dos de esas posiciones son $d(x) + i^2 \pmod{TAMAÑO_D}$ y $d(x) + j^2 \pmod{TAMAÑO_D}$, donde $0 < i, j \leq \lfloor TAMAÑO_D/2 \rfloor$. Supongamos, por vía de contradicción, que esas posiciones son las mismas, pero $i \neq j$. Entonces

$$d(x) + i^2 = d(x) + j^2 \pmod{TAMAÑO_D}$$

$$i^2 = j^2 \pmod{TAMAÑO_D}$$

$$i^2 - j^2 = 0 \pmod{TAMAÑO_D}$$

$$(i-j)(i+j) = 0 \pmod{TAMAÑO_D}$$

Puesto que $TAMAÑO_D$ es primo, se infiere $(i-j)$ o $(i+j)$ es igual a 0 ($\pmod{TAMAÑO_D}$). Como i y j son distintas, la primera opción no es posible. Puesto que $0 < i, j < \lfloor TAMAÑO_D/2 \rfloor$, la segunda opción también es imposible. Así, las primeras $\lfloor TAMAÑO_D/2 \rfloor$ posiciones alternativas son distintas. Como el elemento por insertar también se puede colocar en la celda en la cual se dispersa (si no hay colisión), cualquier elemento tiene $\lceil TAMAÑO_D/2 \rceil$ posiciones en las cuales entrar. Si se toman a lo más $\lfloor TAMAÑO_D/2 \rfloor$ posiciones, siempre se puede encontrar un espacio vacío.

Si la tabla está más que medio llena en uno, la inserción puede fallar (aunque ello es extremadamente improbable). Por tanto, es importante tener esto en mente. También es crucial que la tabla sea de tamaño primo.^{*} Si no es así, el número de

Figura 5.14 Declaración de tipos para tablas de dispersión cerrada

```
type
  clase_de_entrada = (legítima, vacía, eliminada);
  entrada_de_dispersión = record
    elemento: tipo_elemento;
    info: clase_de_entrada;
  end;
  posición = INDICE;
  TABLA_DE_DISPERSIÓN = array [INDICE] of entrada_de_dispersión;
```

* Si el tamaño de la tabla es un primo de la forma $4k+3$, y se usa la estrategia cuadrática de resolución de colisiones $f(i) = \pm i^2$, entonces se puede intentar en toda la tabla. El costo es una rutina ligeramente más compleja.

```
procedure iniciar_tabla(var D: TABLA_DE_DISPERSIÓN);
  var i: integer;

begin
  for i := TAMAÑO_D - 1 downto 0 do
    D[i].info := vacía;
end;
```

Figura 5.15 Rutina para iniciar una tabla de dispersión cerrada

posiciones alternativas se puede reducir notablemente. Por ejemplo, si el tamaño de la tabla fuera 16, las únicas posiciones alternativas pueden estar a distancias 1, 4 o 9.

La eliminación estándar no se puede realizar en una tabla de dispersión cerrada, porque la celda pudo haber causado una colisión en el pasado. Por ejemplo, si retiramos 89, prácticamente todos los *buscar* siguientes fallarán. Así, las tablas de dispersión cerrada requieren eliminación perezosa, aunque en este caso no haya realmente "pereza" implicada.

Las declaraciones de tipos para implantar la dispersión cerrada están en la figura 5.14. La puesta a valores iniciales de la tabla (figura 5.15) se efectúa poniendo el campo *info* a vacío.

Como con la dispersión abierta, *buscar* (*llave*, *D*) devolverá la posición de *llave* en la tabla de dispersión. Si no está presente *llave*, *buscar* devolverá la última celda. Esta celda es donde *llave* estaría insertada si se necesitara. Más aún, como está

Figura 5.16 Rutina *buscar* para la dispersión cerrada con exploración cuadrática

```
function buscar(llave: tipo_elemento; var D: TABLA_DE_DISPERSIÓN): posición;
  label 999;
  var pos_actual, i: posición;

begin
  {1} i := 0;
  pos_actual := dispersión(llave);
  {2} while D[pos_actual].elemento < > llave do
  {3}   if D[pos_actual].info = vacía then
  {4}     goto 999 {llave no encontrada}
  {5}   else
  {6}     begin {resolución cuadrática — buscar otra celda}
  {7}       i := i + 1;
  {8}       pos_actual := pos_actual + 2 * i - 1;
  {9}       if pos_actual >= TAMAÑO_D then
  {10}        pos_actual := pos_actual - TAMAÑO_D;
  end;
  999: buscar := pos_actual;
end;
```

marcada como vacía, es fácil decir que *buscar* falló. Suponemos por conveniencia que la tabla de dispersión es al menos el doble de grande que el número de elementos en la tabla, de modo que la resolución cuadrática siempre funcionará. De otra forma necesitaríamos comprobar el valor de i antes de la línea [6]. En la implantación de la figura 5.16, los elementos que están marcados como eliminados cuentan como si estuvieran en la tabla. Esto puede causar problemas, porque la tabla se llena prematuramente. Estudiaremos esta situación dentro de poco.

Las líneas [6] a [9] representan la forma rápida de hacer la resolución cuadrática. De la definición de la función de resolución cuadrática, $f(i) = f(i-1) + 2i - 1$, se infiere que la siguiente celda a intentar se puede determinar con una multiplicación por dos (en realidad, un desplazamiento de un bit) y un decremento. Si la nueva posición se sale del arreglo, se le puede poner de nuevo en intervalo restándole *TAMAÑO_D*. Este método es más rápido que el obvio porque evita la multiplicación y la división que parecen requerirse. Usar el nombre de variable i no es lo mejor; sólo lo empleamos para ser consistentes con el texto.

La rutina final es la inserción. Como en la dispersión abierta, no hacemos nada si *llave* ya está presente. Es una modificación sencilla hacer algo más. Si no es así, la colocamos en el espacio indicado por la rutina *buscar*. El código se muestra en la figura 5.17.

Aunque la exploración cuadrática elimina el agrupamiento primario, los elementos que se dispersan a la misma posición probarán en las mismas celdas alternas. Esto se denomina *agrupamiento secundario*. Éste es un pequeño defecto teórico. Los resultados de la simulación indican que, en general, se produce menos de medio intento adicional por búsqueda. La siguiente técnica elimina esto, pero lo hace a costa de multiplicaciones y divisiones adicionales.

5.4.3. Dispersion doble

El último método de resolución de colisiones que examinaremos es la *dispersión doble*. Para ésta, la elección común es $f(i) = i \cdot h_2(x)$. Esta fórmula dice que aplicamos una segunda función de dispersión a x y probamos a distancias $h_2(x)$, $2h_2(x)$, ..., y así sucesivamente. Una mala elección de $h_2(x)$ sería desastrosa. Por ejemplo, la

Figura 5.17 Rutina *insertar* para tablas de dispersión cerrada con exploración cuadrática

```
procedure insertar (llave: tipo_elemento; var D: TABLA_DE_DISPERSIÓN);
  var pos: posición;
begin
  pos := buscar(llave, D);
  if D[pos].info <> legítima then [bueno para insertar]
    begin
      D[pos].info := legítima;
      D[pos].elemento := llave;
    end;
end;
```

5.4. DISPERSIÓN CERRADA (DIRECCIONAMIENTO ABIERTO)

	Tabla vacía	Después de 89	Después de 18	Después de 49	Después de 58	Después de 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9	89	89	89	89	89	89

Figura 5.18 Tabla de dispersión cerrada con dispersión doble, después de cada inserción

elección obvia $h_2(x) = x \bmod 9$ podría no ayudar si se insertara 99 en la entrada de los ejemplos previos. Así, la función nunca debe evaluarse a cero. También es importante asegurarse de que todas las celdas pueden ser intentadas (esto no es posible en el ejemplo siguiente, porque el tamaño de la tabla no es primo). Una función como $h_2(x) = R - (x \bmod R)$, con R un primo menor que *TAMAÑO_D*, funcionará bien. Si escogemos $R = 7$, entonces la figura 5.18 muestra los resultados de insertar las mismas llaves de antes.

La primera colisión ocurre cuando se inserta 49. $h_2(49) = 7 - 0 = 7$, así que 49 se inserta en la posición 6. $h_2(58) = 7 - 2 = 5$, así que 58 se inserta en la localidad 3. Por último, 69 entra en colisión y se inserta a una distancia $h_2(69) = 7 - 6 = 1$. Si intentamos insertar 60 en la posición 0, tendríamos una colisión. Puesto que $h_2(60) = 7 - 4 = 3$, intentamos en las posiciones 3, 6, 9 y después 2 hasta encontrar un espacio vacío. En general es posible encontrar algún caso malo, pero aquí no hay demasiados.

Como dijimos antes, el tamaño de nuestra tabla de dispersión de ejemplo no es primo. Se ha hecho así porque conviene para el cálculo de la función de dispersión, pero vale la pena ver por qué hay que asegurarse de que el tamaño de la tabla sea primo cuando se usa la dispersión doble. Si intentamos insertar 23 en la tabla, entraría en colisión con 58. Puesto que $h_2(23) = 7 - 2 = 5$, y el tamaño de la tabla es 10, tenemos en esencia sólo una posición alternativa, y ya está ocupada. Así, si el tamaño de la tabla no es primo, es posible quedarse antes de tiempo sin posiciones alternativas. No obstante, si se implanta correctamente la dispersión doble, las simulaciones implican que el número esperado de intentos es al menos el mismo que en la estrategia aleatoria de resolución de colisiones.

Esto hace interesante la dispersión doble desde un punto de vista teórico. La exploración cuadrática, sin embargo, no requiere el uso de una segunda función de dispersión y así es, probablemente más simple y rápida en la práctica.

5.5. Redispersión

Si la tabla se llena demasiado, el tiempo de ejecución de las operaciones empezará a prolongarse mucho y los *insertar* pueden fallar en la dispersión cerrada con resolución cuadrática. Esto puede ocurrir si hay demasiadas eliminaciones intercaladas con las inserciones. Una solución, entonces, es construir otra tabla de cerca del doble de tamaño (con una nueva función de dispersión asociada) y recorrer toda la tabla de dispersión original, calculando el nuevo valor de dispersión de cada elemento (no eliminado) e insertándolo en la tabla nueva.

Como ejemplo, supongamos que los elementos 13, 15, 28 y 6 se insertan en una tabla de dispersión cerrada de tamaño 7. La función de dispersión es $h(x) = x \bmod 7$. Supongamos que se usa la exploración lineal para resolver colisiones. La tabla de dispersión resultante aparece en la figura 5.19.

Si se inserta 23 en la tabla, la tabla resultante de la figura 5.20 estará llena por encima del 70%. Como la tabla está tan llena se crea una nueva. El tamaño de esta tabla es 17, pues éste es el primer primo cerca del doble del tamaño de la tabla anterior. La nueva función de dispersión es entonces $h(x) = x \bmod 17$. Se recorre la

Figura 5.19 Tabla de dispersión cerrada con exploración lineal con la entrada 13, 15, 6, 24

0	6
1	15
2	
3	24
4	
5	
6	13

Figura 5.20 Tabla de dispersión cerrada con exploración lineal después de insertar el 23

0	6
1	15
2	23
3	24
4	
5	
6	13

tabla anterior, y los elementos 6, 15, 23, 24 y 13 se insertan en la tabla nueva. La tabla resultante aparece en la figura 5.21.

La operación completa se denomina *redispersión*. Es obvio que esta operación es muy costosa: el tiempo de ejecución es $O(n)$, pues hay n elementos por redispersar y la tabla es de tamaño cercano a $2n$, pero esto en realidad no es del todo malo, porque ocurre con poca frecuencia. En particular, deben haber ocurrido $n/2$ *insertar* antes de la última redispersión, así que en esencia ésta agrega un costo constante a cada inserción.[†] Si esta estructura de datos es parte del programa, el efecto no es perceptible. Por otro lado, si la dispersión se efectúa como parte de un sistema interactivo, el usuario desafortunado, cuya inserción ocasiona la redispersión, podría ver una fuerte caída en el rendimiento.

La redispersión se puede implantar en varias formas con exploración cuadrática. Una alternativa es redispersar tan pronto como se llene la tabla a la mitad. El otro extremo es redispersar sólo cuando falla una inserción. Una tercera estrategia intermedia es redispersar cuando la tabla alcanza un cierto factor de carga. Ya que el rendimiento se degrada conforme el factor de carga crece, la tercera estrategia implantada con un buen corte podría ser la mejor.

La redispersión libera al programador de ocuparse del tamaño de la tabla y es importante porque las tablas de dispersión no se pueden hacer arbitrariamente grandes en programas complejos. En los ejercicios se pide al lector investigar el uso de la redispersión en conjunción con la eliminación perezosa. La redispersión se puede usar también en otras estructuras de datos. Por ejemplo, si la estructura de

Figura 5.21 Tabla de dispersión cerrada después de la redispersión

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

[†] Esta es la razón por la cual la tabla se hace del doble del tamaño de la tabla anterior.

datos cola del capítulo 3 se llenara, podríamos declarar un arreglo del doble del tamaño y copiar todo en él, liberando el original.

El pequeño problema con la redispersión es que mientras es fácil hacerla en algunos lenguajes como Ada y C, es difícil en lenguajes como Pascal, el cual requiere que el tamaño del arreglo se declare en tiempo de compilación.

5.6. Dispersion extensible

El último tema de este capítulo se ocupa del caso en que la cantidad de datos es demasiado grande para caber en memoria principal. Como vimos en el capítulo 4, la consideración principal es el número de accesos a disco requeridos para recuperar los datos.

Igual que antes, suponemos que en cualquier punto tenemos n registros para almacenar; el valor de n cambia con el tiempo. Además, a lo más m registros caben en un bloque de disco. Usaremos $m = 4$ en esta sección.

Tanto si se usa la dispersión abierta como la cerrada, el problema principal es que las colisiones pueden ocasionar que varios bloques se examinen durante un *buscar*, aun para tablas de dispersión bien distribuidas. Además, cuando las tablas se llenan demasiado, se debe realizar un paso de redispersión extremadamente costoso, que requiere $O(n)$ accesos a disco.

Una alternativa brillante, conocida como dispersión extensible, permite realizar un *buscar* en dos accesos a disco. Las inserciones también requieren pocos accesos a disco.

Recordemos del capítulo 4, que un árbol-B tiene profundidad $O(\log_{m/2} n)$. Conforme se incrementa m , la profundidad del árbol-B disminuye. En teoría podríamos escoger m tan grande que la profundidad del árbol-B sea 1. Entonces cualquier *buscar* después del primero podría requerir un acceso a disco, ya que, supuestamente, el nodo raíz podría estar en memoria principal. El problema con esta estrategia es que el factor de ramificación es tan alto que requeriría un procesamiento considerable para determinar en qué hoja se encuentra el dato. Si el tiempo para efectuar este paso se puede reducir, tendríamos un esquema práctico. Ésta es exactamente la estrategia usada en la dispersión extensible.

Supongamos, por el momento, que nuestros datos consisten en varios enteros de seis bits. La figura 5.22 muestra un esquema de dispersión extensible para estos datos. La raíz del "árbol" contiene cuatro apuntadores determinados por los primeros dos bits de los datos. Cada hoja tiene hasta $m = 4$ elementos. Ocurre que en cada hoja los primeros dos bits son idénticos; esto se indica en el número entre paréntesis. Más formalmente, D representará el número de bits usados en la raíz, la cual a veces se llama *directorio*. El número de entradas en el directorio es 2^D . d_h es el número de bits de inicio que todos los elementos de alguna hoja h tienen en común. d_h dependerá de la hoja particular, y $d_h \leq D$.

Suponemos que queremos insertar la llave 100100. Esta iría en la tercera hoja, pero como la tercera hoja ya está llena, no queda espacio. Así, partimos esta hoja en otras dos, las cuales ahora están determinadas por los primeros tres bits. Esto

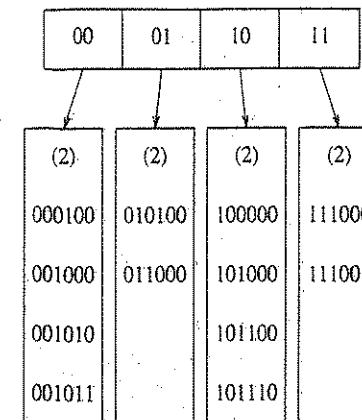


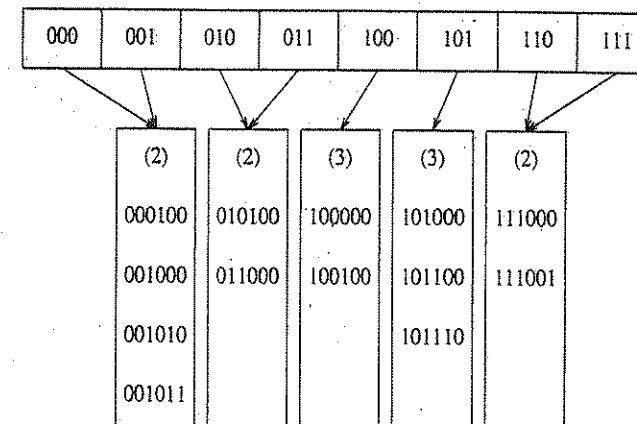
Figura 5.22: Dispersion extensible: datos originales

requiere incrementar el tamaño del directorio a 3. Esos cambios están reflejados en la figura 5.23.

Obsérvese que todas las hojas que no intervienen en la partición están apuntadas ahora por dos entradas de directorio adyacentes. Así, aunque se reescriba un directorio completo, no se tiene acceso real a ninguna de las otras hojas.

Si ahora se inserta la llave 000000, se parte la primera hoja, generando dos hojas con $d_h = 3$. Como $D = 3$, el único cambio requerido en el directorio es la actualización de los apuntadores 000 y 001. Véase la figura 5.24.

Figura 5.23: Dispersion extensible: después de la inserción de 100100 y la partición del directorio



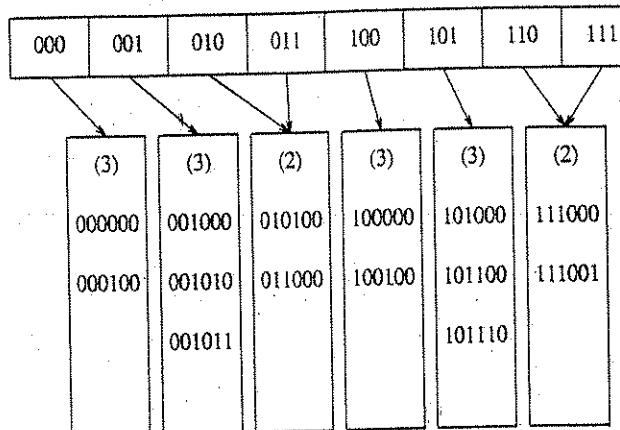


Figura 5.24 Dispersión extensible: después de la inserción de 000000 y la partición de la hoja

Esta estrategia muy sencilla ofrece rápidos tiempos de acceso para las operaciones *insertar* y *buscar* en bases de datos grandes. Hay unos cuantos detalles importantes que no se han considerado.

Primero, es posible que se requieran varias particiones de directorios si los elementos de una hoja concuerdan en más de $D + 1$ bits iniciales. Por ejemplo, iniciando en el ejemplo original, con $D = 2$, si se insertan 111010, 111011 y finalmente 111100, el tamaño del directorio se debe incrementar a 4 para distinguir entre las cinco llaves. Este detalle es fácil de cuidar, pero no debe olvidarse. Segundo, existe la posibilidad de llaves duplicadas; si hay más de m duplicados, el algoritmo no funcionará en absoluto. En este caso, se requieren algunos arreglos adicionales.

Estas posibilidades sugieren que es importante que los bits sean bastante aleatorios. Esto se puede obtener dispersando las llaves en un entero razonablemente grande; de aquí la razón del nombre.

Concluimos mencionando algunas propiedades del rendimiento de la dispersión extensible, las cuales se obtienen después de un análisis muy difícil. Esos resultados se basan en la razonable suposición de que los patrones de bits presentan una distribución uniforme.

El número esperado de hojas es $(n / m) \log_2 e$. Así la hoja media está llena a fin 2 = 0.69. Esto es lo mismo que en los árboles-B, lo que no es sorprendente del todo, ya que para ambas estructuras de datos los nodos nuevos se crean cuando se agrega la $(m+1)$ -ésima entrada.

El resultado más sorprendente es que el tamaño esperado del directorio (en otras palabras, 2^D) es $O(n^{1+1/m} / m)$. Si m es muy pequeña, el directorio puede ser indebidamente grande. En este caso, puede ocurrir que las hojas contengan apuntadores a los registros en vez de los registros mismos, incrementándose así el valor de m . Esto agrega un segundo acceso a disco en cada operación *buscar* a fin de mantener un directorio más pequeño. Si el directorio es demasiado grande para caber en memoria principal, de cualquier forma se necesitaría el segundo acceso a disco.

Resumen

Las tablas de dispersión se pueden usar para implantar las operaciones *insertar* y *buscar* en tiempo medio constante. Es de especial importancia cuidar detalles como el factor de carga cuando se emplean las tablas de dispersión, ya que de otra forma las cotas de tiempo no son válidas. También es importante escoger con cuidado la función de dispersión cuando la llave no es un entero o una cadena corta.

Para la dispersión abierta, el factor de carga debe ser cercano a 1, aunque el rendimiento no se degrada significativamente a menos que el factor de carga crezca mucho. Para dispersión cerrada, el factor de carga no debe exceder 0.5, a menos que sea totalmente inevitable. Si se usa la exploración lineal, el rendimiento se degenera con rapidez conforme el factor de carga se acerca a 1. Si el lector programa en un lenguaje que permite la asignación de arreglos sin conocer su tamaño en tiempo de compilación, entonces se puede implantar la redispersión. Ésta permitirá que la tabla crezca (y se contraiga) para mantener un factor de carga razonable. Esto es importante si el espacio es reducido y no es posible declarar una tabla de dispersión enorme.

Los árboles binarios de búsqueda también se pueden usar para implantar las operaciones *insertar* y *buscar*. Aunque las cotas de tiempo medio resultantes son $O(\log n)$, los árboles binarios de búsqueda también permiten rutinas que requieren orden y por ello son más potentes. Con una tabla de dispersión, no es posible encontrar el elemento mínimo. No es posible buscar cadenas con eficiencia, a menos que se conozca la cadena exacta. Un árbol binario de búsqueda podría encontrar rápidamente todos los elementos en un intervalo dado; esto no es posible en las tablas de dispersión. Además, la cota $O(\log n)$ no necesariamente es mucho más que $O(1)$, en especial porque no se requieren multiplicaciones o divisiones en los árboles de búsqueda.

Por otro lado, en general, el peor caso para la dispersión se debe a un error de implantación, mientras que una entrada ordenada puede provocar un rendimiento deficiente en los árboles binarios. Es muy costoso implantar árboles de búsqueda equilibrados, así que si no se requiere información de ordenamiento y hay alguna sospecha de que la entrada podría estar ordenada, entonces la dispersión es la estructura de datos apropiada.

Las aplicaciones de la dispersión son abundantes. Los compiladores se valen de tablas de dispersión para seguir el rastro de las variables declaradas en el código fuente. La estructura de datos se denomina tabla de *símbolos*. Las tablas de dispersión son la aplicación ideal para este problema porque sólo se efectúan *insertar* y *buscar*. Por lo regular, los identificadores son cortos, así que es rápido calcular la función de dispersión.

Una tabla de dispersión es útil para cualquier problema de teoría de grafos donde los nodos tienen nombres reales en vez de números. Aquí, al leer la entrada, a los vértices se les asigna enteros de 1 en adelante en el orden en que van apareciendo. De nuevo, es probable que la entrada tenga grandes grupos de entradas alfabetizadas. Por ejemplo, los vértices pueden ser computadores. Entonces si una instalación particular lista sus computadores como *ibm1*, *ibm2*, *ibm3*, ..., podría haber un tremendo efecto en la eficiencia si se usan árboles binarios de búsqueda.

Un tercer uso común de las tablas de dispersión son los programas de juegos. Conforme el programa busca a través de diferentes líneas de juego, mantiene las posiciones que ha visto, calculando las funciones de dispersión basadas en la posición (y almacenando su movimiento para esa posición). Si vuelve a ocurrir la misma posición, por lo regular por una simple transposición de los movimientos, el programa puede evitar el costo de volver a hacer los cálculos. Esta capacidad general de todos los programas de juego se conoce como *tabla de transposición*.

Existe aún otro uso de la dispersión: son los revisores de ortografía en línea. Si la detección de alguna falta ortográfica (por oposición a la corrección) es importante, se debe dispersar de antemano un diccionario completo y las palabras se pueden revisar en un tiempo constante. Las tablas de dispersión son bastante apropiadas para esto, porque no es importante alfabetizar palabras; la visualización de los errores ortográficos en el mismo orden en que van apareciendo es, desde luego, aceptable.

Cerramos este capítulo regresando al problema de la sopa de letras del capítulo 1. Si se usa el segundo algoritmo descrito en el capítulo 1, y suponemos que el tamaño máximo es alguna constante pequeña, entonces el tiempo para leer en el diccionario que contiene P palabras y ponerlas en una tabla de dispersión es $O(P)$. Es probable que este tiempo sea dominado por la E/S a disco y no por las rutinas de dispersión. El resto del algoritmo podría examinar la presencia de una palabra por cada cuarteto ordenado (*fila, columna, orientación, número de caracteres*). Como cada revisión podría ser $O(1)$, y sólo hay un número constante de orientaciones (8) y caracteres por palabra, el tiempo de ejecución de esta fase sería $O(r \cdot c)$. El tiempo de ejecución total sería $O(r \cdot c + P)$, que es una clara mejoría sobre el original $O(r \cdot c \cdot P)$. Aún se pueden hacer otras mejoras, que podrían reducir el tiempo de ejecución en la práctica; éstas se describen en los ejercicios.

Ejercicios

- 5.1 Dada la entrada {4371, 1323, 6173, 4199, 4344, 9679, 1989} y una función de dispersión $h(x) = x \pmod{10}$, muestre las resultantes
 - a. tabla de dispersión abierta;
 - b. tabla de dispersión cerrada con exploración lineal;
 - c. tabla de dispersión cerrada con exploración cuadrática;
 - d. tabla de dispersión cerrada con una segunda función de dispersión $h_2(x) = 7 - (x \pmod{7})$.
- 5.2 Demuestre el resultado de redispersar las tablas de dispersión del ejercicio 5.1.
- 5.3 Escriba un programa para calcular el número de colisiones requeridas en una larga secuencia aleatoria de inserciones usando exploración lineal, exploración cuadrática y dispersión doble.
- 5.4 Un número grande de eliminaciones en una tabla de dispersión abierta puede ocasionar que la tabla quede casi vacía, lo cual malgasta espacio. En este caso, podemos redispersar a una tabla de la mitad del tamaño. Suponga que redispersamos a una tabla más grande cuando hay el doble de elementos del

tamaño de la tabla. ¿Qué tan vacía debe estar la tabla abierta antes de redispersar a una tabla más pequeña?

- 5.5 Una estrategia alternativa de resolución de colisiones consiste en definir una secuencia, $f(i) = r_i$, donde $r_0 = 0$ y r_1, r_2, \dots, r_n es una permutación aleatoria de los primeros n enteros (cada entero aparece exactamente una vez).
 - a. Demuestre que con esta estrategia, si la tabla no está llena, siempre se puede resolver la colisión.
 - b. ¿Se podría esperar que esta estrategia elimine el agrupamiento?
 - c. Si el factor de carga de la tabla es λ , ¿cuál es el tiempo esperado para efectuar una inserción?
 - d. Si el factor de carga de la tabla es λ , ¿cuál es el tiempo esperado de una búsqueda exitosa?
 - e. Proporcione un algoritmo eficiente (tanto teórica como prácticamente) para generar la secuencia aleatoria. Explique por qué las reglas de selección de P son importantes.
- 5.6 ¿Cuáles son las ventajas y desventajas de las diferentes estrategias de resolución de colisiones?
- 5.7 Escriba un programa para implantar la siguiente estrategia de multiplicación de dos polinomios dispersos, P_1, P_2 , de tamaño m y n , respectivamente. Cada polinomio se representa como una lista enlazada con celdas consistentes en un coeficiente, un exponente y un apuntador *siguiente* (ejercicio 3.7). Multiplicamos cada término en P_1 por un término en P_2 para un total de mn operaciones. Un método es ordenar los términos y combinar términos comunes, pero esto requiere la clasificación de mn registros, lo cual puede ser costoso, sobre todo en entornos de memoria reducida. Alternativamente, podríamos combinar términos conforme se calculan para después ordenar el resultado.
 - a. Escriba un programa para implantar la estrategia alternativa.
 - b. Si el polinomio de salida tiene cerca de $O(m + n)$ términos, ¿cuál es el tiempo de ejecución de ambos métodos?
- 5.8 Un revisor de ortografía lee un archivo de entrada y visualiza todas las palabras que no están en algún diccionario en línea. Suponga que el diccionario contiene 30 000 palabras y el archivo es de un megabyte, así que el algoritmo puede hacer sólo una pasada a través del archivo de entrada. Una estrategia sencilla es leer el diccionario en una tabla de dispersión y buscar cada palabra de entrada conforme se lea. Suponiendo que una palabra media es de siete caracteres y que es posible almacenar palabras de longitud l en $l + 1$ bytes (así el espacio malgastado no es considerable), y suponiendo una tabla cerrada, ¿cuánto espacio se necesita?
- 5.9 Si la memoria es limitada y el diccionario completo no se puede almacenar en una tabla de dispersión, todavía podemos tener un algoritmo eficiente que casi siempre funcione. Declaramos un arreglo *TABLA_D* de bits (con valor inicial en ceros) de 0 a *TAM_TABLA* - 1. Conforme se lee una palabra, se pone

TABLA_D[dispersión(palabra)] = 1. De las siguientes afirmaciones, ¿qué es verdadero?

- Si una palabra se dispersa a una posición con valor 0, la palabra no está en el diccionario.
- Si una palabra se dispersa a una posición con valor 1, la palabra está en el diccionario.
- ¿Cuánta memoria requiere esto?
- ¿Cuál es la probabilidad de un error en este algoritmo?
- Un documento representativo puede tener cerca de tres faltas ortográficas por página de 500 palabras. ¿Es utilizable este algoritmo?

5.10 *Describa un procedimiento que evite la puesta de valores iniciales en la tabla de dispersión (a expensas de la memoria).

5.11 Supóngase que queremos encontrar la primera aparición de una cadena $p_1p_2 \dots p_k$ en una cadena de entrada grande $a_1a_2 \dots a_n$. Podemos resolver este problema dispersando el patrón de la cadena, obteniendo un valor de dispersión h_p , y comparando este valor con el valor de dispersión formado de $a_1a_2 \dots a_k, a_2a_3 \dots a_{k+1}, a_3a_4 \dots a_{k+2}$, y así sucesivamente hasta $a_{n-k+1}a_{n-k+2} \dots a_n$. Si tenemos una correspondencia entre valores de dispersión, comparamos las cadenas, carácter por carácter, para verificar la correspondencia. Se devuelve la posición (en a) si las cadenas realmente corresponden, y continuamos en el improbable caso de que la correspondencia sea falsa.

- *a. Demuestre que si conocemos el valor de dispersión de $a_1a_{i+1} \dots a_{i+k-1}$, entonces el valor de dispersión de $a_{i+1}a_{i+2} \dots a_{i+k}$ se puede calcular en un tiempo constante.
- b. Demuestre que el tiempo de ejecución es $O(k + n)$ más el tiempo consumido en rechazar las correspondencias falsas.
- *c. Demuestre que el número esperado de correspondencias falsas es insignificante.
- d. Escriba un programa para implantar este algoritmo.
- **e. Describa un algoritmo que se ejecute en un tiempo de $O(k + n)$ para el peor caso.
- **f. Describa un algoritmo que se ejecute en un tiempo medio de $O(n/k)$.

5.12 Un programa en BASIC consiste en una serie de enunciados, cada uno de los cuales se numera en orden ascendente. Se pasa el control con el uso de un *goto* o un *gosub* y un número de enunciado. Escriba un programa que lea un programa legal en BASIC y vuelva a numerar los enunciados de manera que el primero empiece en el número *f* y cada enunciado tenga un número *d* mayor que el enunciado anterior. Puede suponer un límite superior de *n* enunciados, pero los números de enunciado en la entrada pueden ser tan grandes como un entero de 32 bits. Su programa debe ejecutarse en tiempo lineal.

5.13 a. Implante el programa de la sopa de letras usando el algoritmo descrito al final del capítulo.

- Podemos alcanzar una gran velocidad almacenando, además de cada palabra *p*, todos los prefijos de *p*. (Si uno de los prefijos de *p* es otra palabra del diccionario, se almacena como una palabra real). Aunque puede parecer que esto incrementa drásticamente el tamaño de la tabla de dispersión, no lo hace porque muchas palabras tienen los mismos prefijos. Cuando se realiza un recorrido en una dirección particular, si la palabra que se busca no se encuentra en la tabla de dispersión aun como prefijo, la búsqueda en esa dirección puede terminar pronto. Véjase de esta idea para escribir un programa mejorado para resolver la sopa de letras.
- Si estamos dispuestos a sacrificar la pureza del TDA tabla de dispersión, podemos mejorar la velocidad del programa de la parte (b) notando que si, por ejemplo, sólo hemos calculado la función de dispersión para "superar", no necesitamos calcular la función de dispersión para "superar" desde el principio. Ajuste la función de dispersión para que pueda aprovechar los cálculos anteriores.
- En el capítulo 2 sugerimos el uso de la búsqueda binaria. Incorpore la idea de usar prefijos en el algoritmo de búsqueda binaria. La modificación debe ser sencilla. ¿Qué algoritmo es más rápido?
- Muestre el resultado de insertar las llaves 10111101, 00000010, 10011011, 10111110, 01111111, 01010001, 10010110, 00001011, 11001111, 10011110, 11011011, 01010111, 01100001, 11110000, 01101111 en una estructura de datos de dispersión extensible inicialmente vacía con $m = 4$.
- Escriba un programa para implantar la dispersión extensible. Si la tabla es suficientemente pequeña para caber en memoria principal, ¿cómo se compara su rendimiento con la dispersión abierta y cerrada?

Referencias

A pesar de la aparente simplicidad de la dispersión, buena parte del análisis es muy difícil y aún quedan muchas cuestiones no resueltas. También hay muchos resultados teóricos interesantes, que por lo regular intentan hacer improbable que surjan las posibilidades del peor caso en la dispersión.

Uno de los primeros artículos sobre dispersión es [17]. Un caudal de información sobre el tema, incluyendo un análisis de la dispersión cerrada con exploración lineal, se puede encontrar en [11]. Un excelente estudio del tema es [14]; [15] contiene sugerencias y problemas para la elección de funciones de dispersión. En [8] se pueden encontrar resultados analíticos y de simulación precisos para todos los métodos descritos en este capítulo.

Un análisis de la dispersión doble se presenta en [9] y [13]. Otro esquema de resolución de colisiones más es la dispersión unida, descrita en [18]. Yao [20] ha demostrado que la dispersión uniforme, en la que no existe el agrupamiento, es óptima con respecto al costo de una búsqueda exitosa.

Si las llaves de entrada se conocen de antemano, entonces existen las funciones de dispersión perfectas, que no permiten colisiones ([2] y [7]). Algunos

esquemas de dispersión más complejos, para los cuales el peor caso depende no de la entrada particular sino de números aleatorios escogidos por el algoritmo, aparecen en [3] y [4].

La dispersión extensible se estudia en [5], con el análisis en [6] y [19].

Un método de implantación del ejercicio 5.5 se describe en [16]. El ejercicio 5.11 (a-d) proviene de [10]. La parte (e) es de [12] y la parte (f) es de [1].

1. R. S. Boyer y J. S. Moore, "A Fast String Searching Algorithm", *Communications of the ACM*, 20 (1977), págs. 762-772.
2. J. L. Carter y M. N. Wegman, "Universal Classes of Hash Functions", *Journal of Computer and System Sciences*, 18 (1979), págs. 143-154.
3. 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", *Proceedings of the Twenty-ninth IEEE Symposium on Foundations of Computer Science* (1988), págs. 524-531.
4. R. J. Enbody y H. C. Du, "Dynamic Hashing Schemes", *Computing Surveys*, 20 (1988), págs. 85-113.
5. R. Fagin, J. Nievergelt, N. Pippenger, y H. R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems*, 4 (1979), págs. 315-344.
6. P. Flajolet, "On the Performance Evaluation of Extendible Hashing and Trie Searching", *Acta Informatica*, 20 (1983), págs. 345-369.
7. M. L. Fredman, J. Komlos, y E. Szemerédi, "Storing a Sparse Table with O(1) Worst Case Access Time", *Journal of the ACM*, 31 (1984), págs. 538-544.
8. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2a. ed., Addison-Wesley, Reading, 1991.
9. L. J. Guibas y E. Szemerédi, "The Analysis of Double Hashing", *Journal of Computer and System Sciences*, 16 (1978), págs. 226-274.
10. R. M. Karp y M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms", *Aiken Computer Laboratory Report TR-31-81*, Harvard University, Cambridge, MA, 1981.
11. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, 2a. imp., Addison-Wesley, Reading, MA, 1975.
12. D. E. Knuth, J. H. Morris, V. R. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Computing*, 6 (1977), págs. 323-350.
13. G. Lueker y M. Molodowitch, "More Analysis of Double Hashing", *Proceedings of the Twentieth ACM Symposium on Theory of Computing* (1988), págs. 354-359.
14. W. D. Maurer y T. G. Lewis, "Hash Table Methods", *Computing Surveys*, 7 (1975), págs. 5-20.
15. B. J. McKenzie, R. Harries, y T. Bell, "Selecting a Hashing Algorithm", *Software-Practice and Experience*, 20 (1990), págs. 209-224.
16. R. Morris, "Scatter Storage Techniques", *Communications of the ACM*, 11 (1968), págs. 38-44.
17. W. W. Peterson, "Addressing for Random Access Storage", *IBM Journal of Research and Development*, 1 (1957), págs. 130-146.
18. J. S. Vitter, "Implementations for Coalesced Hashing", *Communications of the ACM*, 25 (1982), págs. 911-926.
19. A. C. Yao, "A Note on The Analysis of Extendible Hashing", *Information Processing Letters*, 11 (1980), págs. 84-86.
20. A. C. Yao, "Uniform Hashing is Optimal", *Journal of the ACM*, 32 (1985), págs. 687-693.

Colas de prioridad (montículos)

Aun cuando los trabajos enviados a una impresora de línea se suelen colocar en una cola, esto puede no ser siempre lo mejor. Por ejemplo, un trabajo puede ser de particular importancia, de modo que es deseable permitir que se lleve a cabo tan pronto como la impresora esté disponible. A la inversa, si al quedar disponible la impresora hay varios trabajos de una página y uno de cien, puede ser razonable dejar el trabajo grande para el final, aun cuando no sea el último enviado. (Desafortunadamente, la mayoría de los sistemas no hacen esto, lo cual puede ser muy incómodo en ocasiones.)

De manera similar, en un entorno multiusuario, el planificador del sistema operativo debe decidir entre varios procesos cuál ejecutar. En general se permite que un proceso se ejecute sólo durante un tiempo fijo. Un algoritmo usa una cola. Al inicio los trabajos se colocan al final de la cola. El planificador tomará repetidamente el primer trabajo de la cola, lo ejecutará hasta que termine o alcance su límite de tiempo, y lo colocará al final de la cola si no termina. Por lo general, esta estrategia no es adecuada porque trabajos muy cortos parecerán lentos debido al tiempo de espera para su ejecución. Por lo regular, es importante que los trabajos cortos terminen tan pronto como sea posible, de modo que estos trabajos deberían tener preferencia sobre los trabajos que ya han estado en ejecución. Además, algunos trabajos no cortos son muy importantes y deben también tener preferencia.

Esta aplicación particular parece requerir una clase especial de cola, denominada *cola de prioridad*. En este capítulo estudiaremos:

- La implantación eficiente del TDA cola de prioridad.
- Los usos de las colas de prioridad.
- Las implantaciones avanzadas de las colas de prioridad.

Las estructuras de datos que veremos están entre las más elegantes en la informática.

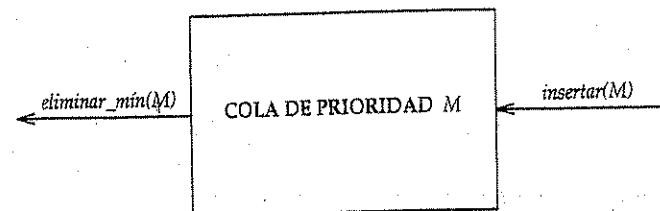


Figura 6.1 Modelo básico de una cola de prioridad

6.1. Modelo

Una cola de prioridad es una estructura de datos que permite al menos las siguientes dos operaciones: *insertar*, que hace la operación obvia, y *eliminar_min*, que busca, devuelve y elimina el elemento mínimo del montículo. La operación *insertar* es el equivalente de *encolar*, y *eliminar_min* es equivalente en la cola de prioridad a la operación *desencolar* de las colas. La función *eliminar_min* también altera su entrada. En la actualidad, entre la comunidad de ingenieros de software se piensa que ésta no es una buena idea. No obstante, continuaremos usando esta función por razones históricas. Muchos programadores esperan que *eliminar_min* opere en esta forma.

Como con la mayoría de las estructuras de datos, a veces es posible agregar otras operaciones, pero son extensiones y no forman parte del modelo básico descrito en la figura 6.1.

Las colas de prioridad tienen muchas aplicaciones más allá de los sistemas operativos. En el capítulo 7, veremos cómo se usan las colas de prioridad en la ordenación externa. Las colas de prioridad también son importantes en la implantación de *algoritmos ávidos*, los cuales operan encontrando un mínimo repetidamente; veremos ejemplos específicos en los capítulos 9 y 10. En éste analizaremos el uso de las colas de prioridad en la simulación de eventos discretos.

6.2. Implantaciones simples

Hay varias formas obvias de implantar colas de prioridad. Podríamos usar una simple lista enlazada efectuando inserciones al frente en $O(1)$ y recorriendo la lista, lo cual requiere un tiempo $O(n)$, para eliminar el mínimo. Alternativamente, podríamos insistir en que la lista siempre esté ordenada, lo que hace que las inserciones sean costosas ($O(n)$) y que las *eliminar_min* no lo sean ($O(1)$). Es posible que la primera sea la mejor idea de las dos, con base en el hecho de que nunca hay más *eliminar_min* que inserciones.

Otra forma de implantar las colas de prioridad consistiría en usar un árbol binario de búsqueda, lo que da un tiempo de ejecución medio $O(\log n)$ para ambas operaciones. Esto es cierto a pesar del hecho de que, aunque las inserciones sean operaciones aleatorias, las eliminaciones no lo serán. Recordemos que el único elemento que se elimina siempre es el mínimo. La eliminación repetida de un nodo que está en el subárbol izquierdo parecería alterar el equilibrio del árbol al hacer más pesado

el subárbol derecho. No obstante, el subárbol derecho es aleatorio. En el peor caso, donde las operaciones *eliminar_min* han agotado el subárbol izquierdo, el subárbol derecho tendría a lo más el doble de elementos de los que debería. Esto sólo incrementa en uno la profundidad esperada. Observe que las cotas pueden llevarse al peor caso usando un árbol equilibrado; esto protege de malas secuencias de inserciones.

Utilizar un árbol de búsqueda podría ser excesivo al permitir una variedad de operaciones que no se requieren. La estructura de datos básica que emplearemos no requerirá apuntable y permitirá ambas operaciones en $O(\log n)$ para el peor caso. En realidad la inserción tardará un tiempo medio constante, y nuestra implantación permitirá construir un montículo de n elementos en un tiempo lineal, si no intervienen eliminaciones. Después estudiaremos cómo implantar montículos para lograr una fusión eficiente. Esta operación adicional parece complicar un poco el asunto y en apariencia requiere el uso de apuntable.

6.3. Montículo binario

La implantación que usaremos se conoce como *montículo binario*, y es tan común para implantar colas de prioridad que cuando se menciona la palabra *montículo* sin calificativo, por lo regular se supone generalmente que se refiere a esta implantación de la estructura de datos. En esta sección, nos referimos a los montículos binarios simplemente como *montículos*. Como los árboles binarios de búsqueda, los montículos tienen dos propiedades, a saber, una propiedad de la estructura y una propiedad de orden de montículo. Como los árboles AVL, una operación sobre un montículo puede destruir una de las propiedades, así que la operación no debe terminar hasta que estén en orden todas las propiedades del montículo. Esto es fácil de lograr.

6.3.1. Propiedad de la estructura

Un montículo es un árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha. Un árbol así se llama *árbol binario completo*. La figura 6.2 muestra un ejemplo.

Es fácil demostrar que un árbol binario completo de altura h tiene entre 2^h y $2^{h+1} - 1$ nodos. Esto implica que la altura de un árbol binario completo es $\lfloor \log n \rfloor$, lo cual es claramente $O(\log n)$.

Una observación importante es que debido a que un árbol binario completo es tan regular, se puede representar en un arreglo sin recurrir a los apuntable. El arreglo de la figura 6.3 corresponde al montículo de la figura 6.2.

Para cualquier elemento en la posición i del arreglo, el hijo izquierdo está en la posición $2i$, el hijo derecho está en la celda siguiente al hijo izquierdo ($2i + 1$), y el padre está en la posición $\lfloor i / 2 \rfloor$. Así no sólo no se requieren los apuntable, sino que las operaciones necesarias para recorrer el árbol son muy sencillas y probablemente muy rápidas en casi cualquier computador. El único problema con esta implantación es que requiere por anticipado un cálculo máximo del montículo, pero

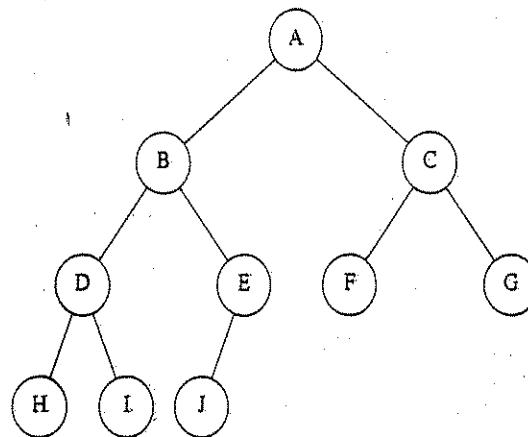


Figura 6.2 Un árbol binario completo

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figura 6.3 Implantación con un arreglo del árbol binario completo

por lo regular no es problemático. En la figura 6.3, el límite del tamaño del montículo es 13 elementos. El arreglo tiene una posición 0; más adelante volveremos a esto.

Entonces, una estructura de datos montículo consistirá en un arreglo (de cualquier tipo que sea la llave) y un entero que represente el tamaño actual del montículo. La figura 6.4 muestra una declaración típica de colas de prioridad.

En este capítulo los montículos se representarán como árboles, en el entendimiento de que la implantación real usa arreglos sencillos.

6.3.2. Propiedad de orden de montículo

La propiedad que permite efectuar rápidamente las operaciones es la *propiedad de orden de montículo*. Como queremos ser capaces de encontrar muy rápido el mínimo, tiene sentido que el mínimo deba estar en la raíz. Si consideramos que cualquier

Figura 6.4 Declaración de colas de prioridad.

```

type
  COLA_DE_PRIORIDAD = record
    elemento: array [0..MÁX_ELEMENTO] of tipo_elemento;
    tamaño: integer;
  end;
  
```

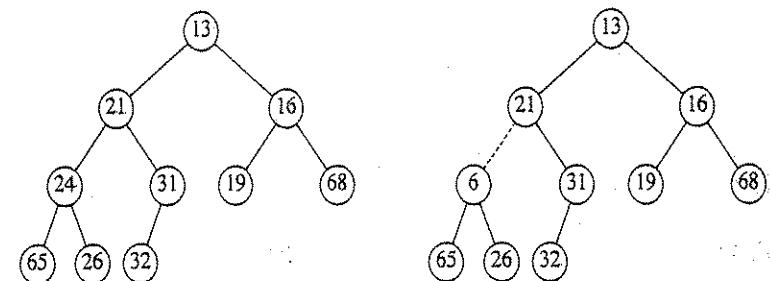


Figura 6.5 Dos árboles completos (sólo el árbol de la izquierda es un montículo)

subárbol debe ser también un montículo, entonces cualquier nodo debe ser menor que todos sus descendientes.

Aplicando esta lógica, se llega a la propiedad de orden de montículo. En un montículo, para todo nodo X , la llave en el padre de X es menor (o igual) que la llave en X , con la excepción obvia de la raíz (la cual no tiene padre).^f En la figura 6.5 el árbol de la izquierda es un montículo, pero el de la derecha no lo es (la línea punteada muestra la violación de la propiedad de orden de montículo). Como es usual, se supondrá que las llaves son enteros, aunque pueden ser arbitrariamente complejas.

Por la propiedad de orden del montículo, el elemento mínimo siempre puede encontrarse en la raíz. Así, obtenemos la operación adicional, *buscar_mín*, en tiempo constante.

6.3.3. Operaciones básicas sobre montículos

Es fácil (conceptual y prácticamente) efectuar las dos operaciones requeridas. Todo el trabajo implica asegurarse de que se mantenga la propiedad de orden de montículo.

Insertar

Para insertar un elemento x en el montículo, creamos un hueco en la siguiente posición disponible, ya que de otra forma el árbol no estará completo. Si se puede colocar x en el hueco sin violar la propiedad de orden de montículo, lo hacemos así y todo queda listo. En otro caso se desliza el elemento que está en el lugar del nodo padre del hueco, subiendo así el hueco hacia la raíz. Seguimos este proceso hasta poder colocar x en el hueco. La figura 6.6 muestra que para insertar 14, se crea un

^f Análogamente podemos declarar un montículo (máx) que permita encontrar y eliminar eficientemente el máximo elemento cambiando la propiedad de orden de montículo. Así, se puede usar una cola de prioridades para encontrar ya sea un mínimo o un máximo, pero esto necesita decidirse de antemano.

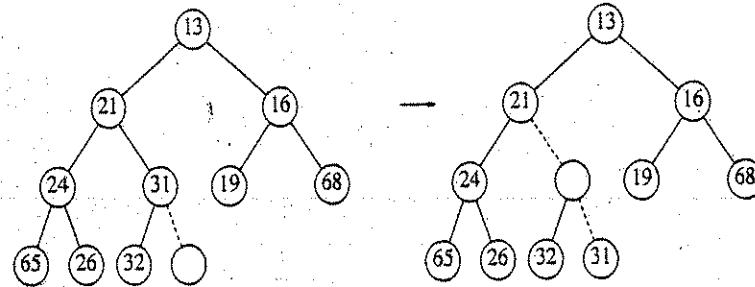


Figura 6.6. Intento de inserción del 14: creación y ascenso del hueco

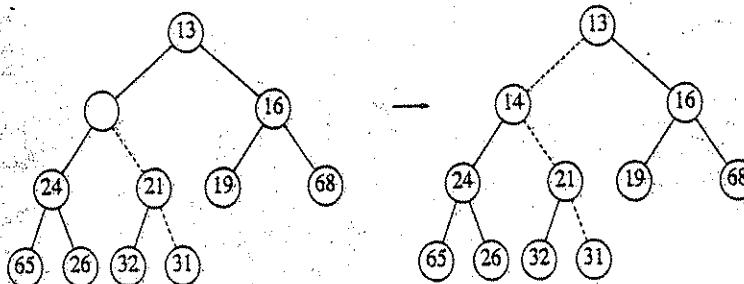


Figura 6.7 Los dos pasos restantes para insertar 14 en el montículo anterior

hueco en la siguiente posición disponible del montículo. La inserción del 14 en el hueco puede violar la propiedad de orden de montículo, así que 31 se baja al lugar del hueco. Esta estrategia continúa en la figura 6.7 hasta encontrar la localidad correcta para el 14.

Esta estrategia general se conoce como *filtrado ascendente*; el elemento nuevo se filtra en el montículo hasta encontrar su posición correcta. La inserción se puede implantar fácilmente con el código mostrado en la figura 6.8.

Podríamos implantar el filtrado en la rutina *insertar* realizando intercambios repetidos hasta establecer el orden correcto, pero un intercambio requiere tres enunciados de asignación. Si un elemento es filtrado hacia arriba d niveles, el número de asignaciones efectuadas por los intercambios podría ser $3d$. Nuestro método usa $d + 1$ asignaciones.

Si el elemento que se ha de insertar es el nuevo mínimo, se llevará hasta arriba en el árbol. En algún punto, i será 1 y se deseará romper el ciclo *while*. Esto se puede hacer con una comprobación explícita, pero hemos preferido poner un valor muy pequeño en la posición 0 a fin de asegurar que el ciclo *while* termine. Se debe garantizar que este valor sea menor (o igual) que cualquier elemento del montículo; se le conoce como *centinela*. Esta idea es semejante a usar nodos cabecera en las listas

[M.elemento[0] es un centinela]

```

procedure insertar(x: tipo_elemento; var M: COLA_DE_PRIORIDAD);
  var i: integer;
begin
  [1] if M.tamaño = MÁX_ELEMENTO then
  [2]   error('El montículo está lleno')
  else
    begin
      [3]   M.tamaño := M.tamaño + 1;
      [4]   i := M.tamaño;

      [5]   while M.elemento[i div 2] > x do
      begin
        [6]     M.elemento[i] := M.elemento[i div 2];
        [7]     i := i div 2
      end;

      [8]   M.elemento[i] := x;
    end;
end;
```

Figura 6.8 Procedimiento para insertar en un montículo binario

enlazadas. Agregando una pieza falsa de información, evitamos hacer una comprobación en cada iteración del ciclo, ahorrando algo de tiempo.^t

El tiempo para hacer la inserción podría llegar a ser $O(\log n)$, si el elemento que se ha de insertar es el nuevo mínimo y es filtrado en todo el camino hacia la raíz. En término medio, el filtrado termina pronto; se ha demostrado que se requieren 2.607 comparaciones en promedio para realizar una inserción, así que el *insertar* medio sube un elemento 1.607 niveles.

Eliminar_mín

Eliminar_mín se maneja en una forma semejante a las inserciones. Encontrar el mínimo es sencillo; la parte difícil es la eliminación. Cuando se retira el mínimo, se crea un hueco en la raíz. Como el tamaño del montículo se reduce en uno, se infiere que el último elemento x en el montículo debe moverse a alguna otra parte del montículo. Si se puede colocar x en el hueco, habremos terminado; sin embargo, ello es improbable, así que se desliza el menor de los hijos del hueco hacia el hueco, empujando el hueco hacia abajo un nivel. Este paso se repite hasta

^t También se ahorra un esfuerzo de codificación, que es más compleja por la falta del cortocircuito booleano en el operador *and* en Pascal.

que x se coloca en el hueco. Así, nuestra acción es colocar x en su lugar correcto sobre una trayectoria desde la raíz que contiene el hijo *mínimo*.

En la parte izquierda de la figura 6.9 se muestra un montículo antes del *eliminar_mín*. Después de eliminar 13, debemos intentar reacomodar 31 en el montículo. 31 no puede ser colocado en el hueco porque violaría el orden de montículo. Así, colocamos el hijo menor (14) en el hueco, deslizando el hueco hacia abajo un nivel (véase la figura 6.10). Repetimos esto, colocando 19 en el hueco y creando un hueco nuevo en un nivel más abajo. Entonces colocamos 26 en el hueco y creamos un hueco nuevo en el nivel más profundo. Por último, es posible colocar 31 en el hueco (figura 6.11). A esta estrategia general se le llama *filtrado descendente*. Utilizamos la misma técnica que en la rutina *insertar* para evitar el uso de intercambios en esta rutina.

Hemos empleado una construcción típica *while not encontrado*, en vez de los *goto* ya vistos en algunas de las codificaciones anteriores. El propósito es mostrar la diferencia de los dos estilos. Si usted tiene la opción, elija el estilo con el que se sienta más cómodo.

Figura 6.9 Creación del hueco en la raíz

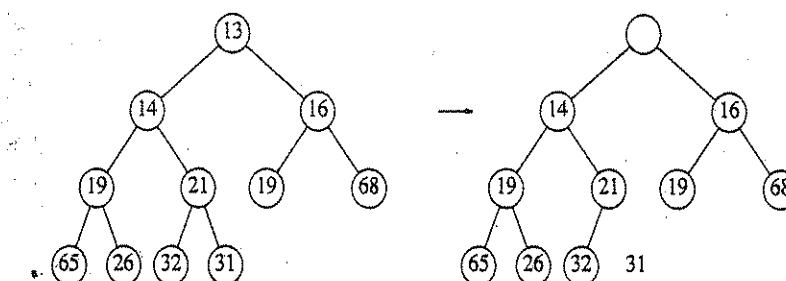


Figura 6.10 Siguientes dos pasos en *eliminar_mín*

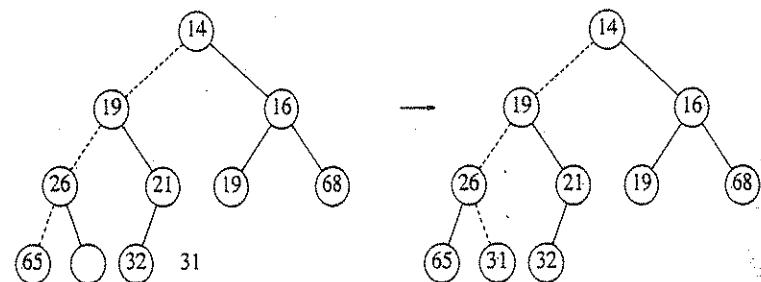
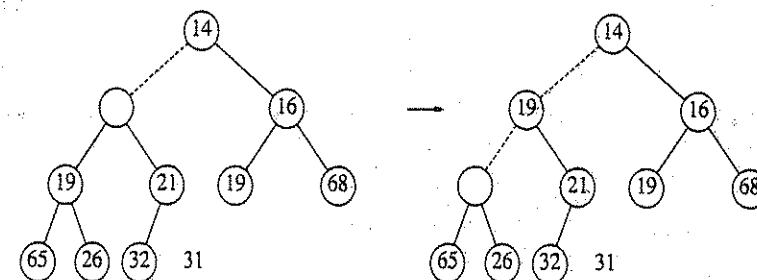


Figura 6.11 Últimos dos pasos en *eliminar_mín*

Un frecuente error de implantación en los montículos se da cuando hay un número par de elementos en él y se encuentra el nodo con sólo un hijo. Hay que asegurarse de no suponer que siempre hay dos hijos, así que en esto suele intervenir una comprobación adicional. En el código, mostrado en la figura 6.12, esta prueba se hizo en la línea 9. Una solución muy trámosa es asegurar que el algoritmo *piense* que todo nodo tiene dos hijos. Esto se hace colocando un centinela, de valor mayor que cualquiera del montículo, en el espacio siguiente al final del montículo, al principio de cada *filtrado descendente* cuando el tamaño del montículo es par. Se debe pensar muy cuidadosamente antes de intentar esto, y hay que poner un comentario muy visible si se usa esta técnica. Aunque esto elimina la necesidad de comprobar la presencia de un hijo derecho, no se puede eliminar la necesidad de comprobar cuándo se alcanza la base porque entonces haría falta un centinela para cada hoja.

El tiempo de ejecución para el peor caso en esta operación es $O(\log n)$. En promedio, un elemento colocado en la raíz se filtra casi hasta la base del montículo (que es el nivel del que viene), así que el tiempo de ejecución medio es $O(\log n)$.

6.3.4. Otras operaciones sobre montículos

Cabe señalar que aunque la búsqueda del mínimo se puede hacer en un tiempo constante, un montículo diseñado para encontrar el elemento mínimo (también conocido como montículo (*min*)) no es de ninguna ayuda para encontrar el elemento máximo. De hecho, un montículo tiene muy poca información sobre el orden, así que no hay forma de encontrar cualquier llave particular sin un recorrido lineal sobre todo el montículo. Para ver esto, consideremos el montículo grande (no se muestran los elementos) de la figura 6.13, donde se ve que la única información conocida acerca del elemento máximo es que está en una de las hojas. La mitad de los elementos se encuentran en las hojas, así que esta información es prácticamente inútil. Por esta razón, si es importante conocer dónde están los elementos, debe usarse alguna otra estructura de datos, como

```

function eliminar_mín(var M: COLA_DE_PRIORIDAD); tipo_elemento;
var i, hijo: integer;
último_elem: tipo_elemento;
parar_filt: boolean;
begin
[1] if M.tamaño = 0 then
[2]   error('La cola de prioridad está vacía')
else
begin
[3]   eliminar_mín := M.elemento[1];
[4]   último_elem := M.elemento[M.tamaño];
[5]   M.tamaño := M.tamaño - 1;

[6]   i := 1; parar_filt := FALSE;
[7]   while (i * 2 < M.tamaño) and (not parar_filt) do
begin
[8]     hijo := i * 2; {buscar hijo menor}
[9]     if hijo > M.tamaño then
[10]       if M.elemento[hijo + 1] < M.elemento[hijo] then
[11]         hijo := hijo + 1;

[12]     if último_elem > M.elemento[hijo] then {filtrar}
begin
[13]       M.elemento[i] := M.elemento[hijo];
[14]       i := hijo;
end;
[15]     else {fuerza la salida del ciclo while}
[16]       parar_filt := TRUE;
end; {while}
M.elemento[i] := último_elem;
end; {if M.tamaño <> 0}
end; {eliminar_mín}

```

Figura 6.12 Función para efectuar *eliminar_mín* en un montículo binario

una tabla de dispersión, además del montículo. (Recuerde que el modelo no permite observar en el interior el montículo.)

Si suponemos que la posición de todo elemento se conoce por algún otro método, entonces varias operaciones resultan económicas. Las tres operaciones siguientes se ejecutan en un tiempo logarítmico para el peor caso.

Decrementar_llave

La operación *decrementar_llave*(x, Δ, M) reduce el valor de la llave en la posición x por una cantidad positiva Δ . Como esto puede violar el orden de montículo, debe arreglarse con un *filtrado ascendente*. Esta operación podría ser útil para adminis-

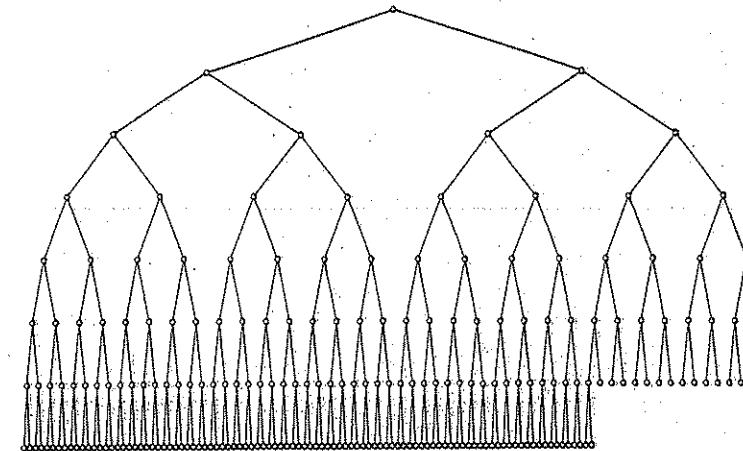


Figura 6.13 Un árbol binario completo muy grande

dores de sistemas: pueden hacer que sus programas se ejecuten con la mayor prioridad.

Incrementar_llave

La operación *incrementar_llave*(x, Δ, M) aumenta el valor de la llave en la posición x en una cantidad positiva Δ . Esto se obtiene con un *filtrado descendente*. Muchos planificadores bajan automáticamente la prioridad de un proceso que consume un tiempo excesivo de UCP.

Eliminar

La operación *eliminar*(x, M) retira el nodo de la posición x del montículo. Esto se hace ejecutando primero *decrementar_llave*(x, ∞, M) y después *eliminar_mín*(M). Cuando un proceso termina por orden del usuario (en vez de terminar normalmente) se debe eliminar de la cola de prioridad.

Construir_montículo

La operación *construir_montículo*(M) toma como entrada n llaves y las coloca en un montículo vacío. Es obvio que esto se puede hacer con n operaciones *insertar* sucesivas. Como cada *insertar* tardará $O(1)$ en el caso medio y $O(\log n)$ en el peor caso, el tiempo de ejecución total de este algoritmo puede ser $O(n)$ en promedio, pero $O(n \log n)$ en el peor caso. Puesto que esta instrucción es especial y no intervienen otras operaciones, y ya se sabe que la instrucción se puede efectuar en tiempo medio lineal, es razonable esperar que con cuidado suficiente se garantice una cota de tiempo lineal.

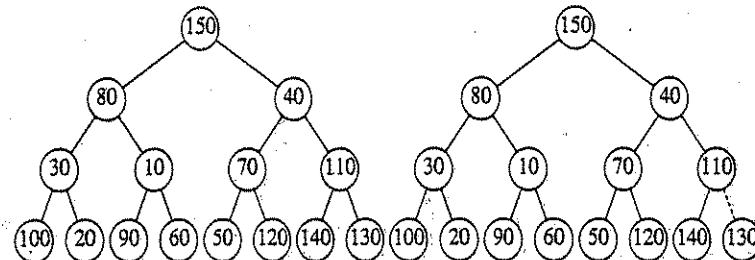
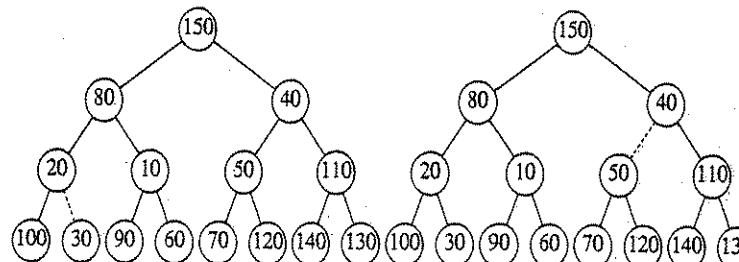
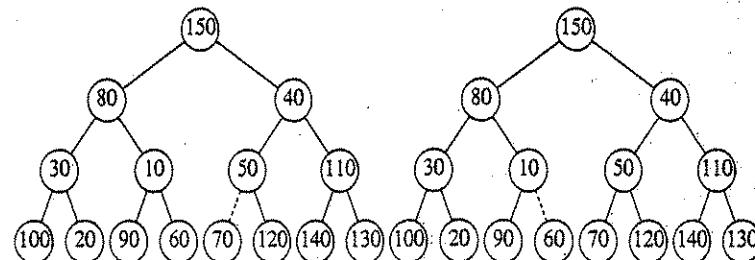
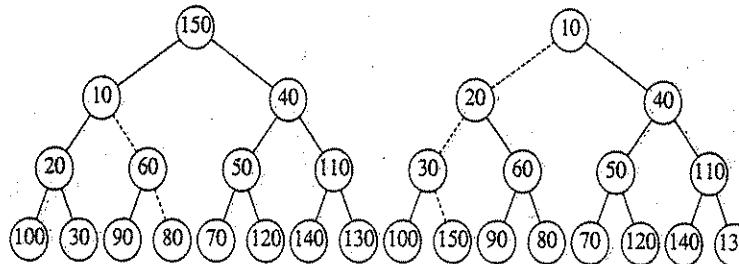
```
for i := n div 2 downto 1 do
    filtrado_descendente(i);
```

Figura 6.14 Bosquejo de construir_montículo

El algoritmo general consiste en colocar las n llaves dentro del árbol en cualquier orden, manteniendo la propiedad de la estructura. Entonces, si $\text{filtrado_descendente}(i)$ filtra por abajo del nodo i , se ejecuta el algoritmo de la figura 6.14 para crear un árbol con orden de montículo.

El primer árbol de la figura 6.15 es el árbol desordenado. Los siete árboles restantes de las figuras 6.15 a 6.18 muestran el resultado de cada uno de los siete $\text{filtrados_descendentes}$. Cada línea punteada corresponde a dos comparaciones: una para encontrar el hijo menor y otra para comparar el hijo menor con el nodo. Observe que sólo hay 10 líneas punteadas en el algoritmo completo (podría haber una décimoprimeras, ¿dónde?) que corresponden a 20 comparaciones.

Para acotar el tiempo de ejecución de *construir_montículo*, se debe acotar el número de líneas punteadas. Esto se puede hacer calculando la suma de alturas de todos los nodos en el montículo, que es el número máximo de líneas punteadas. Lo que quisieramos demostrar es que esta suma es $O(n)$.

Figura 6.15 Izquierda: montículo inicial; derecha: después de *filtrado_descendente(7)*Figura 6.16 Izquierda: después de *filtrado_descendente(6)*; derecha: después de *filtrado_descendente(5)*Figura 6.17 Izquierda: después de *filtrado_descendente(4)*; derecha: después de *filtrado_descendente(3)*Figura 6.18 Izquierda: después de *filtrado_descendente(2)*; derecha: después de *filtrado_descendente(1)***TEOREMA 6.1.**

Para el árbol binario perfecto de altura h que contiene $2^{h+1} - 1$ nodos, la suma de las alturas de los nodos es $2^{h+1} - 1 - (h+1)$.

DEMOSTRACIÓN:

Es fácil ver que este árbol consiste en un nodo de altura h , dos nodos de altura $h-1$, 2^2 nodos de altura $h-2$, y en general 2^i nodos de altura $h-i$. Entonces la suma de las alturas de todos los nodos es

$$\begin{aligned} S &= \sum_{i=0}^h 2^i(h-i) \\ &= h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \dots + 2^{h-1} (1) \end{aligned} \quad (6.1)$$

Multiplicando por dos obtenemos la ecuación

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \dots + 2^h (1) \quad (6.2)$$

Restamos estas dos ecuaciones y se obtiene la ecuación 6.3. Observamos que ciertos términos casi se cancelan. Por ejemplo, se tiene $2h - 2(h-1) = 2, 4(h-1) - 4(h-2) = 4$, y así sucesivamente. El último término de la ecuación 6.2, 2^h , no aparece en la ecuación 6.1; así, aparece en la ecuación 6.3. El primer término en la ecuación 6.1, h , no aparece en la ecuación 6.2; así, $-h$ aparece en la ecuación 6.3.

Obtenemos

$$S = -h + 2 + 4 + 8 + \dots + 2^{h-1} = 2^{h+1} - 1 - (h+1)$$

con lo cual se demuestra el teorema.

Un árbol completo no es un árbol binario perfecto, pero el resultado obtenido es una cota superior de la suma de las alturas de los nodos en un árbol completo. Puesto que un árbol completo tiene entre 2^k y 2^{k+1} nodos, este teorema implica que esta suma es $O(n)$, donde n es el número de nodos.

Aunque el resultado obtenido es suficiente para demostrar que *construir_montículo* es lineal, la cota de la suma de las alturas no es suficientemente fuerte. Para un árbol completo con $n = 2^k$ nodos, la cota obtenida es cercana a $2n$. Puede demostrarse por inducción que la suma de las alturas es $n - b(n)$, donde $b(n)$ es el número de unos en la representación binaria de n .

6.4. Aplicaciones de las colas de prioridad

Ya hemos mencionado cómo usar colas de prioridad en el diseño de sistemas operativos. En el capítulo 9, veremos cómo se usan las colas de prioridad para lograr una implantación eficiente de varios algoritmos de grafos. Aquí se mostrará cómo usar las colas de prioridad para obtener soluciones a dos problemas.

6.4.1. El problema de la selección

El primer problema a examinar es el *problema de la selección* del capítulo 1. Recorremos que la entrada es una lista de n elementos, la cual puede estar ordenada totalmente, y un entero k . El problema de la selección es encontrar el k -ésimo elemento mayor.

En el capítulo 1 se dieron dos algoritmos, pero ninguno es muy eficiente. El primer algoritmo, al que llamaremos algoritmo 1A, consiste en leer los elementos de un arreglo y ordenarlos, devolviendo el elemento apropiado. Soportando un algoritmo de ordenación sencillo, el tiempo de ejecución es $O(n^2)$. El algoritmo alternativo, 1B, consiste en leer k elementos de un arreglo y ordenarlos. El menor es el de la k -ésima posición. Se procesan los elementos restantes uno por uno. Conforme llega un elemento se compara con el k -ésimo del arreglo. Si es mayor, entonces se elimina el k -ésimo elemento, y el nuevo se coloca en el lugar correcto entre los restantes $k-1$ elementos. Cuando termina el algoritmo, la respuesta es el elemento de la k -ésima posición. El tiempo de ejecución es $O(n \cdot k)$ (¿por qué?). Si $k = \lceil n/2 \rceil$, entonces ambos algoritmos son $O(n^2)$. Observe que para cualquier k , podemos

resolver el problema simétrico de encontrar el $(n-k+1)$ -ésimo elemento menor, así que $k = \lceil n/2 \rceil$ es realmente el caso más difícil de estos algoritmos. Éste también es el caso más interesante, ya que este valor de k se conoce como la *mediana*.

A continuación se dan dos algoritmos; ambos se ejecutan en $O(n \log n)$ en el caso extremo de $k = \lceil n/2 \rceil$, lo cual es una clara mejoría.

Algoritmo 6A

Por simplicidad, suponemos que nos interesa encontrar el k -ésimo elemento *menor*. El algoritmo es simple. Leemos los n elementos de un arreglo, aplicamos el algoritmo *construir_montículo* al arreglo, y por último efectuamos k operaciones *eliminar_mín*. El último elemento extraído del montículo será la respuesta. Debe quedar claro que cambiando la propiedad de orden de montículo podríamos resolver el problema original de encontrar el k -ésimo elemento *mayor*.

La corrección del algoritmo debe quedar clara. El tiempo en el peor caso es $O(n)$ para construir el montículo, si se usa *construir_montículo*, y $O(\log n)$ para cada operación *eliminar_mín*. Como hay k *eliminar_mín*, se obtiene un tiempo de ejecución total de $O(n + k \log n)$. Si $k = O(n/\log n)$, el tiempo de ejecución es dominado por la operación *construir_montículo* y es $O(n)$. Para valores más grandes de k , el tiempo de ejecución es $O(k \log n)$. Si $k = \lceil n/2 \rceil$, el tiempo de ejecución es $\Theta(n \log n)$.

Observe que si ejecutamos este programa para $k = n$ y registramos los valores en el orden en que abandonan el montículo, habremos ordenado el archivo de entrada en un tiempo $O(n \log n)$. En el capítulo 7 refinaremos esta idea para obtener un algoritmo de ordenación rápido, denominado *ordenación por montículos*.

Algoritmo 6B

Para el segundo algoritmo, volvemos al problema original y buscamos el k -ésimo elemento *mayor*. Usamos la idea del algoritmo 1B. En cualquier punto en el tiempo mantendremos un conjunto C de los k elementos mayores. Después de leer los primeros k elementos, cuando se lee uno nuevo, se le compara con el k -ésimo elemento mayor, el cual se identifica como C_k . Observe que C_k es el elemento menor en C . Si el nuevo elemento es mayor, reemplazará a C_k en C . C tendrá entonces un nuevo elemento menor, que puede ser o no el recién agregado. Al final de la entrada, se busca el menor elemento en C y se devuelve como respuesta.

En esencia, éste es el mismo algoritmo descrito en el capítulo 1. Aquí, no obstante, usaremos un montículo para implantar C . Los primeros k elementos se colocan en un montículo en un tiempo total $O(k)$ con una llamada a *construir_montículo*. El tiempo de procesar cada uno de los elementos restantes es $O(1)$, para comprobar si el elemento va en C , más $O(\log k)$, para eliminar C_k e insertar el elemento nuevo si es necesario. Así, el tiempo total es $O(k + (n-k) \log k) = O(n \log k)$. Este algoritmo también da una cota de $\Theta(n \log n)$ para encontrar la mediana.

En el capítulo 7, veremos cómo resolver este problema en un tiempo medio $O(n)$. En el capítulo 10 presentaremos un algoritmo elegante, aunque poco práctico, para resolver este problema en un tiempo $O(n)$ para el peor caso.

6.4.2. Simulación de eventos

En la sección 3.4.3 describiremos un problema importante para la construcción de colas. Recordaremos que tenemos un sistema, como podría ser un banco, al que llegan los clientes y esperan en una fila hasta que uno de k cajeros está disponible. La llegada de clientes depende de una función de distribución de probabilidades, como lo es el tiempo de servicio (el tiempo para ser atendido una vez que un cajero está disponible). Nos interesan algunas estadísticas como por ejemplo cuánto tiene que esperar en promedio un cliente, o qué tan larga puede ser la fila.

Con ciertas distribuciones de probabilidad y valores de k , estas respuestas pueden calcularse con exactitud. No obstante, conforme crece k , el análisis se dificulta considerablemente, tanto que se debe recurrir al uso de un computador para simular la operación del banco. En esta forma, los directores del banco pueden determinar cuántos cajeros se necesitan para asegurar un servicio razonablemente cómodo.

Una simulación consiste en el proceso de eventos. Los dos eventos aquí son (a) la llegada de un cliente y (b) la partida de un cliente, liberando así un cajero.

Podemos usar las funciones de probabilidad para generar un flujo de entrada consistente en pares ordenados de hora de llegada y tiempo de servicio para cada cliente, ordenados por la hora de llegada. No necesitamos usar la hora exacta del día. En cambio, podemos usar una cantidad unitaria, a la cual nos referiremos como un tic.

Una forma de hacer esta simulación es arrancar un reloj de simulación en cero tics. Entonces avanzamos el reloj un tic cada vez, para examinar si se presenta un evento. Si es así lo procesamos, o si son muchos eventos los procesamos y compilamos las estadísticas. La simulación termina cuando no hay clientes pendientes en el flujo de entrada y todos los cajeros están libres.

El problema con esta estrategia de simulación es que su tiempo de ejecución no depende del número de clientes o eventos (hay dos eventos por cliente), sino del número de tics, que de hecho no es parte de la entrada. Para ver la importancia de esto, supongamos que cambiamos las unidades de reloj a militics y se multiplican todos los tiempos en la entrada por mil. El resultado sería que la simulación tarde mil veces más.

La clave para evitar este problema es avanzar el reloj al siguiente evento en cada etapa. En términos conceptuales es fácil hacer esto. En cualquier punto, el siguiente evento que puede ocurrir es que (a) llega el siguiente cliente en el archivo de entrada, o (b) uno de los clientes de un cajero se va. Como todos los tiempos en que ocurrirán los eventos están disponibles, sólo necesitamos encontrar el evento más próximo a ocurrir y procesarlo.

Si el evento es una partida, el procesamiento incluye recoger estadísticas para el cliente atendido y revisar la fila (cola) para ver si hay otro cliente esperando. De ser así, se agrega ese cliente, se procesa cualquier estadística requerida, se calcula el tiempo en que el cliente salió, y se agrega esa partida al conjunto de eventos en espera de ocurrir.

Si el evento es una llegada, se busca un cajero disponible. Si no hay ninguno, se coloca la llegada en la fila (cola); de otra forma se le asigna al cliente un cajero,

se calcula la hora de partida del cliente y se agrega la partida al conjunto de eventos en espera de ocurrir.

La fila de espera de clientes se puede implantar como una cola. Puesto que necesitamos encontrar el evento *más próximo* en el futuro, es apropiado que el conjunto de partidas en espera de ocurrir se organice en una cola de prioridad. Así, el siguiente evento es la siguiente llegada o partida (la que ocurra antes); ambas están fácilmente disponibles.

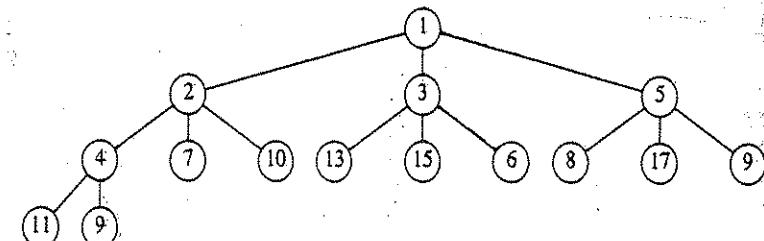
Por tanto, escribir las rutinas de simulación es directo, aunque posiblemente consuma tiempo. Si hay C clientes (y por ello $2C$ eventos) y k cajeros, el tiempo de ejecución de la simulación sería $O(C \log(k+1))$ ^t porque el cálculo y procesamiento de cada evento tarda $O(\log M)$, donde $M = k+1$ es el tamaño del montículo.

6.5. Montículos- d

Los montículos binarios son tan sencillos que casi siempre se usan cuando se necesitan colas de prioridad. Una generalización simple es un *montículo- d* , que es exactamente como un montículo binario, excepto que todos los nodos tienen d hijos (así, un montículo binario es un montículo-2). La figura 6.19 muestra un montículo-3.

Observe que un montículo- d es mucho más bajo que un montículo binario, mejorando el tiempo de ejecución de *insertar* a $O(\log n)$. No obstante, la operación *eliminar_mín* es más costosa, porque aun cuando el árbol es más bajo, debe encontrarse el menor de los d hijos, lo cual requiere $d - 1$ comparaciones con un algoritmo estándar. Esto eleva el tiempo de esta operación a $O(d \log n)$. Si d es una constante, ambos tiempos de ejecución son, por supuesto, $O(\log n)$. Más aún, aunque todavía se puede usar un arreglo, ahora las multiplicaciones y divisiones para encontrar hijos y padres son con d , lo cual incrementa bastante el tiempo de ejecución, porque ya no podemos implantar la división con un desplazamiento de bits. Los montículos- d son interesantes en teoría, porque hay muchos algoritmos donde el número

Figura 6.19 Un montículo- d



^t Utilizamos $O(C \log(k+1))$ en vez de $O(C \log k)$ para evitar confusión en el caso $k=1$.

de inserciones es mucho mayor que el número de *eliminar_mín* (así es posible un incremento teórico en la velocidad). También son de interés cuando la cola de prioridad es demasiado grande para caber completa en memoria principal. En este caso puede ser ventajoso un montículo- d en la misma forma que los árboles B.

La debilidad más evidente de la implantación de montículos, aparte de la incapacidad de efectuar *buscar*, es que la combinación de dos montículos en uno es una operación difícil. Esta operación adicional se conoce como *fusionar*. Hay muy pocas formas de implantar montículos para que el tiempo de ejecución de un *fusionar* sea $O(\log n)$. Ahora estudiaremos tres estructuras de datos de diferente complejidad que permiten la operación eficiente *fusionar*. Dejaremos cualquier análisis complicado hasta el capítulo 11.

6.6. Montículos a izquierda

Parece difícil diseñar una estructura de datos que permita la fusión eficiente (esto es, que procese *fusionar* en un tiempo $O(n)$) y use sólo un arreglo, como en un montículo binario. La razón de esto es que la fusión parece requerir la copia de un arreglo en otro en un tiempo $\Theta(n)$ para montículos de igual tamaño. Por ello todas las estructuras de datos avanzadas que permiten la fusión eficiente requieren el uso de apuntadores. En la práctica, podemos esperar que esto hará más lentas las demás operaciones; la manipulación de apuntadores suele consumir más tiempo que la multiplicación y la división por dos.

Igual que un montículo binario, un *montículo a izquierda* tiene tanto una propiedad estructural como una propiedad de orden. En efecto, un montículo a izquierda, como prácticamente todos los montículos usados, tiene la misma propiedad de orden de montículo vista antes. Más aún, un montículo a izquierda es también un árbol binario. La única diferencia entre un montículo a izquierda y uno binario es que el primero no está perfectamente equilibrado, sino que de hecho intenta ser muy desequilibrado.

6.6.1. Propiedad de montículo a izquierda

Definimos la *longitud de camino nulo*, *lcn(X)*, de cualquier nodo X como la longitud del camino más corto entre X y un nodo sin dos hijos. Así, la *lcn* de un nodo con cero o un hijo es 0, mientras que *lcn(nil) = -1*. En el árbol de la figura 6.20, las longitudes de camino nulo se indican dentro de los nodos del árbol.

Observe que la longitud de camino nulo de cualquier nodo es 1 más que la mínima longitud de camino nulo de sus hijos. Esto se aplica a los nodos con menos de dos hijos porque la longitud de camino nulo de *nil* es -1.

La propiedad de montículo a izquierda es que, para cada nodo X en el montículo, la longitud de camino nulo del hijo izquierdo es al menos tan grande como la del hijo derecho. Esta propiedad la satisface sólo uno de los árboles de la figura 6.20, el de la izquierda. En realidad esta propiedad va más allá, para asegurar que el árbol está desequilibrado, porque claramente desvía al árbol para que esté a más profundidad en el lado izquierdo.

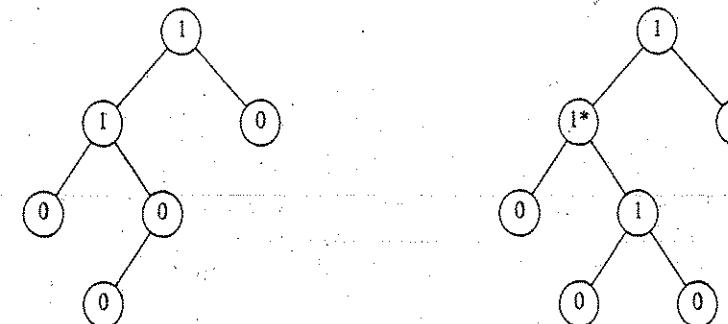


Figura 6.20 Longitudes de camino nulo para dos árboles; sólo el árbol de la izquierda es a izquierda

didad en el lado izquierdo. En efecto, es posible (y de hecho preferible para facilitar la fusión) un árbol consistente en un camino largo de nodos izquierdos; de aquí el nombre de *montículo a izquierda*.

Debido a que los montículos a izquierda tienden a tener caminos izquierdos profundos, se infiere que el camino derecho debe ser corto. En efecto, el camino derecho en un montículo a izquierda es tan corto como cualquiera en el montículo. De otra forma, podría haber un camino que pasa por algún nodo X y pasa por el hijo izquierdo. Entonces X violaría la propiedad a izquierda.

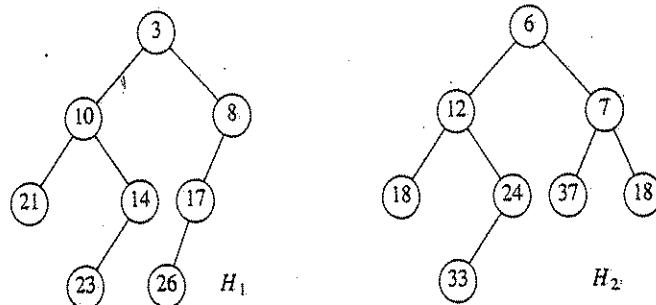
TEOREMA 6.2.

Un árbol a izquierda con d nodos en el camino derecho debe tener al menos $2^d - 1$ nodos.

DEMOSTRACIÓN:

La demostración es por inducción. Si $d = 1$, debe haber al menos un nodo en el árbol. Si no, suponga que el teorema es cierto para 1, 2, ..., d . Considere un árbol a izquierda con $d + 1$ nodos en el camino derecho. Entonces la raíz tiene un subárbol derecho con d nodos en el camino derecho y un subárbol izquierdo con al menos d nodos en el camino derecho (si no, sería a izquierda). Aplicando la hipótesis inductiva a estos subárboles se obtiene un mínimo de $2^d - 1$ nodos en cada subárbol. Esto más la raíz da al menos $2^{d+1} - 1$ nodos en el árbol, con lo cual se demuestra el teorema.

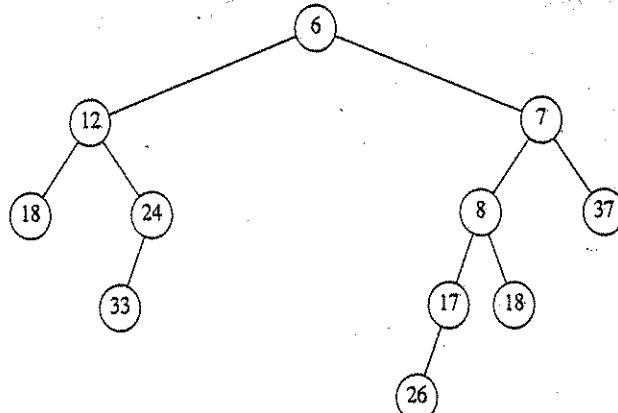
De este teorema se sigue inmediatamente que un árbol a izquierda de n nodos tiene un camino derecho con a lo más $\log(n + 1)$ nodos. La idea general de las operaciones sobre montículos a izquierda consiste en efectuar todo el trabajo sobre el camino derecho, el cual es seguro que es corto. La única parte complicada es que efectuar varias *insertar* y *fusionar* sobre el camino derecho podría destruir la propiedad de montículo a izquierda. Resulta extremadamente fácil restaurar la propiedad.

Figura 6.21 Dos montículos a izquierda M_1 y M_2

6.6.2. Operaciones sobre montículos a izquierda

La operación fundamental sobre montículos a izquierda es la fusión. Observe que la inserción sólo es un caso especial de la fusión, ya que se puede ver la inserción como una *fusión* de un montículo de un nodo con un montículo más grande. Primero daremos una solución recursiva sencilla y después mostraremos cómo hacerlo sin recursión. Nuestra entrada es los dos montículos a izquierda M_1 y M_2 de la figura 6.21. Revise que estos montículos realmente sean a izquierda. Observe que los elementos más pequeños están en las raíces. Además del espacio para los datos y los apuntadores izquierdo y derecho, cada celda contendrá una entrada que indica la longitud de camino nulo.

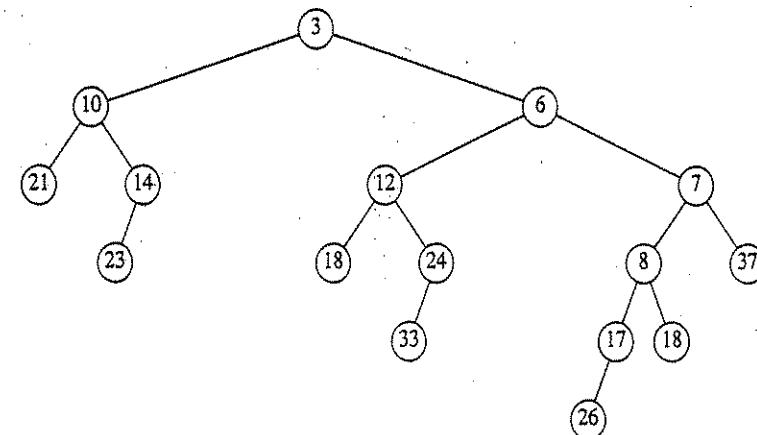
Si alguno de los dos montículos está vacío, entonces podemos devolver el otro montículo. Si no es así, para fusionar los dos montículos comparamos sus raíces.

Figura 6.22 Resultado de la fusión de M_2 con el submontículo derecho de M_1 

Primero, recursivamente se combina el montículo de la raíz mayor con el submontículo derecho del montículo con la raíz menor. En este ejemplo, esto significa que fusionamos M_2 con el submontículo de M_1 de raíz 8, obteniendo el montículo de la figura 6.22.

Puesto que este árbol se forma recursivamente, y aún no hemos terminado la descripción del algoritmo, no podemos en este momento demostrar cómo se obtuvo este montículo. No obstante, es razonable suponer que el árbol resultante es un montículo a izquierda, porque se obtuvo mediante un paso recursivo. Esto se parece mucho a la hipótesis inductiva en la demostración por inducción. Puesto que podemos manejar el caso base (lo que ocurre cuando un árbol está vacío), podemos suponer que el paso recursivo funciona en tanto podamos terminar la fusión; ésta es la regla 3 de la recursión presentada en el capítulo 1. Ahora hacemos de este montículo nuevo el hijo derecho de la raíz de M_1 (véase la figura 6.23).

Aunque el montículo resultante satisface la propiedad de orden de montículo, no es a izquierda, porque el subárbol izquierdo de la raíz tiene una longitud de camino nulo de 1, mientras que el subárbol derecho tiene una longitud de camino nulo de 2. Así, la propiedad de montículo a izquierda se viola en la raíz. Sin embargo, es fácil ver que el resto del árbol debe ser a izquierda. El subárbol derecho de la raíz es a izquierda, gracias al paso recursivo. El subárbol izquierdo de la raíz no cambió, así que también debe seguir siendo a izquierda. Por lo tanto, sólo necesitamos reparar la raíz. Podemos hacer a izquierda todo el árbol con sólo intercambiar los hijos izquierdo y derecho de la raíz (figura 6.24) actualizando la longitud de camino nulo —la nueva longitud de camino nulo es 1 más la longitud de camino nulo del nuevo hijo derecho— completando el *fusionar*. Observe que si no se actualiza la longitud de camino nulo, entonces todas las longitudes de camino nulo

Figura 6.23 Resultado de enlazar el montículo a izquierda de la figura anterior como hijo derecho de M_1 

serán cero, y el montículo no será a izquierda sino aleatorio. En este caso, el algoritmo funcionará, pero la cota de tiempo no será válida.

La descripción del algoritmo se traduce directamente en el código. La definición de tipos (figura 6.25) es la misma que la del árbol binario, excepto que ésta se aumenta con el campo *lcn* (longitud de camino nulo).

La rutina para fusionar (figura 6.26) es un manejador diseñado para eliminar los casos especiales y asegurar que M_1 tenga la raíz más pequeña. La fusión real se efectúa en *fusionar1* (figura 6.27).

El tiempo para ejecutar la fusión es proporcional a la suma de la longitud de los caminos derechos, porque se realiza trabajo constante en cada nodo visitado durante las llamadas recursivas. Así obtenemos una cota de tiempo $O(\log n)$ para fusionar dos montículos a izquierda. También podemos efectuar esta operación no recursivamente en dos pasadas. En la primera pasada creamos un árbol nuevo fusionando los caminos derechos de ambos montículos. Para hacerlo, acomodamos los nodos

Figura 6.24 Resultado del intercambio de los hijos de la raíz de M_1

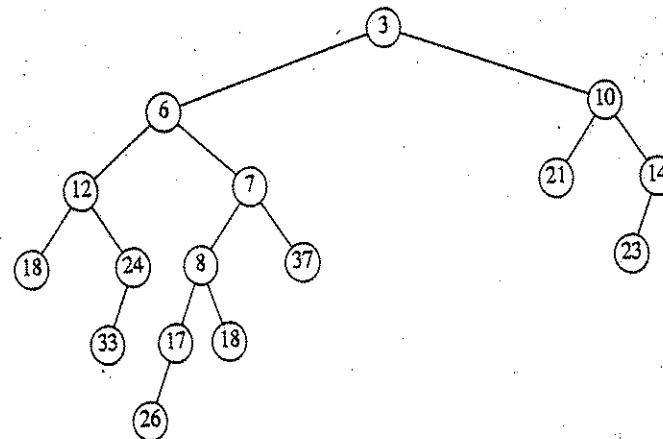


Figura 6.25 Declaraciones de tipos para montículos a izquierda

```

type
  ap_arbol = ^nodo_arbol;
  nodo_arbol = record
    elemento: tipo_elemento;
    izq : ap_arbol;
    der : ap_arbol;
    lcn : integer;
  end;
  COLA_DE_PRIORIDAD = ap_arbol;
  
```

```

function fusionar(m1, m2: COLA_DE_PRIORIDAD): COLA_DE_PRIORIDAD;
begin
  (1) if m1 = nil then
      fusionar := m2
    else
      (2) if m2 = nil then
          fusionar := m1
        else
          (3) if m1^.elemento < m2^.elemento then
              fusionar := fusionar1(m1, m2)
            else
              (4) fusionar := fusionar1(m2, m1);
  end;
  
```

Figura 6.26 Rutina controladora para fusionar montículos a izquierda

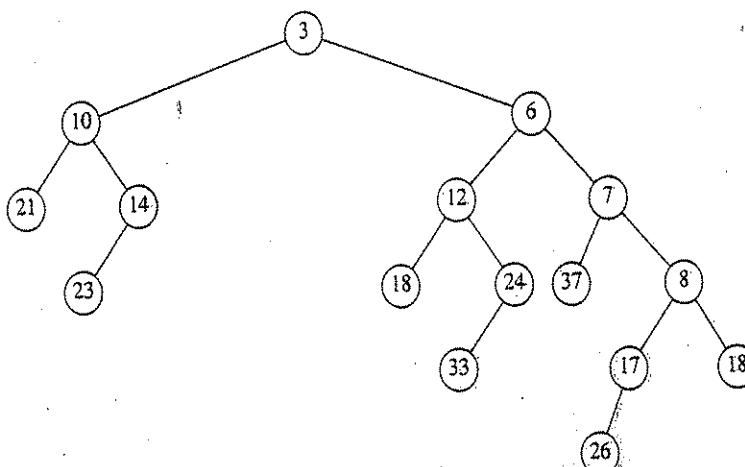
de los caminos derechos de M_1 y M_2 en orden, conservando sus respectivos hijos izquierdos. En este ejemplo, el camino derecho nuevo es 3, 6, 7, 8, 18, y el árbol resultante se muestra en la figura 6.28. Se hace una segunda pasada sobre el montículo, y los intercambios de hijos se realizan en los nodos que violan la propiedad de montículo a izquierda. En la figura 6.28 hay un intercambio en los nodos 7 y 3; y se obtiene el mismo árbol que antes. La versión no recursiva es más fácil de ver pero más difícil de codificar. Se le deja al lector demostrar que los procedimientos recursivo y no recursivo hacen lo mismo.

Como se mencionó antes, podemos llevar a cabo inserciones haciendo del elemento que se ha de insertar un montículo de un solo nodo y realizando un *fusionar*. Para efectuar un *eliminar_mín*, sólo destruimos la raíz, creando dos

Figura 6.27 Rutina real para combinar montículos a izquierda

```

{Para fusionar1, m1 tiene la menor raíz, m1 y m2 no son nulos}
function fusionar1(m1, m2: COLA_DE_PRIORIDAD):
  COLA_DE_PRIORIDAD;
begin
  (1) if m1^.izq = nil then
      (nodo simple)
      m1^.izq := m2
      {m1^.der ya es nil, m1^.lcn ya es 0}
    else
      begin
        (2) m1^.der := fusionar(m1^.der, m2);
        if m1^.izq^.lcn < m1^.der^.lcn then
          intercambiar(m1^.izq, m1^.der);
        (3) m1^.lcn := m1^.der^.lcn + 1;
      end;
    (4) fusionar1 := m1;
  end;
  
```

Figura 6.28 Resultado de fusionar los caminos derechos de M_1 y M_2 .

montículos que luego se puedan fusionar. Así, el tiempo para realizar un *eliminar_mín* es $O(\log n)$. Estas dos rutinas están codificadas en las figuras 6.29 y 6.30.

La llamada a *dispose* en la línea 4 de la figura 6.30 puede parecer peligrosa, pero en realidad es correcta. La llamada no destruye la variable M ; en cambio, indica que la celda a la cual apunta puede ser usada. Esa celda se coloca en la lista de disponibles. Entonces se hace que M , que es un apuntador, apunte a otra cosa en la línea 5. También, observe cómo las cabeceras de estas rutinas son idénticas a las

Figura 6.29 Rutina de inserción para árboles a izquierda

```
procedure insertar(x: tipo_elemento; var M: COLA_DE_PRIORIDAD);
  var nodo_simple: ap_arbol;

begin
  begin
    new(nodo_simple);
    if nodo_simple = nil then
      error_fatal('¡¡¡Memoria agotada!!!')
    else
      begin
        nodo_simple^.elemento := x;
        nodo_simple^.izq := nil;
        nodo_simple^.der := nil;
        nodo_simple^.lcn := 0;
        M := fusionar(nodo_simple, M);
      end;
  end;
end;
```

```
function eliminar_mín(var M: COLA_DE_PRIORIDAD): tipo_elemento;
  var montículo_izq, montículo_der: COLA_DE_PRIORIDAD;
begin
  [1]  eliminar_mín := M^.elemento;
  [2]  montículo_izq := M^.izq;
  [3]  montículo_der := M^.der;
  [4]  dispose(M);
  [5]  M := fusionar(montículo_izq, montículo_der);
end;
```

Figura 6.30 Rutina *eliminar_mín* para montículos a izquierda

usadas en la implantación de montículos binarios. Se podría usar cualquiera de los dos paquetes de colas de prioridad, y la implantación sería completamente transparente para las rutinas que llaman.

Para concluir, podemos construir un montículo a izquierda en un tiempo $O(n)$ creando un montículo binario (obviamente con una implantación con apunadores). Aunque es evidente que un montículo binario es a izquierda, no necesariamente es la mejor solución, porque el montículo obtenido es el peor montículo a izquierda posible. Más aún, recorrer el árbol en orden de nivel inverso no es tan fácil con apunadores. El efecto *construir_montículo* se puede obtener recursivamente construyendo los subárboles izquierdo y derecho y después filtrando desde la raíz hacia abajo. Los ejercicios contienen una solución alternativa.

6.7. Montículos oblicuos

Un *montículo oblicuo* es una versión autoajustable de un montículo a izquierda, y es increíblemente fácil implantarlo. La relación de los montículos oblicuos con los montículos a izquierda es análoga a la relación entre los árboles desplegados y los árboles AVL. Los montículos oblicuos son árboles binarios con orden de montículo, pero sin restricción estructural. A diferencia de los montículos a izquierda, no se mantiene ninguna información acerca de la longitud de camino nulo de cualquier nodo. El camino derecho de un montículo oblicuo puede ser arbitrariamente largo en cualquier instante, así que el tiempo de ejecución de todas las operaciones en el peor caso es $O(n)$. No obstante, como con los árboles desplegados, se puede demostrar (véase el capítulo 11) que para cualesquiera m operaciones consecutivas, el tiempo de ejecución total en el peor caso es $O(m \log n)$. Así, los montículos oblicuos tienen costo amortizado de $O(\log n)$ por operación.

Como con los montículos a izquierda, la operación fundamental sobre montículos oblicuos es la fusión. Una vez más la rutina *fusionar* es recursiva, y realiza exactamente las mismas operaciones que antes, con una excepción. La diferencia es que en los montículos a izquierda revisamos los hijos izquierdo y derecho para ver si satisfacen la propiedad de orden de montículo a izquierda y los intercambiamos si no es así. Con los montículos oblicuos, el intercambio es incondicional: *siempre* lo hacemos, con la única excepción de que el menor de todos los nodos en los caminos

de la derecha no tenga sus hijos intercambiados. Esta excepción es lo que ocurre en la implantación recursiva natural, así que en realidad no es un caso especial en absoluto. Más aún, no es necesario probar los límites, pero como está garantizado que este nodo no tiene un hijo derecho, sería tonto efectuar el intercambio para ponerle uno. (En nuestro ejemplo este nodo no tiene ningún hijo, por lo que no nos preocupamos por ello.) De nuevo, supongamos que las entradas son los mismos dos montículos de antes (figura 6.31).

Si combinamos recursivamente M_2 con el submontículo de M_1 cuya raíz es 8, obtendremos el montículo de la figura 6.32.

De nuevo, esto se hace recursivamente, así que por la tercera regla de la recursión (sección 1.3) no es necesario preocuparse de cómo se obtuvo. Ocurre que el montículo es a izquierda, pero no hay ninguna garantía de que siempre sea así.

Figura 6.31 Dos montículos oblicuos M_1 y M_2

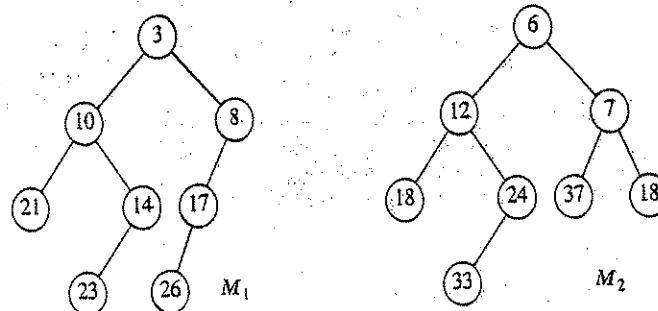


Figura 6.32 Resultado de fusionar M_2 con el submontículo derecho de M_1

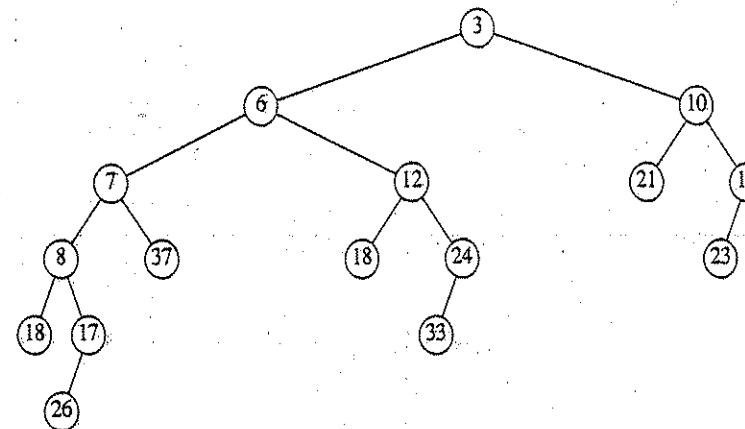
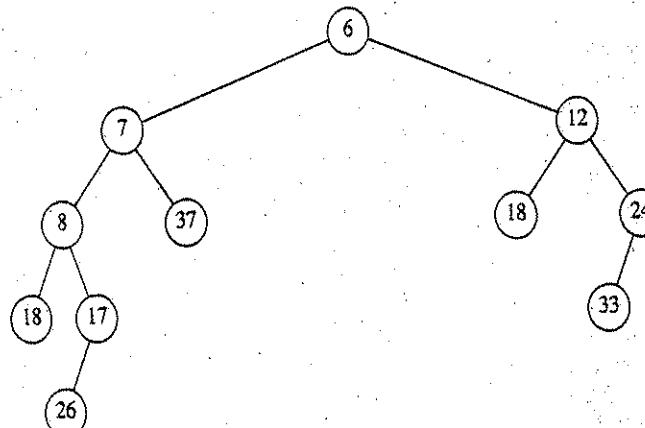


Figura 6.33 Resultado de fusionar los montículos oblicuos M_1 y M_2

Hacemos de este montículo el nuevo hijo izquierdo de M_1 , y el anterior hijo izquierdo de M_1 pasa a ser el nuevo hijo derecho (véase la figura 6.33).

El árbol completo es a izquierda, pero es fácil ver que esto no siempre es cierto: al insertar 15 en este nuevo montículo se destruiría la propiedad de izquierda.

Podemos efectuar todas las operaciones sin recursión, como con los montículos a izquierda, fusionando los caminos derechos e intercambiando los hijos izquierdo y derecho en cada nodo del camino derecho, con excepción del último. Después de unos pocos ejemplos queda claro que como todos los nodos, excepto el último del camino derecho, tienen sus hijos intercambiados, el efecto neto es que éste se convierte en el nuevo camino izquierdo (para convencerse, véase el ejemplo anterior). Esto hace muy fácil fusionar visualmente dos montículos oblicuos.

La implantación de montículos oblicuos se deja como ejercicio (trivial). Los montículos oblicuos tienen la ventaja de no requerir espacio adicional para mantener las longitudes de los caminos, ni comprobaciones para determinar cuándo intercambiar los hijos. Es un problema abierto determinar con precisión la longitud esperada del camino derecho de los montículos a izquierda y oblicuo (el último es, sin duda, más difícil). Tal comparación haría más fácil determinar si la ligera pérdida de información sobre el equilibrio, se compensa con la ausencia de comprobaciones.

6.8. Colas binomiales

Aunque los montículos a izquierda y oblicuo permiten la fusión, inserción y *eliminar_mín* en un tiempo $O(\log n)$ por operación, hay espacio para mejoras porque sabemos que los montículos binarios permiten la inserción en tiempo *promedio constante* por operación. Las colas binomiales permiten estas tres operaciones en un tiempo $O(\log n)$ por operación para el peor caso, pero las inserciones tardan un tiempo constante en promedio.

6.8.1. Estructura de cola binomial

Las colas binomiales difieren de todas las implantaciones de colas de prioridad vistas hasta ahora en que una cola binomial no es un árbol con orden de montículo, sino una colección de árboles con orden de montículo, llamada *bosque*. Cada uno de los árboles con orden de montículo es una forma restringida denominada *árbol binomial* (el nombre será obvio después). A lo más hay un árbol binomial para cada altura. Un árbol binomial de altura cero es un árbol de un nodo; un árbol binomial, B_k , de altura k se forma uniendo un árbol binomial, B_{k-1} , a la raíz de otro árbol binomial, B_{k-1} . La figura 6.34 muestra los árboles binomiales B_0 , B_1 , B_2 , B_3 y B_4 .

Probablemente sea obvio a partir del diagrama que un árbol binomial B_k consiste en una raíz con hijos B_0, B_1, \dots, B_{k-1} . Los árboles binomiales de altura k tienen exactamente 2^k nodos, y el número de nodos a la profundidad d es el coeficiente binomial $\binom{n}{d}$. Si imponemos el orden de montículo a los árboles binomiales y permitimos a lo más un árbol binomial de cualquier altura, podemos representar una cola de prioridad de cualquier tamaño por medio de una colección única de

Figura 6.34 Árboles binomiales B_0, B_1, B_2, B_3 y B_4

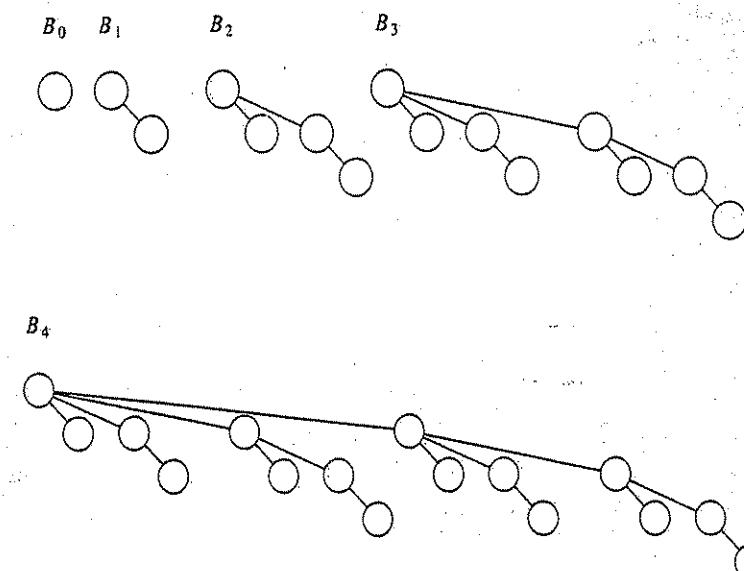
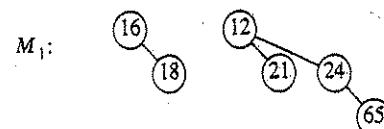


Figura 6.35 Cola binomial M_1 con seis elementos



árboles binomiales. Por ejemplo, una cola de prioridad de tamaño 13 podría ser representada por el bosque B_3, B_2, B_0 . Podríamos escribir esta representación como 1101, que no sólo representa el 13 binario sino que también representa el hecho de que B_3, B_2 y B_0 están presentes en la representación y B_1 no.

Como ejemplo, una cola de prioridad de seis elementos se podría representar como en la figura 6.35.

6.8.2. Operaciones sobre colas binomiales

Así, el elemento mínimo se puede encontrar rastreando en las raíces de todos los árboles. Puesto que hay a lo más $\log n$ árboles diferentes, el mínimo se puede encontrar en un tiempo $O(\log n)$. Alternativamente, podemos seguir conociendo el mínimo y ejecutar la operación en un tiempo $O(1)$, si recordamos actualizar el mínimo cuando cambia durante otras operaciones.

La fusión de dos colas binomiales es una operación conceptualmente sencilla, que describiremos con un ejemplo. Consideraremos las dos colas binomiales M_1 y M_2 con seis y siete elementos, respectivamente, de la figura 6.36.

En esencia, la fusión se realiza juntando las dos colas. Sea M_3 la nueva cola binomial. Como M_1 no tiene ningún árbol binomial de altura 0 y M_2 sí, podemos usar el árbol binomial de altura 0 de M_2 como parte de M_3 . Después, se agregan los árboles binarios de altura 1. Puesto que M_1 y M_2 tienen árboles binomiales de altura 1, los fusionamos haciendo de la raíz mayor un subárbol de la menor, creando así un árbol binomial de altura 2 (figura 6.37). Así, M_3 no tendrá un árbol binomial de altura 1. Ahora hay tres árboles binomiales de altura 2, los originales de M_1 y M_2

Figura 6.36 Dos colas binomiales M_1 y M_2

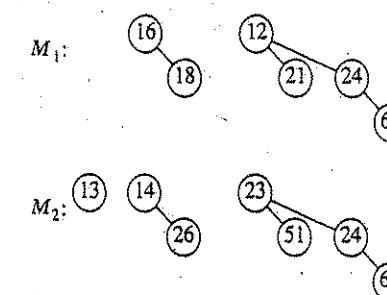
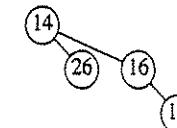
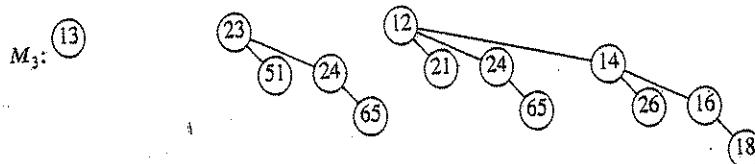


Figura 6.37 Fusión de dos árboles B_1 en M_1 y M_2



Figura 6.38 Cola binomial M_3 : resultado de fusionar M_1 y M_2

más el árbol formado en el paso anterior. Dejamos un árbol binomial de altura 2 en M_3 y fusionamos los otros dos, creando un árbol binomial de altura 3. Puesto que M_1 y M_2 no tienen árboles de altura 3, este árbol se vuelve parte de M_3 y concluimos. La cola binomial resultante se muestra en la figura 6.38.

Puesto que la fusión de dos colas binomiales toma un tiempo constante con casi cualquier implantación razonable y hay $O(\log n)$ árboles binomiales, la fusión tarda un tiempo $O(\log n)$ en el peor caso. Para hacer eficiente esta implantación, necesitamos mantener ordenados los árboles en la cola binomial, de acuerdo con su altura, lo que ciertamente es fácil hacer.

La inserción es sólo un caso especial de fusión, ya que simplemente creamos un árbol de un nodo y efectuamos la fusión. El tiempo del peor caso de esta operación es también $O(\log n)$. Más precisamente, si la cola de prioridad en la cual el elemento se está insertando tiene la propiedad de que el menor árbol binomial no existente es B_i , el tiempo de ejecución es proporcional a $i+1$. Por ejemplo, M_3 (figura 6.38) carece de un árbol binomial de altura 1, así que la inserción terminará en dos pasos. Como la probabilidad de que cada árbol esté en una cola binomial de $\frac{1}{2}$, es de esperar que una inserción termine en dos pasos, de modo que el tiempo medio sea constante. Además, un análisis sencillo demostrará que efectuar n operaciones *insertar* sobre una cola binomial inicialmente vacía se logrará en un tiempo para el peor caso de $O(n)$. En efecto, es posible hacer esta operación usando sólo $n-1$ comparaciones; dejamos esto como ejercicio.

Como ejemplo, en las figuras 6.39 a 6.45 mostramos las colas binomiales que se forman al insertar del 1 al 7 en orden. La inserción de 4 manifiesta ser un caso malo. Fusionamos 4 con B_0 para obtener un árbol nuevo de altura 1. Entonces fusionamos este árbol con B_1 , obteniendo un árbol de altura 2, que es la nueva cola de prioridad. Contamos esto como si fueran tres pasos (dos fusiones de árboles más el caso de paro). La siguiente inserción después de insertar 7 es otro caso malo y requerirá tres fusiones de árboles.

Se puede efectuar un *eliminar_mín* encontrando primero el árbol binomial con la raíz menor. Sea este árbol B_k , y sea M la cola de prioridad original. Retiramos el árbol binomial B_k del bosque de árboles en M , formando la nueva cola binomial M' . También eliminamos la raíz de B_k , creándose así árboles binomiales B_0, B_1, \dots, B_{k-1} los cuales colectivamente forman la cola de prioridad M'' . La operación termina al fusionar M' y M'' .

Figura 6.39 Despues de insertar 1



Figura 6.40 Despues de insertar 2

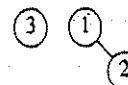


Figura 6.41 Despues de insertar 3

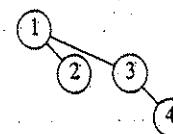


Figura 6.42 Despues de insertar 4

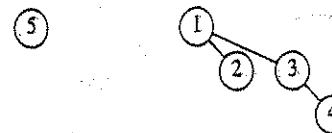


Figura 6.43 Despues de insertar 5

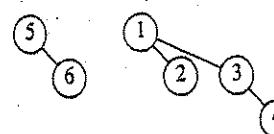


Figura 6.44 Despues de insertar 6

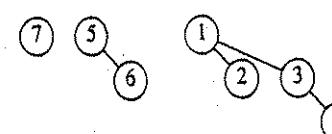
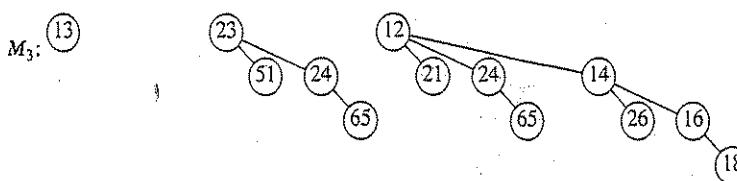
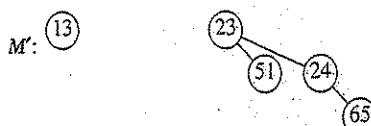
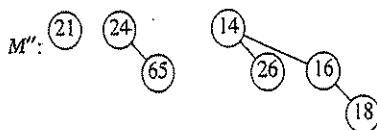
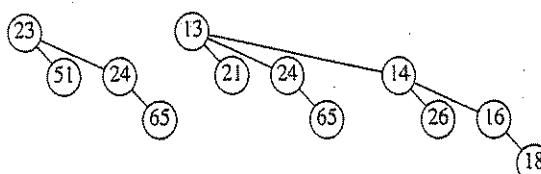


Figura 6.45 Despues de insertar 7

Figura 6.46 Cola binomial M_3 Figura 6.47 Cola binomial M' con todos los árboles binomiales en M_3 excepto B_3

Por ejemplo, supongamos que efectuamos *eliminar_mín* sobre M_3 , que se muestra otra vez en la figura 6.46. La raíz mínima es 12, así que obtenemos las dos colas de prioridad M' y M'' de las figuras 6.47 y 6.48. La cola binomial que resulta de fusionar M' y M'' es la respuesta final, que mostramos en la figura 6.49.

Para el análisis, observe primero que la operación *eliminar_mín* descompone la cola binomial original en dos. Se tarda un tiempo $O(\log n)$ en buscar el árbol con el elemento mínimo y en crear las colas M' y M'' . La fusión de las dos colas tarda un tiempo $O(\log n)$, así que toda la operación *eliminar_mín* lleva un tiempo $O(\log n)$.

Figura 6.48 Cola binomial $M'': B_3$ con la eliminación del 12Figura 6.49 Resultado de *eliminar_mín*(M_3)

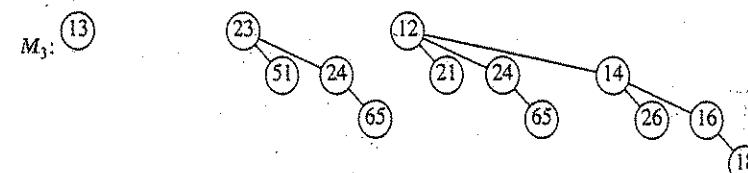
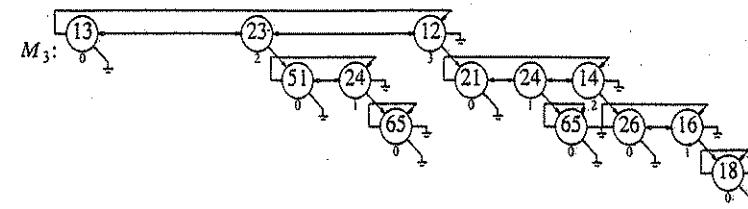
6.8.3: Implementación de colas binomiales

La operación *eliminar_mín* requiere la capacidad de buscar rápidamente todos los subárboles de la raíz, por lo que necesitamos la representación estándar de árboles generales: los hijos de cada nodo se conservan en una lista enlazada, y cada nodo tiene un apuntador a su primer hijo (si existe). Esta operación también requiere que los hijos se ordenen de acuerdo con el tamaño de sus subárboles, esencialmente en la misma forma en que los hemos dibujado. La razón de esto es que cuando se efectúa un *eliminar_mín* los hijos formarán la cola binomial M' .

También es necesario asegurarse de que sea fácil fusionar dos árboles. Dos árboles binomiales se pueden fusionar sólo si tienen el mismo tamaño, así que, si esto ha de hacerse eficientemente, el tamaño del árbol debe almacenarse en la raíz. También, cuando se fusionan dos árboles, uno de ellos se agrega al otro como hijo. Como este árbol nuevo será el último hijo (pues será el subárbol más grande), debemos poder mantener eficientemente el rastro del último hijo de cada nodo. Sólo entonces podremos fusionar dos árboles binarios con eficiencia, y con ello dos colas binomiales. Una forma de hacerlo es usar una lista circular con enlace doble. En esta lista, el hermano izquierdo del primer hijo será el último hijo. El hermano derecho del último hijo podría definirse como el primer hijo, pero sería más fácil definirlo como *nil*. Esto facilita la comprobación de que el hijo al que apunta sea el último.

Para resumir, cada nodo en una cola binomial contendrá los datos, el primer hijo, los hermanos izquierdo y derecho y el número de hijos (a lo que llamaremos el *rango*). Puesto que una cola binomial sólo es una lista de árboles, podemos usar un apuntador al árbol más pequeño como la referencia a la estructura de datos.

La figura 6.51 muestra cómo representar la cola binomial de la figura 6.50. La figura 6.52 presenta las declaraciones de tipos de un nodo en el árbol binomial.

Figura 6.50 Cola binomial M_3 dibujada como un bosqueFigura 6.51 Representación de la cola binomial M_3 

```

type
  ap_arbol = ^nodo_arbol;
  nodo_arbol = record
    elemento: tipo_elemento;
    herm_i: ap_arbol;
    herm_d: ap_arbol;
    p_hijo: ap_arbol;
    rango: integer;
  end;
  COLA_DE_PRIORIDAD = ap_arbol;

```

Figura 6.52 Declaración de tipos de la cola binomial

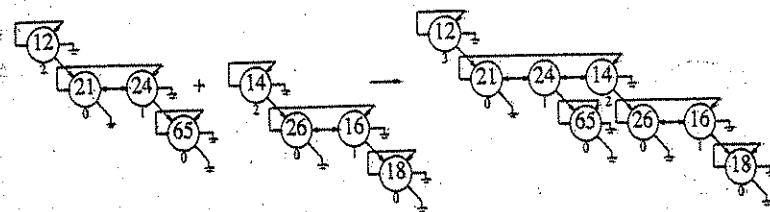


Figura 6.53 Fusión de dos árboles binomiales

A fin de fusionar dos colas binomiales, necesitamos una rutina para fusionar dos árboles binomiales del mismo tamaño. La figura 6.53 muestra cómo cambian los apuntadores cuando se fusionan dos árboles binomiales. Primero, la raíz del árbol nuevo adquiere un hijo, lo que exige actualizar su rango. Después, es necesario cambiar varios apuntadores para unir un árbol a la lista de hijos de la raíz del otro. El código para esto es sencillo y se presenta en la figura 6.54.

La rutina para fusionar dos colas binomiales es relativamente sencilla. Se usa la recursión para mantener pequeño el tamaño del código; un procedimiento no recursivo dará un mejor rendimiento, lo cual se deja para el ejercicio 6.32. Suponemos que el procedimiento *extraer(A, M)*, elimina el primer árbol de la cola de prioridad *M*, colocando el árbol en *A*. Supongamos que el árbol binomial menor está en *M₁*, pero no en *M₂*. Entonces, para fusionar *M₁*, retiramos el primer árbol de *M₁* y le agregamos el resultado de fusionar el resto de *M₁* con *M₂*. Si el árbol más pequeño está tanto en *M₁* como en *M₂*, eliminamos ambos árboles y los fusionamos, para obtener una cola binomial de un árbol *M'*. Entonces fusionamos el resto de *M₁* y *M₂* y fusionamos este resultado con *M'*. Esta estrategia se ilustra en la figura 6.55. Las otras rutinas son implantaciones directas, que dejamos como ejercicios.

Es posible extender las colas binomiales para poder contar con algunas operaciones no estándar que permiten los montículos binarios, como *decrementar_llave* y *eliminar*, cuando se conoce la posición del elemento afectado. Un *decrementar_llave* es un *filtrado ascendente*, que se puede efectuar en un tiempo $O(\log n)$ si agregamos un campo a cada nodo que apunte a su padre. Se puede realizar un *eliminar* arbitrario *decrementar_llave* y *eliminar_mín* en un tiempo $O(\log n)$.

```

function fusionar_arbol(A1, A2: COLA_DE_PRIORIDAD): COLA_DE_PRIORIDAD;
begin
  if A1^.elemento > A2^.elemento then
    intercambiar(A1, A2);
  if A1^.rango = 0 then
    A1^.p_hijo := A2;
  else
    begin
      A2^.herm_i := A1^.p_hijo^.herm_i;
      A2^.herm_i^.herm_d := A2;
      A1^.p_hijo^.herm_i := A2;
    end;
  A1^.rango := A1^.rango + 1;
  fusionar_arbol := A1;
end;

```

Figura 6.54 Rutina para fusionar dos árboles binomiales de igual tamaño

```

function fusionar(M1, M2: COLA_DE_PRIORIDAD): COLA_DE_PRIORIDAD;
var M3: COLA_DE_PRIORIDAD;
  A1, A2, A3: COLA_DE_PRIORIDAD;
begin
  (1) if M1 = nil then
    fusionar := M2
  else
  (2) if M2 = nil then
    fusionar := M1
  else
  (3) if M1^.rango < M2^.rango then
    begin
      (4) extraer(A1, M1);
      (5) M3 := fusionar(M1, M2);
      (6) A1^.herm_i := M3^.herm_i;
      (7) M3^.herm_i^.herm_d := nil;
      (8) A1^.herm_d := M3; M3^.herm_i := A1;
      (9) fusionar := A1;
    end
  else
  (10) if M2^.rango < M1^.rango then
    fusionar := fusionar(M2, M1)
  else
  (11) begin
    (12) extraer(A1, M1); extraer(A2, M2);
    (13) M3 := fusionar(M1, M2);
    (14) A3 := fusionar_arbol(A1, A2);
    (15) A3 := fusionar_arbol(A3, A2);
    (16) fusionar := fusionar(M3, A3);
    (17) end;
  end;
end;

```

Figura 6.55 Rutina para fusionar dos colas de prioridad

Resumen

En este capítulo hemos visto varias implantaciones y usos del TDA cola de prioridad. La implantación estándar del montículo binario es elegante por su simplicidad y velocidad. No requiere apuntadores y sólo necesita una cantidad constante de espacio adicional; así que permite implantar eficientemente las operaciones sobre colas de prioridad.

Estudiamos la operación adicional *fusión* y presentamos tres implantaciones, cada una de las cuales es única a su modo. El montículo a izquierda es un ejemplo excelente del poder de la recursión. El montículo oblicuo representa una notable estructura de datos debido a la falta de criterios de equilibrio. Su análisis, que se presentará en el capítulo 11, es interesante por derecho propio. La cola binomial muestra cómo una idea sencilla se puede usar para alcanzar una buena cota de tiempo.

También hemos visto varios usos de las colas de prioridad, desde la planificación en sistemas operativos hasta la simulación. En los capítulos 7, 9 y 10 veremos de nuevo sus aplicaciones.

Ejercicios

- 6.1 Suponga que sustituimos la función *eliminar_mín* por *buscar_mín*. ¿Se pueden implantar *insertar* y *buscar_mín* en tiempo constante?
- 6.2 a. Muestre el resultado de insertar 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13 y 2, uno cada vez, en un montículo binario inicialmente vacío.
b. Muestre el resultado de usar el algoritmo en tiempo lineal para construir un montículo binario con la misma entrada.
- 6.3 Muestre el resultado de efectuar tres operaciones *eliminar_mín* en el montículo del ejercicio anterior.
- 6.4 Escriba las rutinas para hacer un *filtrado ascendente* y un *filtrado descendente* en un montículo binario.
- 6.5 Escriba y pruebe un programa que realice las operaciones *insertar*, *eliminar_mín*, *construir_montículo*, *buscar_mín*, *decrementar_llave*, *eliminar* e *incrementar_llave* en un montículo binario.
- 6.6 ¿Cuántos nodos hay en el montículo grande de la figura 6.13?
- 6.7 a. Demuestre que en los árboles binarios *construir_montículo* hace a lo más $2n - 2$ comparaciones entre elementos.
b. Muestre que un montículo de 8 elementos se puede construir en 8 comparaciones entre elementos del montículo.
- **c. Proporcione un algoritmo para construir un montículo binario en $\frac{13}{7}n + O(\log n)$ comparaciones de elementos.
- *6.8 Demuestre que la profundidad esperada del k -ésimo elemento menor de un montículo completo grande está acotada por $\log k$ (se puede suponer que $n = 2^k - 1$).

- 6.9 a. Proporcione un algoritmo para buscar todos los nodos menores que algún valor, x , en un montículo binario. Su algoritmo debe ejecutarse en $O(K)$, donde K es el número de nodos encontrados.
b. Su algoritmo, ¿se extiende a cualquier otra estructura de montículo presentada en este capítulo?
- *6.10 Proponga un algoritmo para insertar m nodos en un montículo binario de n elementos en un tiempo $O(m + \log n)$. Demuestre la cota de tiempo.
- 6.11 Escriba un programa para tomar n elementos y hacer lo siguiente:
 - a. insertarlos en un montículo uno por uno;
 - b. construir un montículo en tiempo lineal.
 Compare el tiempo de ejecución de ambos algoritmos para entradas ordenadas, ordenadas inversas y aleatorias.
- 6.12 Cada operación *eliminar_mín* usa $2 \log n$ comparaciones en el peor caso.
 - *a. Proponga un esquema tal que la operación *eliminar_mín* use sólo $\log n + \log \log n + O(1)$ comparaciones entre elementos. Esto no tiene por qué implicar menos movimientos de datos.
 - **b. Extienda el esquema de la parte (a) de modo que sólo se realicen $\log n + \log \log \log n + O(1)$ comparaciones.
 - *c. ¿Qué tan lejos puede llevar esta idea?
 - d. Los ahorros en comparaciones, ¿compensan el incremento en la complejidad de su algoritmo?
- 6.13 Si un montículo d se almacena como un arreglo, para una entrada ubicada en la posición i , ¿dónde se encuentran el padre y los hijos?
- 6.14 Suponga que necesitamos efectuar m *filtrado ascendente* y n *eliminar_mín* sobre un montículo d que inicialmente tiene n elementos.
 - a. ¿Cuál es el tiempo de ejecución total de todas las operaciones en términos de m , n y d ?
 - b. Si $d = 2$, ¿cuál es el tiempo de ejecución de todas las operaciones sobre el montículo?
 - c. Si $d = \Theta(n)$, ¿cuál es el tiempo de ejecución total?
 - *d. ¿Qué elección de d minimiza el tiempo de ejecución total?
- 6.15 Un *montículo mín-máx* es una estructura de datos que permite *eliminar_mín* y *eliminar_máx* en $O(\log n)$ por operación. La estructura es idéntica a un montículo binario, pero la propiedad de montículo es que para cualquier nodo, X , a profundidad par, la llave almacenada en X es menor que el padre pero mayor que el abuelo (cuando esto tiene sentido), y para cualquier nodo X a profundidad impar, la llave almacenada en X es mayor que el padre pero menor que el abuelo. Véase la figura 6.56.
 - a. ¿Cómo se encuentran los elementos mínimo y máximo?
 - *b. Proporcione un algoritmo para insertar un nodo nuevo en el montículo mín-máx.
 - *c. Proporcione un algoritmo para realizar *eliminar_mín* y *eliminar_máx*.

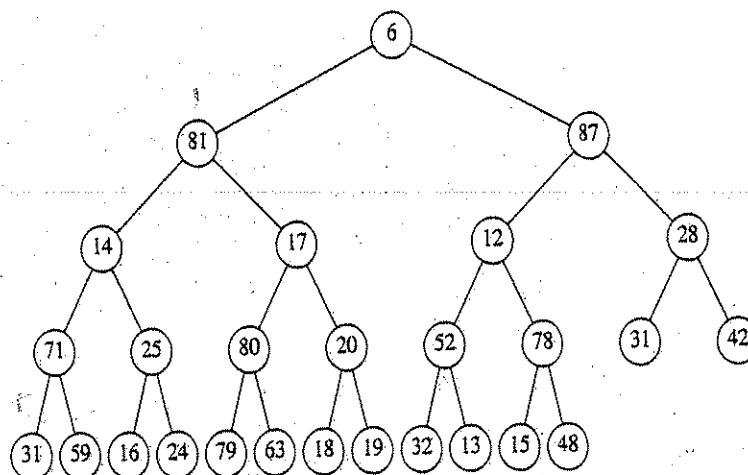


Figura 6.56 Montículo mín-máx.

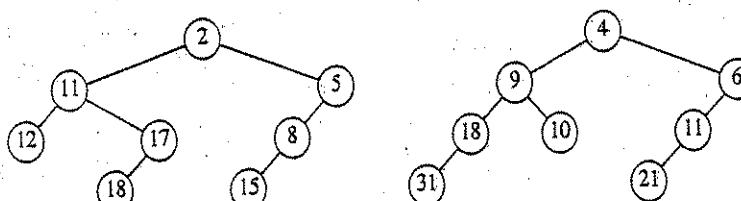


Figura 6.57

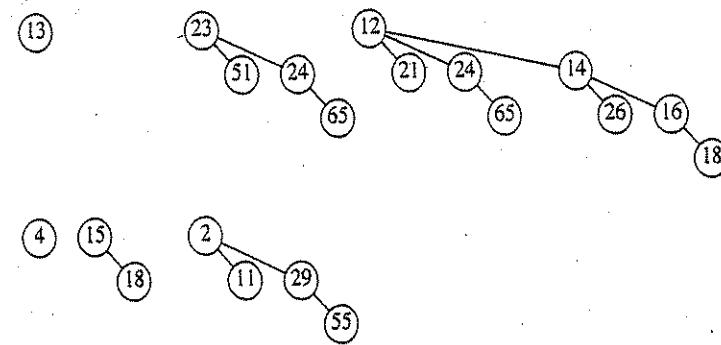
- *d. ¿Es posible construir un montículo mín-máx en tiempo lineal?
- *e. Suponga que desea manejar *eliminar_mín*, *eliminar_máx* y *fusionar*. Proponga una estructura de datos para permitir todas las operaciones en un tiempo $O(\log n)$.
- 6.16 Fusione los dos montículos a izquierda de la figura 6.57.
- 6.17 Muestre el resultado de insertar las llaves del 1 al 15 en orden sobre un montículo a izquierda inicialmente vacío.
- 6.18 Demuestre o rechace: un árbol perfectamente equilibrado se forma si las llaves de 1 a $2^k - 1$ se insertan en orden en un montículo a izquierda inicialmente vacío.
- 6.19 Dé un ejemplo de entrada que genere el mejor montículo a izquierda.
- 6.20 a. ¿Pueden los montículos a izquierda manejar eficientemente *decrementar_llave*?
b. ¿Qué cambios, de ser posibles, se requieren para hacer esto?

6.21 Una forma de eliminar nodos en una posición conocida en un montículo a izquierda es usar una estrategia perezosa. Para eliminar un nodo, simplemente se marca como eliminado. Al efectuar un *buscar_mín* o *eliminar_mín*, hay un problema potencial si la raíz está marcada como eliminada, pues entonces se tiene que eliminar realmente el nodo y encontrar el mínimo real, lo cual puede implicar la eliminación de otros nodos marcados. En esta estrategia, *eliminar* cuesta una unidad, pero el costo de un *eliminar_mín* o *buscar_mín* depende del número de nodos que estén marcados como eliminados. Suponga que después de un *eliminar_mín* o *buscar_mín* hay k nodos marcados menos que antes de la operación.

- *a. Muestre cómo efectuar *eliminar_mín* en un tiempo $O(k \log n)$.
- **b. Proponga una implantación, con un análisis para demostrar que el tiempo para efectuar *eliminar_mín* es $O(k \log(2n/k))$.

- 6.22 Podemos ejecutar *construir_montículo* en tiempo lineal para montículos a izquierda considerando cada elemento como un montículo a izquierda de un nodo, colocando todos esos montículos en una cola, y realizando el siguiente paso: hasta que sólo quede un montículo en la cola, desencolar dos montículos, fusionarlos y encolar el resultado.
- a. Demuestre que este algoritmo es $O(n)$ en el peor caso.
- b. ¿Por qué podría ser preferible este algoritmo al descrito en el texto?
- 6.23 Combine los dos montículos oblicuos de la figura 6.57.
- 6.24 Muestre el resultado de insertar las llaves del 1 al 15 en orden dentro de un montículo oblicuo.
- 6.25 Demuestre o rechace: un árbol perfectamente equilibrado se forma si las llaves 1 a $2^k - 1$ se insertan en orden en un montículo oblicuo inicialmente vacío.

Figura 6.58



11. M. L. Fredman y R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms", *Journal of the ACM*, 34 (1987), págs. 596-615.
12. G. H. Gonnet y J. I. Munro, "Heaps on Heaps", *SIAM Journal on Computing*, 15 (1986), págs. 964-971.
13. A. Hasham y J. R. Sack, "Bounds for Min-max Heaps", *BIT*, 27 (1987), págs. 315-323.
14. D. B. Johnson, "Priority Queues with Update and Finding Minimum Spanning Trees", *Information Processing Letters*, 4 (1975), págs. 53-57.
15. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, 2a. imp., Addison-Wesley, Reading, MA, 1975.
16. C. J. H. McDiarmid y B. A. Reed, "Building Heaps Fast", *Journal of Algorithms*, 10 (1989), págs. 352-365.
17. D. D. Sleator y R. E. Tarjan, "Self-adjusting Heaps", *Journal on Computing*, 15 (1986), págs. 52-69.
18. T. Strothotte, P. Eriksson, y S. Vallner, "A Note on Constructing Min-max Heaps", *BIT*, 29 (1989), págs. 251-256.
19. P. van Emde Boas, R. Kaas, E. Zijlstra, "Design and Implementation of an Efficient Priority Queue", *Mathematical Systems Theory*, 10 (1977), págs. 99-127.
20. J. Vuillemin, "A Data Structure for Manipulating Priority Queues", *Communications of the ACM*, 21 (1978), págs. 309-314.
21. J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM*, 7 (1964), págs. 347-348.

CAPÍTULO 7

Ordenación

En este capítulo estudiaremos el problema de la ordenación de un arreglo de elementos. A fin de simplificar el tema, supondremos en los ejemplos que el arreglo sólo contiene enteros, aunque, como es obvio, puede haber estructuras más complejas. En la mayor parte del capítulo supondremos que la ordenación completa se puede efectuar en la memoria principal, de modo que el número de elementos sea relativamente pequeño (menos de un millón). También son muy importantes las ordenaciones que no se pueden realizar en la memoria principal y deben utilizar disco o cinta. Este tipo de ordenación, llamada ordenación externa, se estudiará al final del capítulo.

Esta investigación de la ordenación interna contemplará lo siguiente:

- Existen varios algoritmos fáciles para ordenar en $O(n^2)$, como la ordenación por inserción.
- Hay un algoritmo, la ordenación de Shell, que es muy simple de codificar, se ejecuta en $O(n^2)$, y en la práctica es eficiente.
- Existen algoritmos de ordenación un poco más complejos con $O(n \log n)$.
- Cualquier algoritmo de ordenación de propósito general requiere $\Omega(n \log n)$ comparaciones.

El resto de este capítulo describirá y analizará los diferentes algoritmos de ordenación. Estos algoritmos contienen ideas importantes y de interés para la optimización de código y para el diseño de algoritmos. La ordenación también es un ejemplo donde se puede efectuar el análisis con precisión. Entiéndase que donde se requiera, se analizará tanto como sea posible.

- 6.26 Un montículo oblicuo de n elementos se puede construir usando el algoritmo estándar de montículo binario. ¿Podemos usar la misma estrategia de fusión descrita en el ejercicio 6.22 sobre los montículos oblicuos para obtener un tiempo de ejecución $O(n)$?
- 6.27 Demuestre que un árbol binomial B_k tiene árboles binomiales B_0, B_1, \dots, B_{k-1} como hijos de la raíz.
- 6.28 Demuestre que un árbol binomial de altura k tiene $\binom{k}{d}$ nodos a la profundidad d .
- 6.29 Fusione las dos colas binomiales de la figura 6.58.
- 6.30 a. Demuestre que n operaciones *insertar* en una cola binomial inicialmente vacía toma un tiempo $O(n)$ en el peor caso.
b. Proporcione un algoritmo para construir una cola binomial de n elementos, usando a lo más $n-1$ comparaciones entre elementos.
- *6.31 Proponga un algoritmo para insertar m nodos en una cola binomial de n elementos en un tiempo $O(m + \log n)$ para el peor caso. Demuestre la cota.
- 6.32 Escriba rutinas no recursivas para efectuar *fusionar*, *insertar* y *eliminar_mín* usando colas binomiales.
- **6.33 Suponga que extendemos las colas binomiales para permitir a lo más dos árboles de la misma altura por estructura. ¿Podemos obtener un tiempo $O(1)$ en el peor caso para la inserción y conservar $O(\log n)$ para las otras operaciones?
- 6.34 Suponga que tiene un número de cajas, cada una de las cuales puede contener un peso total P y elementos $e_1, e_2, e_3, \dots, e_n$, con pesos $p_1, p_2, p_3, \dots, p_n$. El objeto es empaquetarlos todos sin exceder la capacidad de cada caja y usando el menor número de cajas posible. Por ejemplo, si $P = 5$, y los elementos tienen pesos 2, 2, 3, 3, entonces podemos resolver el problema con dos cajas.
En general, este problema es muy difícil y no se conoce ninguna solución eficiente. Escriba programas para implantar eficientemente las siguientes estrategias de aproximación:
a. Colocar el peso en la primera caja en la cual quepa (creando una caja nueva si no hay caja con espacio suficiente). (Esta estrategia y todo lo que sigue podría dar tres cajas, lo cual es subóptimo.)
b. Colocar el peso en la caja con el mayor espacio para éste.
c. Colocar el peso en la caja más llena que puede alojarlo sin desbordamiento.
**d. ¿Se mejora alguna de las estrategias preordenando los elementos de acuerdo con el peso?
- 6.35 Suponga que queremos agregar la operación *decrementar_todas_las_llaves*(Δ) al repertorio del montículo. El resultado de esta operación es que todas las llaves del montículo ven reducido su valor en una cantidad Δ . Para la implantación de montículos que se deseé, explique las modificaciones necesarias para que las demás operaciones mantengan su tiempo de ejecución y *decrementar_todas_las_llaves* se ejecute en $O(1)$.
- 6.36 ¿Cuál de los dos algoritmos de selección tiene la mejor cota de tiempo?

Referencias

El montículo binario fue descrito por primera vez en [21]. La explicación del algoritmo en tiempo lineal proviene de [9].

La primera descripción de montículos-d está en [14]. Los montículos a izquierda fueron inventados por Crane [7] y descritos por Knuth [15]. Los montículos oblicuos fueron desarrollados por Sleator y Tarjan [17]. Las colas binomiales las inventó Vuillemin [20]; Brown ofreció un análisis detallado y un estudio empírico donde demostró que operan bien en la práctica [2], si se implantan con cuidado.

El ejercicio 6.7 (b-c) se tomó de [12]. En [16] se describe un método para construir montículos binarios que usa en promedio cerca de 1.52n comparaciones. La eliminación perezosa en los montículos a izquierda (ejercicio 6.21) proviene de [6]. Una solución al ejercicio 6.33 se puede encontrar en [5].

Los montículos mÍn-máx (ejercicio 6.15) se describieron originalmente en [1]. Implantaciones más eficientes de las operaciones están en [13] y [18]. Una representación alternativa de las colas de prioridad con doble extremo es el *deap* (del inglés *double heap*). Los detalles están en [3] y [4].

Una representación teórica interesante de las colas de prioridad es el *montículo de Fibonacci* [11], que describiremos en el capítulo 11. El montículo de Fibonacci permite todas las operaciones en un tiempo amortizado de $O(1)$, excepto para las eliminaciones, las cuales son $O(\log n)$. Los *montículos relajados* [8] logran cotas idénticas en el peor caso. Otra implantación interesante es el *montículo apareado* [10]. Por último, una cola de prioridad que funciona cuando los datos consisten en enteros pequeños se describe en [19].

1. M. D. Atkinson, J. R. Sack, N. Santoro, y T. Strothotte, "Min-Max Heaps and Generalized Priority Queues", *Communications of the ACM*, 29 (1986), págs. 996–1000.
2. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms", *SIAM Journal on Computing*, 7 (1978), págs. 298–319.
3. S. Carlsson, "The Deap – A Double-ended Heap to Implement Double-ended Priority Queues", *Information Processing Letters*, 26 (1987), págs. 33–36.
4. S. Carlsson, J. Chen, y T. Strothotte, "A Note on the Construction of the Data Structure 'Deap'", *Information Processing Letters*, 31 (1989), págs. 315–317.
5. S. Carlsson, J. I. Munro, y P. V. Poblete, "An Implicit Binomial Queue with Constant Insertion Time", *Proceedings of First Scandinavian Workshop on Algorithm Theory*, 1988, págs. 1–13.
6. D. Cherdon y R. E. Tarjan, "Finding Minimum Spanning Trees", *SIAM Journal on Computing*, 5 (1976), págs. 724–742.
7. C. A. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees", *Technical Reports STAN-CS-72-259*, Computer Science Department, Stanford University, Stanford, CA, 1972
8. J. R. Driscoll, H. N. Gabow, R. Shraiman, y R. E. Tarjan, "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation", *Communications of the ACM*, 31 (1988), págs. 1343–1354.
9. R. W. Floyd, "Algorithm 245: Treesort 3", *Communications of the ACM*, 7 (1964), págs. 701.
10. 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), págs. 111–129.

7.1. Preliminares

Todos los algoritmos que vamos a describir son intercambiables. A cada uno se le pasará un arreglo que contiene los elementos y un entero para indicar el número de elementos.

Supondremos que n , el número de elementos pasados a las rutinas de ordenación, ya se ha revisado y es legal. Para algunas de las rutinas de ordenación, será conveniente colocar un centinela en la posición 0, por lo cual supondremos que los índices de los arreglos van de 0 a n . Los datos reales empiezan en 1 para todos los métodos.

También se supondrá la existencia de los operadores " $<$ " y " $>$ ", los cuales se pueden usar para establecer un orden consistente en la entrada. Además del operador de asignación, éstas son las únicas operaciones permitidas sobre los datos de entrada. La ordenación en esas condiciones se denomina ordenación *basada en comparaciones*.

7.2. Ordenación por inserción

7.2.1. El algoritmo

Uno de los algoritmos de ordenación más sencillos es la *ordenación por inserción*. Ésta consiste en $n - 1$ *pasadas*. Para las pasadas $p = 2$ a n , la ordenación por inserción garantiza que los elementos en las posiciones 1 a p estén ordenados. La ordenación por inserción se vale del hecho de que ya se sabe que los elementos en las posiciones 1 a $p - 1$ están ordenados. La figura 7.1 muestra un archivo de ejemplo después de cada pasada de la ordenación por inserción.

La figura 7.1 muestra la estrategia general. En la pasada p , movemos a la izquierda el elemento p -ésimo hasta encontrar su posición correcta entre los primeros p elementos. El código de la figura 7.2 implanta esta estrategia. El centinela en $a[0]$ terminará el ciclo *while* en el momento en que un elemento se mueva durante alguna pasada, todo el camino hasta llegar al frente. Las líneas [4] a [7] implantan el movimiento de datos sin el uso explícito de intercambios. El elemento en la posición p se guarda en tmp , y todos los elementos mayores (anteriores a la posición p) se mueven un espacio a la derecha. Después se coloca tmp en el lugar correcto. Esta es la misma técnica usada en la implantación de montículos binarios.

Figura 7.1 Ordenación por inserción después de cada pasada

Original	34	8	64	51	32	21	Posiciones movidas
Después de $p = 2$	8	34	64	51	32	21	1
Después de $p = 3$	8	34	64	51	32	21	0
Después de $p = 4$	8	34	51	64	32	21	1
Después de $p = 5$	8	32	34	51	64	21	3
Después de $p = 6$	8	21	32	34	51	64	4

7.3. UNA COTA INFERIOR PARA ALGORITMOS DE ORDENACIÓN SIMPLES

```

procedure ordenación_por_inserción(var a: datos_entrada; n: integer);
  var j, p: integer;
      tmp: tipo_entrada;

begin
  {1}   a[0] := DATO_MÍN; {centinela}
  {2}   for p := 2 to n do begin
  {3}     j := p; tmp := a[p];
        while tmp < a[j-1] do begin
          {4}         a[j] := a[j-1];
          {5}         j := j-1;
        end; {while}
        {6}         a[j] := tmp;
        {7}   end; {for}
    end;

```

Figura 7.2 Rutina de ordenación por inserción

7.2.2. Análisis de la ordenación por inserción

Debido a los ciclos anidados, cada uno de los cuales hace n iteraciones, la ordenación por inserción es $O(n^2)$. Además, esta cota es ajustada, porque de hecho, la entrada en orden inverso puede alcanzarla. Un cálculo preciso muestra que la comprobación de la línea [4] se puede ejecutar a lo más p veces para cada valor de p . Sumando para todos los valores de p da un total de

$$\sum_{p=2}^n p = 2 + 3 + 4 + \dots + n = \Theta(n^2)$$

Por otro lado, si la entrada está ordenada de antemano, el tiempo de ejecución es $O(n)$ porque la condición al inicio del ciclo *while* siempre falla de inmediato. En efecto, si la entrada está casi ordenada (este término se definirá con más rigor en la sección siguiente), la ordenación por inserción se ejecutará rápidamente. Debido a esta enorme variación, vale la pena analizar el comportamiento del algoritmo en el caso medio. Resulta que el caso medio de la ordenación por inserción es $\Theta(n^2)$, al igual que el de una gran variedad de algoritmos de ordenación, como se muestra en la sección siguiente.

7.3. Una cota inferior para algoritmos de ordenación simples

Una *inversión* en un arreglo de números es cualquier par ordenado (i, j) con la propiedad de que $i < j$ pero $a[i] > a[j]$. En el ejemplo de la última sección, la lista de entrada 34, 8, 64, 51, 32, 21 tiene nueve inversiones, a saber: (34, 8), (34, 32),

(34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21) y (32, 21). Observe que éste es exactamente el número de intercambios que se necesita efectuar (implícitamente) en la ordenación por inserción. Éste es siempre el caso, porque el intercambio de dos elementos adyacentes que están fuera de lugar elimina exactamente una inversión, y un archivo ordenado no tiene inversiones. Puesto que hay otro trabajo $O(n)$ implicado en el algoritmo, el tiempo de ejecución de la ordenación por inserción es $O(I+n)$, donde I es el número de inversiones en el archivo original. Así, la ordenación por inserción se ejecuta en un tiempo lineal si el número de inversiones es $O(n)$.

Calculando el número medio de inversiones en una permutación se pueden obtener cotas precisas sobre el tiempo de ejecución medio de la ordenación por inserción. Como es usual, definir *medio* es una tarea difícil. Supondremos que no hay elementos duplicados (si se permitieran duplicados, ni siquiera quedaría claro cuál es el número medio de duplicados). Con esta suposición, podemos dar por sentado que la entrada es alguna permutación de los primeros n enteros (ya que sólo es importante el orden relativo) y que son igualmente probables. Bajo esas suposiciones, se tiene el siguiente teorema:

TEOREMA 7.1

El número medio de inversiones en un arreglo de n números diferentes es $n(n - 1)/4$.

DEMOSTRACIÓN:

Para cualquier lista, L , de números, consideremos L_i la lista en orden inverso. La lista inversa del ejemplo es 21, 32, 51, 64, 34, 8. Consideremos cualquier par de dos números en la lista (x, y) , con $y > x$. Es obvio que en exactamente una de L y L_i este par ordenado representa una inversión. El número total de esos pares en una lista L y su inversa L_i es $n(n - 1)/2$. Así, una lista media tiene la mitad de esta cantidad, o sea $n(n - 1)/4$ inversiones.

Este teorema implica que la ordenación por inserción es cuadrática en promedio. También da una cota inferior muy fuerte para cualquier algoritmo que sólo intercambia elementos adyacentes.

TEOREMA 7.2

Cualquier algoritmo que ordena intercambiando elementos adyacentes requiere un tiempo $\Omega(n^2)$ en promedio.

DEMOSTRACIÓN:

Al principio número medio de inversiones es $n(n - 1)/4 = \Omega(n^2)$. Cada intercambio elimina sólo una inversión, así que se requieren $\Omega(n^2)$ intercambios.

Éste es un ejemplo de una demostración de la cota inferior. Es válido no sólo para la ordenación por inserción, que efectúa implícitamente intercambios adyacentes, sino para otros algoritmos sencillos como la ordenación por el método de la burbuja y la ordenación por selección, que no describiremos aquí. De hecho, es válido para una clase completa de algoritmos de ordenación, incluyendo aquellos no descubiertos, que sólo realizan intercambios adyacentes. Debido a esto, esta

demonstración no se puede confirmar empíricamente. Aunque esta demostración de la cota inferior es muy sencilla, en general la demostración de las cotas inferiores es mucho más compleja que la demostración de las cotas superiores y en algunos casos hasta parece cuestión de vudú.

Esta cota inferior demuestra que para que un algoritmo de ordenación se ejecute en tiempo subcuadrático, o $O(n^k)$, debe realizar comparaciones y, en particular, intercambios entre elementos lejanos. Un algoritmo de ordenación progresivamente elimina inversiones, y para ejecutarse con eficiencia, debe eliminar más de una inversión por intercambio.

7.4. Ordenación de Shell

La ordenación de Shell (*Shellsort*, en inglés), llamada así por su inventor, Donald Shell, fue uno de los primeros algoritmos en romper la barrera del tiempo, aunque no fue sino hasta varios años después de su descubrimiento que se demostró una cota de tiempo subcuadrática. Como se indicó en la sección anterior, funciona comparando elementos que están distantes; la distancia entre comparaciones decrece conforme el algoritmo se ejecuta hasta la última fase, en la cual se comparan los elementos adyacentes. Por esta razón, la ordenación de Shell se llama a veces ordenación por *disminución de incrementos*.

La ordenación de Shell usa una secuencia, h_1, h_2, \dots, h_k , conocida como la *secuencia de incrementos*. Cualquier secuencia de incrementos funcionará en tanto que $h_1 = 1$, pero es obvio que algunas elecciones son mejores que otras (este asunto se revisará después). Después de una fase, usando algún incremento h_k , para toda i , tenemos que $a[i] \leq a[i + h_k]$ (donde esto tenga sentido); todos los elementos espaciados por una distancia h_k están ordenados. Entonces se dice que el archivo está h_k -ordenado. Por ejemplo, la figura 7.3 muestra un arreglo después de varias fases de la ordenación de Shell. Una propiedad importante de la ordenación de Shell (que enunciamos sin demostración) es que un archivo h_k -ordenado que después es h_{k-1} -ordenado sigue siendo h_k -ordenado. Si no fuera así, el algoritmo probablemente sería de poco valor, ya que las últimas fases desharían el trabajo hecho por las fases anteriores.

La estrategia general para h_k -ordenar es que para cada posición i en $h_k + 1, h_k + 2, \dots, n$, se coloca el elemento en el espacio correcto entre $i, i - h_k, i - 2h_k$, etc. Aunque esto no afecta la implantación, una revisión cuidadosa muestra que la acción de una h_k -ordenación consiste en efectuar una ordenación por inserción sobre h_k subarreglos independientes. Esta observación será importante cuando analicemos el tiempo de ejecución de la ordenación de Shell.

Figura 7.3 Ordenación de Shell después de cada pasada

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
después de 5-ordenar	35	17	11	28	12	41	75	15	96	58	81	94	95
después de 3-ordenar	28	12	11	35	15	41	58	17	94	75	81	96	95
después de 1-ordenar	11	12	15	17	28	35	41	58	75	81	94	95	96

Una elección muy común (pero no tan eficiente) para la secuencia de incrementos es adoptar la secuencia sugerida por Shell: $h_t = \lfloor n/2 \rfloor$, y $h_k = \lfloor h_{k+1}/2 \rfloor$. La figura 7.4 contiene un programa que implanta la ordenación de Shell usando esta secuencia. Más adelante veremos que hay secuencias que ofrecen mejoras significativas en el tiempo de ejecución del algoritmo.

El programa de la figura 7.4 evita el uso explícito de intercambios en la misma forma que nuestra implantación de la ordenación por inserción. Desafortunadamente, no es posible usar un centinela para la ordenación de Shell, así que el código de las líneas [5] a [9] no es tan limpio como el código correspondiente en la ordenación por inserción (líneas [4] a [6]).

7.4.1. Análisis del peor caso de la ordenación de Shell

Aunque es fácil codificar la ordenación de Shell, no ocurre lo mismo con el análisis de su tiempo de ejecución. El tiempo de ejecución de la ordenación de Shell depende de qué secuencia de incrementos se elige y las demostraciones pueden ser bastante complicadas. El análisis del caso medio de la ordenación de Shell ha sido un problema abierto por largo tiempo, excepto en las secuencias de incrementos más triviales. Demostraremos cotas ajustadas del peor caso para dos secuencias de incrementos particulares.

Figura 7.4 Rutina de ordenación de Shell usando los incrementos de Shell (es posible tener mejores incrementos)

```

procedure shellsort(var a: datos_entrada; n: integer);
label 99;
var incremento, j: integer;
    tmp: tipo_entrada;
begin
[1]   incremento := n div 2;
[2]   while incremento > 0 do begin
[3]       for i := incremento + 1 to n do begin
[4]           tmp := a[i]; j := i;
[5]           while j - incremento > 0 do begin
[6]               if tmp < a[j - incremento] then begin
[7]                   a[j] := a[j - incremento];
[8]                   j := j - incremento;
[9]               end
[10]              else {rompe el while}
[11]                  goto 99;
[12]          end; {while}
[13]      99:      a[j] := tmp;
[14]      end; {for}
[15]      incremento := incremento div 2;
[16]  end; {while}
end; {Shellsort}
```

TEOREMA 7.3.

El tiempo de ejecución del peor caso en la ordenación de Shell, usando los incrementos de Shell, es $\Theta(n^2)$.

DEMOSTRACIÓN:

Para la prueba no basta con una cota superior en el tiempo de ejecución del peor caso; hay que demostrar también que existe alguna entrada que realmente tarda en ejecutarse $\Omega(n^2)$. Primero se demuestra la cota inferior, construyendo un caso malo. Primero escogemos que n sea una potencia de 2. Esto hace que todos los incrementos sean pares, excepto el último, que es 1. Ahora, daremos como entrada un arreglo, *datos_entrada*, con los $n/2$ números más grandes en las posiciones pares y los $n/2$ números menores en las posiciones impares. Como todos los incrementos excepto el último son pares, cuando llegamos a la última pasada los $n/2$ números mayores siguen en las posiciones pares y los $n/2$ menores permanecen en las posiciones impares. Entonces el i -ésimo número menor ($i \leq n/2$) está en la posición $2i-1$ antes de iniciarse la última pasada. Para devolver el i -ésimo elemento menor a su posición correcta hay que moverlo $i-1$ espacios en el arreglo. Así, sólo para colocar los $n/2$ elementos menores en el lugar correcto se requiere un trabajo $\sum_{i=1}^{n/2} i-1 = \Omega(n^2)$ como mínimo. Por ejemplo, la figura 7.5 muestra una entrada mala (pero no la peor) cuando $n = 16$. El número de inversiones restantes después de 2-ordenar es exactamente $1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$; así, la última pasada tarda bastante.

Para terminar la demostración, demostraremos que la cota superior es $O(n^2)$. Como se observó antes, una pasada con incrementos h_k consiste en h_k ordenaciones por inserción de unos n/h_k elementos. Puesto que la ordenación por inserción es cuadrática, el costo total de una pasada es $O(h_k(n/h_k)^2) = O(n^2/h_k)$. Sumando todas las pasadas se obtiene una cota total de $O(\sum_{i=1}^t n^2/h_i) = O(n^2 \sum_{i=1}^t 1/h_i)$. Puesto que los incrementos forman una serie geométrica con un factor común 2, y el término mayor en la serie es $h_1 = 1$, $\sum_{i=1}^t 1/h_i < 2$. Así se obtiene una cota total de $O(n^2)$.

El problema con los incrementos de Shell es que los pares de incrementos no son necesariamente primos relativos, y así el menor incremento puede tener poco efecto. Hibbard sugirió una secuencia de incrementos un poco diferente, la cual da mejores resultados en la práctica (y en la teoría). Sus incrementos son de la forma: 1, 3, 7, ..., $2^k - 1$. Aunque esos incrementos son casi idénticos, la diferencia clave es que los incrementos consecutivos no tienen factores comunes. Ahora analizamos el tiempo de ejecución para el peor caso del algoritmo de Shell para esta secuencia de incrementos. La demostración es bastante complicada.

TEOREMA 7.4.

El tiempo de ejecución del peor caso en la ordenación de Shell, usando los incrementos de Hibbard, es $\Theta(n^{3/2})$.

DEMOSTRACIÓN:

Demostraremos sólo la cota superior y dejaremos como ejercicio la demostración de la cota inferior. La demostración requiere algunos resultados conocidos de la

Original	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
Después de 8-ordenar	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
Después de 4-ordenar	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
Después de 2-ordenar	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
Después de 1-ordenar	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figura 7.5 Caso malo para la ordenación de Shell con los incrementos de Shell

teoría de los números aditivos. Al final del capítulo se encuentran referencias a esos resultados.

Para la cota superior, como antes, se acota el tiempo de ejecución de cada pasada y se suman los de todas las pasadas. Para incrementos $h_k > n^{1/2}$, usaremos la cota $O(n^2/h_k)$ del teorema anterior. Aunque esta cota sirve para los otros incrementos, es demasiado grande para ser útil. Intuitivamente, se debe aprovechar el hecho de que *esta* secuencia de incrementos es *especial*. Lo que necesitamos demostrar es que para cualquier elemento a_p en la posición p , en el momento de realizar una h_k -ordenación, sólo hay unos cuantos elementos a la izquierda de la posición p que son mayores que a_p .

Al llegar a la h_k -ordenación del arreglo de entrada, sabemos que ya se pasó por h_{k+1} -ordenación y por h_{k+2} -ordenación. Antes de la h_k -ordenación, consideremos los elementos de las posiciones p y $p-i$, con $i < p$. Si i es un múltiplo de h_{k+1} o h_{k+2} , entonces claramente $a[p-i] < a[p]$. Se puede decir más, sin embargo. Si i es expresable como una combinación lineal (en enteros no negativos) de h_{k+1} y h_{k+2} , entonces $a[p-i] < a[p]$. Como ejemplo, cuando llegamos a 3-ordenar, el archivo ya está 7- y 15-ordenado. 52 es expresable como una combinación lineal de 7 y 15, porque $52 = 1 \cdot 7 + 3 \cdot 15$. Así, $a[100]$ como una combinación lineal de 7 y 15, porque $a[100] \leq a[107] \leq a[122] \leq a[137] \leq a[152]$.

Ahora, $h_{k+2} = 2h_{k+1} + 1$, así h_{k+1} y h_{k+2} no pueden compartir un factor común. En este caso, es posible demostrar que todos los enteros que son al menos tan grandes como $(h_{k+1} - 1)(h_{k+2} - 1) = 8h_k^2 + 4h_k$ se pueden expresar como una combinación lineal de h_{k+1} y h_{k+2} (véase la referencia al final del capítulo).

Esto indica que el ciclo *while* de la línea [5] se puede ejecutar a lo más $8h_k + 4 = O(h_k)$ veces para cada una de las $n - h_k$ posiciones. Esto da una cota de $O(n h_k)$ por pasada.

Con base en el hecho de que cerca de la mitad de los incrementos satisfacen $h_k < \sqrt{n}$, y suponiendo que t es par, el tiempo de ejecución total es

$$O\left(\sum_{k=1}^{t/2} nh_k + \sum_{k=t/2+1}^t n^2/h_k\right) = O\left(n \sum_{k=1}^{t/2} h_k + n^2 \sum_{k=t/2+1}^t 1/h_k\right)$$

Puesto que ambas sumas son series geométricas, y como $h_{t/2} = \Theta(\sqrt{n})$, esto se simplifica a

$$= O(nh_{t/2}) + O\left(\frac{n^2}{h_{t/2}}\right) = O(n^{3/2}) = O(n^{3/2})$$

Se cree que el tiempo de ejecución del caso promedio de la ordenación de Shell, usando incrementos de Hibbard, es $O(n^{5/4})$, con base en simulaciones, pero nadie ha podido demostrarlo. Pratt demostró que la cota $\Theta(n^{3/2})$ se aplica a un amplio conjunto de secuencias de incrementos.

Sedgewick ha propuesto varias secuencias de incrementos que dan un tiempo de ejecución para el peor caso de $O(n^{4/3})$ (también alcanzable). Se conjectura que el tiempo de ejecución medio es $O(n^{7/6})$ para estas secuencias de incrementos. Hay estudios empíricos que demuestran que estas secuencias funcionan significativamente mejor que la de Hibbard. La mejor de esas secuencias es $\{1, 5, 19, 41, 109, \dots\}$, en la cual los términos son de la forma $9 \cdot 4^i - 9 \cdot 2^i + 1$ o bien $4^i - 3 \cdot 2^i + 1$. La manera más fácil de implantar esto es colocando estos valores en un arreglo. En la práctica, esta secuencia de incrementos es la mejor conocida, aunque hay una posibilidad permanente de que exista alguna secuencia de incrementos que dé una mejoría significativa al tiempo de ejecución de la ordenación de Shell.

Hay muchos otros resultados sobre la ordenación de Shell que (por lo regular) requieren teoremas difíciles de la teoría de números y de combinatoria, y son de interés teórico sobre todo. La ordenación de Shell es un buen ejemplo de un algoritmo muy sencillo con un análisis extremadamente complejo.

El rendimiento de la ordenación de Shell es bastante aceptable en la práctica, aun para n en las decenas de miles. La simplicidad del código la hace el algoritmo adecuado para clasificar entradas moderadamente grandes.

7.5. Ordenación por montículos

Como se mencionó en el capítulo 6, las colas de prioridad se pueden usar para ordenar en un tiempo $O(n \log n)$. El algoritmo basado en esta idea se denomina *ordenación por montículo* (*heapsort*) y da el mejor tiempo de ejecución. O grande de los vistos hasta ahora. No obstante, en la práctica es más lento que una versión de la ordenación de Shell que utilice la secuencia de incrementos de Sedgewick.

Recordaremos, del capítulo 6, que la estrategia básica es construir un montículo binario de n elementos. Esta etapa lleva un tiempo $O(n)$. Después efectuaremos n operaciones *eliminar_min*. Los elementos menores dejan primero el montículo, ordenadamente. Almacenando estos elementos en un segundo arreglo y después copiándolos de regreso al arreglo original, ordenamos n elementos. Como cada *eliminar_min* tarda $O(\log n)$, el tiempo de ejecución total es $O(n \log n)$.

El problema principal con este algoritmo es que usa un arreglo adicional, así que se duplica el requerimiento de memoria. Esto podría ser un problema en algunos casos. Observe que el tiempo extra consumido al copiar el segundo arreglo de regreso al primero es sólo $O(n)$, así que es probable que no afecte significativamente el tiempo de ejecución. El problema es el espacio.

Una forma ingeniosa de evitar el uso de un segundo arreglo aprovecha el hecho de que en cada *eliminar_min*, el montículo se contrae en uno. Así la celda

que era última en el montículo se puede usar para almacenar el elemento recién eliminado. Como ejemplo, supongamos que tenemos un montículo con seis elementos. El primer *eliminar_mín* produce a_1 . Ahora el montículo sólo tiene cinco elementos, así que se puede colocar a_1 en la posición 6. El siguiente *eliminar_mín* produce a_2 . Puesto que ahora el montículo tendrá cuatro elementos, podríamos colocar a_2 en la posición 5.

Con esta estrategia, después del último *eliminar_mín* el arreglo contendrá los elementos en orden *decreciente*. Si queremos los elementos en el orden *creciente*, más típico, podemos cambiar la propiedad de orden para que el padre tenga una llave mayor que los hijos. Así tenemos un montículo (*máx*).

En nuestra implantación, usaremos un montículo (*máx*), pero se evitará el TDA real por fines de velocidad. Como es usual, todo se hace en un arreglo. El primer paso construye el montículo en tiempo lineal. Después efectuamos $n - 1$ operaciones *eliminar_máx* intercambiando el último elemento del montículo por el primero, disminuyendo el tamaño del montículo, y filtrando hacia abajo. Al terminar el algoritmo, el arreglo contiene los elementos en orden. Por ejemplo, consideremos la secuencia de entrada 31, 41, 59, 26, 53, 58, 97. El montículo resultante se muestra en la figura 7.6.

La figura 7.7 muestra el montículo que se obtiene después del primer *eliminar_máx*. Como implica la figura, el último elemento del montículo es 31; 97 se colocó en una parte del arreglo del montículo que técnicamente ya no forma parte del montículo. Después de 5 operaciones *eliminar_máx* el montículo tendrá realmente sólo un elemento, pero los elementos depositados en el arreglo del montículo estarán ordenados.

El código para realizar la ordenación por montículo se da en la figura 7.8. Como es usual, tenemos el problema de la ruptura del ciclo más interno. Ya hemos visto dos formas de hacerlo: las construcciones *goto* y *while not encontrado*. La rutina

Figura 7.6 Montículo (*máx*) después de la fase *construir_montículo*.

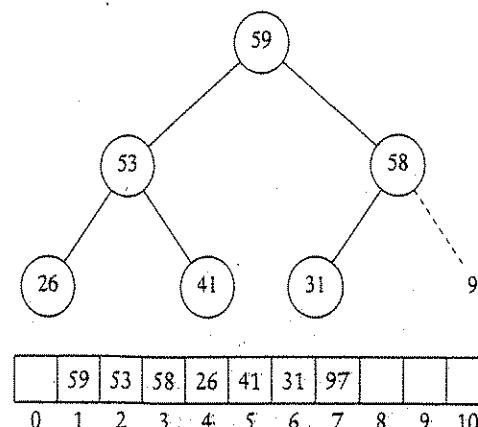
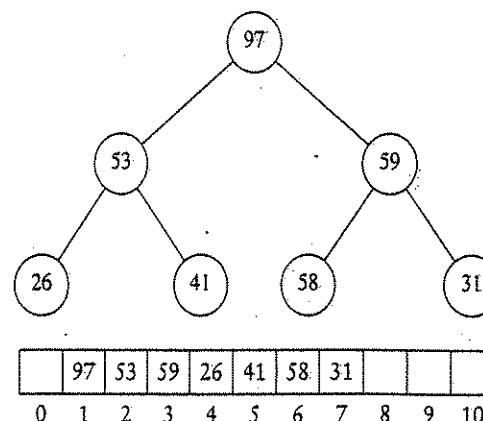


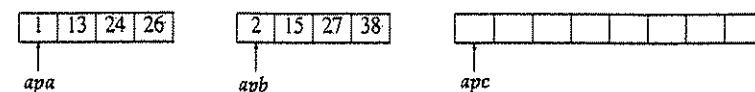
Figura 7.7 Montículo después del primer *eliminar_máx*

correspondiente en el capítulo 6 usó un *while not encontrado*, por lo que aquí presentamos la construcción *goto* por diversidad.

7.6. Ordenación por intercalación

Ahora volcaremos nuestra atención hacia la *ordenación por intercalación (mergesort)*. Esta ordenación requiere un tiempo de ejecución, en el peor caso, de $O(n \log n)$, y el número de comparaciones es casi óptimo. Se trata de un buen ejemplo fino de un algoritmo recursivo.

La operación fundamental en este algoritmo es la intercalación de dos listas ordenadas. Debido a que las listas están ordenadas, esto se puede hacer en una pasada sobre la entrada, si la salida se pone en una tercera lista. El algoritmo de intercalación básico toma dos arreglos de entrada a y b , un arreglo de salida c , y tres contadores, apa , apb y apc , los cuales primero se ponen al inicio de sus arreglos respectivos. El menor de $a[apa]$ y $b[apb]$ se copia en la siguiente entrada de c , y se avanzan los contadores apropiados. Cuando se agota cualquier lista de entrada, los datos que queden en la otra lista se copian en c . Un ejemplo de cómo funciona la rutina de intercalación se da para la siguiente entrada.



```

procedure filtrado_desc(var a:datos_entrada; i, n: integer);
label 999;
var hijo: integer;
tmp: tipo_entrada;

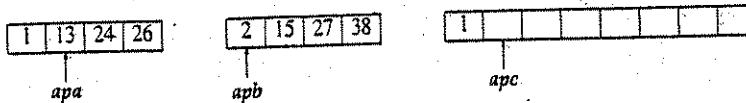
begin
[1] tmp := a[i];
[2] while i * 2 <= n do
begin
[3] hijo := i * 2;
if hijo <> n then
[5] if a[hijo + 1] > a[hijo] then
hijo := hijo + 1;
[6] if tmp < a[hijo] then
begin
[8] a[i] := a[hijo];
i := hijo;
end
else {fuerza la salida del ciclo while};
[10] goto 999;
end;
[11] 999: a[i] := tmp;
end;

procedure heapsort(var a: datos_entrada; n: integer);
var i: integer;
begin
for i := n div 2 downto 1 do {construye_montículo}
[1] filtrado_desc(a, i, n);
for i := n downto 2 do
begin
[4] intercambiar(a[1], a[i]);
{elimina_máx}
[5] filtrado_desc(a, 1, i-1);
end;
end;

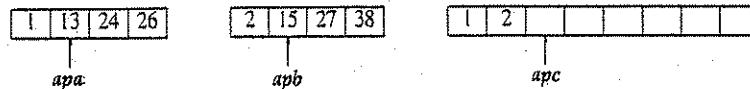
```

Figura 7.8. Ordenación por montículos

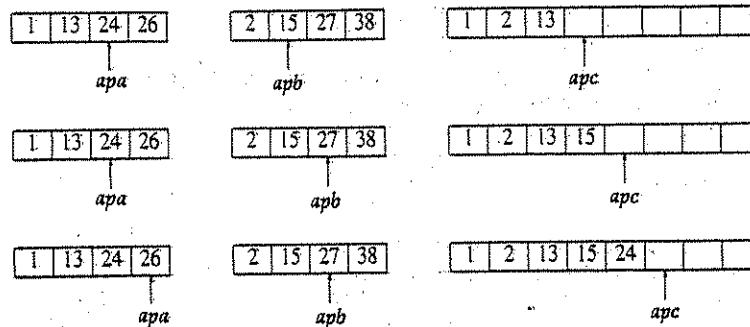
Si el arreglo *a* contiene 1, 13, 24, 26 y *b* contiene 2, 15, 27, 38, el algoritmo procede como sigue: primero, se hace una comparación entre 1 y 2. 1 se agrega a *c*, y después se comparan 13 y 2.



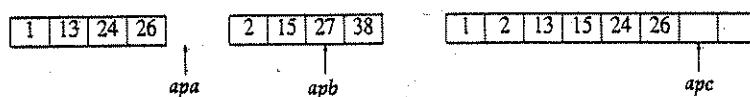
2 se agrega a *c*, y luego se comparan 13 y 15.



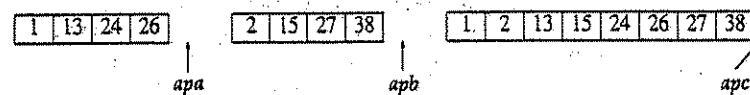
13 se agrega a *c*, y después se comparan 24 y 15. Esto continúa hasta comparar 26 y 27.



26 se agrega a *c*, y se agota el arreglo *a*.



El resto del arreglo *b* se copia en *c*.



Desde luego, el tiempo para combinar dos arreglos ordenados es lineal porque a lo más se hacen $n - 1$ comparaciones, donde n es el número total de elementos. Para ver esto, observe que todas las comparaciones agregan un elemento al arreglo *c*, excepto la última, que agrega dos.

Por lo tanto, es fácil describir el algoritmo *mergesort*. Si $n = 1$, sólo hay un elemento por ordenar, y la respuesta está a la mano. Si no, se hace una ordenación recursiva por intercalación de la primera mitad del arreglo con la segunda mitad. Esto da dos mitades ordenadas, las cuales pueden intercalarse usando el algoritmo de ordenación antes descrito. Por ejemplo, para clasificar el arreglo de ocho elemen-

```

procedure ord_intercala(var a, arreglo_tmp: datos_entrada; izq, der: integer);
  var centro: integer;
begin
  if izq < der then
    begin
      centro := (izq + der) div 2;
      ord_intercala(a, arreglo_tmp, izq, centro);
      ord_intercala(a, arreglo_tmp, centro+1, der);
      intercala(a, arreglo_tmp, izq, centro+1, der);
    end;
  end;

procedure mergesort(var a: datos_entrada; n: integer);
  var arreglo_tmp: datos_entrada;

begin
  ord_intercala(a, arreglo_tmp, 1, n);
end;

```

Figura 7.9 Rutina mergesort

tos 24, 13, 26, 1, 2, 27, 38, 15, clasificamos recursivamente los primeros cuatro y los últimos cuatro elementos, obteniendo 1, 13, 24, 26, 2, 15, 27, 38. Entonces, combinamos las dos mitades como antes, obteniendo la lista final 1, 2, 13, 15, 24, 26, 27, 38. Este algoritmo es una estrategia clásica de "divide y vencerás". El problema se divide en dos problemas menores y se resuelve recursivamente. La fase de *vencer* consiste en pegar las dos respuestas. "Divide y vencerás" es un uso muy potente de la recursión que veremos muchas veces.

Una implantación de la ordenación por intercalación se da en la figura 7.9. El procedimiento llamado *mergesort* es sólo un manejador de la rutina recursiva *ord_intercala*.

La rutina *intercala* es sutil. Si se declara localmente un arreglo temporal para cada llamada recursiva de *intercala*, podría haber $\log n$ arreglos temporales activos en cualquier punto. Esto podría ser fatal en una máquina con memoria pequeña. Un detenido examen muestra que como *intercala* es la última línea de *ord_intercala*, sólo necesita un arreglo temporal activo en cualquier punto. Más aún, podemos usar cualquier parte del arreglo temporal; usaremos la misma porción que el arreglo de entrada *a*. Esto permite la mejoría descrita al final de esta sección. La figura 7.10 implanta la rutina *intercala*.

7.6.1. Análisis de la ordenación por intercalación

La ordenación por intercalación es un ejemplo clásico de las técnicas con que se analizan rutinas recursivas. No es obvio que es fácil reescribir sin recursión la ordenación por intercalaciones (pero se puede), así que es necesario escribir una

ordenación por intercalaciones (pero se puede), así que es necesario escribir una relación recurrente para el tiempo de ejecución. Supondremos que *n* es una potencia de 2, así que siempre dividimos en mitades pares. Para *n* = 1, el tiempo de la ordenación por intercalación es constante y se denotará con 1. Si no, el tiempo para ordenar por intercalación *n* números es igual al tiempo para hacer dos ordenaciones recursivas por intercalación de tamaño *n*/2, más el tiempo para intercalar, que es lineal. Las siguientes ecuaciones indican esto con exactitud:

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n$$

Esta es una relación recurrente estándar, que se puede resolver de varias formas. Mostraremos dos métodos. La primera idea es dividir la relación recurrente entre *n*. La razón para hacerlo será evidente pronto. Esto da

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

Esta ecuación es válida para cualquier *n* que sea potencia de 2, así que podemos escribir

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

y

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Ahora, se suman todas las ecuaciones. Esto significa que se suman todos los términos del lado izquierdo y que el resultado se iguala a la suma de todos los términos del lado derecho. Observe que el término $T(n/2)/(n/2)$ aparece en ambos lados, así que se anulan. De hecho, prácticamente todos los términos aparecen en ambos lados y se ordenan. Esto se conoce como *proyectar* una suma. Después de sumar todo, el resultado final es

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n$$

```

{pos_i = inicio de la mitad izquierda, pos_d = inicio de la mitad derecha}

procedure intercala(vaia, arreglo_tmp: datos_entrada; pos_i, pos_d, der: integer);
  var izq, num_elementos, pos_tmp: integer;

begin
  izq := pos_d - 1;
  pos_tmp := pos_i;
  num_elementos := der - pos_i + 1;

  while (pos_i <= izq) and (pos_d <= der) do (ciclo principal)
  begin
    if a[pos_i] <= a[pos_d] then
      begin
        arreglo_tmp[pos_tmp] := a[pos_i];
        pos_i := pos_i + 1;
      end
    else
      begin
        arreglo_tmp[pos_tmp] := a[pos_d];
        pos_d := pos_d + 1;
      end;
    pos_tmp := pos_tmp + 1;
  end;
  while pos_i <= izq do (copia el resto de la primera mitad)
  begin
    arreglo_tmp[pos_tmp] := a[pos_i];
    pos_tmp := pos_tmp + 1; pos_i := pos_i + 1;
  end;
  while pos_d <= der do (copia el resto de la segunda mitad)
  begin
    arreglo_tmp[pos_tmp] := a[pos_d];
    pos_tmp := pos_tmp + 1; pos_d := pos_d + 1;
  end;
  for i := 1 to num_elementos do {copia de nuevo arreglo_tmp a}
  begin
    a[der] := arreglo_tmp[der];
    der := der - 1;
  end;
end;

```

Figura 7.10 Rutina intercala

porque todos los demás términos se anulan y hay $\log n$ ecuaciones, así que todos los unos al final de esas ecuaciones se suman hasta dar $\log n$. Multiplicando por n se obtiene la respuesta final.

$$T(n) = n \log n + n = O(n \log n)$$

Observe que si no se dividiera entre n al inicio de las soluciones, la suma no se podría proyectar. Por ello fue necesario dividir entre n .

Un método alternativo es sustituir la relación recurrente continuamente sobre el lado derecho. Tenemos

$$T(n) = 2T(n/2) + n$$

Puesto que podemos sustituir $n/2$ en la ecuación principal,

$$2T(n/2) = 2(2T(n/4)) + n/2 = 4T(n/4) + n/2$$

tenemos

$$T(n) = 4T(n/4) + 2n$$

De nuevo, sustituyendo $n/4$ en la ecuación principal, vemos que

$$4T(n/4) = 4(2T(n/8)) + n/4 = 8T(n/8) + n/4$$

Así se tiene

$$T(n) = 8T(n/8) + 3n$$

Continuando de este modo obtenemos

$$T(n) = 2^k T(n/2^k) + k \cdot n$$

Usando $k = \log n$, obtenemos

$$T(n) = nT(1) + n \log n + n = n \log n + n$$

La elección del método a usar es cuestión de gusto. El primer método tiende a producir un trabajo pequeño que cabe mejor en una cuartilla, lleva a pocos errores matemáticos, pero requiere cierta experiencia. El segundo método parece más un enfoque de fuerza bruta.

Recuerde que supusimos $n = 2^k$. El análisis se puede refinar para manejar casos donde n no es una potencia de 2. La respuesta resulta ser casi idéntica (por lo regular tal es el caso).

Aunque el tiempo de ejecución de la ordenación por intercalación es $O(n \log n)$, casi nunca se usa para ordenar en memoria principal. El mayor problema es que la intercalación de dos listas ordenadas requiere memoria lineal, y el trabajo extra consumido en la copia al y del arreglo temporal, durante el algoritmo, tiene el efecto de reducir considerablemente la velocidad de la ordenación. Esta copia se puede evitar intercambiando con cuidado los papeles de a y $arreglo_tmp$ en niveles alternos de la recursión. Se puede implantar no recursivamente (ejercicio 7.13) una variante de la ordenación por intercalación, pero aun así, para aplicaciones serias de ordenación interna, el algoritmo adecuado es la ordenación rápida (quicksort), que describimos en la siguiente sección. No obstante, como veremos después en

en este capítulo, la rutina de intercalación es la piedra angular para la mayoría de los algoritmos de ordenación externa.

7.7. Ordenación rápida

Como su nombre lo indica, la *ordenación rápida* (*quicksort*) es el algoritmo de ordenación más rápido conocido en la práctica. Su tiempo de ejecución promedio es $O(n \log n)$. Es muy rápido, sobre todo debido a un ciclo interno muy cerrado y altamente optimizado. Tiene un rendimiento de $O(n^2)$ para el peor caso, pero con un poco de esfuerzo se puede volver exponencialmente improbable. Es sencillo entender el algoritmo de ordenación rápida y demostrar su corrección, aunque por muchos años tuvo fama de ser un algoritmo que en teoría podría ser altamente optimizado, pero que en la práctica era imposible codificar correctamente (sin duda a causa de FORTRAN). Como la ordenación por intercalación, la ordenación rápida es un algoritmo recursivo de "divide y vencerás". El algoritmo básico para ordenar un arreglo S consiste en los siguientes cuatro pasos:

1. Si el número de elementos en S es 0 o 1, terminar.
2. Elegir cualquier elemento v en S . Llámesele *pivote*.
3. Particionar $S - \{v\}$ (los demás elementos de S) en dos grupos ajenos (o disjuntos): $S_1 = \{x \in S - \{v\} \mid x \leq v\}$, y $S_2 = \{x \in S - \{v\} \mid x \geq v\}$.
4. Devolver {ordenación_rápida(S_1) seguida de v seguido de ordenación_rápida(S_2)}.

Puesto que el paso de la partición describe ambiguamente qué hacer con los elementos iguales al pivote, esto se torna una decisión de diseño. Parte de una buena implantación es manejar este caso con tanta eficiencia como sea posible. Intuitivamente esperaríamos que cerca de la mitad de las llaves que son iguales al pivote caigan en S_1 y la otra mitad en S_2 , de la misma forma que nos gusta que los árboles binarios de búsqueda estén equilibrados.

La figura 7.11 muestra la acción de la ordenación rápida sobre un conjunto de números. El pivote elegido (al azar) es 65. Los elementos restantes en el conjunto se parten en dos conjuntos de menor tamaño. Ordenando recursivamente el conjunto de números menores se obtiene 0, 13, 26, 31, 43 y 57 (por la regla 3 de la recursión). El conjunto de números grandes se ordena en una forma semejante, y entonces es trivial obtener la ordenación del conjunto completo.

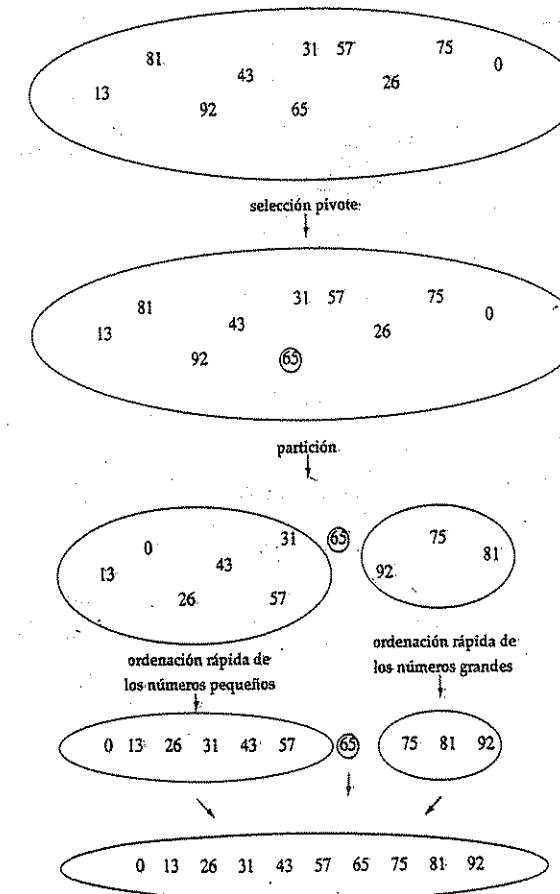
Debe quedar claro que este algoritmo funciona, pero no es claro por qué es más rápido que la ordenación por intercalación. Como esta última, la ordenación rápida resuelve recursivamente dos subproblemas y requiere trabajo lineal adicional (paso 3), pero, a diferencia de la ordenación por intercalación, no está garantizado que los subproblemas sean de igual tamaño, lo cual es potencialmente malo. La razón de que la ordenación rápida tarde menos es que el paso de partición se puede realizar en el mismo lugar y con mucha eficiencia y esta eficiencia más que compensa la falta de llamadas recursivas de tamaños iguales.

Le faltan unos cuantos detalles a la descripción del algoritmo, que ahora explicaremos. Hay muchas formas de implantar los pasos 2 y 3; el método aquí presentado es el resultado de un amplio análisis y de estudios empíricos y representa una forma muy eficiente de implantar la ordenación rápida. Incluso las más ligeras desviaciones de este método pueden provocar resultados sorprendentemente malos.

7.7.1. Selección del pivote

Aunque el algoritmo funciona tal como se describió, independientemente de qué elemento se escoja como pivote, es obvio que algunas elecciones son mejores que otras.

Figura 7.11 Pasos de la ordenación rápida ilustrados con un ejemplo.



Una forma errónea

La selección usual no informada es usar el primer elemento como pivote. Esto es aceptable si la entrada es aleatoria, pero si la entrada está ordenada de antemano o en orden inverso, entonces el pivote da una partición deficiente, porque prácticamente todos los elementos caerán en S_1 o S_2 . Lo peor es que esto ocurre consistentemente a través de las llamadas recursivas. El efecto práctico es que si el primer elemento es el pivote y la entrada está ordenada por adelantado, la ordenación rápida tarda un tiempo cuadrático para esencialmente no hacer nada, lo cual es indeseable. Más aún, la entrada ordenada (o la entrada con una gran sección ordenada previamente) es muy frecuente, así que usar el primer elemento como pivote es una idea absolutamente horrible y se debe descartar de inmediato. Una alternativa es escoger como pivote la mayor de las primeras dos llaves, pero esto tiene las mismas propiedades malas que la selección de la primera llave. Tampoco se debe usar esta estrategia.

Una maniobra segura

Un procedimiento seguro es simplemente escoger el pivote al azar. Por lo regular esta estrategia es segura, a menos que el generador de números aleatorios tenga un defecto (lo cual no es tan raro como se pudiera pensar), ya que es muy improbable que un pivote aleatorio pueda dar consistentemente una partición deficiente. Por otro lado, la generación de números aleatorios suele ser costosa y no reduce el tiempo de ejecución promedio del resto del algoritmo.

Partición de la mediana de tres

La mediana de un grupo de n números es el $\lceil n/2 \rceil$ -ésimo número mayor. La mejor selección del pivote sería la mediana del archivo. Desafortunadamente, es costoso calcular esto y reduciría considerablemente la velocidad del algoritmo. Se puede obtener un buen cálculo escogiendo tres elementos al azar y usando como pivote la mediana de estos tres. La aleatoriedad no ayuda demasiado, así que el curso común es usar como pivote la mediana de los elementos de la izquierda, la derecha y el centro. Por ejemplo, con la entrada 8, 1, 4, 9, 6, 3, 5, 2, 7, 0, como antes, el elemento de la izquierda es 8, el de la derecha es 0 y el del centro (en la posición $(izquierda + derecha) / 2$) es 6. Así, el pivote sería $v = 6$. Con este método se elimina claramente el caso malo para la entrada ordenada (las particiones son de igual tamaño en este caso) y de hecho reduce el tiempo de ejecución de la ordenación rápida en cerca del 5 por ciento.

7.7.2. Estrategia de partición

Hay varias estrategias de partición usadas en la práctica, pero se sabe que la descrita aquí da buenos resultados. Es muy fácil, como se verá, hacerlo mal o sin eficiencia, pero es seguro si se usa un método conocido. El primer paso es tomar el pivote e

intercambiarlo con el penúltimo. i inicia en el primer elemento y j inicia en el penúltimo elemento. Si la entrada original fue la misma que antes, la siguiente figura muestra la situación actual.

Por ahora supondremos que todos los elementos son diferentes. Más adelante nos ocuparemos de qué hacer en la presencia de duplicados. Como un caso límite, este algoritmo debe comportarse correctamente si todos los elementos son idénticos. Resulta sorprendente lo fácil que es hacer lo *erróneo*.

Lo que esta etapa de partición quiere hacer es mover todos los elementos pequeños a la parte izquierda del arreglo y todos los grandes a la parte derecha. "Pequeño" y "grande" son, por supuesto, relativos al pivote.

Mientras i está a la izquierda de j , moveremos i a la derecha, hasta llegar a un elemento menor que el pivote. j se mueve a la izquierda, hasta llegar a un elemento mayor que el pivote. Cuando se detienen i y j , i está apuntando a un elemento grande y j a uno pequeño. Si i está a la izquierda de j , se intercambian esos elementos. El efecto es empujar un elemento grande a la derecha y uno pequeño a la izquierda. En el ejemplo anterior, i no se movería y j se deslizaría sobre un lugar. La situación es como sigue.

Entonces intercambiamos los elementos apuntados por i y j y repetimos el proceso hasta que i y j se crucen.

Antes del segundo intercambio									
2	1	4	9	0	3	5	8	7	
			↑			↓			

Después del segundo intercambio									
2	1	4	5	0	3	9	8	7	
			↑			↓			
			i			j			

Antes del tercer intercambio								
2	1	4	5	0	3	9	8	7
					↑	↑		
	j	i						

En esta etapa, i y j se han cruzado, así que no se realiza el intercambio. La parte final de la partición consiste en intercambiar el elemento pivote con el elemento apuntado por i .

Después del intercambio con el pivote								
2	1	4	5	0	3	6	8	7
					↑		↑	
	i					pivote		

Cuando el pivote se intercambia con i en el último paso, sabemos que todo elemento en una posición $p < i$ debe ser menor. Esto es así porque la posición p contenía un elemento pequeño para empezar, o el elemento grande originalmente en p fue reemplazado durante un intercambio. Un razonamiento semejante demuestra que los elementos en las posiciones $p > i$ deben ser grandes.

Un detalle importante que debemos considerar es cómo manejar las llaves iguales al pivote. Las preguntas son si i debe o no parar cuando alcanza una llave igual al pivote y si j debe o no parar cuando alcanza una llave igual al pivote. Intuitivamente, i y j deben hacer lo mismo, ya que de otra forma el paso de la partición es sesgado. Por ejemplo, si i para y j no, entonces todas las llaves iguales al pivote caerán en S_2 .

Para tener una idea de lo que puede ser bueno, consideraremos el caso en el que todas las llaves de un archivo son idénticas. Si i y j paran, habrá muchos intercambios entre elementos idénticos. Aunque esto parece inútil, el efecto positivo es que i y j se cruzarán en la mitad, así que cuando se reemplaza el pivote, la partición crea dos subarchivos casi iguales. El análisis de la ordenación por intercalación indica que el tiempo de ejecución total entonces será $O(n \log n)$.

Si ni i ni j paran, y está presente el código para evitar que vayan más allá del final del arreglo, no se realizará ningún intercambio. Aunque esto parece bueno, una implantación correcta intercambiaria entonces el pivote en el último espacio que i alcance, lo cual sería la penúltima posición (o la última, dependiendo de la implantación exacta). Esto creará archivos muy desiguales. Si todas las llaves son idénticas, el tiempo de ejecución es $O(n^2)$. El efecto es el mismo que usar el primer elemento de una entrada ordenada de antemano. Esto lleva un tiempo cuadrático y no hace nada.

Así, encontraremos que es mejor hacer intercambios innecesarios y crear subarchivos similares que arriesgarse a archivos exageradamente dispares. Por lo tanto, haremos parar i y j si encuentran una llave igual al pivote. Esta es la única de las cuatro posibilidades que no toma tiempo cuadrático con esta entrada.

A primera vista puede parecer que preocuparse por un archivo que tenga elementos idénticos es tonto. Después de todo, ¿por qué alguien querría ordenar

5 000 elementos idénticos? No obstante, recordaremos que la ordenación rápida es recursiva. Suponga que hay 100 000 elementos, de los cuales 5 000 son idénticos. Tarde o temprano, la ordenación rápida hará la llamada recursiva sobre sólo estos 5 000 elementos. Entonces realmente será importante asegurarse de que 5 000 elementos idénticos se puedan ordenar con eficiencia.

7.7.3. Archivos pequeños

Para archivos muy pequeños ($n \leq 20$), la ordenación rápida no funciona tan bien como la ordenación por inserción. Más aún, debido a que la ordenación rápida es recursiva, estos casos ocurrirán con frecuencia. Una solución común es no usar la ordenación rápida recursivamente para archivos pequeños, sino emplear un algoritmo de ordenación que sea eficiente para archivos pequeños, como la ordenación por inserción. Una idea aún mejor es dejar el archivo ligeramente desordenado y terminarlo con la ordenación por inserción. Esto funciona bien porque la ordenación por inserción es eficiente para archivos casi ordenados. Mediante esta estrategia, de hecho se puede ahorrar cerca del 15% del tiempo de ejecución del que se tarda (sin hacer ninguna mejora). Un buen nivel de corte es $n = 10$, aunque es probable que cualquier corte entre 5 y 20 produzca resultados semejantes. Esto también evita molestos casos degenerados, como sacar la mediana de tres elementos cuando hay sólo uno o dos. Por supuesto, si hay un error en la rutina básica de ordenación rápida, la ordenación por inserción será muy, muy lenta.

7.7.4. Rutinas reales de ordenación rápida

El manejador de la ordenación rápida se muestra en la figura 7.12.

La forma general de las rutinas será pasar el arreglo y los límites del arreglo (*izquierda* y *derecha*) que se ha de ordenar. La primera rutina a tratar es la selección del pivote. La forma más fácil de hacer esto es ordenar *a[izquierda]*, *a[derecha]* y *a[centro]* en su lugar. Esto tiene la ventaja adicional de que el menor de los tres se lleva a *a[izquierda]*, que es donde la partición lo pondría de todos modos. El más grande se lleva a *a[derecha]*, que también es su lugar correcto, ya que es mayor que el pivote. Por lo tanto, podemos colocar el pivote en *a[derecha - 1]* e iniciar i y j con *izquierda + 1* y *derecha - 2* en la fase de partición. Otro beneficio es que como *a[izquierda]* es menor que el pivote, actuará como un centinela para j . Así, no es necesario preocuparse de que j salga por el extremo. Puesto que i se parará sobre llaves iguales al pivote, guardar el pivote en *a[derecha - 1]* da un centinela para i . El

Figura 7.12 Manejador de la ordenación rápida

```
procedure quick_sort(var a: datos_entrada; n: integer);
begin
    q_sort(a, 1, n);
    ordenación_por_inserción(a, n);
end;
```

```

{devuelve la mediana de la izquierda, centro y derecha. Los ordena y oculta el pivote}
procedure mediana3(var a: datos_entrada; izquierda, derecha: integer;
var pivot: tipo_entrada);
var centro: integer;

begin
centro := (izquierda+derecha) div 2;

if a[izquierda] > a[centro] then
    intercambiar(a[izquierda], a[centro]);
if a[izquierda] > a[derecha] then
    intercambiar(a[izquierda], a[derecha]);
if a[centro] > a[derecha] then
    intercambiar(a[centro], a[derecha]);

{invariante: a[izquierda] <= a[centro] <= a[derecha]}

pivot := a[centro];
intercambiar(a[centro], a[derecha-1]); {oculta el pivote}
end;

```

Figura 7.13 Codificación para efectuar la partición con base en la mediana de tres

código de la figura 7.13 hace la partición de la mediana de tres con todos los efectos laterales descritos. Puede parecer que sólo es ligeramente inefficiente calcular el pivote por un método que realmente no ordena $a[\text{izquierda}]$, $a[\text{centro}]$ y $a[\text{derecha}]$, pero, sorprendentemente, esto produce malos resultados (véase el ejercicio 7.37).

El núcleo real de la ordenación rápida está en la figura 7.14. Incluye la partición y las llamadas recursivas. Hay varias cosas que vale la pena considerar en esta implantación. La línea [3] inicia i y j con valores que se diferencian en 1 de sus valores correctos, así que no hay casos especiales que considerar. Este inicio depende del hecho de que la partición con base en la mediana de tres tiene algunos efectos laterales; este programa no funcionará si intenta usarlo sin cambiarlo utilizando estrategia de pivote sencilla, porque i y j empiezan en el lugar incorrecto y ya no hay centinela para j .

La línea [8] es necesaria porque se hace un intercambio no deseado después de haberse cruzado i y j . Esto se podría haber comprobado en la línea [6] (como en el ejemplo), pero eso puede retardar la rutina, ya que esta comprobación tendría éxito sólo una vez y fallaría muchas veces. Si esta rutina se escribiera en un lenguaje que permitiera salidas tempranas de ciclos (como C o Ada), esto no sería necesario y el código sería más claro.

El *intercambiar* de la línea [6] a veces se escribe explícitamente, con fines de velocidad. Para que el algoritmo sea rápido, es necesario forzar al compilador a compilar esta codificación en línea. La mayoría de los compiladores lo hacen automáticamente, si se les indica, pero para los que no lo hacen la diferencia puede ser significativa.

```

procedure q_sort(var a: datos_entrada; izquierda, derecha: integer);
var i, j, pivot: integer;

begin
if izquierda + CORTE <= derecha then begin
    mediana3(a, izquierda, derecha, pivot);

    i := izquierda; j := derecha - 1;
repeat
    repeat i := i + 1 until a[i] >= pivot;
    repeat j := j - 1 until a[j] <= pivot;
    intercambiar(a[i], a[j]);
until j <= i;

    intercambiar(a[i], a[j]); {deshace el último intercambio}
    intercambiar(a[i], a[derecha-1]); {restaura el pivote}

    q_sort(a, izquierda, i-1);
    q_sort(a, i+1, derecha);
end; {if}
end;

```

Figura 7.14 Rutina principal de la ordenación rápida

Por último, las líneas [4] y [5] muestran por qué la ordenación rápida tarda tan poco. El ciclo interno del algoritmo consiste en un incremento/decuento (de 1, lo cual es rápido), una condición y un salto. No hay ninguna manipulación adicional como en la ordenación por intercalación. Esta codificación es sorprendentemente trampa. Es tentador sustituir las líneas [3] a [7] por los enunciados de la figura 7.15. Esto no funciona porque habría un infinito si $a[i] = a[j] = \text{pivot}$.

7.7.5. Análisis de la ordenación rápida

Como la ordenación por intercalación, la ordenación rápida es recursiva, y por ello su análisis requiere resolver una fórmula recurrente. Haremos el análisis de la ordenación rápida, suponiendo un pivote aleatorio (sin la partición con base en la mediana de tres).

Figura 7.15 Un pequeño cambio a la ordenación rápida, que rompe el algoritmo

```

[3] i := izquierda + 1; j := derecha - 2;
repeat
[4] while a[i] < pivot do i := i + 1;
[5] while a[j] > pivot do j := j - 1;
[6] intercambiar(a[i], a[j]);
[7] until j <= i;

```

mediana de tres) y ningún corte para archivos pequeños. Se tomará $T(0) = T(1) = 1$, como en la ordenación por intercalación. El tiempo de ejecución de la ordenación rápida es igual al tiempo de ejecución de dos llamadas recursivas más el tiempo lineal consumido en la partición (la selección del pivote toma sólo tiempo constante). Esto da la relación básica de la ordenación rápida

$$T(n) = T(i) + T(n-i-1) + cn \quad (7.1)$$

donde $i = |C_1|$ es el número de elementos en C_1 . Examinaremos los tres casos.

Análisis del peor caso

El pivote es el elemento menor, todo el tiempo. Entonces $i = 0$, y si ignoramos $T(0) = 1$, lo cual es insignificante, la recurrencia es

$$T(n) = T(n-1) + cn, \quad n > 1 \quad (7.2)$$

Proyectamos, usando la ecuación 7.2 repetidamente. Así

$$T(n-1) = T(n-2) + c(n-1) \quad (7.3)$$

$$T(n-2) = T(n-3) + c(n-2) \quad (7.4)$$

$$\dots \quad T(2) = T(1) + c(2) \quad (7.5)$$

Sumando todas estas ecuaciones se obtiene

$$T(n) = T(1) + c \sum_{i=2}^n i = O(n^2) \quad (7.6)$$

como se sosténía antes.

Análisis del mejor caso

En el mejor caso, el pivote está a la mitad. Para simplificar los cálculos, suponemos que los dos subarchivos son exactamente de la mitad del tamaño del original, y aunque esto da una ligera sobreestimación, es aceptable porque sólo nos interesa la respuesta O grande.

$$T(n) = 2T(n/2) + cn \quad (7.7)$$

Dividimos ambos lados de la ecuación 7.7 entre n .

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c \quad (7.8)$$

Proyectaremos usando esta ecuación.

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c \quad (7.9)$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c \quad (7.10)$$

$$\dots \quad \frac{T(2)}{2} = \frac{T(1)}{1} + c \quad (7.11)$$

Sumamos todas las ecuaciones desde 7.7 hasta 7.11, y observamos que hay $\log n$ ecuaciones:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c \log n \quad (7.12)$$

lo cual da

$$T(n) = cn \log n + n = O(n \log n) \quad (7.13)$$

Observe que éste es exactamente el mismo análisis que se hizo en la ordenación por intercalación; por tanto, obtendremos la misma respuesta.

Análisis del caso medio

Ésta es la parte más difícil. Para el caso medio, suponemos que cada tamaño del archivo para C_1 es igualmente probable, por lo que tiene una probabilidad $1/n$. Esta suposición es realmente válida para nuestra estrategia de pivoteo y partición, pero no es válida para algunas otras. Las estrategias de partición que no preservan la aleatoriedad de los subarchivos no pueden servirse de este análisis. Es un hecho interesante que esas estrategias parecen proporcionar programas que toman más tiempo de ejecución en la práctica.

Con esta suposición, el valor promedio de $T(i)$, y por lo tanto, de $T(n-i-1)$, es $\frac{1}{n} \sum_{j=0}^{n-1} T(j)$. La ecuación 7.1 se convierte entonces en

$$T(n) = \frac{2}{n} \left[\sum_{i=0}^{n-1} T(i) \right] + cn \quad (7.14)$$

Si la ecuación 7.14 se multiplica por n , se obtiene

$$nT(n) = 2 \left[\sum_{i=0}^{n-1} T(i) \right] + cn^2 \quad (7.15)$$

Necesitamos eliminar el signo de la sumatoria con fines de simplificación. Observe que se puede proyectar con una ecuación más.

$$(n-1)T(n-1) = 2 \left[\sum_{j=0}^{n-2} T(j) \right] + c(n-1)^2 \quad (7.16)$$

Si restamos 7.16 de 7.15, obtenemos

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c \quad (7.17)$$

Reordenando los términos y quitando el insignificante $-c$ de la derecha, obtenemos

$$nT(n) = (n+1)T(n-1) + 2cn \quad (7.18)$$

Ahora tenemos una fórmula para $T(n)$ en términos de $T(n-1)$ solamente. De nuevo la idea es proyectar, pero la ecuación 7.18 está en una forma errónea. Dividiendo 7.18 entre $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \quad (7.19)$$

Ahora proyectamos.

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n} \quad (7.20)$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1} \quad (7.21)$$

...

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \quad (7.22)$$

Sumando las ecuaciones 7.19 a la 7.22 se obtiene

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{n+1} \frac{1}{i} \quad (7.23)$$

La suma es aproximadamente $\log(n+1) + \gamma - \frac{3}{2}$, donde $\gamma = 0.577$ se conoce como la constante de Euler; así

$$\frac{T(n)}{n+1} = O(\log n) \quad (7.24)$$

Y por tanto

$$T(n) = O(n \log n) \quad (7.25)$$

Aunque este análisis parece complejo, en realidad no lo es: los pasos son naturales una vez que se han visto algunas relaciones recurrentes. De hecho, el análisis puede llevarse más lejos. La versión altamente optimizada que se describió antes también ha sido analizada, y este resultado se vuelve extremadamente difícil, pues implica complicadas recurrencias y matemáticas avanzadas. También se han analizado en detalle los efectos de llaves iguales, y resulta que los códigos presentados hacen la tarea correcta.

7.7.6. Un algoritmo de selección con un tiempo esperado lineal

La ordenación rápida se puede modificar para resolver el *problema de la selección*, que hemos visto en los capítulos 1 y 6. Recuerde que con una cola de prioridad podemos encontrar el k -ésimo elemento mayor (o menor) en $O(n + k \log n)$. Para el caso especial de encontrar la mediana, esto da un algoritmo $O(n \log n)$.

Puesto que podemos ordenar el archivo en un tiempo $O(n \log n)$, uno podría esperar obtener una mejor cota de tiempo para la selección. El algoritmo que presentamos para encontrar el k -ésimo elemento menor de un conjunto C es casi idéntico a la ordenación rápida. De hecho, los primeros tres pasos son los mismos. Llamaremos a este algoritmo *SelecciónRápida*. Sea $|S_i|$ el número de elementos en S_i . Los pasos de SelecciónRápida son:

1. Si $|C| = 1$, entonces $k = 1$ y devuelve el elemento en C como respuesta. Si se usa un corte de archivos pequeños y $|C| \leq \text{CORTE}$, entonces se ordena C y se devuelve el k -ésimo elemento menor.
2. Se toma un elemento pivote, $v \in C$.
3. Se parte $C - \{v\}$ en C_1 y C_2 , como se hizo con la ordenación rápida.
4. Si $k \leq |C_1|$, entonces el k -ésimo elemento menor debe estar en C_1 . En este caso, se devuelve SelecciónRápida(C_1, k). Si $k = 1 + |C_1|$, entonces el pivote es el k -ésimo elemento menor y se puede devolver como respuesta. Si no, el k -ésimo elemento menor cae en C_2 y es el $(k - |C_1| - 1)$ -ésimo elemento menor en C_2 . Se hace una llamada recursiva y se devuelve SelecciónRápida($C_2, k - |C_1| - 1$).

En contraste con la ordenación rápida, la selección rápida hace sólo una llamada recursiva en vez de dos. El peor caso de la selección rápida es idéntico al de la ordenación rápida y es $O(n^2)$. Intuitivamente, esto se debe a que el peor caso de la ordenación rápida se da cuando uno de C_1 y C_2 está vacío; así, la selección rápida no ahorra realmente una llamada recursiva. El tiempo de ejecución medio, no obstante, es $O(n)$. El análisis es semejante al de la ordenación rápida y se deja como ejercicio.

La implantación de la selección rápida es aún más sencilla de lo que puede implicar la descripción abstracta. La codificación para hacer esto se muestra en la figura 7.16. Cuando el algoritmo termina, el k -ésimo elemento menor está en la posición k . Esto destruye el orden original; si no es deseable, hay que hacer una copia.

```

{selec_r coloca el k-ésimo elemento en a[k]}

procedure selec_r(var a: datos_entrada; k, izq, der: integer);
var i, j, pivot: integer;

begin
[1]   if (izq + CORTE <= der) then begin
[2]     mediana3(a, izq, der, pivot);

[3]     i := izq; j := der - 1;
repeat
[4]       repeat i := i + 1 until a[i] >= pivot;
[5]       repeat j := j - 1 until a[j] <= pivot;
[6]       intercambiar(a[i], a[j]);
[7]       until j <= i;

[8]       intercambiar(a[i], a[j]); {deshace el último intercambio}
[9]       intercambiar(a[i], a[der - 1]); {restaura el pivote}

[10]      if k < i then
[11]        selec_r(a, k, izq, i - 1)
[12]      else
[13]        selec_r(a, k, i + 1, der);
end {if}
[14]      ordenación_por_inserción(a, izq, der);
end;

```

Figura 7.16 Rutina principal de la selección rápida

Con una estrategia de pivoteo con la mediana de tres la probabilidad de que se dé el peor caso es casi despreciable. Eligiendo con cuidado el pivote, sin embargo, podemos eliminar el peor caso cuadrático y asegurar un algoritmo $O(n)$. La sobre carga implícita de hacer esto es considerable, así que el algoritmo resultante es de interés teórico principalmente. En el capítulo 10 examinaremos el algoritmo de la selección de tiempo lineal para el peor caso, y veremos también una interesante técnica de elección del pivote que produce una selección un poco más rápida en la práctica.

7.8. Ordenación de registros grandes

A lo largo de nuestro estudio de la ordenación, hemos supuesto que los elementos por ordenar son sólo enteros. Con frecuencia, necesitamos ordenar registros grandes por cierta llave. Por ejemplo, podríamos tener los registros de una nómina, donde cada registro consiste en un nombre, dirección, número telefónico, datos financieros como salarios e información fiscal. Quizá se quiera ordenar esta infor-

mación por un campo particular, como el nombre. Para todos nuestros algoritmos, la operación fundamental es el intercambio, pero aquí intercambiar dos registros puede ser muy costoso, porque los registros son potencialmente grandes. Si éste es el caso, una solución práctica es que el arreglo de entrada contenga apuntadores a los registros. Se ordena comparando las llaves que los apuntadores indican, intercambiando apuntadores cuando es necesario. Esto significa que, en esencia, todo el movimiento de datos es el mismo que cuando ordenábamos enteros. Esto se denomina *ordenación indirecta*; podemos usar esta técnica para la mayoría de las estructuras de datos descritas. Esto justifica nuestra suposición de que los registros complejos se pueden manejar sin mucha pérdida de eficiencia.

7.9. Una cota inferior general para la ordenación

Aunque tenemos algoritmos $O(n \log n)$ para la ordenación, no está claro que sean de lo mejor. En esta sección, demostramos que cualquier algoritmo de ordenación que sólo usa comparaciones requiere $\Omega(n \log n)$ de éstas (y de aquí el tiempo) en el peor de los casos, así que las ordenaciones por intercalación y por montículo son óptimas con un factor constante. Se puede extender la prueba para demostrar que se requieren $\Omega(n \log n)$ comparaciones, incluso en el promedio, para cualquier algoritmo de ordenación que sólo use comparaciones, lo que significa que la ordenación rápida es óptima en promedio dentro de un factor constante.

Específicamente, demostraremos el siguiente resultado: cualquier algoritmo de ordenación que use sólo comparaciones requiere $\lceil \log n! \rceil$ comparaciones en el peor caso y $\log n!$ en el promedio. Supondremos que todos los n elementos son distintos, ya que cualquier algoritmo de ordenación debe funcionar para este caso.

7.9.1. Árboles de decisión

Un *árbol de decisión* es una abstracción con que se demuestran cotas inferiores. En nuestro contexto, un árbol de decisión es un árbol binario. Cada nodo representa un conjunto de ordenamientos posibles, consistente con las comparaciones realizadas entre los elementos. Los resultados de las comparaciones son las aristas del árbol.

El árbol de decisión de la figura 7.17 representa un algoritmo que ordena los tres elementos a , b y c . El estado inicial del algoritmo está en la raíz. (Usaremos los términos *estado* y *nodo* indistintamente.) No se han hecho comparaciones, así que todos los ordenamientos son legales. La primera comparación que efectúa *este algoritmo particular* es la comparación de a con b . Los dos resultados conducen a dos estados posibles. Si $a < b$, entonces sólo quedan tres posibilidades. Si el algoritmo alcanza el nodo 2, entonces comparará a con c . Otros algoritmos pueden hacer cosas diferentes; un algoritmo distinto tendría un árbol de decisión diferente. Si $a > c$, el algoritmo entra al estado 5. Puesto que sólo hay un ordenamiento consistente, el algoritmo puede terminar e informar que la ordenación ha concluido. Si $a < c$, el algoritmo no puede hacer esto porque hay dos ordenamientos posibles y no

puede estar seguro de cuál es el correcto. En tal caso, el algoritmo requerirá una comparación más.

Todo algoritmo que ordena sólo con comparaciones se puede representar por medio de un árbol de decisión. Por supuesto, sólo es posible dibujar el árbol para tamaños de entrada extremadamente pequeños. El número de comparaciones usadas por el algoritmo de ordenación es igual a la profundidad de la hoja más baja. En este caso, este algoritmo usa tres comparaciones en el peor caso. El número medio de comparaciones usadas es igual a la profundidad media de las hojas. Puesto que un árbol de decisión es grande, se infiere que debe haber algunos caminos largos. Para probar las cotas inferiores, todo lo que se necesita demostrar son algunas propiedades de árbol básicas.

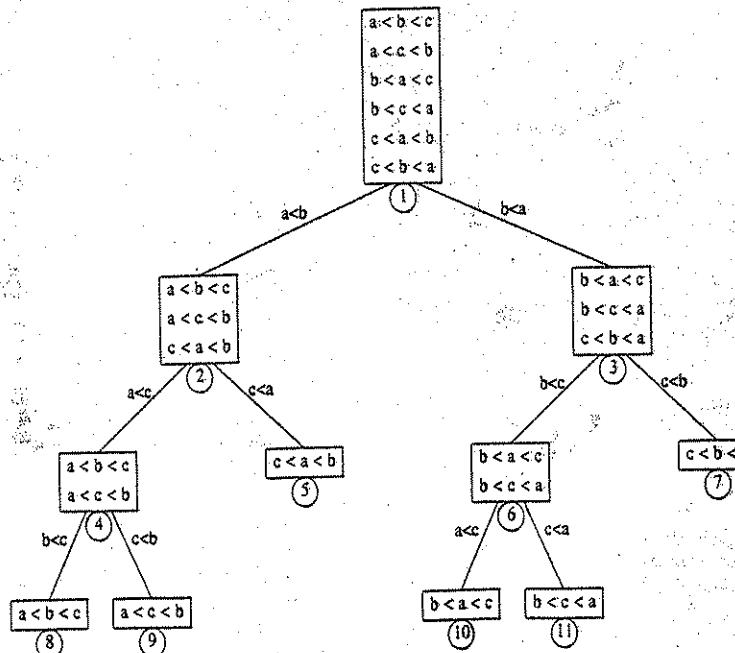
LEMA 7.1.

Sea A un árbol binario de profundidad p . Entonces A tiene a lo más 2^p hojas.

DEMOSTRACIÓN:

La demostración es por inducción. Si $p = 0$, entonces hay a lo más una hoja, así que el caso base es verdadero. De otra forma, se tiene una raíz, la cual no puede ser hoja, y un subárbol izquierdo y uno derecho, cada uno de profundidad

Figura 7.17 Árbol de decisión para ordenación por inserción de tres elementos



máxima $p - 1$. Por la hipótesis de inducción, éstos pueden tener a lo más 2^{p-1} hojas, dando un total de 2^p hojas como máximo. Esto demuestra el lema.

LEMA 7.2.

Un árbol binario con H hojas debe tener una profundidad mínima de $\lceil \log H \rceil$.

DEMOSTRACIÓN:

Inmediata a partir del lema anterior.

TEOREMA 7.5.

Cualquier algoritmo de ordenación que sólo usa comparaciones entre elementos requiere al menos $\lceil \log n! \rceil$ comparaciones en el peor caso.

DEMOSTRACIÓN:

Un árbol de decisión para ordenar n elementos debe tener $n!$ hojas. El teorema se infiere del lema precedente.

TEOREMA 7.6.

Cualquier algoritmo de ordenación que sólo usa comparaciones entre elementos requiere $\Omega(n \log n)$ comparaciones.

DEMOSTRACIÓN:

Del teorema anterior, se requieren $\log n!$ comparaciones:

$$\begin{aligned} \log n! &= \log(n(n-1)(n-2)\cdots(2)(1)) \\ &= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 + \log 1 \\ &\geq \log n + \log(n-1) + \log(n-2) + \cdots + \log n/2 \\ &\geq \frac{n}{2} \log \frac{n}{2} \\ &\geq \frac{n}{2} \log n - \frac{n}{2} \\ &= \Omega(n \log n) \end{aligned}$$

Este tipo de razonamientos de la cota inferior, cuando se usan para demostrar un resultado del peor caso, se denominan a veces cota inferior de información teórica. El teorema general dice que si hay P casos diferentes posibles por distinguir, y las preguntas son de la forma SÍ/NO, entonces se requieren siempre $\lceil \log P \rceil$ preguntas en algún caso por cualquier algoritmo que resuelva el problema. Es posible demostrar un resultado semejante para el tiempo de ejecución del caso medio de cualquier algoritmo de ordenación basado en comparaciones. Este resultado está implícito por el siguiente lema, el cual se deja como ejercicio: Cualquier árbol binario con H hojas tiene una profundidad media de por lo menos $\log H$.

7.10. Ordenación por cubetas

Aunque en la sección anterior demostramos que cualquier algoritmo general de ordenación que sólo use comparaciones requerirá un tiempo $\Omega(n \log n)$ en el peor

caso, recordamos que aún es posible ordenar en tiempo lineal en algunos casos especiales.

Un ejemplo sencillo es la ordenación por cubetas. Para que funcione la ordenación por cubetas se requiere información adicional. La entrada a_1, a_2, \dots, a_n debe consistir sólo en enteros positivos menores que m . (Obviamente, es posible hacer extensiones a esto.) Si tal es el caso, el algoritmo es sencillo: se tiene un arreglo llamado *cuenta*, de tamaño m , el cual está con valores iniciales en 0. Así, cuenta tiene m celdas, o cubetas, que están vacías inicialmente. Cuando se lee a_i , se incrementa *cuenta*[a_i] en 1. Después de leer toda la entrada, se recorre el arreglo *cuenta*, visualizando una representación de la lista ordenada. Este algoritmo tarda $O(m + n)$; la demostración se deja como ejercicio. Si m es $O(n)$, el tiempo de ejecución total es $O(n)$.

Aunque parece que este algoritmo viola la cota inferior, no es así porque usa una operación más potente que simples comparaciones. Al incrementar la cubeta adecuada, el algoritmo efectúa esencialmente una comparación m -camino en una unidad de tiempo. Esto es semejante a la estrategia con la dispersión extensible (sección 5.6). Desde luego, esto no está en el modelo para el cual se probaron las cotas inferiores.

Este algoritmo, sin embargo, cuestiona la validez del modelo usado en la demostración de la cota inferior. En realidad, el modelo es fuerte porque un algoritmo de ordenación de *propósito general* no puede hacer suposiciones acerca del tipo de entrada que espera ver, sino que sólo debe tomar decisiones basadas en la información de ordenamiento. Naturalmente, si hay información adicional disponible, debemos esperar encontrar un algoritmo más eficiente, ya que de otra forma la información adicional se desperdiciaría.

Aunque la ordenación por cubetas parece demasiado trivial para ser un algoritmo útil, resulta que hay muchos casos en los que la entrada sólo está constituida por enteros pequeños, así que el uso de un método como la ordenación rápida es realmente excesivo.

7.11. Ordenación externa

Hasta aquí, todos los algoritmos examinados requieren que la entrada quepa en memoria principal. No obstante, hay aplicaciones donde la entrada es demasiado grande para caber en ella. Esta sección presentará algoritmos de *ordenación externa*, los cuales están diseñados para manejar entradas muy grandes.

7.11.1. ¿Por qué necesitamos algoritmos nuevos?

La mayoría de los algoritmos de ordenación interna aprovechan el hecho de que la memoria es directamente direccionable. La ordenación de Shell compara los elementos $a[i]$ y $a[i - h_k]$ en una unidad de tiempo. La ordenación por montículo compara los elementos $a[i]$ y $a[i + 2]$ en una unidad de tiempo. La ordenación rápida, con la partición con base en la mediana de tres, requiere comparar $a[izq]$, $a[centro]$ y $a[der]$ en un número constante de unidades de tiempo. Si la entrada está en una cinta,

7.11. ORDENACIÓN EXTERNA

todas esas operaciones pierden su eficiencia, ya que sólo se puede tener acceso secuencial a los elementos en una cinta. Aun si los datos están en disco, todavía hay una pérdida práctica de eficiencia a causa del retardo requerido para girar el disco y mover la cabeza lectora.

Para ver lo lento que son realmente los accesos externos, creamos un archivo aleatorio grande, pero no tanto como para que no quepa en memoria. Se lee el archivo y se ordena usando un algoritmo eficiente. El tiempo que toma en ordenar la entrada es ciertamente insignificante comparado con el tiempo para leer la entrada, aun cuando la ordenación es una operación $O(n \log n)$ y la lectura de la entrada es sólo $O(n)$.

7.11.2. Modelo para ordenación externa

La gran variedad de dispositivos de almacenamiento masivo hacen que la ordenación externa sea mucho más dependiente del dispositivo que la ordenación interna. Los algoritmos que consideramos funcionan sobre cintas, que probablemente son los medios de almacenamiento más restrictivos. Puesto que el acceso a un elemento en la cinta se hace corriendo la cinta a la posición correcta, sólo es posible tener un acceso eficiente a las cintas en orden secuencial (en una u otra dirección).

Supondremos que al menos tenemos tres controladores de cinta para realizar la ordenación. Se necesitan dos controladores para hacer una ordenación eficiente; el tercer controlador simplifica la operación. Si sólo se cuenta con una unidad de cinta, entonces estamos en problemas: cualquier algoritmo requerirá $\Omega(n^2)$ accesos a cinta.

7.11.3. El algoritmo sencillo

El algoritmo básico de ordenación externa usa la rutina *intercalar* de la ordenación por intercalación. Suponemos que tenemos cuatro cintas $C_{a1}, C_{a2}, C_{b1}, C_{b2}$ con dos de entrada y dos de salida. Dependiendo del momento, las cintas a y b son ya sea de entrada o de salida. Supóngase que los datos están inicialmente en C_{a1} . Suponga además que la memoria interna puede contener (y ordenar) m registros a un tiempo. Un primer paso natural es leer m registros de la cinta de entrada cada vez, ordenarlos internamente y después escribir los registros ordenados alternadamente en C_{b1} y C_{b2} . Llamaremos a cada conjunto de registros ordenados una *serie*. Cuando se ha hecho esto, se rebobinan todas las cintas. Suponga que tenemos la misma entrada del ejemplo de la ordenación de Shell.

Si $m = 3$, entonces, después de construir las series, las cintas contendrán los datos

C_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
C_{a2}													
C_{b1}													
C_{b2}													

indicados en la siguiente figura.

C_{a1}							
C_{a2}							
C_{b1}	11	81	94	17	28	99	15
C_{b2}	12	35	96	41	58	75	

Ahora C_{b1} y C_{b2} contienen un grupo de series. Tomamos la primera serie de cada cinta y las intercalamos, escribiendo el resultado, que es una serie del doble de largo, en C_{a1} . Entonces se toma la siguiente serie de cada cinta, se intercalan, y se escribe el resultado en C_{a2} . Continuamos este proceso, alternando entre C_{a1} y C_{a2} , hasta que C_{b1} o C_{b2} esté vacía. En este momento o ambas están vacías o queda sólo una serie. En el último caso, se copia esta serie a la cinta apropiada. Se rebobinan las cuatro cintas y repetimos los mismos pasos, esta vez usando las cintas a como entrada y las cintas b como salida. Esto dará series de $4m$. Continuamos el proceso hasta obtener un serie de longitud n .

Este algoritmo requerirá $\lceil \log(n/m) \rceil$ pasadas, más la pasada inicial de construcción de series. Por ejemplo, si tenemos 10 millones de registros de 128 bytes cada uno, y cuatro megabytes de memoria interna, la primera pasada crea 320 series. Entonces necesitaremos nueve pasadas más para completar la ordenación. Nuestro ejemplo requiere $\lceil \log(13/3) \rceil = 3$ pasadas más, lo cual se muestra en la figura siguiente.

C_{a1}	11	12	35	81	94	96	15	
C_{a2}	17	28	41	58	75	99		
C_{b1}								
C_{b2}								
C_{a1}	11	12	17	28	35	51	58	75
C_{a2}	81	94	96	99				
C_{b1}								
C_{b2}	15							
C_{a1}	11	12	15	17	28	35	41	58
C_{a2}	75	81	94	96	99			
C_{b1}								
C_{b2}								

7.11.4. Intercalación de vías múltiples

Si tenemos cintas adicionales, entonces podemos esperar reducir el número de pasadas requeridas para ordenar la entrada. Hacemos esto extendiendo la intercalación básica (de dos vías) a una intercalación de k vías.

La intercalación de dos series se hace rebobinando la cinta al inicio de cada serie. Entonces se encuentra el elemento menor y se coloca en una cinta de salida, y se hace avanzar la cinta de entrada adecuada. Si hay k cintas de entrada, esta estrategia funciona en la misma forma, con la única diferencia de que es ligeramente más complejo encontrar el menor de los k elementos. Podemos encontrar el menor de estos elementos usando una cola de prioridad. Para obtener el siguiente elemento por escribir en la cinta de salida, ejecutamos una operación *eliminar min*. Se hace avanzar la cinta de entrada adecuada, y si no se completa la serie sobre la cinta de entrada, se *inserta* el elemento nuevo en la cola de prioridad. Usando el mismo ejemplo de antes, distribuimos la entrada en las tres cintas.

C_{a1}							
C_{a2}							
C_{a3}							
C_{b1}	11	81	94	41	58	75	
C_{b2}	12	35	96	15			
C_{b3}	17	28	99				

Ahora necesitamos dos pasadas más de intercalación de tres vías para completar la ordenación.

C_{a1}	11	12	17	28	35	81	94	96	99	
C_{a2}	15	41	58	75						
C_{a3}										
C_{b1}										
C_{b2}										
C_{b3}										

C_{a1}	11	12	15	17	28	35	41	58	75	81	94	96	99	
C_{a2}														
C_{a3}														
C_{b1}														
C_{b2}														
C_{b3}														

Después de la fase inicial de construcción de series, el número de pasadas requeridas usando la intercalación de vías múltiples es $\lceil \log_k(n/m) \rceil$, porque las series tardan k veces más en cada pasada. Para el ejemplo anterior, la fórmula se verifica, porque $\lceil \log_3 13/3 \rceil = 2$. Si tenemos 10 cintas, entonces $k = 5$, y nuestro ejemplo grande de la sección previa requerirá $\lceil \log_5 320 \rceil = 4$ pasadas.

7.11.5. Intercalación polifásica

La estrategia de intercalación de k vías presentada en la última sección requiere el uso de $2k$ cintas. Esto podría ser prohibitivo para algunas aplicaciones. Es posible hacerlo con sólo $k + 1$ cintas. Como ejemplo mostraremos cómo efectuar una intercalación de dos vías con sólo tres cintas.

Supongamos que tenemos tres cintas, C_1, C_2 y C_3 , un archivo de entrada en C_1 que producirá 34 series. Una opción es poner 17 series en C_2 y C_3 . Entonces podríamos intercalar este resultado en C_1 , obteniendo una cinta con 17 series. El problema es que como todas las series están en una cinta, debemos poner algunas de esas series en C_2 para realizar otra intercalación. La manera lógica para hacer esto es copiar las primeras ocho series de C_1 a C_2 y después efectuar la intercalación. Esto tiene el efecto de agregar una pasada adicional por cada pasada realizada.

Un método alternativo es partir las 34 series en forma desigual. Supongamos que ponemos 21 series en C_2 y 13 series en C_3 . Se podrían intercalar 13 series en C_1 antes de que C_3 quede vacía. En este momento, podríamos rebobinar C_1 y C_3 , e intercalar C_1 con 13 series, y C_2 , la cual tiene 8 series, en C_3 . Entonces podríamos intercalar 8 series hasta que C_2 quede vacía, lo que dejaría 5 series en C_1 y 8 series en C_3 . Entonces podríamos intercalar C_1 y C_3 , y así sucesivamente. La siguiente tabla muestra el número de series en cada cinta después de cada pasada.

	Const. serie	Después de $C_1 + C_2$	Después de $C_1 + C_2$	Después de $C_1 + C_3$	Después de $C_2 + C_3$	Después de $C_1 + C_2$	Después de $C_1 + C_3$	Después de $C_2 + C_3$
C_1	0	13	5	0	3	1	0	1
C_2	21	8	0	5	2	0	1	0
C_3	13	0	8	3	0	2	1	0

La distribución original de las series marca una gran diferencia. Por ejemplo, si se colocan 22 series en C_2 , con 12 en C_3 , entonces después de la primera intercalación obtendremos 12 series en C_1 y 10 en C_2 . Después de otra intercalación, hay 10 series en C_1 y 2 en C_3 . En este momento se torna lento el proceso porque sólo podemos intercalar dos conjuntos de series antes de que se agote C_3 . Entonces C_1 tiene 8 series y C_2 tiene dos series. De nuevo, sólo podemos intercalar dos conjuntos de series, obteniendo C_1 con 6 series y C_3 con 2 series. Después de tres pasadas más, C_2 tiene dos series y las otras cintas están vacías. Debemos copiar una serie en otra cinta, para después terminar la intercalación.

Resulta que la primera distribución que se dio es óptima. Si el número de series es un número de Fibonacci F_n , la mejor forma de distribuir las es partiendo las en dos números de Fibonacci F_{n-1} y F_{n-2} . Si no, es necesario llenar la cinta con series falsas a fin de que el número de series llegue hasta un número de Fibonacci. Los detalles para colocar el conjunto inicial de series en las cintas se dejan como ejercicio.

Podemos extender esto a una intercalación de k vías, en cuyo caso necesitaremos números de Fibonacci de orden k -ésimo para la distribución, donde el número de Fibonacci de orden k se define como $F^{(k)}(n) = F^{(k)}(n-1) + F^{(k)}(n-2) + \dots + F^{(k)}(n-k)$, con las condiciones iniciales adecuadas $F^{(k)}(n) = 0, 0 \leq n \leq k-2, F^{(k)}(k-1) = 1$.

7.11.6. Selección de sustitución

El último punto que consideraremos es la construcción de las series. La estrategia que hemos usado es la más sencilla posible: leemos tantos registros como sea posible y los ordenamos, escribiendo el resultado en alguna cinta. Esto parece ser el mejor enfoque posible, hasta que uno se da cuenta de que tan pronto como el primer registro está escrito en la cinta de salida, la memoria ocupada por él queda disponible para otro registro. Si el siguiente registro de la cinta de entrada es más grande que el registro recién sacado, entonces se puede incluir en la serie.

A partir de esta observación, podemos crear un algoritmo para producir series. A esta técnica suele llamársele *selección de sustitución*. Al principio se leen en memoria m registros y se colocan en una cola de prioridad. Efectuamos un *eliminar_mín*, escribiendo el registro menor a la cinta de salida. Leemos el siguiente registro de la cinta de entrada. Si es mayor que el registro recién escrito, podemos agregarlo a la cola de prioridad. De otra forma, no se puede colocar en la serie actual. Puesto que la cola de prioridad es menor por un elemento, se puede almacenar este elemento nuevo en el espacio muerto de la cola de prioridad hasta que la serie se complete y use el elemento para la siguiente serie. Almacenar un elemento en el espacio muerto es semejante a lo hecho en la ordenación por montículo. Continuamos haciendo esto hasta que el tamaño de la cola de prioridad es cero, punto en el cual la serie se acaba. Iniciamos una serie nueva construyendo una cola de prioridad nueva, usando todos los elementos del espacio muerto. La figura 7.18 muestra la construcción de la serie para el pequeño ejemplo que hemos estado usando, con $m = 3$. Los elementos muertos se indican con un asterisco.

En este ejemplo, la selección de sustitución produce sólo tres series, en comparación con las cinco series obtenidas con ordenación. Debido a esto, una intercalación

Figura 7.18 Ejemplo de la construcción de series

	3 elementos en el arreglo del montículo			Salida	Siguiente elemento leído
	M[1]	M[2]	M[3]		
Serie 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	96	17*
	17*	35*	12*	Fin de serie	Reconstruir montículo
Serie 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75*
	58	99	75*	58	fin de cinta
	99	75*	99	99	
Serie 3	75			75	Reconstruir montículo

ción de tres vías termina en una pasada en vez de dos. Si la entrada está distribuida aleatoriamente, se puede demostrar que la selección de sustitución produce series de longitud media $2m$. Para el ejemplo grande, esperaríamos 160 series en vez de 320, así una intercalación de cinco vías requeriría cuatro pasadas. En este caso, no hemos ahorrado una pasada, aunque podríamos si tenemos suerte y tenemos 125 series o menos. Puesto que la ordenación externa tarda tanto, cada pasada ahorrada puede marcar una diferencia significativa en el tiempo de ejecución.

Como hemos visto, es posible que la selección de sustitución no haga nada mejor que el algoritmo estándar. No obstante con frecuencia la entrada está ordenada, o casi, desde el inicio, en cuyo caso la selección de sustitución produce sólo unas pocas series muy grandes. Esta clase de entrada es común para ordenaciones externas y hace que la selección de sustitución tenga muchísimo valor.

Resumen

Para la mayoría de las aplicaciones generales de ordenación interna, la ordenación por inserción, la ordenación de Shell o la ordenación rápida será el método a elegir, y la decisión de cuál usar dependerá casi totalmente del tamaño de la entrada. La figura 7.19 muestra el tiempo de ejecución obtenido por cada algoritmo en varios tamaños de archivo.

Se escogió que los datos fueran permutaciones aleatorias de n enteros, y los tiempos dados sólo incluyen el tiempo real tardado en ordenar. La codificación dada en la figura 7.2 se utilizó para la ordenación por inserción. La ordenación de Shell usó el código de la sección 7.4 modificado para ejecutarse con los incrementos de Sedgewick. Con base en literalmente millones de ordenaciones, con tamaños variantes entre 100 y 25 millones, el tiempo de ejecución esperado de la ordenación de Shell con esos incrementos es próximo a $O(n^{7/6})$. La rutina de la ordenación por montículo es la misma que en la sección 7.5. Se dan dos versiones de la ordenación rápida. La primera usa una estrategia de pivoteo sencilla y no hace corte. Por fortuna, los archivos de entrada fueron aleatorios. La segunda usa la partición con base en la mediana de tres y un corte a diez. Fueron posibles mayores optimizaciones. Podríamos haber codificado una rutina de la mediana de tres en línea, en vez

Figura 7.19 Comparación de diferentes algoritmos de ordenación
(todos los tiempos son en segundos)

n	Ordenación por inserción $O(n^2)$	Ordenación de Shell $O(n^{7/6})$	Ordenación por montículo $O(n \log n)$	Ordenación rápida (opt.) $O(n \log \log n)$	Ordenación rápida (opt.) $O(n \log n)$
10	0,00044	0,00041	0,00057	0,00052	0,00046
100	0,00675	0,00171	0,00420	0,00284	0,00244
1.000	0,59564	0,02927	0,05565	0,03153	0,02587
10.000	58,864	0,42998	0,71650	0,36765	0,31532
100.000	NA	5,7298	8,8591	4,2298	3,5882
1.000.000	NA	71,164	104,68	47,065	41,282

de usar una función, y podríamos haber escrito una versión no recursiva de la ordenación rápida. Hay algunas otras optimizaciones para codificar que son bastante difíciles de implantar, y por supuesto podríamos haber usado un lenguaje ensamblador. Hemos hecho un honesto intento por codificar todas las rutinas con eficiencia, pero desde luego el rendimiento puede variar algo de una máquina a otra.

La versión altamente optimizada de la ordenación rápida es tan rápida como la ordenación de Shell aun para tamaños de archivo muy pequeños. La versión mejorada de la ordenación rápida todavía tiene un peor caso de $O(n^2)$ (un ejercicio le pide que construya un ejemplo pequeño), pero las posibilidades de que aparezca el peor caso son tan insignificantes como para no tenerlo en cuenta. Si necesita ordenar archivos grandes, el método adecuado es la ordenación rápida. Pero nunca debe seguir el camino fácil de usar el primer elemento como pivote. En efecto, no va sin peligros suponer que la entrada será aleatoria. Si no quiere preocuparse por esto, use la ordenación de Shell, la cual dará una penalización menor en el rendimiento pero también podría ser aceptable, en especial si se requiere simplicidad. En el peor caso es sólo $O(n^{4/3})$; la oportunidad de que ocurra el peor caso es igualmente mínima.

La ordenación por montículo, aunque es un algoritmo $O(n \log n)$ con un ciclo interno aparentemente cerrado, es más lenta que la de Shell. Una profunda revisión cercana del algoritmo revela que, a fin de mover los datos, la ordenación por montículo hace dos comparaciones. Carlsson ha analizado una mejora sugerida por Floyd que mueve los datos con una sola comparación, pero implantar esta mejoría hace un poco más larga la codificación. Dejamos como ejercicio al lector decidir si vale la pena el esfuerzo de la codificación adicional para incrementar la velocidad (ejercicio 7.39).

La ordenación por inserción es útil sólo para archivos pequeños o archivos casi ordenados. No hemos incluido la ordenación por intercalación, porque su rendimiento no es tan bueno como la ordenación rápida para ordenación en memoria principal y no es más fácil de codificar. Hemos visto, no obstante, que la intercalación es la idea central de la ordenación externa.

Ejercicios

- 7.1 Ordene la secuencia 3, 1, 4, 1, 5, 9, 2, 6, 5 usando la ordenación por inserción.
- 7.2 ¿Cuál es el tiempo de ejecución de la ordenación por inserción si todas las llaves son iguales?
- 7.3 Suponga que intercambiamos los elementos $a[i]$ y $a[i + k]$, los cuales originalmente estaban desordenados. Compruebe que se eliminan al menos 1 y a lo más $2k - 1$ inversiones.
- 7.4 Muestre que el resultado de ejecutar la ordenación de Shell sobre la entrada 9, 8, 7, 6, 5, 4, 3, 2, 1 usando los incrementos {1, 3, 7}.
- 7.5 ¿Cuál es el tiempo de ejecución de la ordenación de Shell usando la secuencia de dos incrementos {1, 2}?

- 7.6 *a. Demuestre que el tiempo de ejecución de la ordenación de Shell es $\Omega(n^2)$ usando incrementos de la forma $1, c, c^2, \dots, c^t$ para cualquier entero c .
- **b. Demuestre que, para esos incrementos, el tiempo de ejecución promedio es $\Theta(n^{3/2})$.
- *7.7 Demuestre que si un archivo k -ordenado es después h -ordenado permanece k -ordenado.
- **7.8 Demuestre que el tiempo de ejecución de la ordenación de Shell, usando la secuencia de incrementos sugerida por Hibbard, es $\Omega(n^{3/2})$ en el peor caso. *Sugerencia:* Se puede demostrar la cota considerando el caso especial de lo que la ordenación de Shell hace cuando todos los elementos son 0 o 1. Ponga `datos_entrada[i] = 1 si i es expresable como una combinación lineal de $h_i, h_{i-1}, \dots, h_{\lfloor i/2 \rfloor}$ y 0 en otro caso.`
- 7.9 Determine el tiempo de ejecución de la ordenación de Shell para:
- la entrada ordenada,
 - la entrada ordenada en orden inverso.
- 7.10 Muestre cómo es que la ordenación por montículo procesa la entrada 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.
- 7.11 a. ¿Cuál es el tiempo de ejecución de la ordenación por montículo para la entrada previamente ordenada?
- **b. ¿Hay alguna entrada para la cual la ordenación por montículo se ejecute en $O(n \log n)$? En otras palabras, ¿existen entradas particularmente buenas para la ordenación por montículos?
- 7.12 Clasifique 3, 1, 4, 1, 5, 9, 2, 6 usando la ordenación por intercalación.
- 7.13 ¿Cómo implantaría la ordenación por intercalación sin recursión?
- 7.14 Determine el tiempo de ejecución de la ordenación por intercalación sobre:
- la entrada ordenada,
 - la entrada ordenada en orden inverso,
 - la entrada aleatoria.
- 7.15 En el análisis de la ordenación por intercalación, hemos omitido las constantes. Demuestre que el número de comparaciones usadas en el peor caso por la ordenación por intercalación es $n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$.
- 7.16 Clasifique 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 usando la ordenación rápida con la partición con base en la mediana de 3 y un corte de 3.
- 7.17 Mediante la implantación de la ordenación rápida de este capítulo, determine el tiempo de ejecución de la ordenación rápida para:
- la entrada ordenada,
 - la entrada ordenada en orden inverso,
 - la entrada aleatoria.
- 7.18 Repita el ejercicio 7.17 cuando el pivote se escoge como:
- el primer elemento,

- b. la mayor de las dos primeras llaves no distintas,
- c. un elemento aleatorio,
- d. el promedio de todas las llaves del conjunto.
- 7.19 a. Para la implantación de la ordenación rápida de este capítulo, ¿cuál es el tiempo de ejecución cuando todas las llaves son iguales?
- b. Suponga que cambiamos la estrategia de partición de modo que ni i ni j se detengan cuando se encuentra un elemento con la misma llave que el pivote. ¿Qué correcciones se necesitan en el código para garantizar que la ordenación rápida funcione, y cuál es el tiempo de ejecución, si todas las llaves son iguales?
- c. Suponga que cambiamos la estrategia de partición de modo que i se detenga en un elemento con la misma llave que el pivote, pero que j no se detenga en un caso semejante. ¿Qué ajustes se requieren para garantizar que la ordenación rápida funcione, y cuando todas las llaves son iguales, cuál es el tiempo de ejecución de la ordenación rápida?
- 7.20 Suponga que escogemos el elemento de la mitad como pivote. ¿Esto hace improbable que la ordenación rápida requiera un tiempo cuadrático?
- 7.21 Construya una permutación de 20 elementos tan mala como sea posible para la ordenación rápida usando la partición con base en la mediana de tres y un corte de 3.
- 7.22 Escriba un programa para implantar el algoritmo de selección.
- 7.23 Resuelva la siguiente ecuación de recurrencia: $T(n) = \frac{1}{n} [\sum_{i=0}^{n-1} T(i)] + cn, T(0) = 0$.
- 7.24 Un algoritmo de ordenación es *estable* si los elementos con llaves iguales se dejan en el mismo orden en que aparecieron en la entrada. ¿Cuáles de los algoritmos de ordenación de este capítulo son estables y cuáles no? ¿Por qué?
- 7.25 Suponga una lista ordenada de n elementos seguida de $f(n)$ elementos ordenados aleatoriamente. ¿Cómo ordenaría la lista completa dado lo siguiente?
- $f(n) = O(1)$
 - $f(n) = O(\log n)$
 - $f(n) = O(\sqrt{n})$
 - ¿Qué tan grande puede ser $f(n)$ para que la lista completa permanezca ordenable en un tiempo $O(n)$?
- 7.26 Demuestre que cualquier algoritmo que encuentre un elemento x en una lista ordenada de n elementos requiere $\Omega(\log n)$ comparaciones.
- 7.27 Usando la fórmula de Stirling, $n! \approx (n/e)^n \sqrt{2\pi n}$ proporcione un cálculo preciso para $\log n!$
- 7.28 *a. ¿En cuántas formas se pueden intercalar dos arreglos ordenados de n elementos?
- b. Proporcione una cota inferior no trivial sobre el número de comparaciones requeridas para intercalar dos listas ordenadas de n elementos.

- 7.29 Demuestre que la ordenación de n elementos con llaves enteras en el intervalo $1 \leq \text{llave} \leq m$ tarda un tiempo $O(m+n)$ usando la ordenación por cubetas.
- 7.30 Suponga que tiene un arreglo de n elementos que contiene sólo dos llaves distintas, *cierto* y *falso*. Dé un algoritmo $O(n)$ para reordenar la lista, de modo que todos los elementos *falso* precedan a los elementos *cierto*. Puede usar sólo un espacio adicional constante.
- 7.31 Suponga que tiene un arreglo de n elementos, con 3 llaves distintas, *cierto*, *falso* y *quizás*. Proporcione un algoritmo $O(n)$ para reordenar la lista de modo que todos los elementos *falso* precedan a los elementos *quizás*, y que éstos precedan a los elementos *cierto*. Sólo puede usar un tiempo adicional constante.
- 7.32 a. Demuestre que cualquier algoritmo basado en comparaciones para ordenar 4 elementos requiere 5 comparaciones.
 b. Proporcione un algoritmo para ordenar 4 elementos en 5 comparaciones.
- 7.33 a. Demuestre que se requieren 7 comparaciones para ordenar 5 elementos usando cualquier algoritmo basado en comparaciones.
 b. Proporcione un algoritmo para ordenar 5 elementos con 7 comparaciones.
- 7.34 Escriba una versión eficiente de la ordenación de Shell y compare el rendimiento cuando se usan las siguientes secuencias de incrementos:
- secuencia original de la ordenación de Shell
 - incrementos de Hibbard
 - incrementos de Knuth: $h_i = \frac{1}{2}(3^i + 1)$
 - incrementos de Gonnet: $h_i = \lfloor \frac{n}{2^i} \rfloor$ y $h_k = \lfloor \frac{h_{k+1}}{2^2} \rfloor$ (con $h_1 = 1$ y si $h_2 = 2$)
 - incrementos de Sedgewick
- 7.35 Implante una versión optimizada de la ordenación rápida y experimente con combinaciones de lo siguiente:
- Pivote: primer elemento, elemento medio, elemento aleatorio, mediana de tres y mediana de cinco.
 - Valores de corte de 0 a 20.
- 7.36 Escriba una rutina que lea dos archivos ordenados alfabéticamente y los intercale, formando un tercer archivo ordenado alfabéticamente.
- 7.37 Suponga que implantamos la rutina de la mediana de tres como sigue: encontrar la mediana de $a[\text{izquierda}], a[\text{centro}]$ y $a[\text{derecha}]$, e intercambiarla con $a[\text{derecha}]$. Proceda con el paso de partición normal, empezando i en izquierda y j en $\text{derecha} - 1$ (en vez de $\text{izquierda} + 1$ y $\text{derecha} - 2$). Suponga que $a[0] = \text{DATO_MÍN}$, de modo que haya sentinelas.
- Suponga que la entrada es $2, 3, 4, \dots, n - 1, n, 1$. ¿Cuál es el tiempo de ejecución de esta versión de la ordenación rápida?
 - Suponga que la entrada está en orden inverso. ¿Cuál es el tiempo de ejecución de esta versión de la ordenación rápida?
- 7.38 Compruebe que cualquier algoritmo de ordenación basado en comparaciones requiere $\Omega(n \log n)$ comparaciones en promedio.

7.39 Considere la siguiente estrategia de *filtrado_descen*. Tenemos un hueco en el nodo X . La rutina normal es comparar los hijos de X y después mover el hijo a X si es mayor (en el caso de un montículo (*máx*)) que el elemento que tratamos de colocar, con lo cual desciende el hueco; paramos cuando sea seguro colocar el elemento nuevo en el hueco. La estrategia alterna es mover los elementos hacia arriba y los huecos hacia abajo tan lejos como sea posible, sin comprobar si se puede insertar la celda nueva. Esto colocaría la celda nueva en una hoja y probablemente violaría el orden del montículo; para corregir el orden, se llena hacia arriba la celda nueva en la forma normal. Escriba una rutina para incluir esta idea, y compare el tiempo de ejecución con una implantación estándar de la ordenación por montículo.

7.40 Proponga un algoritmo para ordenar un archivo grande usando sólo dos cintas.

Referencias

El libro de Knuth [12] es una amplia, aunque algo antigua, referencia sobre la ordenación. Gonnet y Baeza-Yates [6] tienen resultados más recientes, así como una bibliografía enorme.

El artículo original que detalla la clasificación de Shell es [21]. El artículo de Hibbard [7] sugirió el uso de los incrementos $2^k - 1$ y ajustó la codificación evitando intercambios. El teorema 7.4 proviene de [14]. La cota inferior de Pratt, la cual usa un método más complejo que el sugerido en el texto, se puede encontrar en [15]. En [11], [20] y [23] aparecen secuencias de incrementos mejoradas y cotas superiores; en [24] se muestra la correspondencia de las cotas inferiores. Un reciente resultado inédito de Poonen muestra que ninguna secuencia de incrementos da un tiempo de ejecución para el peor caso de $O(n \log n)$. El tiempo de ejecución para el caso medio de la ordenación de Shell permanece sin solución. Yao [26] ha realizado un análisis extremadamente complejo del caso de tres incrementos. El resultado ha de ser extendido todavía a más incrementos. En [22] aparecen los experimentos con varias secuencias de incrementos.

La ordenación por montículo fue inventada por Williams [25]; Floyd [3] proporcionó el algoritmo en tiempo lineal para la construcción del montículo. El análisis exacto del algoritmo permanece incompleto. Algunos resultados se dan en [1] y [2].

Un análisis exacto del caso medio de la ordenación por intercalación se ha mostrado en [5]; el artículo que detalla los resultados está aún por salir. Un algoritmo que ejecuta la intercalación en tiempo lineal sin espacio adicional está descrito en [10].

La ordenación rápida es de Hoare [8]. Este artículo analiza el algoritmo básico, describe la mayoría de las mejoras e incluye el algoritmo de selección. Un análisis detallado y un estudio empírico fue el tema de la tesis doctoral de Sedgewick [19]. Muchos de los resultados importantes aparecen en los tres artículos [16], [17] y [18].

Ford y Johnson estudian los árboles de decisión y la optimidad de la ordenación en [4]. Este artículo también ofrece un algoritmo que casi satisface la cota

inferior en términos del número de comparaciones (pero no de otra operación). Manacher [13] demostró finalmente que este algoritmo es un poco subóptimo.

La ordenación externa se trata en detalle en [12]. Horvath [9] estudió la ordenación estable, descrita en el ejercicio 7.24.

1. S. Carlsson, "Average-Case Results on Heapsort", *BIT*, 27 (1987), págs. 2-17.
2. E. E. Doberkat, "An Average Case Analysis of Floyd's Algorithm to Construct Heaps", *Information and Control*, 61 (1984), págs. 114-131.
3. R. W. Floyd, "Algorithm 245: Treesort 3", *Communications of the ACM*, 7 (1964), págs. 701.
4. L. R. Ford y S. M. Johnson, "A Tournament Problem", *American Mathematics Monthly*, 66 (1959), págs. 387-389.
5. M. Golin y R. Sedgewick, "Exact Analysis of Mergesort", *Fourth SIAM Conference on Discrete Mathematics*, 1988.
6. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2a. ed., Addison-Wesley, Reading MA, 1991.
7. T. H. Hibbard, "An Empirical Study of Minimal Storage Sorting", *Communications of the ACM*, 6 (1963), págs. 206-213.
8. C. A. R. Hoare, "Quicksort", *Computer Journal*, 5 (1962), págs. 10-15.
9. E. C. Horvath, "Stable Sorting in Asymptotically Optimal Time and Extra Space", *Journal of the ACM*, 25 (1978), págs. 177-199.
10. B. Huang y M. Langston, "Practical In-place Merging", *Communications of the ACM*, 31 (1988), págs. 348-352.
11. J. Incerpi y R. Sedgewick, "Improved Upper Bounds on Shellsort", *Journal of Computer and System Science*, 31 (1985), págs. 210-224.
12. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2a. imp., Addison-Wesley, Reading, MA, 1975.
13. G. K. Manacher, "The Ford-Johnson Sorting Algorithm Is Not Optimal", *Journal of the ACM*, 26 (1979), págs. 441-456.
14. A. A. Papernov y G. V. Stasevich, "A Method of Information Sorting in Computer Memories", *Problems of Information Transmission*, 1 (1965), págs. 63-75.
15. V. R. Pratt, *Shellsort and Sorting Networks*, Garland Publishing, Nueva York, 1979. (Originalmente presentado como tesis de doctorado del autor, Stanford University, 1971.)
16. R. Sedgewick, "Quicksort with Equal Keys", *SIAM Journal on Computing*, 6 (1977), págs. 240-267.
17. R. Sedgewick, "The Analysis of Quicksort Programs", *Acta Informatica*, 7 (1977), págs. 327-355.
18. R. Sedgewick, "Implementing Quicksort Programs", *Communications of the ACM*, 21 (1978), págs. 847-857.
19. R. Sedgewick, *Quicksort*, Garland Publishing, Nueva York, 1978. (Originalmente presentado como tesis de doctorado del autor, Stanford University, 1975.)
20. R. Sedgewick, "A New Upper Bound for Shellsort", *Journal of Algorithms*, 2 (1986), págs. 159-173.
21. D. L. Shell, "A High-Speed Sorting Procedure", *Communications of the ACM*, 2 (1959), págs. 30-32.
22. M. A. Weiss, "Empirical Results of the running Time of Shellsort", *Computer Journal*, 34 (1991), págs. 88-91.
23. M. A. Weiss y R. Sedgewick, "More On Shellsort Increment Sequences", *Information Processing Letters*, 34 (1990), págs. 267-270.

24. M. A. Weiss y R. Sedgewick, "Tight Lower Bounds For Shellsort", *Journal of Algorithms*, 11 (1990), págs. 242-251.
25. J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM*, 7 (1964), págs. 347-348.
26. A. C. Yao, "An Analysis of (h,k,l) Shellsort", *Journal of Algorithms*, 1 (1980), págs. 14-50.

El TDA conjunto ajeno

En este capítulo describiremos una estructura de datos eficiente para resolver el problema de la equivalencia. Es simple implantar esta estructura de datos. Cada rutina requiere sólo unas cuantas líneas de código, y se puede usar un simple arreglo. También la implantación es extremadamente rápida, pues requiere un tiempo medio constante por operación. Esta estructura de datos también es muy interesante desde un punto de vista teórico, porque su análisis es sumamente difícil; la forma funcional del peor caso es diferente de cualquiera que hayamos visto antes. Para el TDA conjunto ajeno (o disjunto),

- mostraremos cómo implantarlo con un esfuerzo de codificación mínimo;
- incrementaremos bastante su velocidad, usando sólo dos observaciones sencillas;
- analizaremos el tiempo de ejecución de una implantación rápida;
- veremos una aplicación sencilla.

8.1. Relaciones de equivalencia

Una *relación R* se define en un conjunto *C* si para todo par de elementos (a, b) , $a, b \in C$, $a \ R \ b$ es verdadera o falsa. Si $a \ R \ b$ es verdadera, entonces se dice que *a* está relacionada con *b*.

Una *relación de equivalencia* es una relación *R* que satisface tres propiedades:

1. (*Reflexiva*) $a \ R \ a$, para todo $a \in C$.
2. (*Simétrica*) $a \ R \ b$ si y sólo si $b \ R \ a$.
3. (*Transitiva*) $a \ R \ b$ y $b \ R \ c$ implica que $a \ R \ c$.

Consideremos varios ejemplos.

La relación \leq no es una relación de equivalencia. Aunque es reflexiva, porque $a \leq a$, y transitiva, porque $a \leq b$ y $b \leq c$ implica que $a \leq c$, no es simétrica, ya que $a \leq b$ no implica que $b \leq a$.

La *conectividad eléctrica*, donde todas las conexiones son a través de alambres metálicos, es una relación de equivalencia. Claro está, la relación es reflexiva, puesto que cualquier componente está conectado a sí mismo. Si a está conectado eléctricamente a b , entonces b debe estar conectado a a , así que la relación es simétrica. Por último, si a está conectado a b y b está conectado a c , entonces a está conectado a c . Así la conectividad eléctrica es una relación de equivalencia.

Dos ciudades están relacionadas si están en el mismo país. Se verifica fácilmente que ésta es una relación de equivalencia. Supongamos que el poblado a está relacionado con b si es posible viajar de a a b siguiendo caminos. Esta relación es de equivalencia si todos los caminos son de doble circulación.

8.2. El problema de la equivalencia dinámica

Dada una relación de equivalencia \sim , el problema natural es decidir si $a \sim b$, para cualesquiera a y b . Si la relación se almacena como un arreglo bidimensional de booleanos, entonces, por supuesto, esto se puede hacer en un tiempo constante. El problema es que la relación no suele definirse explícitamente, sino implícitamente.

Por ejemplo, suponga que la relación de equivalencia se define sobre el conjunto de cinco elementos $\{a_1, a_2, a_3, a_4, a_5\}$. Entonces hay 25 pares de elementos, cada uno de los cuales está o no relacionado. No obstante, la información $a_1 \sim a_2, a_3 \sim a_4, a_5 \sim a_1, a_4 \sim a_2$ implica que todos los pares están relacionados. Nos gustaría poder inferir esto rápidamente.

La *clase de equivalencia* de un elemento $a \in C$ es el subconjunto de C que contiene todos los elementos relacionados con a . Observe que las clases de equivalencia forman una partición de C : todo miembro de C aparece en exactamente una clase de equivalencia. Para decidir si $a \sim b$, sólo necesitamos verificar si a y b están en la misma clase de equivalencia. Esto proporciona la estrategia para resolver el problema de la equivalencia.

Al principio es una colección de n conjuntos, cada uno con un elemento. Esta representación inicial es que todas las relaciones son falsas (excepto las relaciones reflexivas). Cada conjunto tiene un elemento diferente, así que $C_i \cap C_j = \emptyset$; esto hace *ajenos* los conjuntos.

Hay dos operaciones válidas. La primera es *búsqueda*, que devuelve el nombre del conjunto (es decir, la clase de equivalencia) con un elemento dado. La segunda operación añade relaciones. Si queremos añadir la relación $a \sim b$, primero vemos si a y b ya están relacionados. Esto se hace efectuando operaciones de *búsqueda* en a y en b y revisando si están en la misma clase de equivalencia. Si no lo están, se aplica la *unión*. Esta operación combina dos clases de equivalencia que contienen a y b en una clase de equivalencia nueva. Desde un punto de vista de conjuntos, el resultado de la \cup es crear un nuevo conjunto $C_k = C_i \cup C_j$, destruyéndose los originales y preservándose la calidad de ser *ajenos* de todos los conjuntos. Con frecuencia, por esa razón al algoritmo para hacer esto se le llama *algoritmo unión/búsqueda* para conjuntos ajenos.

Este algoritmo es *dinámico* porque, durante el curso del algoritmo, los conjuntos pueden cambiar vía la operación *unión*. El algoritmo debe operar también *en línea*: cuando se efectúa una *búsqueda*, debe dar una respuesta antes de continuar. Otra posibilidad sería ser un algoritmo *fuera de línea*. Tal algoritmo tendría permitido ver la secuencia completa de *uniones* y *búsquedas*. La respuesta que otorga para cada *búsqueda* debe permanecer consistente con todas las *uniones* que fueron realizadas antes de la *búsqueda*, pero el algoritmo puede dar todas sus respuestas después de ver *todas* las preguntas. La diferencia es semejante a hacer un examen escrito (que en general es fuera de línea sólo hay que dar las respuestas antes de que el tiempo expire), y un examen oral (que es en línea porque se debe dar la respuesta a la pregunta actual antes de proceder con la siguiente).

Observe que no se realizan operaciones comparando los valores relativos de los elementos, sino que sólo requiere conocer su localización. Por esta razón, podemos suponer que todos los elementos se han numerado en secuencia de 1 a n y que la numeración puede determinarse fácilmente por medio de algún esquema de dispersión. Así, al principio se tiene $C_i = \{i\}$ para $i = 1$ hasta n .

Nuestra segunda observación es que de hecho, el nombre del conjunto devuelto por *búsqueda* es bastante arbitrario. Todo lo que realmente importa es que *búsqueda*(x) = *búsqueda*(y) si y sólo si x y y están en el mismo conjunto.

Esas operaciones son importantes en muchos problemas de la teoría de grafos y también en compiladores que procesan declaraciones (o tipos) de equivalencia. Más adelante veremos una aplicación.

Hay dos estrategias para resolver este problema. Una asegura que la instrucción *búsqueda* se puede ejecutar en un tiempo constante para el peor caso, y la otra asegura que la instrucción *unión* se puede ejecutar en un tiempo constante para el peor caso. Hay demostraciones recientes de que ambas no se pueden ejecutar simultáneamente en un tiempo constante para el peor caso.

Ahora estudiaremos brevemente el primer enfoque. Para que la operación *búsqueda* sea rápida, podemos mantener, en un arreglo, el nombre de la clase de equivalencia de cada elemento. Entonces *búsqueda* es una simple consulta $O(1)$. Supongamos que queremos ejecutar *unión*(a, b). Supongamos también que a está en la clase de equivalencia i y b está en la clase de equivalencia j . Entonces recorremos el arreglo de arriba a abajo, cambiando toda i a j . Desafortunadamente, este recorrido toma $\Theta(n)$. Así, una secuencia de $n - 1$ uniones (la máxima, ya que entonces todo está en un conjunto) podría tomar un tiempo $\Theta(n^2)$. Si hay $\Omega(n^2)$ operaciones *búsqueda*, este rendimiento es bueno, porque el tiempo de ejecución total ascendería a $O(1)$ por cada operación *unión* o *búsqueda* en el curso del algoritmo. Si hay menos *búsquedas*, esta cota no es aceptable.

Una idea es conservar todos los elementos que están en la misma clase de equivalencia en una lista enlazada. Esto ahorrará tiempo durante la actualización porque no tenemos que buscar en todo el arreglo. Por sí mismo, esto no reduce el tiempo de ejecución asintótico, porque aún es posible efectuar $\Theta(n^2)$ actualizaciones de la clase de equivalencia en el curso del algoritmo.

Si también conservamos el tamaño de cada clase de equivalencia, y al ejecutar *uniones* cambiamos el nombre de la clase de equivalencia menor por el de mayor, entonces el tiempo total consumido para $n - 1$ combinaciones es $O(n \log n)$. La razón

de esto es que cada elemento puede cambiar su clase de equivalencia cuando mucho $\log n$ veces, pues cada vez que cambia su clase, su nueva clase de equivalencia es al menos el doble de grande que la anterior. Usando esta estrategia, cualquier secuencia de m operaciones *búsqueda* y hasta $n - 1$ *uniones* tarda un tiempo $O(m + n \log n)$, cuando mucho.

En lo que resta del capítulo, examinaremos una solución al problema *unión/búsqueda* que hace fácil la *unión*, pero difícil la *búsqueda*. Aun así, el tiempo de ejecución para cualesquier secuencias de m operaciones *búsqueda* como máximo y hasta $n - 1$ *uniones* será un poco más que $O(m + n)$.

8.3. Estructura de datos básica

Recordemos que el problema no requiere que la operación *búsqueda* devuelva algún nombre específico, sólo que las operaciones *búsqueda* sobre dos elementos devuelvan el mismo nombre si y sólo si están en el mismo conjunto. Una idea podría consistir en utilizar un árbol para representar cada conjunto, pues cada elemento en un árbol tiene la misma raíz. Así, la raíz puede usarse para nombrar el conjunto. Representaremos cada conjunto con un árbol. (Recuerde que una colección de árboles se denomina *bosque*.) Al principio, cada conjunto contiene un elemento. Los árboles que se usarán no necesariamente son binarios, pero su representación es fácil porque la única información que necesitaremos es un apuntador al padre. El nombre de un conjunto está dado por el nodo en la raíz. Como sólo se requiere el nombre del padre, podemos suponer que este árbol está almacenado implícitamente en un arreglo: cada entrada $p[i]$ en el arreglo representa el padre del elemento i . Si i es una raíz, entonces $p[i] = 0$. En el bosque de la figura 8.1, $p[i] = 0$ para $1 \leq i \leq 8$. Como en los montículos, dibujaremos el árbol explícitamente, a sabiendas de que se está usando un arreglo. La figura 8.1 muestra la representación explícita. Por conveniencia se dibujará verticalmente el apuntador al padre de la raíz.

Para ejecutar una *unión* de dos conjuntos se combinan los dos árboles haciendo que la raíz de un árbol apunte a la raíz del otro. Debe estar claro que esta operación toma un tiempo constante. Las figuras 8.2, 8.3 y 8.4 representan el bosque después de las operaciones *unión*(5, 6), *unión*(7, 8), *unión*(5, 7), donde se ha adoptado el convenio de que la raíz nueva después de *unión*(x, y) es x . La representación implícita del último bosque se muestra en la figura 8.5.

Una *búsqueda*(x) sobre el elemento x se efectúa devolviendo la raíz del árbol que contiene x . El tiempo para ejecutar esta operación es proporcional a la profundidad del nodo que representa a x , suponiendo, desde luego, que podemos encontrar el nodo que representa a x en un tiempo constante. Usando esta estrategia, es posible crear un árbol de profundidad $n - 1$, para que el tiempo de ejecución en el peor caso de una *búsqueda* sea $O(n)$. Por lo regular, el tiempo de ejecución se calcula para una secuencia de m instrucciones intercaladas. En este caso, m operaciones consecutivas tomarían un tiempo $O(mn)$ en el peor caso.

El código de las figuras 8.6 a la 8.9 representa una implantación del algoritmo básico, suponiendo que ya se han efectuado las comprobaciones de errores. En esta rutina, las *uniones* se realizan en las raíces de los árboles. Algunas veces la operación

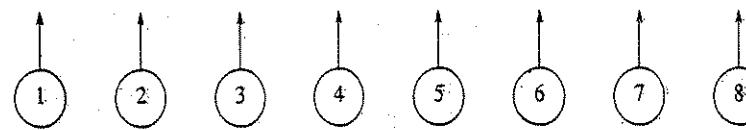


Figura 8.1 Ocho elementos, inicialmente en conjuntos diferentes

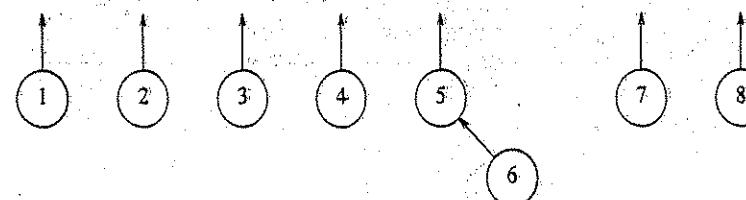


Figura 8.2 Despues de *unión*(5, 6)

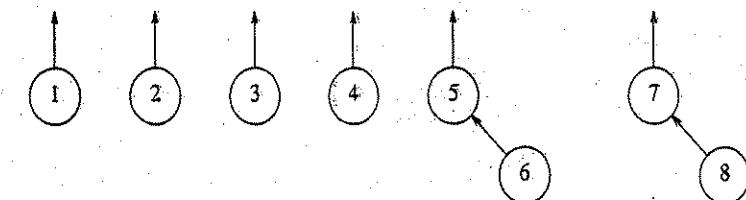


Figura 8.3 Despues de *unión*(7, 8)

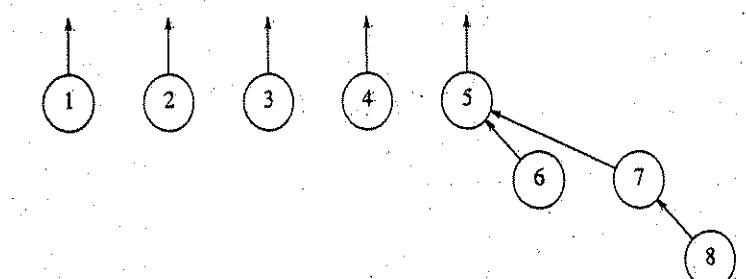


Figura 8.4 Despues de *unión*(5, 7)

0	0	0	0	0	5	5	7
1	2	3	4	5	6	7	8

Figura 8.5 Representación implícita del árbol anterior

se ejecuta pasando cualesquiera dos elementos, y haciendo que la *unión* efectúe dos búsquedas para determinar las raíces.

El análisis del caso medio es bastante difícil de hacer. El menor de los problemas es que la respuesta depende de cómo definir *medio* (con respecto a la operación *unión*). Por ejemplo, en el bosque de la figura 8.4 podríamos decir que como hay cinco árboles, hay $5 \cdot 4 = 20$ resultados igualmente probables de la siguiente *unión* (ya que cualesquiera dos árboles diferentes pueden ser *unidos*). Por supuesto, la implicación de este modelo es que hay sólo una oportunidad de $\frac{1}{5}$ de que la *unión* siguiente implique el árbol grande. Segundo modelo, todas las *uniones* entre dos

```
type
  CONJ_AJENO = array [1..NÚM_CONJS] of integer;
  tipo_conj = integer;
  tipo_elemento = integer;
```

Figura 8.6 Declaración de tipos del conjunto ajeno

```
procedure inicia(var C: CONJ_AJENO);
var i: tipo_conj;

begin
  for i := 1 to NÚM_CONJS do
    C[i] := 0;
end;
```

Figura 8.7 Rutina de inicio del conjunto ajeno

{supone que raíz 1 y raíz 2 son raíces}

```
procedure unión(var C: CONJ_AJENO; raíz1, raíz2: tipo_conj);
begin
  C[raíz2] := raíz1;
end;
```

Figura 8.8 Unión (no en la mejor forma)

```
function búsqueda(x: tipo_elemento; var C: CONJ_AJENO): tipo_conj;
begin
  if C[x] < 0 then
    búsqueda := x
  else
    búsqueda := búsqueda(C[x], C);
end;
```

Figura 8.9 Un algoritmo *búsqueda* sencillo para conjuntos ajenos

elementos cualesquiera en árboles diferentes son igualmente probables, así que es más probable que un árbol mayor intervenga en la siguiente *unión* que un árbol menor. En el ejemplo anterior, hay $\frac{1}{5}$ oportunidades de que el árbol grande intervenga en la siguiente unión, ya que (ignorando simetrías) hay 6 formas de combinar dos elementos de $\{1, 2, 3, 4\}$, y 16 formas de combinar un elemento de $\{5, 6, 7, 8\}$ con un elemento de $\{1, 2, 3, 4\}$. Hay aún más modelos y ningún acuerdo general de cuáles es el mejor. El tiempo de ejecución promedio depende del modelo; las cotas $\Theta(n)$, $\Theta(m \log n)$ y $\Theta(mn)$ realmente han sido demostradas para tres modelos diferentes de árbol, aunque la última cota parece más realista.

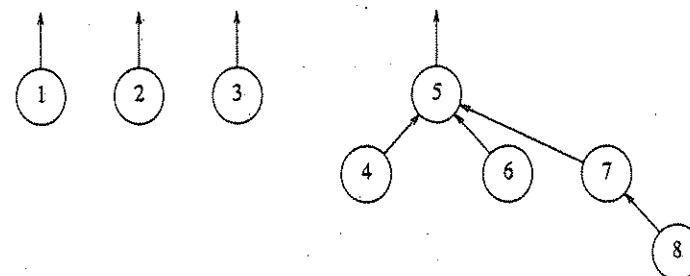
En general el tiempo de ejecución cuadrático para una secuencia de operaciones es inaceptable. Por fortuna hay varias formas fáciles de asegurar que este tiempo de ejecución no ocurra.

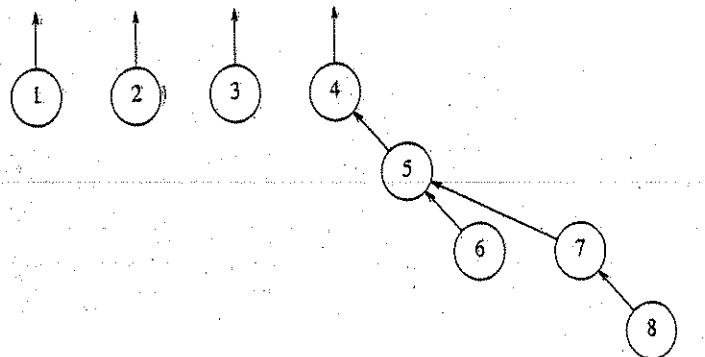
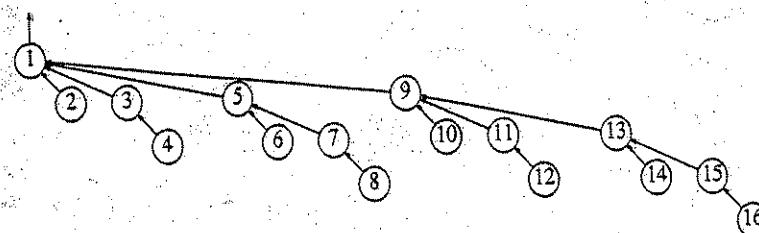
8.4. Algoritmos de unión refinados

Las *uniones* anteriores se efectuaban de un modo más bien arbitrario, haciendo del segundo árbol un subárbol del primero. Una mejora sencilla es hacer siempre que el subárbol menor sea un subárbol del mayor, deshaciendo los empates por cualquier método; a este enfoque se le llama *unión por tamaño*. Las tres uniones en el ejemplo anterior fueron empates todos, y así podemos considerar que se realizaron por tamaño. Si la siguiente operación fuera *unión(4, 5)*, se formaría el bosque de la figura 8.10. Si no se hubiera usado la heurística del tamaño, se habría formado un bosque más profundo (figura 8.11).

Podemos demostrar que si las *uniones* se hacen por tamaño, la profundidad de cualquier nodo nunca es mayor que $\log n$. Para ver esto, observe que un nodo está inicialmente a la profundidad 0. Cuando su profundidad se incrementa como resultado de una *unión*, se coloca en un árbol que es al menos el doble de grande que antes. Así, su profundidad se puede incrementar a lo más, $\log n$ veces. (Utilizaremos este razonamiento en el algoritmo de búsqueda rápida al final de la sección 8.2.) Esto implica que el tiempo de ejecución de una operación *búsqueda* es $O(\log n)$, y una secuencia de m operaciones tarda $O(m \log n)$. El árbol de la figura 8.12 muestra el peor árbol posible después de 16 *uniones* y se obtiene si todas las *uniones* son entre árboles de igual tamaño (los árboles del peor caso son árboles binomiales, estudiados en el capítulo 6).

Figura 8.10 Resultado de la unión por tamaño



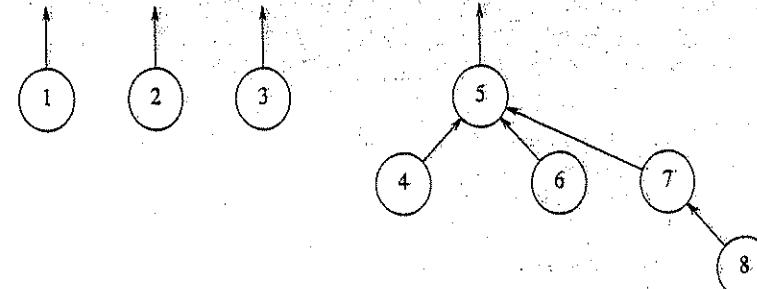
Figura 8.11 Resultado de una *unión* arbitrariaFigura 8.12 Árbol del peor caso para $n = 16$

Para implantar esta estrategia, necesitamos conservar el tamaño de cada árbol. Como de hecho sólo estamos usando un arreglo, es posible que las entradas del arreglo correspondientes a raíces contengan el *negativo* del tamaño de su árbol. Así, inicialmente la representación vectorial del árbol es todo -1 s. Cuando se efectúa una *unión*, se revisan los tamaños; el nuevo tamaño es la suma de los anteriores. Así, no hay ninguna dificultad en implantar la unión por tamaño y no requiere espacio adicional. También es rápida, en promedio. Para prácticamente todos los modelos razonables, se ha demostrado que una secuencia de m operaciones requiere un tiempo promedio de $O(m)$ si se usa la unión por tamaño. Esto se debe a que al ejecutar *uniones* aleatorias, por lo regular se combinan conjuntos muy pequeños (casi siempre de un elemento) con conjuntos grandes, a lo largo de todo el algoritmo.

Una implantación alternativa, que también garantiza que todos los árboles tendrán una profundidad máxima de $O(\log n)$, es la *unión por altura*. Se conserva la altura, en vez del tamaño, de cada árbol y se realizan *uniones* haciendo del árbol menos profundo un subárbol del árbol más profundo. Éste es un algoritmo fácil, ya que la altura de un árbol se incrementa sólo cuando se unen dos árboles de igual

profundidad (entonces la altura se incrementa en uno). Así, la unión por altura es una modificación trivial de la unión por tamaño.

Las siguientes figuras muestran un árbol y su representación implícita para la unión por tamaño y por altura. La codificación de la figura 8.13 implementa la unión por altura.



0	0	0	5	-2	5	5	7
1	2	3	4	5	6	7	8

Figura 8.13 Código de la unión por altura (rango)

```

[supone que raíz 1 y raíz 2 son raíces]
procedure unión(var C: CONJ_AJENO; raíz1, raíz2: tipo_conj);
begin
  if C[raíz2] < C[raíz1] then      [raíz2 es más profundo]
    C[raíz1] := raíz2
  else
    begin
      if C[raíz2] = C[raíz1] then   [misma altura, actualiza]
        C[raíz1] := C[raíz1] - 1;
      C[raíz2] := raíz1;          [hace de raíz la nueva raíz1]
    end;
end;
  
```

8.5. Compresión de caminos

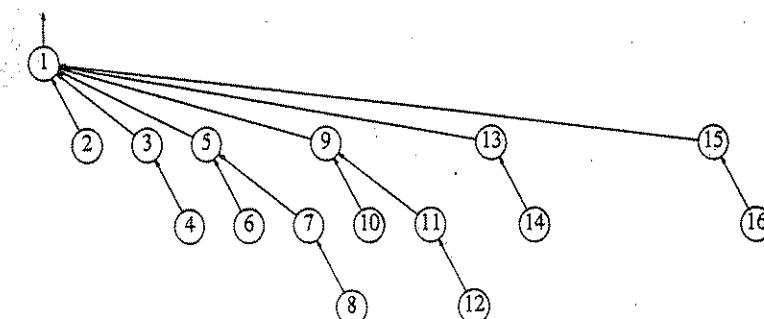
El algoritmo *unión/búsqueda*, como lo hemos descrito hasta aquí, es bastante aceptable para la mayoría de los casos. Es muy sencillo y lineal en promedio para una secuencia de m instrucciones (para todos los modelos). No obstante, es bastante fácil, y natural, que se presente el peor caso de $O(m \log n)$. Por ejemplo, si ponemos todos los conjuntos en una cola y se desencolan repetidamente los dos primeros conjuntos y se encola la unión, ocurre el peor caso. Si hay muchos más *búsqueda* que *uniones*, este tiempo de ejecución es peor que el del algoritmo *búsqueda_rápida*. Además, debe quedar claro que probablemente no haya más mejoras para el algoritmo *unión*. Esto se basa en la observación de que cualquier método que efectúe *uniones* producirá los mismos árboles del peor caso, pues debe deshacer empates arbitrariamente. Por lo tanto, la única forma de acelerar el algoritmo, sin rehacer por completo la estructura de datos, es hacer algo más ingenioso sobre la operación *búsqueda*.

La operación ingeniosa se conoce como *compresión de caminos*. Se ejecuta durante una operación *búsqueda* y es independiente de la estrategia con que se efectúen las *uniones*. Suponga que la operación es *búsqueda(x)*. Entonces el efecto de la compresión de caminos es que todo nodo en el camino de x a la raíz cambia su padre por la raíz. La figura 8.14 muestra el efecto de la compresión de caminos después de *búsqueda(15)* en el peor árbol genérico de la figura 8.12.

El efecto de la compresión de caminos es que con un movimiento adicional de dos apuntadores, los nodos 13 y 14 están ahora una posición más cerca de la raíz y los nodos 15 y 16 están dos posiciones más cerca. Así, los rápidos accesos futuros sobre esos nodos compensarán (esperamos) el trabajo adicional de hacer la compresión de los caminos.

Como lo muestra la codificación de la figura 8.15, la compresión de caminos es un cambio trivial del algoritmo *búsqueda* básico. El único cambio a la rutina *búsqueda* (aparte de un par adicional *begin/end*) es que $C[x]$ se hace igual al valor devuelto por *búsqueda*; así, después de que la raíz del conjunto se encuentre recursivamente, se hace que x apunte directamente a ésta. Esto sucede recursivamente con todo nodo en el camino a la raíz, así que se planta la compresión de caminos. Como afirmamos al implantar pilas y colas, modificar un parámetro de los pasados a una función llamada no necesariamente está en línea con las reglas actuales de la

Figura 8.14 Ejemplo de compresión de caminos



8.6. PEOR CASO DE LA UNIÓN POR RANGOS Y COMPRESIÓN DE CAMINOS

```

function búsqueda(x: tipo_elemento; var C: CONJ_AJENO): tipo_conj;
begin
[1]  if C[x] <= 0 then
[2]    búsqueda := x
      else
        begin
[3]          C[x] := búsqueda(C[x], C);
[4]          búsqueda := C[x];
        end;
end;

```

Figura 8.15 Codificación para *búsqueda* en conjuntos ajenos con compresión de caminos

ingeniería de software. Algunos lenguajes no permitirán esto, así que este código bien puede necesitar cambios.

Cuando efectuamos *uniones* en forma arbitraria, la compresión de caminos es una buena idea, porque hay abundantes nodos profundos y éstos se acercan a la raíz gracias a la compresión. Se ha demostrado que cuando se hace la compresión de caminos en este caso, una secuencia de m operaciones requiere un tiempo $O(m \log n)$ como máximo. Es todavía un problema abierto determinar cuál es el comportamiento medio en esta situación.

La compresión de caminos es perfectamente compatible con la unión por tamaño, y así ambas rutinas se pueden implantar en el mismo momento. Como se espera que al hacer la unión por tamaño en sí misma se ejecute una secuencia de m operaciones en tiempo lineal, no está claro que valga la pena, en general, la pasada extra contenida en la compresión de caminos. En efecto, este problema sigue abierto. Sin embargo, como veremos después, la combinación de la compresión de caminos y una regla de unión buena garantiza un algoritmo muy eficiente en todos los casos.

La compresión de caminos no es del todo compatible con la unión por altura, porque la compresión puede cambiar las alturas de los árboles. No está totalmente claro cómo volver a calcularlos con eficiencia. La respuesta es no hacerlo. Entonces las alturas almacenadas para cada árbol se convierten en alturas estimadas (algunas veces llamadas *rangos*), pero ocurre que la unión por rango (que es en lo que ahora se ha convertido) es tan eficiente en teoría como la unión por tamaño. Además, las alturas se actualizan menos que los tamaños. Como con la unión por tamaño, no está claro si la compresión de caminos, en general, vale la pena. Lo que mostraremos en la siguiente sección es que, con una u otra heurística de unión, la compresión de caminos logra una reducción significativa del tiempo de ejecución del peor caso.

8.6. Peor caso de la unión por rangos y compresión de caminos

Cuando se usan ambas heurísticas, el algoritmo es casi lineal en el peor caso. Específicamente, el tiempo requerido en el peor caso es $\Theta(ma(m, n))$ (siempre que

$m \geq n$), donde $\alpha(m, n)$ es una inversa funcional de la función de Ackerman, la cual se define en seguida:^t

$$A(1, j) = 2^j \text{ para } j \geq 1;$$

$$A(i, 1) = A(i - 1, 2) \text{ para } i \geq 2;$$

$$A(i, j) = A(i - 1, A(i, j - 1)) \text{ para } i, j \geq 2.$$

A partir de esto, definimos

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$$

Quizá quiera calcular algunos valores, pero para todos los propósitos prácticos, $\alpha(m, n) \leq 4$, que es todo lo que realmente importa aquí. La función inversa de Ackerman de una variable, que a veces se escribe como $\log^* n$, es el número de veces que el logaritmo de n necesita ser aplicado hasta que $n \leq 1$. Así, $\log^* 65536 = 4$, porque $\log \log \log 65536 = 1$, $\log^* 2^{65536} = 5$, pero tenga en mente que 2^{65536} es un número de 20 000 dígitos. $\alpha(m, n)$ de hecho crece aún más lentamente que $\log^* n$. Sin embargo, $\alpha(m, n)$ no es una constante, así que el tiempo de ejecución no es lineal.

En el resto de esta sección, demostraremos un resultado ligeramente más débil: que cualquier secuencia de $m = \Omega(n)$ operaciones *unión/búsqueda* toma un tiempo de ejecución total $O(m \log^* n)$. La misma cota se mantiene si la unión por rangos se sustituye por la unión por tamaño. Es probable que este análisis sea el más complejo del libro y, uno de los primeros análisis del peor caso verdaderamente complejos, que se hayan realizado para un algoritmo que en esencia resulta trivial de implantar.

8.6.1. Análisis del algoritmo *unión/búsqueda*

En esta sección se establece una cota muy ajustada del tiempo de ejecución de una secuencia de $m = \Omega(n)$ operaciones *unión/búsqueda*. Las *uniones* y las *búsquedas* pueden ocurrir en cualquier orden, pero la *unión* se hace por rangos y las *búsquedas* se hacen con compresión de caminos.

Empecemos estableciendo algunos lemas concernientes al número de nodos de *rango r*. Intuitivamente, debido a la regla de la unión por rango, hay muchos más nodos de rango pequeño que de rango grande. En particular, puede haber a lo más un nodo de rango $\log^* n$. Lo que nos gustaría hacer es producir una cota tan precisa como sea posible del número de nodos de cualquier rango *r* particular. Puesto que los rangos cambian sólo cuando se ejecutan las *uniones* (y entonces sólo cuando los dos árboles tienen el mismo rango), podemos demostrar esta cota ignorando la compresión de caminos.

^t Con frecuencia la función de Ackerman se define con $A(1, j) = j + 1$ para $j \geq 1$. La forma en este texto crece más rápido; así, la inversa crece más lentamente.

LEMA 8.1.

Al ejecutar una secuencia de instrucciones *unión*, un nodo de rango *r* debe tener 2^r descendientes (incluyéndose a sí mismo).

DEMOSTRACIÓN:

Por inducción. La base, $r = 0$, es claramente cierta. Sea *A* el árbol de rango *r* con el menor número de descendientes y sea *x* la raíz de *A*. Suponga que la última *unión* en que intervino *x* fue entre *A*₁ y *A*₂. Suponga que la raíz de *A*₁ fue *x*. Si *A*₁ tenía rango *r*, entonces *A*₁ sería un árbol de altura *r* con menos descendientes que *A*, lo cual contradice la suposición de que *A* es el árbol con el menor número de descendientes. Por tanto, el rango de *A*₁ $\leq r - 1$. El rango de *A*₂ \leq rango de *A*₁. Puesto que *A* tiene rango *r* y el rango sólo podría incrementarse a causa de *A*₂, se infiere que el rango de *A*₂ = *r* - 1. Entonces el rango de *A*₁ = *r* - 1. Por la hipótesis de la inducción, cada árbol tiene al menos 2^{r-1} descendientes, dando un total de 2^r y así demostramos el lema.

El lema 8.1 dice que si no se efectúa ninguna compresión de caminos, entonces cualquier nodo de rango *r* debe tener al menos 2^r descendientes. Por supuesto, la compresión de caminos puede cambiar esto, pues puede eliminar descendientes de un nodo. Sin embargo, al ejecutar *uniones*, aún con compresión de caminos, usamos los rangos, los cuales son alturas aproximadas. Estos rangos se comportan como si no hubiera compresión de caminos. Así, al acotar el número de nodos de rango *r* se puede ignorar la compresión de caminos.

De este modo el siguiente lema es válido con o sin compresión de caminos.

LEMA 8.2.

El número de nodos de rango *r* es $n/2^r$ como máximo.

DEMOSTRACIÓN:

Si la compresión de caminos, cada nodo de rango *r* es la raíz de un subárbol de al menos 2^r nodos. Ningún nodo puede tener rango *r*. Así, todos los subárboles de los nodos de rango *r* son ajenos. En consecuencia, hay cuando mucho $n/2^r$ subárboles ajenos y, por tanto, $n/2^r$ nodos de rango *r*.

El siguiente lema parece obvio, pero es crucial en el análisis.

LEMA 8.3.

En cualquier punto del algoritmo *unión/búsqueda*, los rangos de los nodos de un camino de la hoja a la raíz se incrementan monótonamente.

DEMOSTRACIÓN:

El lema es obvio si no hay compresión de caminos (véase el ejemplo). Si, después de la compresión de caminos, algún nodo *v* es descendiente de *w*, entonces está claro que *v* debe haber sido un descendiente de *w* cuando sólo se consideraron *uniones*. Aquí el rango de *v* es menor que el de *w*.

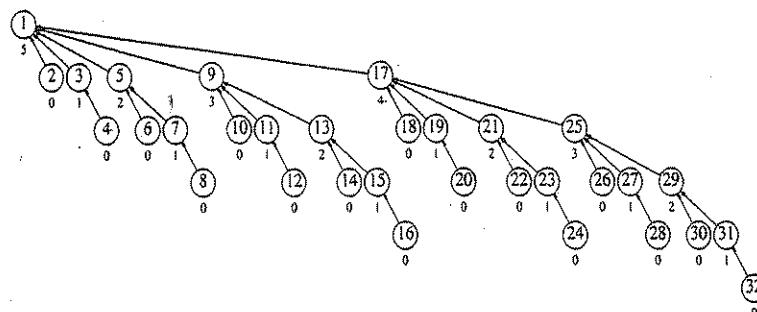


Figura 8.16: Un gran árbol de conjunto ajeno (los números bajo los nodos son rangos)

Resumamos los resultados preliminares. El lema 8.2 nos dice cuántos nodos se pueden asignar al rango r . Como los rangos son asignados sólo por las *uniones*, lo cual no tiene idea de compresión de caminos, el lema 8.2 es válido en cualquier etapa del algoritmo *unión/búsqueda*; aun a la mitad de la compresión de caminos. La figura 8.16 muestra que en tanto hay muchos nodos en los rangos 0 y 1, hay menos nodos de rangos r conforme r crece.

El lema 8.2 es ajustado, en el sentido de que es posible que haya $n/2^r$ nodos en cualquier rango r . Esto es ligeramente abierto, porque no es posible que la cota se mantenga para todos los rangos r al mismo tiempo. Mientras que el lema 8.2 describe el *número* de nodos en un rango r , el lema 8.3 establece su *distribución*. Como se podría esperar, el rango de los nodos es estrictamente creciente a lo largo del camino de una hoja a la raíz.

Ahora estamos listos para demostrar el teorema principal. La idea básica es como sigue: una *búsqueda* de cualquier nodo v cuesta un tiempo proporcional al número de nodos en el camino de v a la raíz. Entonces, carguemos una unidad de costo por cada nodo en el camino de v a la raíz por cada *búsqueda*. Para ayudarnos a contar los cargos, depositaremos una moneda imaginaria en cada nodo del camino. Ésta es estrictamente una estratagema de contabilidad, que no forma parte del programa. Cuando el algoritmo termina, recolectamos todas las monedas depositadas; éste es el costo total.

Como estrategia de contabilidad adicional, depositamos monedas estadounidenses y canadienses. Demostraremos que durante la ejecución del algoritmo podemos depositar sólo cierto número de monedas estadounidenses durante cada *búsqueda*. También demostraremos que se puede depositar sólo cierto número de monedas canadienses a cada nodo. Sumando los dos totales se obtiene una cota del total de monedas que se pueden depositar.

Ahora bosquejamos el esquema de contabilidad con un poco más de detalle. Clasificaremos los nodos por rangos, y luego dividiremos los rangos en grupos de rangos. En cada *búsqueda*, depositaremos algunas monedas estadounidenses en el fondo general y algunas monedas canadienses en vértices específicos. Para calcular el número total de monedas canadienses depositadas, calcularemos los depósitos por nodo. Sumando todos los depósitos de cada nodo en el rango r , obtendremos

el total de depósitos por rango r . Entonces sumaremos todos los depósitos por cada rango r en el grupo g , obteniéndose el total de depósitos por cada grupo de rangos g . Por último, sumaremos todos los depósitos por cada grupo de rangos g para obtener el número total de monedas canadienses depositadas en el bosque. Sumar esto al número de monedas estadounidenses totales nos da la respuesta.

Se hace una partición de los rangos en grupos. El rango r va al grupo $G(r)$, y se determinará después. El rango más grande en cualquier 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, $g > 0$, es así $F(g) - F(g - 1)$. Claro está, $G(r)$ es una cota superior muy abierta sobre el grupo de rangos más grande. Por ejemplo, supongamos que hacemos una partición de rangos como en la figura 8.17. En este caso, $G(r) = \sqrt{r}$. El rango más grande del grupo g es $F(g) = g^2$, y se observa que el grupo $g > 0$ contiene los rangos $F(g - 1) + 1$ hasta $F(r)$, inclusive. Esta fórmula no se aplica al grupo 0 de rangos, así que, por conveniencia, aseguraremos que el grupo de rango 0 contiene sólo elementos de rango 0. Observe que los grupos se hacen a base de rangos consecutivos.

Como se mencionó antes, cada instrucción *unión* tarda un tiempo constante, en la medida en que cada raíz haga el seguimiento de qué tan grandes son sus subárboles. Así, las *uniones* son esencialmente libres, hasta donde llega esta demostración.

Cada *búsqueda*(i) tarda un tiempo proporcional al número de vértices en el camino del vértice que representa i hasta la raíz. Así, depositaremos una moneda por cada vértice en el camino. Si esto es todo lo que hacemos, sin embargo, no podemos esperar mucho de la cota, porque no se está aprovechando la compresión de caminos. Así, necesitamos aprovechar dicha compresión en nuestros análisis. Se usará una contabilidad imaginaria.

Para cada vértice, v , en el camino del vértice que representa i a la raíz, depositaremos una moneda en una de dos cuentas:

1. Si v es la raíz, o si el padre de v es la raíz, o si el padre de v está en un grupo de rangos diferente de v , entonces se le carga una unidad por esta regla. Esto deposita una moneda estadounidense en el fondo común.
2. Si no, se deposita una moneda canadiense en el vértice.

Figura 8.17 Partición posible de rangos en grupos

Grupo	Rango
0	0
1	1
2	2, 3, 4
3	5 a 9
4	10 a 16
i	$(i - 1)^2 + 1$ a i^2

LEMA 8.4.

Para cualquier búsqueda(v), el número total de monedas depositadas, ya sea en el fondo común o en un vértice, es exactamente igual al número de nodos en el camino de v a la raíz.

DEMOSTRACIÓN:

Obvia.

Así, todo lo que necesitamos hacer es sumar todas las monedas estadounidenses depositadas por la regla 1 con todas las monedas canadienses depositadas por la regla 2.

Estamos haciendo m búsquedas como máximo. Necesitamos acotar el número de monedas que se pueden depositar en el fondo común durante una búsqueda.

LEMA 8.5.

Para el algoritmo completo, el total de depósitos de monedas estadounidenses por la regla 1 es $m(G(n) + 2)$.

DEMOSTRACIÓN

Esto es fácil. Para cualquier búsqueda, se depositan dos monedas estadounidenses, a causa de la raíz y su hijo. Por el lema 8.3, los vértices que ascienden en el camino crecen monótonamente en el rango, y como hay $G(n)$ grupos de rangos cuando mucho, sólo $G(n)$ otros vértices se pueden calificar como un depósito de la regla 1 para cualquier búsqueda particular. Así, durante cualquier búsqueda, a lo más $G(n) + 2$ monedas estadounidenses se pueden colocar en el fondo común. Así, cuando mucho se pueden depositar $m(G(n) + 2)$ monedas estadounidenses por la regla 1 para una secuencia de m búsquedas.

Para obtener un buen cálculo de todos los depósitos canadienses por la regla 2, sumaremos los depósitos por vértices en vez de por instrucciones búsqueda. Si se deposita una moneda en el vértice v por la regla 2, v se moverá por la compresión de caminos y obtendrá un parente nuevo de rango más alto que su parente anterior. (Es aquí donde se está aplicando la compresión de caminos.) Así, un vértice v en el grupo de rangos $g > 0$ se puede mover $F(g) - F(g-1)$ veces cuando mucho, antes de que su parente quede fuera del grupo g , ya que ése es el tamaño del grupo de rangos.[†] Después de ocurrir esto, todos los cargos futuros a v serán por la regla 1.

LEMA 8.6.

El número de vértices, $N(g)$, en el grupo de filas $g > 0$ es $n/2^{F(g-1)}$ como máximo.

DEMOSTRACIÓN:

Por el lema 8.2, hay a lo más $n/2^r$ vértices de la fila r . Sumando para todos los rangos del grupo g , se obtiene

[†] Esto se puede reducir en 1. No lo hacemos por claridad; la cota no se mejora por ser más cuidadoso aquí.

$$\begin{aligned} N(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^r} \\ &\leq \sum_{r=F(g-1)+1}^{\infty} \frac{n}{2^r} \\ &\leq n \cdot \sum_{r=F(g-1)+1}^{\infty} \frac{1}{2^r} \\ &\leq \frac{n}{2^{F(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \\ &\leq \frac{2n}{2^{F(g-1)+1}} \\ &\leq \frac{n}{2^{F(g-1)}} \end{aligned}$$

LEMA 8.7.

El número máximo de monedas canadienses depositadas en todos los vértices en el grupo de rangos g es $n F(g)/2^{F(g-1)}$ como máximo.

DEMOSTRACIÓN:

Cada vértice del grupo de rangos puede recibir cuando mucho $F(g) - F(g-1) \leq F(g)$ monedas canadienses mientras su parente permanezca en su grupo de rangos, y el lema 8.6 indica cuántos de tales vértices hay. El resultado se obtiene por una simple multiplicación.

LEMA 8.8.

El depósito total por la regla es $2 \cdot n \sum_{g=1}^{G(n)} F(g)/2^{F(g-1)}$ monedas canadienses como máximo.

DEMOSTRACIÓN:

En virtud de que el grupo de rango 0 sólo contiene elementos de rango 0, no puede contribuir a los cargos de la regla 2 (no puede tener un parente en el mismo grupo de rangos). La cota se obtiene sumando los otros grupos de rangos.

Así, se tienen los depósitos por las reglas 1 y 2. El total es

$$m(G(n) + 2) + n \sum_{g=1}^{G(n)} F(g)/2^{F(g-1)} \quad (8.1)$$

Aún no hemos especificado $G(n)$ o su inversa $F(n)$. Es obvio que somos libres de escoger virtualmente cualquiera que queramos, pero tiene sentido escoger $G(n)$ para minimizar la cota anterior. No obstante, si $G(n)$ es demasiado pequeña, $F(n)$ será grande, afectando la cota. Al parecer, una buena elección es escoger $F(i)$ como la función definida recursivamente por $F(0) = 0$ y $F(i) = 2^{F(i-1)}$. Esto da $G(n) = 1 + \lfloor \log^* n \rfloor$. La figura 8.18 muestra cómo esto hace una partición de los rangos. Obsérvese que el grupo 0 contiene sólo el rango 0, que requeríamos en el lema anterior. F es muy semejante a la función de Ackerman de una variable, la cual sólo difiere en la definición del caso base ($F(0) = 1$).

TEOREMA 8.1.

El tiempo de ejecución de m uniones y búsquedas es $O(m \log^ n)$.*

DEMOSTRACIÓN:

Conectamos las definiciones de F y G en la ecuación 8.1. El número total de monedas estadounidenses es $O(mG(n)) = O(m \log^* n)$. El número total de monedas canadienses es $n \sum_{g=1}^{G(n)} F(g)/2^{F(g-1)} = n \sum_{g=1}^{G(n)} 1 = nG(n) = O(n \log^* n)$. Puesto que $m = \Omega(n)$, se obtiene la cota en consecuencia.

Lo que demuestra el análisis es que hay pocos nodos que se podrían mover con frecuencia con la compresión de caminos, y así el tiempo total consumido es relativamente pequeño.

8.7. Una aplicación

Como ejemplo del uso de esta estructura de datos, consideremos el siguiente problema. Tenemos una red de computadores y una lista de conexiones bidireccionales; cada una de estas conexiones permite la transferencia de archivos de un computador a otro. ¿Es posible enviar un archivo desde cualquier computador de

Figura 8.18 Partición real de rangos en grupos para la demostración

Grupo	Rango
0	0
1	1
2	2
3	3, 4
4	5 a 16
5	17 a 2^{16}
6	65537 a 2^{65536}
7	rangos verdaderamente enormes

la red a otro? Una restricción adicional es que el problema se debe resolver *en línea*. Así, la lista de conexiones se presenta una a la vez, y el algoritmo debe prepararse para dar una respuesta en cualquier punto.

Un algoritmo para resolver este problema puede poner inicialmente cada computador en su propio conjunto. Nuestra invariante es que dos computadores pueden transferir archivos si y sólo si están en el mismo conjunto. Podemos ver que la capacidad de transferir archivos forma una relación de equivalencia. Entonces se leen conexiones, una a la vez. Cuando se lee alguna conexión, digamos (u, v) , evaluaremos si u y v están en el mismo conjunto y no se hace nada si lo están. Si están en diferentes conjuntos, combinamos sus conjuntos. Al final del algoritmo, el grafo es conexo si y sólo si hay exactamente un conjunto. Si hay m conexiones y n computadores, el requerimiento de espacio es $O(n)$. Usando la unión por tamaño y la compresión de caminos, obtenemos un tiempo de ejecución de $O(m\alpha(m, n))$ para el peor caso, ya que hay $2m$ búsquedas y $n - 1$ uniones como máximo. Este tiempo de ejecución es lineal para todos los propósitos prácticos.

Veremos una aplicación mucho mejor en el siguiente capítulo.

Resumen

Hemos visto una estructura de datos muy sencilla para manejar conjuntos ajenos. Cuando se efectúa la operación de *unión*, no importa, en lo que concierne a la corrección, qué conjunto retiene su nombre. Una lección valiosa que debe aprenderse aquí es que puede ser muy importante considerar las alternativas cuando no se especifica totalmente un paso particular. El paso *unión* es flexible; aprovechándonos de esto, podemos obtener un algoritmo mucho más eficiente.

La compresión de caminos es una de las formas más antiguas de *autoajuste*, que ya hemos visto en otra parte (árboles desplegados, montículos oblicuos). Su uso es de sumo interés, en especial desde un punto de vista teórico, porque fue uno de los primeros ejemplos de un algoritmo sencillo con un análisis no tan sencillo para el peor caso.

Ejercicios

8.1 Muestre el resultado de la siguiente secuencia de instrucciones: $\text{unión}(1, 2)$, $\text{unión}(3, 4)$, $\text{unión}(3, 5)$, $\text{unión}(1, 7)$, $\text{unión}(3, 7)$, $\text{unión}(8, 9)$, $\text{unión}(1, 8)$, $\text{unión}(3, 10)$, $\text{unión}(3, 11)$, $\text{unión}(3, 12)$, $\text{unión}(3, 13)$, $\text{unión}(14, 15)$, $\text{unión}(16, 17)$, $\text{unión}(14, 16)$, $\text{unión}(1, 3)$, $\text{unión}(1, 14)$ cuando las *uniones* se ejecutan

- a. arbitrariamente,
- b. por altura,
- c. por tamaño.

8.2 Para cada uno de los árboles del ejercicio anterior, efectúe una *búsqueda* con compresión de caminos del nodo más profundo.

- 8.3 Escriba un programa para determinar los efectos de la compresión de caminos y las diferentes estrategias de *unión*. Su programa debe procesar un secuencia larga de operaciones de equivalencia usando las seis posibles estrategias.
- 8.4 Demuestre que si las *uniones* se realizan por altura, entonces la profundidad de cualquier árbol es $O(\log n)$.
- 8.5 a. Demuestre que si $m = n^2$, entonces el tiempo de ejecución de m operaciones *unión/búsqueda* es $O(m)$.
 b. Demuestre que si $m = n \log n$, entonces el tiempo de ejecución de m operaciones *unión/búsqueda* es $O(m)$.
 *c. Suponga que $m = \Theta(n \log \log n)$. ¿Cuál es el tiempo de ejecución de m operaciones *unión/búsqueda*?
 *d. Suponga que $m = \Theta(n \log n)$. ¿Cuál es el tiempo de ejecución de m operaciones *unión/búsqueda*?
- 8.6 Muestre el funcionamiento del programa de la sección 8.7 sobre el siguiente grafo: $(1, 2), (3, 4), (3, 6), (5, 7), (4, 6), (2, 4), (8, 9), (5, 8)$. ¿Cuáles son los componentes conexos?
- 8.7 Escriba un programa para implantar el algoritmo de la sección 8.7.
- *8.8 Suponga que queremos agregar una operación, *desunión*, que deshaga la última operación de *unión* que no se haya deshecho ya.
 a. Demuestre que si hacemos la *unión* por altura y *búsqueda* sin compresión de caminos, entonces la *desunión* es fácil y una secuencia de m operaciones *unión, búsqueda* y *desunión* toma un tiempo $O(m \log n)$.
 b. ¿Por qué la compresión de caminos dificulta la *desunión*?
 **c. Muestre cómo implantar las tres operaciones para que la secuencia de m operaciones tome un tiempo $O(m \log n / \log \log n)$.
- *8.9 Suponga que queremos agregar una operación, *eliminación(x)*, que elimine x de su conjunto actual y lo coloque en uno propio. Muestre cómo modificar el algoritmo *unión/búsqueda* para que el tiempo de ejecución de una secuencia de m operaciones *unión, búsqueda* y *eliminación* sea $O(m \alpha(m, n))$.
- **8.10 Proporcione un algoritmo que tome como entrada un árbol de n vértices y una lista de n pares de vértices y que determine para cada par (v, w) el antecesor común más cercano de v y w . Su algoritmo se debe ejecutar en tiempo $O(n \log^* n)$.
- *8.11 Demuestre que si todos las *uniones* preceden a *búsquedas*, entonces el algoritmo de conjuntos ajenos con compresión de caminos requiere un tiempo lineal, aun si las *uniones* se hacen arbitrariamente.
- **8.12 Demuestre que si las *uniones* se hacen arbitrariamente, pero la compresión de caminos se hace en las operaciones *búsqueda*, el tiempo de ejecución para el peor caso es $\Theta(m \log n)$.
- 8.13 Compruebe que si las *uniones* se hacen por tamaño y se efectúa compresión de caminos, el tiempo de ejecución para el peor caso es $O(m \log^* n)$.

- 8.14 Suponga que implantamos la compresión parcial de caminos en *búsqueda(i)* haciendo que un nodo sí y otro no en el camino, desde i a la raíz, apunte a su abuelo (cuando ello tenga sentido). Esto se llama *división de caminos por mitades*.
 a. Escriba un procedimiento para hacer esto.
 b. Compruebe que si se ejecuta la división de caminos por mitades en las *búsquedas* y se usa la unión por altura o por tamaño, el tiempo de ejecución para el peor caso es $O(m \log^* n)$.

Referencias

Varias soluciones al problema *unión/búsqueda* se pueden encontrar en [5], [8] y [10]. Hopcroft y Ullman demostraron la cota $O(m \log^* n)$ de la sección 8.6. Tarjan [14] obtuvo la cota $O(m \alpha(m, n))$. Una cota más precisa (pero asintóticamente idéntica) para $m < n$ aparece en [2] y [17]. Otras estrategias para la compresión de caminos y las *uniones* también alcanzan la misma cota; para detalles véase [17].

Una cota inferior que muestra que con ciertas restricciones se requiere un tiempo $\Theta(m \alpha(m, n))$ para procesar m operaciones *unión/búsqueda* fue dada por Tarjan [15]. Se ha mostrado más recientemente que hay cotas idénticas en condiciones menos restrictivas, en [6] y [13].

En [1] y [9] aparecen aplicaciones de la estructura de datos *unión/búsqueda*. Ciertos casos especiales del problema *unión/búsqueda* se pueden resolver en tiempo $O(m)$ en [7]. Esto reduce el tiempo de ejecución de varios algoritmos, como [1], dominio de grafos y reducibilidad (véase las referencias del capítulo 9) con un factor de $\alpha(m, n)$. Otros, como [9] y el problema de la conectividad de grafos en este capítulo no resultan afectados. El artículo lista diez ejemplos. Tarjan usó la compresión de caminos para obtener algoritmos eficientes para varios problemas de grafos [16].

Los resultados del caso medio del problema *unión/búsqueda* aparecen en [4], [11] y [19]. Los resultados de acotar el tiempo de ejecución para cualquier operación sencilla (en oposición a la secuencia completa) aparecen en [3] y [12].

El ejercicio 8.8 está resuelto en [18].

1. A. V. Aho, J. E. Hopcroft y J. D. Ullman, "On Finding Lowest Common Ancestors in Trees", *SIAM Journal on Computing*, 5 (1976), págs. 115–132.
2. L. Banachowski, "A Complement to Tarjan's Result about the Lower Bound on the Complexity of the Set Union Problem", *Information Processing Letters*, 11 (1980), págs. 59–65.
3. N. Blum, "On the Single-operation Worst-case Time Complexity of the Disjoint Set Union Problem", *SIAM Journal on Computing*, 15 (1986), págs. 1021–1024.
4. J. Doyle y R. L. Rivest, "Linear Expected Time of a Simple Union Find Algorithm", *Information Processing Letters*, 5 (1976), págs. 146–148.
5. M. J. Fischer, "Efficiency of Equivalence Algorithms", *Complexity of Computer Computation* (eds. R. E. Miller y J. W. Thatcher), Plenum Press, 1972, págs. 153–168.
6. M. L. Fredman y M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures", *Proceedings of the Twenty-first Annual Symposium on Theory of Computing* (1989), págs. 345–354.

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), págs. 209–221.
8. B. A. Galler y M. J. Fischer, "An Improved Equivalence Algorithm", *Communications of the ACM* 7(1964), págs. 301–303.
9. J. E. Hopcroft y R. M. Karp, "An Algorithm for Testing the Equivalence of Finite Automata", *Technical Report TR-71-114*, Department of Computer Science, Cornell University, Ithaca, NY, 1971.
10. J. E. Hopcroft y J. D. Ullman, "Set Merging Algorithms", *SIAM Journal on Computing*, 2 (1973), págs. 294–303.
11. D. E. Knuth y A. Schonhage, "The Expected Linearity of a Simple Equivalence Algorithm", *Theoretical Computer Science*, 6 (1978), págs. 281–315.
12. J. A. LaPoutre, "New Techniques for the Union-Find Problem", *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), págs. 54–63.
13. J. A. LaPoutre, "Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines", *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing* (1990), págs. 34–44.
14. R. E. Tarjan, "Efficiency of a Good but Not Linear Set Union Algorithm", *Journal of the ACM*, 22 (1975), págs. 215–225.
15. R. E. Tarjan, "A Class of Algorithms Which Require Nonlinear Time to Maintain Disjoint Sets", *Journal of Computer and System Sciences*, 18 (1979), págs. 110–127.
16. R. E. Tarjan, "Applications of Path Compression on Balanced Trees", *Journal of the ACM*, 26 (1979), págs. 690–715.
17. R. E. Tarjan y J. van Leeuwen, "Worst Case Analysis of Set Union Algorithms", *Journal of the ACM*, 31 (1984), págs. 245–281.
18. J. Westbrook y R. E. Tarjan, "Amortized Analysis of Algorithm for Set Union with Backtracking", *SIAM Journal on Computing*, 18 (1989), págs. 1–11.
19. A. C. Yao, "On the Average Behavior of Set Merging Algorithms", *Proceedings of the Eighth Annual ACM Symposium on the Theory of Computation*, (1976), págs. 192–195.

Algoritmos de grafos

En este capítulo estudiaremos varios problemas comunes de la teoría de grafos. Estos algoritmos no sólo son útiles en la práctica, sino que son interesantes porque en muchas aplicaciones reales son demasiado lentos, a menos que se ponga una cuidadosa atención a la elección de la estructura de datos.

- Mostrarémos varios problemas reales que se pueden convertir en problemas de grafos.
- Proporcionaremos algoritmos para resolver algunos problemas comunes de grafos.
- Mostrarémos cómo se puede reducir drásticamente el tiempo de ejecución de estos algoritmos con la elección adecuada de estructuras de datos.
- Veremos una técnica importante conocida como búsqueda en profundidad, y se mostrará cómo usarla para resolver en tiempo lineal varios problemas que parecen no triviales.

9.1. Definiciones

Un *grafo* $G = (V, A)$ consta de un conjunto de *vértices*, V , y un conjunto de *aristas*, A . Cada arista es un par (v, w) , donde $v, w \in V$. A veces a las aristas se les llama *arcos*. Si el par es ordenado, entonces el grafo es dirigido. En ocasiones a los grafos dirigidos se les denomina *digráficos*. El vértice w es *adyacente* a v si y sólo si $(v, w) \in A$. En un grafo no dirigido con arista (v, w) , y por lo tanto (w, v) , w es adyacente a v y v es adyacente a w . A veces una arista tiene un tercer componente, llamado *peso* o *costo*.

Un *camino* en un grafo es una secuencia de vértices w_1, w_2, \dots, w_n tal que $(w_i, w_{i+1}) \in A$ para $1 \leq i < n$. La *longitud* de tal camino es el número de aristas del camino, que es igual a $n - 1$. Se permiten caminos de un vértice a sí mismo; si este camino no contiene aristas, entonces el camino tiene longitud 0. Esta es una forma conveniente de definir un caso que de otro modo sería especial. Si el grafo contiene una arista

(v, v) de un vértice a sí mismo, entonces al camino v, v se le conoce como *ciclo*. Los grafos que vamos a considerar suelen carecer de ciclos. Un camino *simple* es un camino tal que todos los vértices son distintos, excepto que el primero y el último pueden ser el mismo.

En un grafo dirigido un *ciclo* es un camino de longitud mínima igual a 1, tal que $w_1 = w_n$; este ciclo es simple si el camino es simple. Para grafos no dirigidos, requerimos que las aristas sean diferentes. La lógica de esos requerimientos es que el camino u, v, u en un grafo no dirigido no debe considerarse como ciclo, porque (u, v) y (v, u) son la misma arista. En un grafo dirigido, estas aristas son diferentes, así que tiene sentido llamarle ciclo a éste. Un grafo dirigido es *acíclico* si no tiene ciclos. En ocasiones, a un grafo dirigido acíclico se le llama por su abreviatura, *GDA*.

Un grafo no dirigido es conexo si hay un camino desde cualquier vértice a cualquier otro. Un grafo dirigido con esta propiedad se denomina *fueramente conexo*. Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin dirección en los arcos) es conexo, se dice que el grafo es *débilmente conexo*. Un *grafo completo* es un grafo en el cual hay una arista entre cualquier par de vértices.

Un ejemplo de una situación real modelable con un grafo es el sistema de un aeropuerto. Cada aeropuerto es un vértice, y dos vértices están conectados por una arista si hay un vuelo directo entre los aeropuertos representados por los vértices. La arista podría tener un peso, que representa el tiempo, distancia o el costo del vuelo. Es razonable suponer que tal grafo es dirigido, ya que puede ser más largo o costar más (dependiendo de los impuestos locales, por ejemplo) volar en direcciones diferentes. Es probable que quisieramos asegurarnos de que el sistema del aeropuerto sea fuertemente conexo, de manera que siempre sea posible volar de cualquier aeropuerto a cualquier otro. También podríamos querer determinar rápidamente el mejor vuelo entre dos aeropuertos cualesquiera. "Mejor" podría significar el camino con el menor número de aristas o podría considerarse con respecto a una, o todas, las medidas de ponderación.

El flujo de tráfico se puede modelar con un grafo. Cada intersección de calles representa un vértice, y cada calle es una arista. Los costos de las aristas podrían representar, entre otras cosas, un límite de velocidad o una capacidad (número de carriles). Entonces podríamos preguntar por la ruta más corta o usar esta información para encontrar la ubicación más probable para un embotellamiento.

En el capítulo veremos más aplicaciones de grafos. Muchos de esos grafos pueden ser muy grandes, así que es importante que los algoritmos usados sean eficientes.

9.1.1. Representación de grafos

Consideramos grafos dirigidos (los grafos no dirigidos se representan en una forma semejante).

Supongamos, por ahora, que podemos numerar los vértices, iniciando en 1. El grafo de la figura 9.1 representa 7 vértices y 12 aristas.

Una forma sencilla de representar un grafo es con un arreglo bidimensional. Esta representación se denomina *matriz de adyacencia*. Para cada arista (u, v) , se pone $a[u, v] = 1$; si no, la entrada del arreglo es 0. Si la arista tiene un peso asociado,

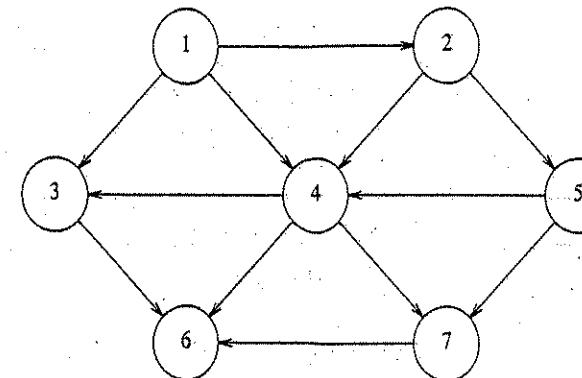


Figura 9.1 Grafo dirigido

entonces se puede poner $a[u, v]$ igual al peso y usar un peso muy grande o muy pequeño como centinela para indicar la inexistencia de aristas. Por ejemplo, si se buscara la ruta más económica de una línea aérea, podríamos representar la no existencia de vuelos con un costo ∞ . Si se buscara, por alguna razón extraña, la ruta más costosa, podríamos usar $-\infty$ (o quizás 0) para representar aristas inexistentes.

Aunque esta representación tiene el mérito de extrema simplicidad, el requerimiento de espacio es $\Theta(|V|^2)$, que puede ser prohibitivo si el grafo no tiene muchas aristas. Una matriz de adyacencia es una representación adecuada si el grafo es *denso*: $|A| = \Theta(|V|^2)$. En la mayoría de las aplicaciones que veremos, esto no es verdadero. Por ejemplo, supongamos que el grafo representa un mapa de calles, las cuales tienen una orientación como en Manhattan, donde casi todas las calles corren de norte a sur o de este a oeste. Por lo tanto, casi cualquier intersección está determinada por cuatro calles, así que si el grafo es dirigido y todas las calles son bidireccionales, entonces $|A| \approx 4|V|$. Si hay 3 000 intersecciones, se tiene un grafo de 3 000 vértices con 12 000 entradas de aristas, lo cual requeriría un arreglo de un tamaño de nueve millones. La mayoría de las entradas contendrían cero. Esto es malo intuitivamente, porque queremos que nuestras estructuras de datos representen realmente los datos que haya y no los datos que no estén presentes.

Si el grafo no es denso, en otras palabras, si es *disperso*, una mejor solución es la representación por medio de *listas de adyacencia*. Para cada vértice, mantenemos una lista de todos los vértices adyacentes. Entonces el requerimiento de espacio es $O(|A| + |V|)$. La estructura del extremo izquierdo de la figura 9.2 es simplemente un arreglo de celdas de cabecera. La representación debe ser clara en la figura 9.2.

Las listas de adyacencia son la forma estándar de representar grafos. Los grafos no dirigidos pueden representarse en una forma semejante; cada arista (u, v) aparece en dos listas, duplicándose el espacio en uso. Una necesidad común en los algoritmos de grafos es encontrar todos los vértices adyacentes a algún vértice dado v , y esto se puede hacer, en tiempo proporcional al número de tales vértices encontrados, con un simple recorrido de la lista de adyacencia apropiada.

En la mayoría de las aplicaciones reales, los vértices tienen nombres, desconocidos en tiempo de compilación, en vez de números. Como no podemos indizar un arreglo de acuerdo con un nombre desconocido, debemos dar una correspondencia

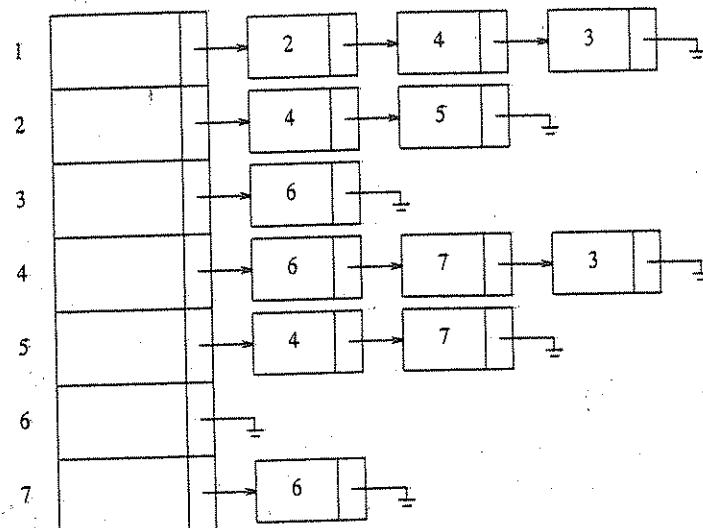


Figura 9.2 Representación de un grafo con listas de adyacencia.

entre nombres y números. La forma más sencilla de hacer esto es usar una tabla de dispersión, en la cual se almacene un nombre y un número interno en el intervalo de 1 a $|V|$ para cada vértice. Los números se asignan conforme se lee el grafo. El primer número asignado es 1. Con cada arista que entra, se comprueba si a cada uno de los dos vértices se le ha asignado un número, viendo si está en la tabla de dispersión. Si es así, usamos el número interno. En otro caso, asignamos el siguiente número disponible al vértice e insertamos el nombre del vértice y su número en la tabla de dispersión.

Con esta transformación, todos los algoritmos de grafos usarán sólo los números internos. Puesto que tarde o temprano necesitaremos tener los nombres reales de los vértices como salida y no los números internos, por cada número interno debemos grabar también el nombre del vértice correspondiente. Una forma es empleando un arreglo de cadenas. Si los nombres de los vértices son largos, esto puede consumir espacio considerable porque los nombres de los vértices se almacenan dos veces. Una alternativa es mantener un arreglo de apuntadores a la tabla de dispersión. El precio de esta alternativa es una ligera pérdida de pureza en el TDA, tabla de dispersión.

El código que presentaremos en este capítulo será pseudocódigo usando TDA tanto como sea posible. Haremos esto para ahorrar espacio y, por supuesto, hacer más clara la presentación algorítmica de los algoritmos.

9.2. Ordenación topológica

La ordenación topológica es una ordenación de los vértices de un grafo dirigido acíclico, tal que si hay un camino de v_i a v_j , entonces v_i aparece después de v_j en la

ordenación. El grafo de la figura 9.3 representa la estructura de requisitos previos de los cursos en una universidad estatal de Miami. Una arista dirigida (v, w) indica que el curso v debe cubrirse antes de intentar cursar w . Una ordenación topológica de esos cursos es cualquier secuencia de cursos que no viole el requerimiento de requisitos.

Está claro que una ordenación topológica no es posible si el grafo tiene un ciclo, ya que para dos vértices v y w en el ciclo, v precede a w y w precede a v . Además, la ordenación no es necesariamente única; cualquier ordenación legal servirá. En el grafo de la figura 9.4, tanto $v_1, v_2, v_5, v_4, v_3, v_7, v_6$ como $v_1, v_2, v_5, v_4, v_7, v_3, v_6$ son ordenaciones topológicas.

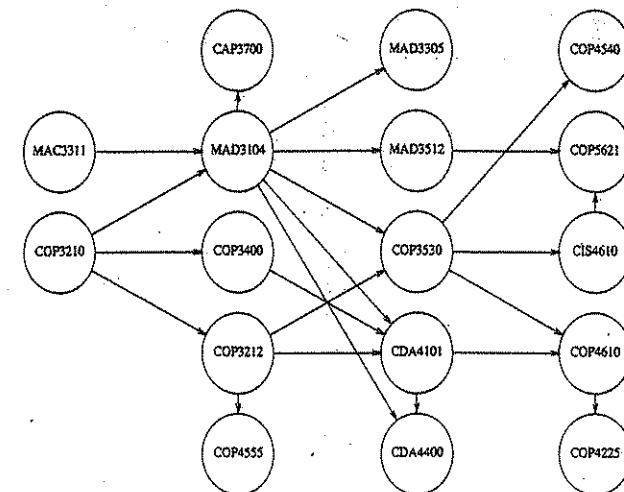
Un algoritmo sencillo para encontrar una ordenación topológica consiste en encontrar primero cualquier vértice al que no entran aristas. Entonces podemos imprimir el vértice, y eliminarlo del grafo junto con sus aristas. Despues se aplica esta misma estrategia al resto del grafo.

Para formalizar esto, definimos el *grado de entrada* de un vértice v como el número de aristas (u, v) . Calculamos los grados de entrada de todos los vértices en el grafo. Suponiendo que el arreglo *gradoent* se inicia y que el grafo está en una lista de adyacencia, podemos aplicar el algoritmo de la figura 9.5 para generar una ordenación topológica.

La función *buscar_nuevo_vértice_de_gradoent_cero* recorre el arreglo *gradoent* buscando un vértice con grado de entrada igual a 0 al que todavía no se le haya asignado un número topológico. Devuelve 0 si no existe tal vértice; esto indica que el grafo tiene un ciclo.

Cuando *buscar_nuevo_vértice_de_gradoent_cero* es un simple recorrido secuencial del arreglo *gradoent*, cada llamada a la función toma un tiempo $O(|V|)$. Puesto que hay $|V|$ llamadas, el tiempo de ejecución del algoritmo es $O(|V|^2)$.

Figura 9.3 Grafo acíclico para representar la estructura de requisitos de cursos



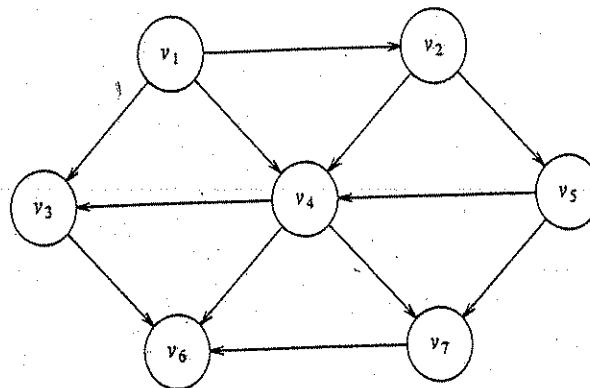


Figura 9.4 Grafo acíclico

```

procedure ordentop(G: grafo);
  var contador: integer;
      v, w: vértice;
begin
  for contador := 1 to |V| do
    begin
      v := buscar_nuevo_vértice_de_gradoent_cero;
      if v = 0 then
        error('el grafo tiene un ciclo') [se supone que esto también termina
                                         la iteración]
      else
        begin
          num_top[v] := contador;
          for cada w adyacente a v do
            gradoent[w] := gradoent[w]-1;
        end;
    end;
end;
  
```

Figura 9.5 Ordenación topológica simple

Poniendo más cuidado en las estructuras de datos es posible hacerlo mejor. La causa del deficiente tiempo de ejecución es el recorrido secuencial del arreglo *gradoent*. Si el grafo es disperso, esperaríamos que sólo se actualicen los grados de

Podemos eliminar esta inefficiencia manteniendo todos los vértices (no asignados) de grado de entrada 0 en una caja especial. Entonces la función *buscar_nuevo_vértice_de_gradoent_cero* devuelve (y elimina) cualquier vértice de la caja. Cuando decrementamos los grados de entrada de los vértices adyacentes, revisamos cada vértice, y lo colocamos en la caja si su grado de entrada cae a 0.

Para implantar la caja se puede usar una pila o una cola. Primero, para cada vértice se calcula el grado de entrada. Después todos los vértices de grado de entrada 0 se colocan en una cola inicialmente vacía. Mientras la colá no esté vacía, se elimina un vértice *v*, y todas las aristas adyacentes a *v* ven decrementados sus grados de entrada. Un vértice se agrega a la cola tan pronto como su grado de entrada cae a 0. Entones la ordenación topológica es el orden en el cual se desencolan los vértices. La figura 9.6 muestra el estado después de cada fase.

En la figura 9.7 se da una implantación con pseudocódigo de este algoritmo. Igual que antes, supondremos que el grafo ya fue leído en una lista de adyacencia y que los grados de entrada ya se calcularon y colocaron en el arreglo. En la práctica, una forma conveniente de hacer esto sería colocar el grado de entrada de cada vértice en la celda de cabecera. También suponemos un arreglo *num_top*, en el cual se coloca la numeración topológica.

El tiempo para ejecutar este algoritmo es $O(|A| + |V|)$ si se usan listas de adyacencia. Esto es evidente al darse cuenta de que el cuerpo del ciclo *for* se ejecuta cuando mucho una vez por arista. Las operaciones de la cola se hacen cuando mucho una vez por vértice, y los pasos de asignación de valores iniciales también toman un tiempo proporcional al tamaño del grafo.

Figura 9.6 Resultado de aplicar la ordenación topológica al grafo de la figura 9.4

Vértice	Grado de entrada antes de desencolar #						
	1	2	3	4	5	6	7
<i>v</i> ₁	0	0	0	0	0	0	0
<i>v</i> ₂	1	0	0	0	0	0	0
<i>v</i> ₃	2	1	1	1	0	0	0
<i>v</i> ₄	3	2	1	0	0	0	0
<i>v</i> ₅	1	1	0	0	0	0	0
<i>v</i> ₆	3	3	3	3	2	1	0
<i>v</i> ₇	2	2	2	1	0	0	0
<i>encolar</i>	<i>v</i> ₁	<i>v</i> ₂	<i>v</i> ₅	<i>v</i> ₄	<i>v</i> ₃ , <i>v</i> ₇		
<i>desencolar</i>	<i>v</i> ₁	<i>v</i> ₂	<i>v</i> ₅	<i>v</i> ₄	<i>v</i> ₃	<i>v</i> ₇	<i>v</i> ₆

```

procedure ordenop(G: grafo);
var C: COLA;
    contador: integer;
    v,w: vértice;

begin
{1} crear_nula(C); contador := 1;
{2} for cada vértice v do
{3}     if gradoent[v] = 0 then
{4}         encolar(v, C);

{5} while not está_vacia(C) do
begin

{6}     v := desencolar(C);
{7}     num_top[v] := contador; {asigna el siguiente número}
{8}     contador := contador + 1;

{9}     for cada w adyacente a v do begin
{10}         gradoent[w] := gradoent[w] - 1;
{11}         if gradoent[w] = 0 then
{12}             encolar(w, C);
        end;
    end;
{13} if contador <= |V| then
{14}     error('El grafo tiene un ciclo');

end;

```

Figura 9.7 Código para efectuar la ordenación topológica

9.3. Algoritmos del camino más corto

En esta sección examinaremos varios problemas del camino más corto. La entrada es un grafo ponderado: asociado a cada arista (v_i, v_j) hay un costo c_{ij} para cruzar el arco. El costo de un camino v_1, v_2, \dots, v_n es $\sum_{i=1}^{n-1} c_{i,i+1}$. A esto se le llama *longitud ponderada del camino*. La *longitud no ponderada del camino* es simplemente el número de aristas del camino, a saber, $n - 1$.

PROBLEMA DEL CAMINO MÁS CORTO CON ORIGEN ÚNICO:

Dado como entrada un grafo ponderado, $G=(V, A)$, y un vértice diferenciado, s , encontrar el camino ponderado más corto de s a cada uno de los demás vértices de G .

Por ejemplo, en el grafo de la figura 9.8, el camino ponderado más corto de v_1 a v_6 tiene un costo de 6 y ya de v_1 a v_4 a v_7 a v_6 . El costo del camino no ponderado más corto entre esos dos vértices es 2. En general, cuando no se especifica si nos referimos a un camino ponderado o no ponderado, el camino es ponderado si el grafo también lo es. Observe que en este grafo no hay camino de v_6 a v_1 .

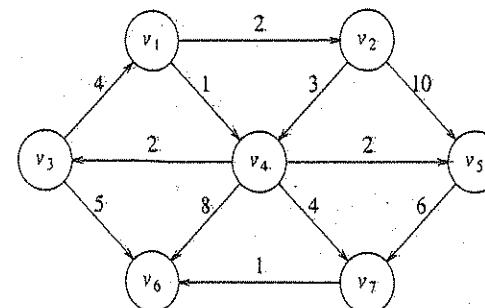


Figura 9.8 Grafo dirigido G

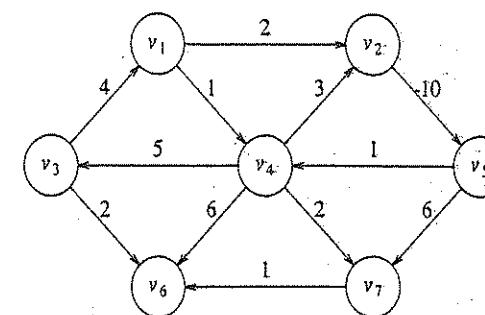


Figura 9.9 Grafo con un ciclo de costo negativo

El grafo del ejemplo anterior no tiene aristas de costo negativo. El grafo de la figura 9.9 muestra el problema que pueden ocasionar las aristas negativas. El camino de v_5 a v_4 tiene un costo 1, pero existe un camino más corto siguiendo el ciclo v_5, v_4, v_2, v_5, v_4 , el cual tiene un costo de -5. Este camino aún no es el más corto, porque podríamos permanecer indefinidamente en el ciclo. Así, el camino más corto entre los dos puntos está indefinido. De manera semejante, el camino más corto de v_1 a v_6 está indefinido porque se puede caer en el mismo ciclo. A este ciclo se le conoce como *ciclo de costo negativo*; cuando en un grafo hay alguno, los caminos más cortos no están definidos. Las aristas de costo negativo no son necesariamente malas, como lo son los ciclos, pero su presencia parece crear un problema más difícil. Por conveniencia, en la ausencia de un ciclo de costo negativo, el camino más corto de s a s es cero.

Hay muchos ejemplos donde es interesante resolver el problema del camino más corto. Si los vértices representan computadores, las aristas representan un enlace entre ellos, y los costos representan costos de comunicación (cuenta telefónica por cada 1 000 bytes de datos), costos de tiempo (número de segundos requeridos para transmitir 1 000 bytes), o una combinación de éstos y otros factores, entonces podemos usar el algoritmo del camino más corto para encontrar la vía más eco-

nómica para enviar noticias electrónicas de un computador a un conjunto de otros computadores.

Es posible modelar rutas aéreas u otras rutas de tránsito masivo por medio de grafos y usar un algoritmo del camino más corto para calcular la mejor ruta entre dos puntos. En ésta y muchas aplicaciones prácticas, se podría querer encontrar el camino más corto de un vértice, s , a sólo un vértice diferente, t . En la actualidad, no hay algoritmos en los cuales el camino de s a un vértice sea más rápido (en más de un factor constante) que encontrar el camino de s a todos los vértices.

Examinaremos algoritmos para resolver cuatro versiones de este problema. Primero, consideraremos el problema del camino más corto no ponderado y mostraremos cómo resolverlo en $O(|A| + |V|)$. Después mostraremos cómo resolver el problema del camino más corto ponderado si suponemos que no hay aristas negativas. El tiempo de ejecución de este algoritmo es $O(|A|\log|V|)$ cuando se implementa con estructuras de datos razonables.

Si el grafo tiene aristas negativas, proporcionaremos una solución sencilla, la cual desafortunadamente tiene una deficiente cota de tiempo de $O(|A|\cdot|V|)$. Por último, resolveremos el problema ponderado para el caso especial de grafos acíclicos en tiempo lineal.

9.3.1. Caminos más cortos no ponderados

La figura 9.10 muestra un grafo no ponderado, G . Usando algún vértice, s , el cual es un parámetro de entrada, quisiéramos encontrar el camino más corto de s a todos los demás vértices. Sólo nos interesa el número de aristas contenidas en el camino, así que no hay pesos en las aristas. Claro está, éste es un caso especial del problema del camino más corto ponderado, pues podríamos asignar un peso de 1 a todas las aristas.

Por ahora, supongamos que sólo nos interesa la longitud de los caminos más cortos, no los caminos reales mismos. El seguimiento de los caminos reales resultará ser una simple cuestión de gestión interna.

Figura 9.10 Grafo dirigido no ponderado G

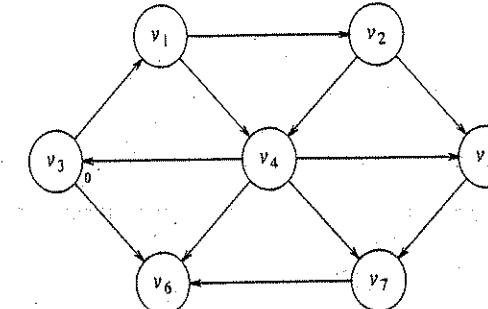
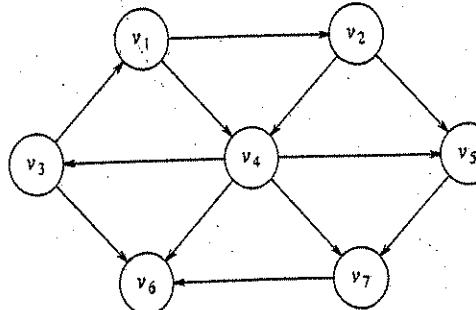


Figura 9.11: Grafo después de marcar el nodo de inicio como alcanzable en cero aristas

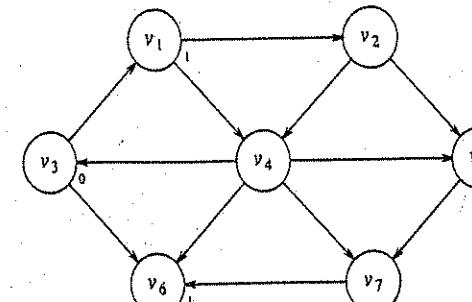
Supongamos que escogemos que s sea v_3 . De inmediato, podemos decir que el camino más corto de s a v_3 tiene una longitud 0. Podemos marcar esta información obteniendo el grafo de la figura 9.11.

Ahora se puede empezar buscando todos los vértices que están a una distancia 1 de s . Se pueden encontrar buscando en los vértices adyacentes a s . Al hacer esto, vemos que v_1 y v_6 están a una arista de distancia de s . Esto se muestra en la figura 9.12.

Ahora podemos encontrar vértices cuyo camino más corto desde s sea 2 exactamente encontrando todos los vértices adyacentes a v_1 y v_6 (los vértices a la distancia 1), cuyos caminos más cortos no se conocen todavía. Esta búsqueda nos dice que el camino más corto a v_2 y v_7 es 2. La figura 9.13 muestra el progreso hecho hasta ahora.

Por último, podemos encontrar, examinando los vértices adyacentes a los recién evaluados v_2 y v_7 , que v_5 y v_4 tienen un camino más corto de tres aristas. Ahora todos los vértices han sido calculados, y la figura 9.14 muestra el resultado final del algoritmo.

Figura 9.12: Grafo después de encontrar todos los vértices cuya longitud de camino desde s es 1



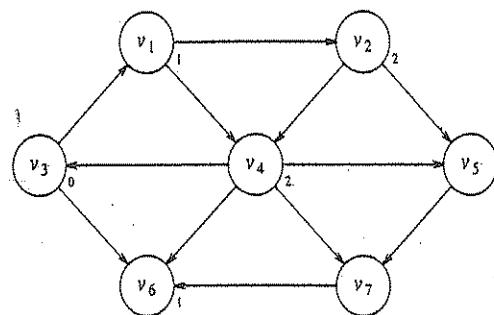


Figura 9.13: Grafo después de encontrar todos los vértices cuyo camino más corto es 2.

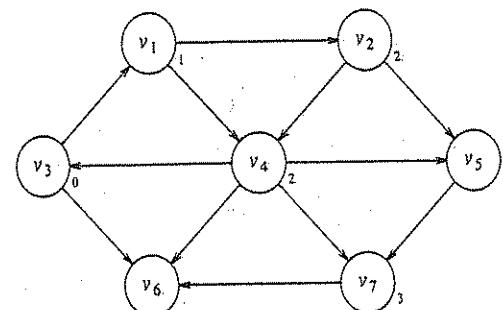


Figura 9.14 Caminos más cortos finales

Esta estrategia de recorrer el grafo se conoce como *búsqueda en amplitud*. Ésta opera procesando vértices en capas: se evalúan primero los vértices más cercanos al inicio, y al final los vértices más distantes. Esto se parece mucho al recorrido en orden de nivel de los árboles.

Dada esta estrategia, debemos traducirla en código. La figura 9.15 muestra la configuración inicial de la tabla que usará nuestro algoritmo para hacer el seguimiento de su proceso.

Por cada vértice llevaremos tres partes de información. Primero, guardaremos su distancia a s en la entrada d_v . Al principio, todos los vértices son inalcanzables excepto s , cuya longitud de camino es 0. La entrada en p_v es la variable de manejo interno, la cual nos permitirá visualizar los caminos reales. La entrada *conocido* se pone a 1 después de procesar un vértice. Al principio, todas las entradas son *desconocidas*, incluyendo el vértice inicial. Cuando un vértice es *conocido*, se tiene la garantía de que nunca encontrará un camino más económico, y así, en esencia, el procesamiento de ese vértice está completo.

v	Conocido	d_v	p_v
v_1	0	∞	0
v_2	0	∞	0
v_3	0	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figura 9.15: Configuración inicial de la tabla usada en el computador del camino más corto no ponderado.

El algoritmo básico puede describirse en la figura 9.16. El algoritmo de la figura 9.16 imita los diagramas declarando como *conocidos* los vértices a distancia $d = 0$, después a $d = 1$, después a $d = 2$, y así sucesivamente, y poniendo todos los vértices adyacentes w que aún tengan $d_w = \infty$ a una distancia $d_w = d + 1$.

Se puede visualizar el camino real regresando a través de la variable p_v . Veremos cómo hacerlo cuando estudiemos el caso ponderado.

El tiempo de ejecución del algoritmo es $O(|V|^2)$, debido a los ciclos *for* doblemente anidados. Una inefficiencia obvia es que el ciclo externo continúa hasta que $NUM_VERT - 1$ aun si todos los vértices se tornan conocidos mucho antes. Aunque se podría hacer una comprobación adicional para evitarlo, esto no afecta el tiempo

Figura 9.16 Seudocódigo del algoritmo del camino más corto no ponderado

```

procedure no ponderado(var T: TABLA); {suponer que T tiene valores iniciales}
    var dist_act: integer;
        v, w: vértice;
    begin
        for dist_act := 0 to NUM_VERT - 1 do
            for cada vértice v do
                if (T[v].conocido = false) and (T[v].dist = dist_act) then
                    begin
                        T[v].conocido := true;
                        for cada w adyacente a v do
                            if T[w].dist = MÁXINT then
                                begin
                                    T[w].dist := dist_act + 1;
                                    T[w].camino := v;
                                end;
                    end;
    end;

```

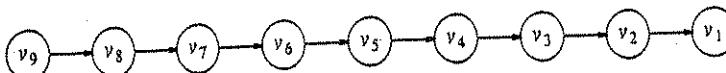


Figura 9.17 Caso deficiente para el algoritmo del camino más corto no ponderado sin estructuras de datos

de ejecución del peor caso, como se puede ver al generalizar lo que ocurre cuando la entrada es el grafo de la figura 9.17 con el vértice inicial v_9 .

La ineeficiencia se puede eliminar en una forma muy parecida a como se hizo para la ordenación topológica. En cualquier momento, sólo hay dos tipos de vértices desconocidos que tienen $d_v \neq \infty$. Algunos tienen $d_v = dist_actual$, y el resto tienen $d_v = dist_actual + 1$. Debido a esta estructura adicional, es muy costoso buscar en toda la tabla un vértice adecuado en las líneas [2] y [3].

Una solución muy sencilla, pero abstracta, es manejar dos cajas. La caja #1 tendrá los vértices desconocidos con $d_v = dist_actual$, y la caja #2 tendrá $d_v = dist_actual + 1$. La comprobación de las líneas [2] y [3] puede sustituirse encontrando cualquier vértice en la caja #1. Después de la línea [8] (dentro del bloque *if*), podemos agregar vértice en la caja #1. Después de terminar el ciclo *for* externo, la caja #1 está vacía, y la caja #2 puede transferirse a la caja #1 para la siguiente pasada por el ciclo *for*.

Esta idea se puede refinar aún más usando sólo una cola. Al inicio de la pasada, la cola contiene sólo vértices de distancia $dist_actual$. Cuando se agregan los vértices adyacentes de distancia $dist_actual + 1$, que se encolan por la parte posterior,

Figura 9.18 Seudocódigo para el algoritmo del camino más corto no ponderado

```
procedure noponderado(var T: TABLA); {supone que T tiene valores iniciales
                                         (fig. 9.30)}
```

```

var v, w: vértice;
    C: COLA;
begin
    crear-nula(C);
    encolar(s, C); {encola el vértice inicial s, el cual se determina en otra
                    parte}
    while not está_vacia(C) do
        begin
            v := desencolar(C);
            T[v].conocido := true; {realmente no se necesita nada más}
            for cada w adyacente a v do
                if T[w].dist = MÁXINT then
                    begin
                        T[w].dist := T[v].dist + 1;
                        T[w].camino := v;
                        encolar(w, C);
                    end;
            end;
        end;
end;
```

tenemos la seguridad de que no serán procesados hasta después de haber procesado todos los vértices de distancia $dist_actual$. Después de desencolar y procesar el último vértice de distancia $dist_actual$, la cola sólo contiene vértices de distancia $dist_actual + 1$, así que este proceso se perpetúa. Sólo necesitamos iniciar el proceso colocando en la cola el nodo inicial.

El algoritmo depurado se muestra en la figura 9.18. En el seudocódigo, hemos supuesto que el vértice inicial, s , es conocido y $T[s].dist$ es cero. Una rutina en Pascal puede pasar s como argumento. También, es posible que la cola se vacíe prematuramente si algunos vértices son inalcanzables desde el nodo de inicio. En este caso, se notificará una distancia $MÁXINT$ para esos nodos, lo cual es perfectamente razonable. Por último, no se usa el campo *conocido*; una vez procesado un vértice, nunca puede volver a entrar a la cola, así que el hecho de que no necesita ser reprocesado queda marcado implícitamente. Por tanto, se puede descartar el campo

Figura 9.19 Cambios de la estructura de datos durante el algoritmo del camino más corto no ponderado

v	Estado inicial	v_3 desencolado	v_1 desencolado	v_6 desencolado
	Conocido d_v p_v	Conocido d_v p_v	Conocido d_v p_v	Conocido d_v p_v
v_1	0 ∞ 0	0 1 v_3	1 1 v_3	1 1 v_3
v_2	0 ∞ 0	0 ∞ 0	0 2 v_1	0 2 v_1
v_3	0 0 0	1 0 0	1 0 0	1 0 0
v_4	0 ∞ 0	0 ∞ 0	0 2 v_1	0 2 v_1
v_5	0 ∞ 0	0 ∞ 0	0 ∞ 0	0 ∞ 0
v_6	0 ∞ 0	0 1 v_3	0 1 v_3	1 1 v_3
v_7	0 ∞ 0	0 ∞ 0	0 ∞ 0	0 ∞ 0
C:	v_3	v_1, v_6	v_6, v_2, v_4	v_2, v_4

v	v_2 desencolado	v_4 desencolado	v_5 desencolado	v_7 desencolado
	Conocido d_v p_v	Conocido d_v p_v	Conocido d_v p_v	Conocido d_v p_v
v_1	1 1 v_3	1 1 v_3	1 1 v_3	1 1 v_3
v_2	1 2 v_1	1 2 v_1	1 2 v_1	1 2 v_1
v_3	1 0 0	1 0 0	1 0 0	1 0 0
v_4	0 2 v_1	1 2 v_1	1 2 v_1	1 2 v_1
v_5	0 3 v_2	0 3 v_2	1 3 v_2	1 3 v_2
v_6	1 1 v_3	1 1 v_3	1 1 v_3	1 1 v_3
v_7	0 ∞ 0	0 3 v_4	0 3 v_4	1 3 v_4
C:	v_4, v_5	v_5, v_7	v_7	vacía

conocido. La figura 9.19 muestra cómo cambian los valores del grafo que se ha estado usando durante el algoritmo. Se conserva el campo *conocido* para facilitar el seguimiento de la tabla, y por consistencia con el resto de esta sección.

Usando el mismo análisis que en la ordenación topológica, vemos que el tiempo de ejecución es $O(|\bar{A}| + |V|)$, en la medida en que se usen las listas de adyacencia.

9.3.2. Algoritmo de Dijkstra

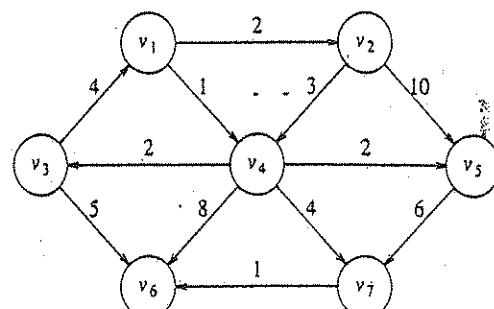
Si el grafo es ponderado, el problema se torna (al parecer) más difícil, pero aún se pueden usar las ideas del caso no ponderado.

Se maneja la misma información que antes. Así, cada vértice se marca como conocido o desconocido. Se usa una distancia provisional d_v por cada vértice, igual que antes. Esta distancia resulta ser la longitud del camino más corto de s a v usando como intermediarios sólo vértices conocidos. Como antes, registramos p_v , que es el último vértice que ocasiona un cambio de d_v .

Al método general para resolver el problema del camino más corto con origen único se le llama *algoritmo de Dijkstra*. Esta solución de hace treinta años es un ejemplo excelente de *algoritmo ávido*. En general, los algoritmos ávidos resuelven un problema en etapas haciendo lo que parece ser mejor en cada etapa. Por ejemplo, para cambiar monedas en los Estados Unidos, la mayoría de la gente cuenta primero las monedas de 25 centavos, y luego las de 10, las de 5 y de un centavo. Este algoritmo ávido da cambio usando el número mínimo de monedas. El problema principal con los algoritmos ávidos es que no siempre funcionan. La adición de una moneda de 12 centavos rompe el algoritmo de cambio de monedas, porque la respuesta que da (una moneda de 12 centavos y tres de un centavo) no es óptima (una moneda de 10 centavos y una de cinco).

El algoritmo de Dijkstra procede en etapas, igual que el algoritmo del camino más corto no ponderando. En cada etapa, el algoritmo de Dijkstra selecciona un vértice v , que tiene la d_v menor de entre todos los vértices desconocidos, y declara conocido el camino más corto de s a v . El resto de una etapa consiste en actualizar los valores de d_w .

Figura 9.20 El grafo dirigido G (otra vez)



9.3. ALGORITMOS DEL CAMINO MÁS CORTO

En el caso no ponderado, se pone $d_{vw} = d_v + 1$ si $d_{vw} = \infty$. Así, en esencia se disminuyó el valor de d_{vw} si el vértice v ofreció un camino más corto. Si aplicamos la misma lógica al caso ponderado, se debe poner $d_{vw} = d_v + c_{vw}$ si este valor nuevo de d_{vw} puede ofrecer una mejoría. En una palabra, el algoritmo decide si es o no buena idea usar v en el camino a w . El costo original, d_{vw} , es el costo sin usar v ; el costo antes calculado es nuestro camino más económico usando v (y sólo vértices conocidos).

El grafo de la figura 9.20 es nuestro ejemplo. La figura 9.21 representa la configuración inicial, suponiendo que el nodo inicial, s , es v_1 . El primer vértice elegido es v_1 , con una longitud de camino 0. Este vértice se marca como conocido. Ahora que se conoce v_1 , algunas entradas necesitan ajustes. Los vértices adyacentes a v_1 son v_2 y v_3 . Las entradas de ambos vértices se ajustan como se indica en la figura 9.22.

Después se selecciona v_4 y se marca como conocido. Los vértices v_3 , v_5 , v_6 y v_7 son adyacentes, y resulta que todos requieren ajustes, como se muestra en la figura 9.23.

A continuación se elige v_5 . v_5 es adyacente pero ya conocido, así que no se trabaja con él. v_5 es adyacente pero no ajustado, porque el costo de ir a través de

v	Conocido	d_v	p_v
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figura 9.21 Configuración inicial de la tabla usada en el algoritmo de Dijkstra

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	∞	0
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figura 9.22 Despues de declarar v_1 conocido

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

Figura 9.23 Despues de declarar v_4 conocido

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

Figura 9.24 Despues de declarar v_2 conocido

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	8	v_3
v_7	0	5	v_4

Figura 9.25 Despues de declarar v_5 y v_3 conocidos

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	6	v_7
v_7	1	5	v_4

Figura 9.26 Despues de declarar v_7 conocido

v_2 es $2 + 10 = 12$ y ya se conoce un camino de longitud 3. La figura 9.24 muestra la tabla después de elegir esos vértices.

El siguiente vértice seleccionado es v_5 a costo 3. v_7 es el único vértice adyacente, pero no se ajusta porque $3 + 6 > 5$. Entonces se elige v_3 y la distancia de v_6 se ajusta a $3 + 5 = 8$. La tabla resultante se bosqueja en la figura 9.25.

A continuación se elige v_7 , v_6 se actualiza a $5 + 1 = 6$. La tabla resultante es la figura 9.26.

Para concluir, se elige v_6 . La tabla final se muestra en la figura 9.27. La figura 9.28 indica gráficamente cómo se marcan conocidas las aristas y se actualizan los vértices durante el algoritmo de Dijkstra.

Si queremos visualizar el camino real de un vértice inicial a algún vértice v , podemos escribir una rutina recursiva para seguir el rastro dejado en el arreglo p .

Ahora proporcionamos el pseudocódigo para implantar el algoritmo de Dijkstra. Supondremos que los vértices están numerados, por conveniencia, de 1 a NÚM_VERTICE (véase la figura 9.29) y que el grafo puede leerse en una lista de adyacencia con la rutina *leer_grafo*.

Figura 9.27 Despues de declarar v_6 conocido y de terminar el algoritmo

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	1	6	v_7
v_7	1	5	v_4

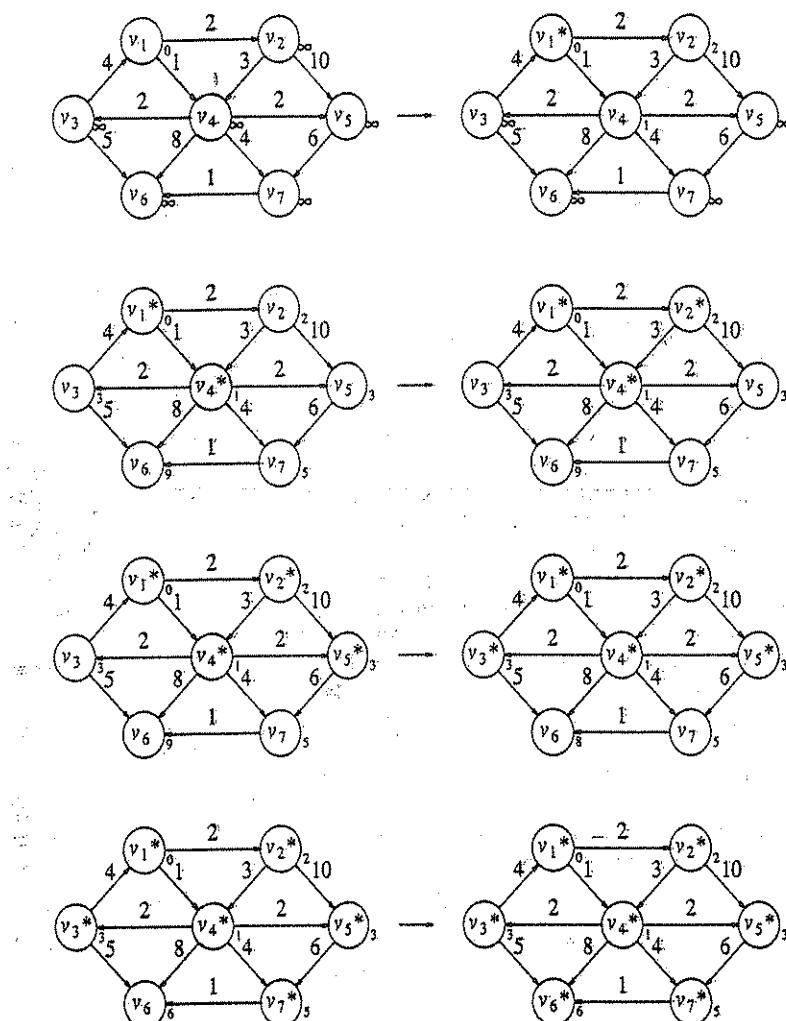


Figura 9.28 Etapas del algoritmo de Dijkstra

En la rutina de la figura 9.30, el vértice inicial se pasa a la rutina de asignación de valores iniciales. Este es el único lugar en el código donde debe conocerse el vértice inicial. Para que tenga sentido la línea [5], el 0 debe ser un vértice permisible; esto explica su declaración de tipos.

El camino puede visualizarse usando la rutina recursiva de la figura 9.31. La rutina imprime recursivamente todo el camino al vértice anterior a v en el camino, y después sólo imprime v . Esto funciona porque el camino es sencillo.

```

type
  vértice = 0 .. NÚM_VÉRTICE;

entrada_tabla = record
  cabecera: ^nodo;      {cabecera de la lista de adyacencia}
  conocido: boolean;
  dist: tipo_dist;
  camino: vértice;
end;

TABLA = array [vértice] of entrada_tabla;

```

Figura 9.29 Declaraciones para el algoritmo de Dijkstra

```

procedure inic_tabla(inicio: vértice; var G: grafo; var T: TABLA);
  var i: integer;

begin
  [1] lee_grafo(G, T); {lee el grafo de alguna forma}
  [2] for i := 1 to NÚM_VÉRTICE do begin
  [3]   T[i].conocido := FALSE;
  [4]   T[i].dist := MÁXINT;
  [5]   T[i].camino := 0;
  [6] end;
  T[inicio].dist := 0;
end;

```

Figura 9.30 Rutina de inicio de la tabla

{visualiza el camino más corto a v después de haber ejecutado Dijkstra}
{se supone que el camino existe}

```

procedure imprimir_camino(v: vértice; var T: TABLA);
begin
  if T[v].camino <> 0 then
  begin
    imprimir_camino(T[v].camino, T);
    write(' ');
  end;
  write(v);
end;

```

Figura 9.31 Rutina para imprimir el camino más corto real

```

procedure dijkstra(var T: TABLA);
var i: integer;
v, w: vértice;

begin
[1] for i := 1 to NÚM_VÉRTICE do begin
[2]   v := vértice con la distancia más corta desconocida;
[3]   T[v].conocido := true;
[4]   for cada w adyacente a v do
[5]     if T[w].conocido = false then
begin
[6]       if T[v].dist + c(v, w) < T[w].dist then {actualiza w}
begin
[7]         reducir(T[w].dist a T[v].dist + c(v, w));
[8]         T[w].camino := v;
end;
end;
end;
end;

```

Figura 9.32. Seudocódigo para el algoritmo de Dijkstra

La figura 9.32 muestra el algoritmo principal, el cual sólo es un ciclo *for* para llenar la tabla usando la regla de selección ávida.

Una demostración por contradicción comprobará que este algoritmo siempre funciona en tanto no tenga costos negativos. Si cualquier arista tiene costo negativo, el algoritmo podría producir la respuesta incorrecta (véase el ejercicio 9.7a). El tiempo de ejecución depende de cómo se manipula la tabla, lo cual todavía tenemos que considerar. Si se usa el algoritmo obvio de recorrido descendente sobre la tabla para encontrar la d_v mínima, cada fase tardará un tiempo $O(|V|)$ para encontrar el mínimo, consumiéndose así un tiempo $O(|V|^2)$ para encontrar el mínimo durante el algoritmo. El tiempo por actualización de d_w es constante, y hay cuando mucho una actualización por arista para un total de $O(|A|)$. Así, el tiempo de ejecución total es $O(|A| + |V|^2) = O(|V|^2)$. Si el grafo es denso, con $|A| = \Theta(|V|^2)$, este algoritmo no sólo es sencillo sino óptimo, ya que se ejecuta en un tiempo lineal sobre el número de aristas.

Si el grafo es disperso, con $|A| = \Theta(|V|)$, este algoritmo es demasiado lento. En este caso, necesitaremos mantener las distancias en una cola de prioridad. De hecho, hay dos formas de hacer esto; ambas son semejantes.

Las líneas [2] y [3] se combinan para formar una operación *eliminar_min*, ya que una vez encontrado el vértice mínimo desconocido, deja de ser desconocido y debe eliminarse de consideraciones futuras. La actualización de la línea [7] se puede implantar en dos formas.

Un esquema trata la actualización como una operación *decrementar_llave*. Entonces el tiempo para encontrar el mínimo es $O(\log |V|)$, como lo es el tiempo para ejecutar actualizaciones, que equivale a las operaciones *decrementar_llave*. Esto da un tiempo

de ejecución de $O(|A| \log |V| + |V| \log |V|) = O(|A| \log |V|)$, una mejoría sobre la cota anterior para grafos dispersos. Puesto que las colas de prioridad no permiten con eficiencia la operación *buscar*, necesitará mantenerse la ubicación de cada valor de d_i en la cola de prioridad y actualizarse siempre que d_i cambie en la cola de prioridad. Esto no es representativo del TDA cola de prioridad y por ello se le considera feo.

El método alternativo consiste en insertar en la cola de prioridad w y el valor nuevo de d_w cada vez que se ejecute la línea [7]. Así, puede haber más de un representante de cada vértice en la cola de prioridad. Cuando la operación *eliminar_min* quita el vértice menor de la cola de prioridad, debe asegurarse que no es conocido ya. Así, la línea [2] se convierte en un ciclo que ejecutará *eliminar_min* hasta que aparezca un vértice desconocido. Aunque este método es superior desde el punto de vista del software, y es ciertamente mucho más fácil de codificar, el tamaño de la cola de prioridad podría ser tan grande como $|A|$. Esto no afecta las cotas de tiempo asintóticas, ya que $|A| \leq |V|^2$ implica que $\log |A| \leq 2 \log |V|$. Así, todavía tendremos un algoritmo $O(|A| \log |V|)$. No obstante, el requerimiento de espacio se incrementa, y esto podría ser importante en algunas aplicaciones. Más aún, debido a que este método requiere $|A|$ *eliminar_min* en vez de sólo $|V|$, es probable que sea más lento en la práctica.

Cabe señalar que para problemas representativos, como el correo por computador y la comutación de tráfico masivo, los grafos suelen ser muy dispersos porque la mayoría de los vértices sólo tienen una pareja de aristas, así que en muchas aplicaciones es importante usar colas de prioridad para resolver el problema.

Con el algoritmo de Dijkstra hay mejores cotas de tiempo si se usan estructuras de datos diferentes. En el capítulo 11 veremos otra estructura de datos de cola de prioridad, llamada montículo de Fibonacci. Cuando se usa, el tiempo de ejecución es $O(|A| + |V| \log |V|)$. Los montículos de Fibonacci tienen buenas cotas de tiempo teóricas pero una gran cantidad de sobrecarga, no quedando claro si realmente es mejor, en la práctica, que el algoritmo de Dijkstra con los montículos binarios. No es necesario decir que no hay resultados del caso medio para este problema, ya que aún no es obvio cómo modelar un grafo aleatorio.

9.3.3. Grafos con aristas de costo negativo

Si el grafo tiene aristas de costo negativo, el algoritmo de Dijkstra no funciona. El problema es que una vez que un vértice u se declara conocido, es posible que desde algún otro vértice v desconocido haya un camino de regreso a u que sea muy negativo. En tal caso, tomar un camino de s a v con regreso a u será mejor que hacerlo de s a u sin usar v .

Una combinación de algoritmos ponderados y no ponderados resolverá el problema, pero al costo de un crecimiento drástico en el tiempo de ejecución. Olvidemos el concepto de vértices conocidos, ya que el algoritmo necesita ser capaz de cambiar de idea. Empezamos poniendo s en una cola. Luego, en cada etapa, se desencola un vértice v . Encontramos todos los vértices w adyacentes a v tales que $d_w > d_v + c_{vw}$. Actualizamos d_w y p_{vw} , y colocamos w en una cola si todavía no está allí. Puede ponerse un bit en cada vértice para indicar la presencia en la cola. Repetimos

el proceso hasta que la cola esté vacía. La figura 9.33 (casi) implanta este algoritmo.

Aunque el algoritmo funciona si no hay ciclos de costo negativo, ya no es cierto que el código de las líneas [7] a [11] se ejecute una vez por cada arista. Cada vértice se puede desencolar a lo más $|V|$ veces, así que el tiempo de ejecución es $O(|A| \cdot |V|)$ si se usan listas de adyacencia (ejercicio 9.7b). Esto es una gran mejoría respecto al algoritmo de Dijkstra, por lo que es una fortuna que en la práctica las aristas sean no negativas. Si existen ciclos de costo negativo, entonces el algoritmo tal como está escrito iterará indefinidamente. Al detener el algoritmo después de desencolar $|V| + 1$ veces, podemos asegurar su terminación.

9.3.4. Grafos acíclicos

Si sabemos que el grafo es acíclico, es posible mejorar el algoritmo de Dijkstra cambiando el orden en que se declaran conocidos los vértices, a lo que se llama la regla de selección de vértices. La regla nueva consiste en seleccionar vértices en orden topológico. El algoritmo se puede hacer en una pasada, pues las selecciones y actualizaciones pueden ocurrir conforme se efectúa la ordenación topológica.

Esta regla de selección funciona porque al elegir un vértice v , su distancia, d_v , ya no puede ser reducida, porque por la regla de ordenación topológica, no tiene aristas que provengan de nodos desconocidos.

Figura 9.33 Seudocódigo del algoritmo del camino más corto ponderado con aristas de costo negativo

```

procedure ponderado_negativo(var T: TABLA); {supone que T tiene valores
iniciales (como en la fig. 9.18)}
    var v, w: vértice;
        C: COLA;
    begin
        [1] crear_nula(C);
        [2] encolar(s, C); {encola el vértice inicial s}
        [3] while not está_vacia(C) do
            begin
                [4] v := desencolar(C);
                [5] for cada w adyacente a v do
                    if T[w].dist > T[v].dist + cv,w then
                        begin
                            [8] T[w].dist := T[v].dist + cv,w;
                            [9] T[w].camino := v;
                            [10] if w no está todavía en C then
                                encolar(w, C);
                        end;
                end;
            end;
    end;

```

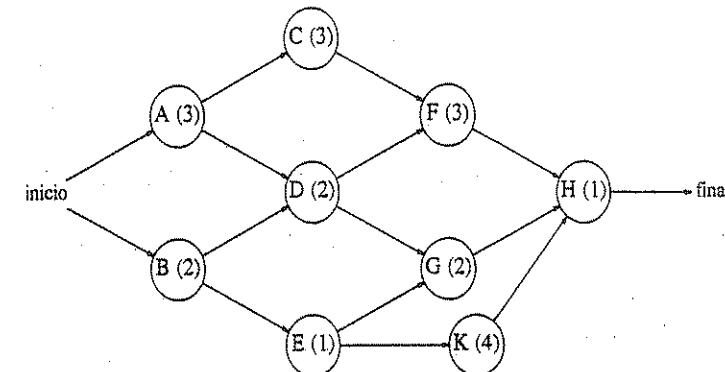


Figura 9.34 Grafo de nodos de actividad

Con esta regla de selección no se necesita la cola de prioridad; el tiempo de ejecución es $O(|A| + |V|)$, ya que la selección tarda un tiempo constante.

Un grafo acíclico podría modelar algún problema de descenso de montaña en esquí: queremos llegar del punto a a b ; pero sólo se puede ir cuesta abajo, así que, desde luego, no hay ciclos. Otra aplicación posible sería en modelar reacciones químicas (no reversibles). Podríamos representar con cada vértice un estado particular de un experimento. Las aristas representarían una transición de un estado a otro, y los pesos de las aristas podrían representar la energía liberada. Si sólo se permiten transiciones de un estado de mayor energía a otro de menor, el grafo es acíclico.

Un uso más importante de los grafos acíclicos es el análisis de caminos críticos. El grafo de la figura 9.34 servirá como ejemplo. Cada nodo representa una actividad a realizar, junto con el tiempo que lleva completar la actividad. Por ello este grafo se conoce como grafo de nodos de actividad. Las aristas representan relaciones de precedencia: una arista (v, w) significa que la actividad v debe terminarse antes que se inicie la actividad w . Por supuesto, esto implica que el grafo debe ser acíclico. Suponemos que cualesquiera actividades que no dependan (directa o indirectamente) una de la otra pueden efectuarse en paralelo por diferentes servidores.

Este tipo de grafo se podría utilizar (y se le emplea con frecuencia) para modelar proyectos de construcción. En este caso, hay varias preguntas importantes que sería de interés responder. Primero, ¿cuál es el menor tiempo de terminación del proyecto? A partir del grafo podemos ver qué en el camino A, C, F, H se requieren 10 unidades de tiempo. Otra cuestión importante es determinar qué actividades pueden retrasarse, y por cuánto tiempo, sin afectar el tiempo de terminación mínimo. Por ejemplo, la posposición de cualquiera de A, C, F o H podría retardar el tiempo de terminación diez unidades de tiempo. Por el contrario, la actividad B es menos crítica y puede retardarse hasta dos unidades de tiempo sin afectar el tiempo de terminación final.

Para realizar esos cálculos, convertimos el grafo de nodos de actividad en un grafo de nodos de evento. Cada evento corresponde a la terminación de una actividad

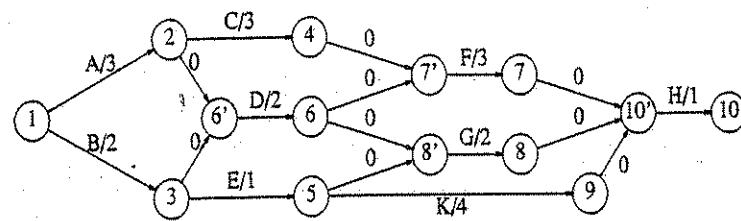


Figura 9.35 Grafo de nodos de evento

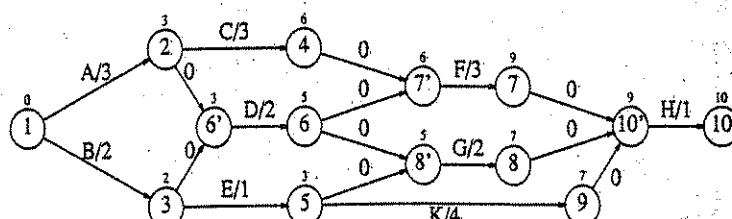
y todas sus actividades dependientes. Los eventos alcanzables desde un nodo v en el grafo de nodos de evento no pueden comenzar hasta después de terminar el evento v . Este grafo puede construirse en forma automática o manual. Podría requerirse la inserción de aristas y nodos falsos para el caso en que una actividad dependa de varias otras. Esto es necesario con el fin de evitar dependencias falsas (o carencia falsa de dependencias). El grafo de nodos de evento correspondiente al grafo de la figura 9.34 se muestra en la figura 9.35.

Para encontrar el tiempo mínimo de terminación del proyecto, sólo necesitamos encontrar la longitud del camino más largo desde el primero hasta el último evento. Para grafos generales, el problema del camino más largo no suele tener sentido, debido a la posibilidad de ciclos de costo positivo. Éstos son equivalentes a ciclos de costo negativo en problemas del camino más corto. Si existen ciclos de costo positivo, podríamos pedir el camino simple más largo, pero no se conoce solución satisfactoria a este problema. Puesto que el grafo de nodos de evento es acíclico, no es necesario preocuparse de los ciclos. En este caso, es fácil adaptar el algoritmo del camino más corto para calcular el tiempo mínimo de terminación para todos los nodos del grafo. Si EC_v es el tiempo mínimo de terminación del nodo v , entonces las reglas aplicables son

$$EC_1 = 0$$

$$EC_v = \max_{(v,w) \in E} (EC_v + C_{vw})$$

Figura 9.36 Tiempos mínimos de terminación



La figura 9.36 muestra el tiempo mínimo de terminación para cada evento en nuestro ejemplo del grafo de nodos de evento.

También podemos calcular el tiempo máximo, LC_i , en que cada evento puede terminar sin afectar el tiempo de terminación final. Las fórmulas para hacerlo son:

$$LC_v = EC_v$$

$$LC_v = \min_{(v,w) \in E} (LC_w + C_{vw})$$

Esos valores pueden calcularse en tiempo lineal manejando, para cada vértice, una lista de todos los vértices adyacentes y precedentes. Los tiempos mínimos de terminación de los vértices se calculan para los vértices de acuerdo con su orden topológico, y los tiempos máximos se calculan en orden topológico inverso. Los tiempos máximos de terminación se muestran en la figura 9.37.

El tiempo de *inactividad* para cada arista en el grafo de nodos de eventos representa la cantidad de tiempo que la terminación de la actividad correspondiente se puede retardar sin aplazar la terminación global. Es fácil ver que:

$$\text{Inactividad}_{v,w} = LC_w - EC_v - C_{vw}$$

La figura 9.38 muestra la inactividad (como tercera entrada) de cada actividad en el grafo de nodos de evento. Para cada nodo, el número superior es el tiempo mínimo de terminación y la entrada inferior es el tiempo máximo de terminación.

Figura 9.37 Tiempos máximos de terminación

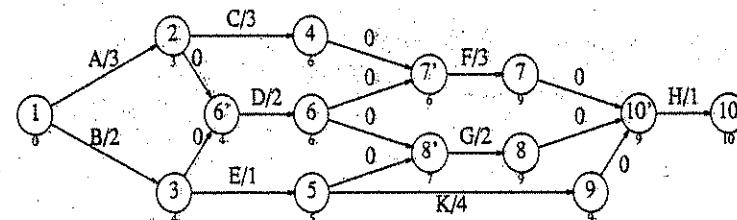
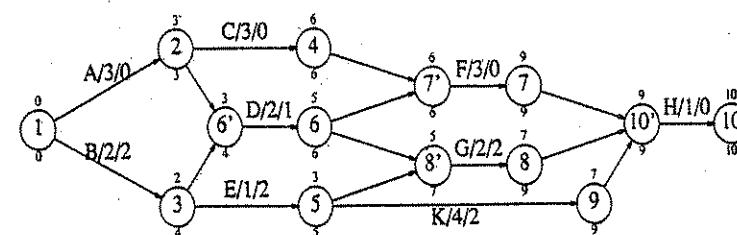


Figura 9.38 Tiempo mínimo de terminación, tiempo máximo de terminación último e inactividad



Algunas actividades tienen inactividad cero. Esas actividades son críticas, y deben terminar de acuerdo con el programa. Hay al menos un camino que consiste por completo en aristas de cero inactividad; tal camino se considera *crítico*.

9.3.5. Camino más corto entre todos los pares

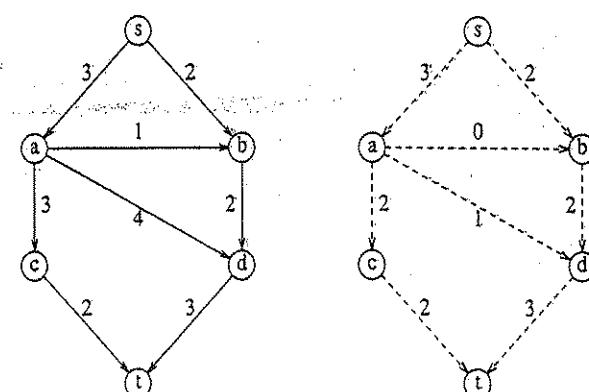
A veces es importante encontrar el camino más corto entre todos los pares de vértices del grafo. Aunque podríamos ejecutar simplemente el algoritmo adecuado de origen único $|V|$ veces, se puede esperar una solución algo más rápida, en especial en un grafo denso, si se calcula toda la información a la vez.

En el capítulo 10 veremos un algoritmo $O(|V|^3)$ para resolver este problema para grafos ponderados. Aunque, para grafos densos, ésta es la misma cota que se obtiene de la ejecución de un simple algoritmo de Dijkstra (con cola sin prioridad) $|V|$ veces, los ciclos son tan ajustados que el algoritmo especializado de todos los pares probablemente sea más rápido en la práctica. Sobre grafos dispersos, por supuesto, es más rápido ejecutar $|V|$ algoritmos de Dijkstra codificados con colas de prioridad.

9.4. Problemas de flujo en redes

Supongamos que tenemos un grafo dirigido $G = (V, A)$ con capacidades en las aristas c_{vw} . Estas capacidades podrían representar la cantidad de agua que puede fluir a través de una tubería o la cantidad de tráfico que puede circular en una calle entre dos cruces. Se tienen dos vértices: s , al cual llamamos *origen*, y t , que es el *destino*. A través de cualquier arista, (v, w) , pueden "fluir" cuando mucho c_{vw} unidades. En cualquier vértice, v , que no sea s o t , el flujo total que llega es igual al flujo total que sale. El problema de flujo máximo es determinar la cantidad máxima de flujo que puede pasar de s a t . Como ejemplo, para el grafo de la parte izquierda de la figura 9.39 el flujo máximo es 5, como se indica en el grafo de la derecha.

Figura 9.39: Un grafo (izquierda) y su flujo máximo



Como exige el planteamiento del problema, ninguna arista lleva más flujo que su capacidad. El vértice a tiene tres unidades de flujo que llega, el cual se distribuye a c y d . El vértice d toma tres unidades de flujo de a y b y las combina, enviándolas a t . Un vértice puede combinar y distribuir flujo en cualquier forma que desee, en tanto no sean violadas las capacidades y se mantenga la conservación del flujo (lo que entra debe salir).

9.4.1. Un algoritmo simple de flujo máximo

Un primer intento para resolver el problema procede por etapas. Empezamos con nuestro grafo G , y construimos un grafo de flujo G_f . Éste indica el flujo conseguido en cualquier etapa del algoritmo. Al principio, todas las aristas en G_f no tienen flujo, y esperamos que cuando el algoritmo termine, G_f contenga un flujo máximo. También construimos un grafo G_r , llamado grafo residual. G_r dice, para cada arista, cuánto flujo más puede agregarse. Podemos calcular esto restando el flujo actual de la capacidad de cada arista. Una arista en G_r se denomina arista *residual*.

Figura 9.40: Etapas iniciales del grafo, grafo de flujo y grafo residual

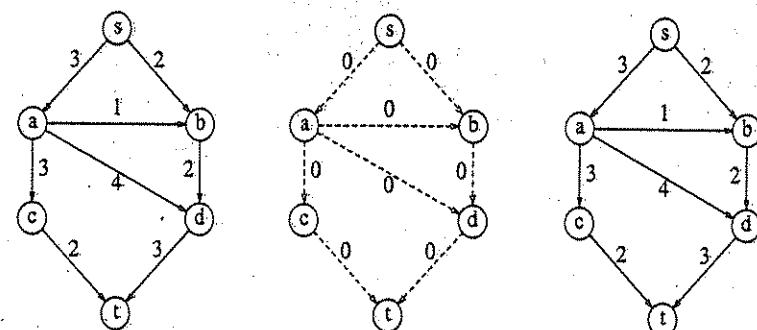
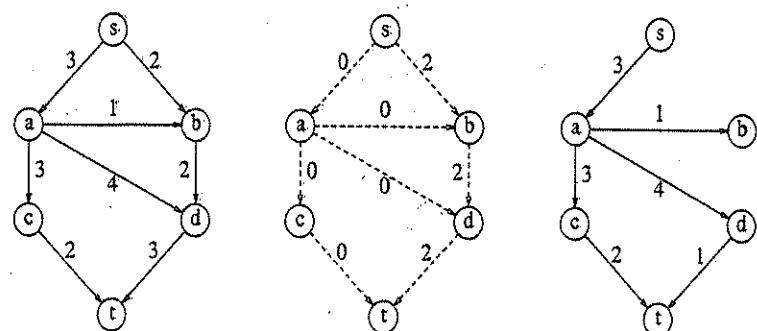


Figura 9.41: G , G_f y G_r después de dos unidades de flujo agregadas a lo largo de s, b, d, t



En cada etapa, encontramos un camino en G , de s a t . Este camino se conoce como *camino creciente*. La arista mínima en este camino es la cantidad de flujo que se puede agregar a cada arista del camino. Esto lo hacemos ajustando G_f y recalcando G_r . Cuando no encontramos ningún camino de s a t en G_r , terminamos. Este algoritmo es no determinista, en el sentido de que somos libres de escoger *cualquier* camino de s a t ; es obvio que algunas elecciones son mejores que otras, y nos ocuparemos de esto después. Ejecutaremos este algoritmo en nuestro ejemplo. Los grafos siguientes son G , G_f y G_r , respectivamente. Recordemos que hay una pequeña imperfección en este algoritmo. La configuración inicial está en la figura 9.40.

Hay muchos caminos de s a t en el grafo residual. Supongamos que elegimos s, b, d, t . Entonces podemos enviar dos unidades de flujo a través de cada arista de este camino. Adoptaremos el convenio de que una vez que hemos llenado (*saturado*) una arista, se elimina del grafo residual. Entonces se obtiene el grafo de la figura 9.41.

A continuación podemos elegir el camino s, a, c, t , el cual también permite dos unidades de flujo. Haciendo los ajustes requeridos se obtienen los grafos de la figura 9.42.

Figura 9.42 G , G_f y G_r después de dos unidades de flujo agregadas a lo largo de s, a, c, t

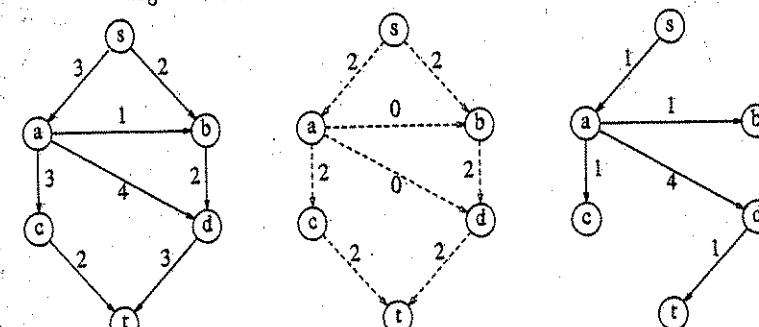
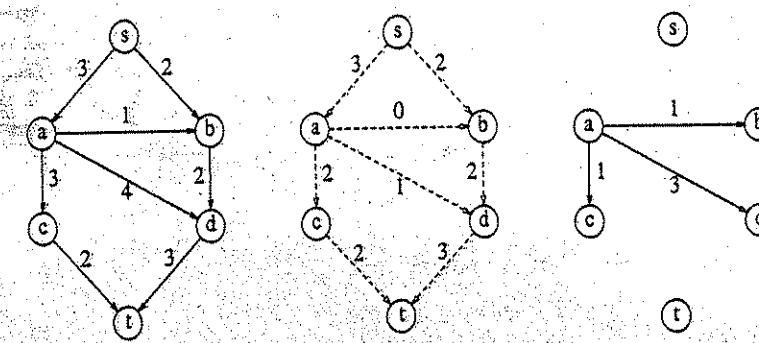


Figura 9.43 G , G_f y G_r después de una unidad de flujo agregada a lo largo de s, a, d, t : el algoritmo termina



El único camino que queda por elegir es s, a, d, t , el cual permite una unidad de flujo. Los grafos resultantes se muestran en la figura 9.43.

El algoritmo termina aquí porque t queda inalcanzable desde s . El flujo resultante de 5 es el máximo. Para ver cuál es el problema, supongamos que en el grafo inicial se elige el camino s, a, d, t . Este camino permite 3 unidades de flujo y así parece ser una buena elección. El resultado de esta elección, sin embargo, es que ahora ya no hay camino alguno de s a t en el grafo residual, y nuestro algoritmo no logra encontrar una solución óptima. Éste es un ejemplo de algoritmo ávido que no funciona. La figura 9.44 muestra por qué falla el algoritmo.

A fin de hacer que este algoritmo funcione, necesitamos permitir que el algoritmo cambie de idea. Para hacerlo, en cada arista (v, w) con flujo f_{vw} en el grafo de flujo, agregaremos una arista en el grafo residual (w, v) de capacidad f_{vw} . En efecto, con ello permitimos al algoritmo deshacer sus decisiones enviando de regreso el flujo en la dirección opuesta. Esto se ve mejor con el ejemplo. Iniciando en el grafo original y seleccionando el camino creciente s, a, d, t , obtenemos los grafos de la figura 9.45.

Figura 9.44 G , G_f y G_r si la acción inicial es agregar tres unidades de flujo a lo largo de s, a, d, t : el algoritmo termina con una solución subóptima

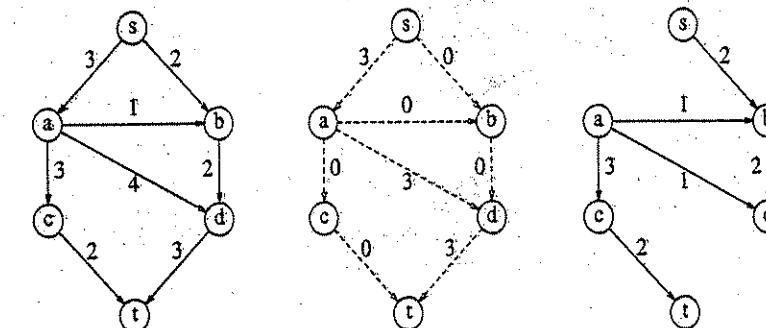
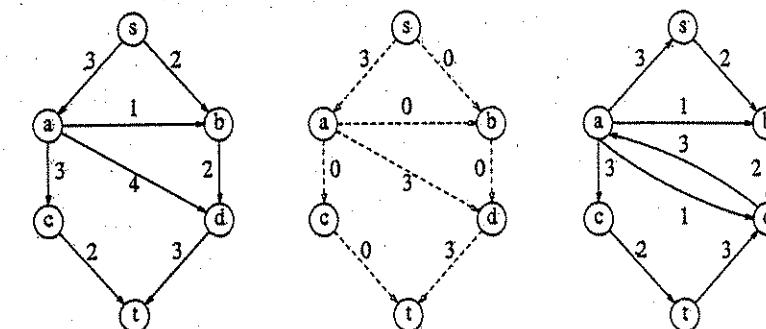


Figura 9.45 Grafos después de agregar tres unidades de flujo a lo largo de s, a, d, t usando el algoritmo correcto



Observe que en el grafo residual hay aristas en ambas direcciones entre a y d . Puede enviarse un flujo de una unidad más de a a d , o hasta tres unidades pueden ponerse de regreso; podemos deshacer el flujo. Ahora el algoritmo encuentra el camino creciente s, b, d, a, c, t de flujo 2. Envíando dos unidades de flujo de d a a , el algoritmo tarda dos unidades de flujo de la arista (a, d) y, en esencia, está cambiando su idea. La figura 9.46 muestra los grafos nuevos.

No hay camino creciente en este grafo, así que el algoritmo termina. Lo que es sorprendente es que se puede demostrar que si las capacidades de las aristas son números racionales, este algoritmo siempre termina con un flujo máximo. Esta demostración es algo difícil y rebasa el alcance de este libro. Aunque el ejemplo resultó ser acíclico, no es un requerimiento para que el algoritmo funcione. Hemos usado grafos acíclicos simplemente para conservar la sencillez.

Si las capacidades son todas enteras y el flujo máximo es f , entonces, como cada camino creciente incrementa el valor del flujo en al menos 1, f etapas son suficientes, y el tiempo de ejecución total es $O(f \cdot |A|)$, ya que un camino creciente se puede encontrar en un tiempo $O(|A|)$ por medio de un algoritmo del camino más corto no ponderado. El ejemplo clásico de por qué este tiempo de ejecución es malo se muestra en el grafo de la figura 9.47.

Por inspección se encuentra que el flujo máximo es de 2 000 000 enviando a cada lado 1 000 000. Los aumentos aleatorios podrían aumentar continuamente a lo largo de un camino que incluye la arista conectada por a y b . Si esto ocurriera repetidamente, se podrían requerir 2 000 000 aumentos, cuando se podría obtener con sólo 2.

Un método sencillo para eludir este problema es escoger siempre el camino creciente que permita el mayor incremento de flujo. Encontrar tal camino es semejante a resolver un problema del camino más corto ponderado y el truco lo hará una modificación de una línea en el algoritmo de Dijkstra. Si cap_{\max} es la capacidad máxima de la arista, se puede demostrar que $O(|A| \log cap_{\max})$ aumentos serán suficientes para encontrar el flujo máximo. En este caso, como se usa un tiempo $O(|A| \log |V|)$ para cada cálculo de un camino aumentado, se obtiene una cota total de $O(|A|^2 \log |V| \log cap_{\max})$. Si las capacidades son enteros pequeños, esto se reduce a $O(|A|^2 \log |V|)$.

Figura 9.46. Grafo después de agregar dos unidades de flujo a lo largo de s, b, d, a, c, t usando el algoritmo correcto

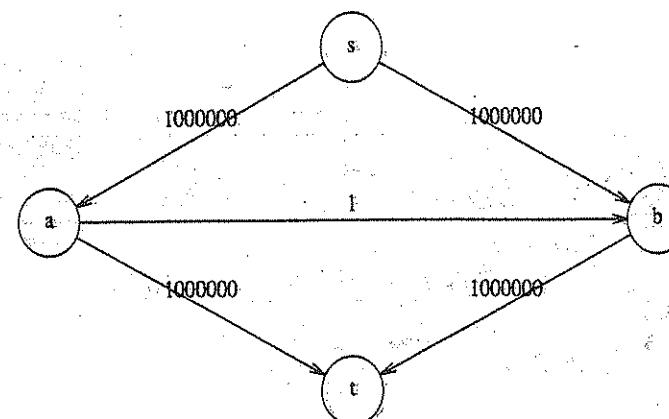
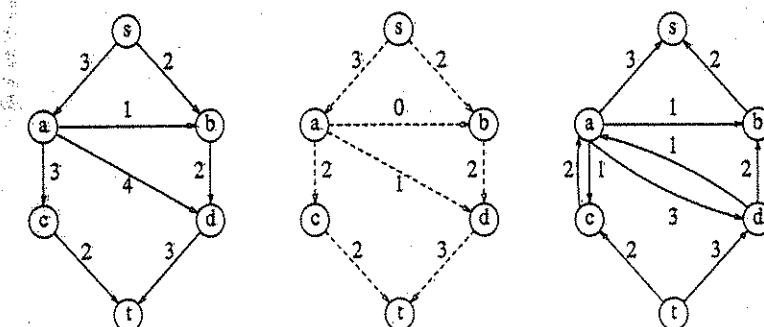


Figura 9.47. El clásico caso deficiente del aumento

Otra forma de escoger caminos crecientes es tomar siempre el camino con el menor número de aristas, con la expectativa plausible de que al escoger un camino en esta forma, es menos probable que aparezca en el camino una pequeña arista restrictiva de flujo. Con esta regla, se puede demostrar que se requieren $O(|A| \cdot |V|)$ pasos de aumento. Cada paso tarda $O(|A|)$, de nuevo utilizando un algoritmo del camino más corto no ponderado, lo que produce una cota $O(|A|^2 \cdot |V|)$ en su tiempo de ejecución.

Es posible hacer más mejoras a las estructuras de datos en este algoritmo, y hay varios algoritmos más complejos. Una larga historia de cotas mejoradas ha reducido la cota mejor conocida hasta ahora para este problema hasta llegar a $O(|A| \cdot |V| \cdot \log(|V|^2 / |A|))$ (ver las referencias). También hay una variedad de muy buenas cotas para casos especiales. Por ejemplo, en un tiempo $O(|A| \cdot |V|^{1/2})$ se encuentra un flujo máximo en un grafo, teniendo la propiedad de que todos los vértices excepto el origen y el destino tienen una sola arista de llegada con capacidad 1 o una sola arista de salida con capacidad 1. Estos grafos aparecen en muchas aplicaciones.

Los análisis requeridos para producir esas cotas son ciertamente intrincados, y no queda claro cómo se relacionan los resultados del peor caso con los tiempos de ejecución encontrados en la práctica. Un problema relacionado aún más complejo es el problema del *flujo de costo mínimo*. Cada arista no sólo tiene una capacidad sino un costo por unidad de flujo. El problema es encontrar, entre todos los flujos máximos, el de costo mínimo. Ambos problemas están siendo investigados intensamente.

9.5. Árbol de extensión mínimo

El siguiente problema que consideraremos es el de encontrar un *árbol de extensión mínimo* en un grafo no dirigido. El problema tiene sentido para grafos dirigidos pero

parece más difícil. En términos informales, un árbol de extensión mínimo de un grafo no dirigido G es un árbol formado a partir de las aristas que conectan todos los vértices de G con un costo total mínimo.

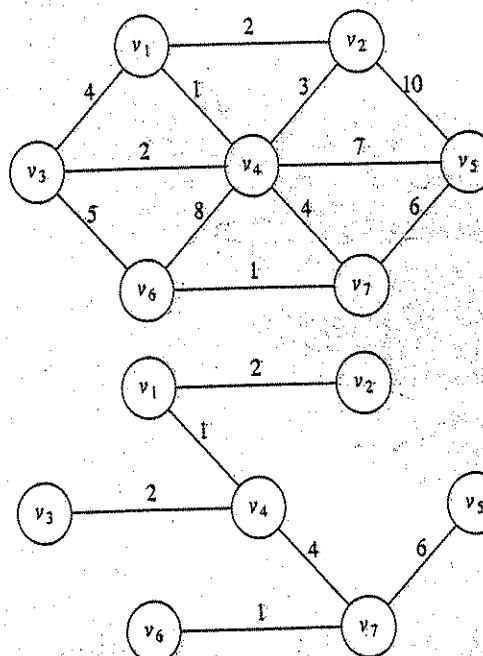
En la figura 9.48 el segundo grafo es un árbol de extensión mínimo del primero (ocurre que es único, pero esto es raro). Observe que el número de aristas del árbol de extensión mínimo es $|V| - 1$. El árbol de extensión mínimo es un *árbol* porque es acíclico, es *de extensión* porque cubre todas las aristas y es *mínimo* por razones obvias. Si necesitamos cablear una casa con un mínimo de cable, entonces se necesita resolver un problema de árbol de extensión mínimo. Hay dos algoritmos básicos para resolver este problema; ambos son ávidos, y ahora los describiremos.

9.5.1. Algoritmo de Prim

Una forma de calcular un árbol de extensión mínimo es hacer crecer el árbol en etapas sucesivas. En cada etapa se elige un nodo como raíz y agregamos al árbol una arista, y con ella su vértice asociado.

En cualquier punto del algoritmo, podemos ver que tenemos un conjunto de vértices que ya se han incluido en el árbol, pero no el resto de los vértices. Entonces el algoritmo encuentra, en cada etapa, un vértice nuevo que agregar al árbol eligiendo la arista (u, v) de tal modo que el costo de (u, v) sea el menor entre todas

Figura 9.48 Grafo G y su árbol de extensión mínimo



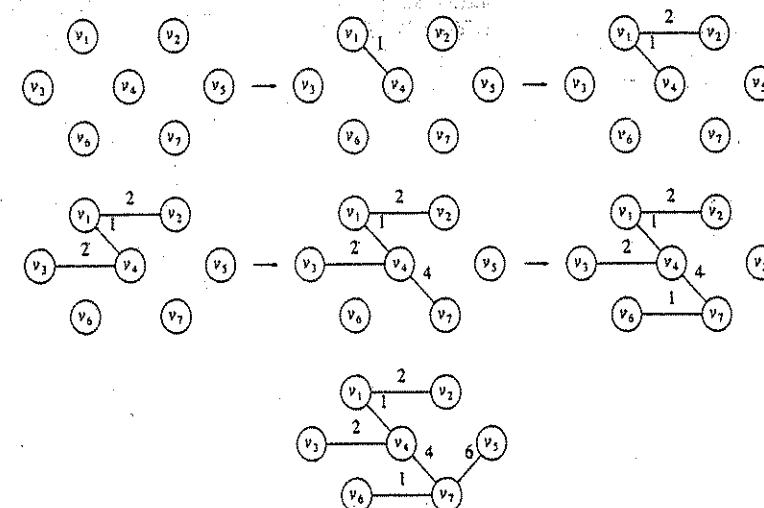
las aristas donde u está en el árbol y v no. La figura 9.49 muestra cómo es que este algoritmo construiría el árbol de extensión mínimo, empezando en v_1 . Al principio, v_1 está en el árbol como una raíz sin aristas. Cada paso agrega una arista y un vértice al árbol.

Podemos ver que el algoritmo de Prim es, en esencia, idéntico al de Dijkstra para caminos más cortos. Como antes, para cada vértice mantenemos los valores d_v y p_v y una indicación de si es conocido o no. d_v es el peso del arco más corto que conecta v con un vértice conocido, y p_v , como antes, es el último vértice que ocasiona un cambio en d_v . El resto del algoritmo es exactamente el mismo, con la excepción de que como la definición de d_v es diferente, también lo es la regla de actualización. Para este problema, la regla de actualización es aún más sencilla que antes: después de elegir un vértice v , para cada w desconocido adyacente a v , $d_w = \min(d_w, c_{vw})$.

La configuración inicial de la tabla se muestra en la figura 9.50. Se elige v_1 , y se actualizan v_2 , v_3 y v_4 . La tabla resultante de esto se muestra en la figura 9.51. El siguiente vértice elegido es v_4 . Todo vértice es adyacente a v_4 . No se examina v_1 porque es conocido. v_2 permanece sin cambio porque tiene $d_2 = 2$ y el costo de la arista de v_4 a v_2 es 3; el resto se actualiza. La figura 9.52 muestra la tabla resultante. El siguiente vértice elegido es v_2 (haciendo un desempate arbitrario). Esto no afecta ninguna de las distancias. Despues se elige v_3 , lo cual afecta la distancia en v_6 , produciendo la figura 9.53. La figura 9.54 resulta de la elección de v_7 , lo que obliga a ajustar v_6 y v_5 . Se eligen v_6 y despues v_5 , terminando el algoritmo.

La tabla final se muestra en la figura 9.55. Las aristas del árbol de extensión mínimo se pueden leer de la tabla: $(v_2, v_1), (v_3, v_4), (v_4, v_1), (v_5, v_7), (v_6, v_7), (v_7, v_4)$. El costo total es 16.

Figura 9.49 Algoritmo de Prim después de cada etapa



v	Conocido	d_v	p_v
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figura 9.50 Configuración inicial de la tabla usada en el algoritmo de Prim.

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	4	v_1
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figura 9.51 La tabla después de declarar a v_1 conocido

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	8	v_4
v_7	0	4	v_4

Figura 9.52 La tabla después de declarar a v_4 conocido

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	5	v_3
v_7	0	4	v_4

Figura 9.53 La tabla después de declarar v_2 y v_3 conocidos

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	6	v_7
v_6	0	1	v_7
v_7	1	4	v_4

Figura 9.54 La tabla después de declarar a v_7 conocido

v	Conocido	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	6	v_7
v_6	1	1	v_7
v_7	1	4	v_4

Figura 9.55 La tabla después de elegir v_6 y v_5 (termina el algoritmo de Prim)

La implantación completa del algoritmo es prácticamente idéntica al algoritmo de Dijkstra, y todo lo dicho sobre el análisis del algoritmo de Dijkstra se aplica aquí. Tenga en mente que el algoritmo de Prim se ejecuta con grafos no dirigidos, así que al codificarlo se deben poner todas las aristas en dos listas de adyacencia. El tiempo de ejecución es $O(|V|^2)$ sin montículos, lo cual es óptimo para grafos densos, y $O(|A| \log |V|)$ usando montículos binarios, que es bueno para grafos dispersos.

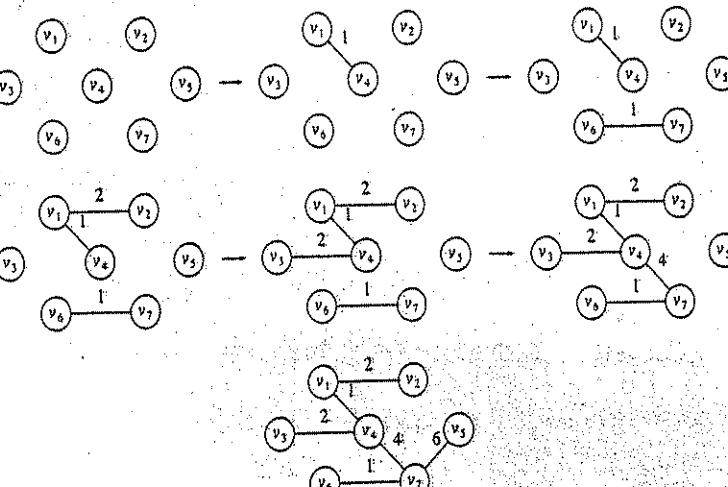
9.5.2. Algoritmo de Kruskal

Una segunda estrategia ávida es elegir continuamente las aristas en orden ascendente por peso y aceptar una arista si no ocasiona un ciclo. La acción del algoritmo sobre el grafo del ejemplo precedente se muestra en la figura 9.56.

Figura 9.56 Acción del algoritmo de Kruskal sobre G

Arista	Peso	Acción
(v_1, v_4)	1	Aceptado
(v_6, v_7)	1	Aceptado
(v_1, v_2)	2	Aceptado
(v_3, v_4)	2	Aceptado
(v_2, v_4)	3	Rechazado
(v_1, v_3)	4	Rechazado
(v_4, v_7)	4	Aceptado
(v_3, v_6)	5	Rechazado
(v_5, v_7)	6	Aceptado

Figura 9.57 Algoritmo de Kruskal después de cada etapa



Formalmente, los algoritmos de Kruskal mantienen un bosque: una colección de árboles. Al principio, hay $|V|$ árboles de un solo nodo. Al agregar una arista se combinan dos árboles en uno. Cuando termina el algoritmo sólo hay un árbol, y éste es el árbol de extensión mínimo. La figura 9.57 muestra el orden en el cual se agregan las aristas al bosque.

El algoritmo termina cuando se aceptan suficientes aristas. Es sencillo decidir si una arista (u, v) debe aceptarse o rechazarse. La estructura de datos adecuada es el algoritmo *unión/búsqueda* del capítulo anterior.

La invariante que se usará es que, en cualquier punto del proceso, dos vértices pertenecen al mismo conjunto si y sólo si están conectados en el bosque de extensión actual. Así, al principio, cada vértice está en su propio conjunto. Si u y v están en el mismo conjunto, la arista es rechazada porque, como ya están conectados, agregar (u, v) formaría un ciclo. Si no, la arista es aceptada, y se ejecuta una *unión* de los dos conjuntos que contienen a u y v . Es fácil ver que esto mantiene el conjunto invariante, porque una vez agregada la arista (u, v) al bosque de extensión, si w estaba conectado con u y x con v , entonces x y w deben estar ahora conectados y, por tanto, pertenecer al mismo conjunto.

Las aristas se podrían ordenar para facilitar la selección, pero la construcción de un montículo en tiempo lineal es una idea mucho mejor. Entonces operaciones *eliminar_mín* dan en orden las aristas que se han de probar. Por lo regular, sólo una fracción pequeña de las aristas necesita ser evaluada antes de que el algoritmo

Figura 9.58 Seudocódigo del algoritmo de Kruskal

```

procedure kruskal(var G: grafo);
var
  aristas_aceptadas: integer;
  C: CONJ_AJENO;
  M: COLA_DE_PRIORIDAD;
  u, v: vértice;
  conj_u, conj_v: tipo_conj;
  a: arista;

begin
  (1) iniciar(C);
  (2) leer_grafo_en_arreglo_monticulo(G, M);
    (no es del todo una operación del TDA)
  (3) construir_monticulo(M);

  (4) aristas_aceptadas := 0;
  while aristas_aceptadas < NÚM_VERTICE-1 do begin
    (5) a := eliminar_mín(M); {a = (u, v)}
    (6) conj_u := buscar(u, C);
    (7) conj_v := buscar(v, C);
    (8) if conj_u <> conj_v then begin
        (9) writein(u, v); {acepta la arista}
        (10) aristas_aceptadas := aristas_aceptadas + 1;
        (11) unión(C, conj_u, conj_v);
    end;
    (12) end;
  end;
end;
```

pueda terminar, aunque siempre es posible que se deban probar todas las aristas. Por ejemplo, si hubiera un vértice adicional v_8 y una arista (v_5, v_8) de costo 100, todas las aristas tendrían que ser examinadas. El procedimiento *Kruskal* de la figura 9.58 encuentra un árbol de extensión mínimo.[†] Debido a que una arista consta de tres datos, en algunas máquinas es más eficiente implantar la cola de prioridad como un arreglo de apuntadores a aristas, en vez de como un arreglo de aristas. El efecto de esta implantación es que, para reacomodar el montículo, sólo se necesita mover apuntadores, y no registros grandes.

El tiempo de ejecución de este algoritmo para el peor caso es $O(|A| \log |A|)$, el cual es dominado por las operaciones del montículo. Observe que como $|A| = O(|V|^2)$, este tiempo de ejecución es en realidad $O(|A| \log |V|)$. En la práctica, el algoritmo es mucho más rápido de lo que esta cota podría indicar.

9.6. Aplicaciones de la búsqueda en profundidad

La búsqueda en profundidad es una generalización del recorrido en orden previo. Iniciando en algún vértice, v , lo procesamos y después recursivamente se recorren todos los vértices adyacentes a él. Si este proceso se hace sobre un árbol, entonces todos los vértices se visitan sistemáticamente en un tiempo total de $O(|A|)$, ya que $|A| = \Theta(|V|)$. Si efectuamos este proceso sobre un grafo arbitrario, se debe tener cuidado de evitar ciclos. Para ello, al visitar un vértice v , lo *marcamos* como visitado, pues ahora ya hemos estado en él, y recursivamente llamamos a la búsqueda en profundidad sobre todos los vértices adyacentes que todavía no estén marcados. Implicitamente suponemos que, para grafos no dirigidos, cualquier arista (v, w) aparece dos veces en las listas de adyacencia: una como (v, w) y una como (w, v) . El procedimiento de la figura 9.59 realiza una búsqueda en profundidad (y no hace nada más en absoluto) y es una plantilla del estilo general.

El arreglo booleano (global) *visitado*[] se inicia con FALSE. Llamando recursivamente a los procedimientos sólo en los nodos que no han sido visitados, se garantiza que no se hará un ciclo infinito. Si el grafo es no dirigido y no conexo, o

Figura 9.59. Plantilla de la búsqueda en profundidad

```
procedure bep(v: vértice);
begin
    visitado[v] := TRUE;
    for cada w adyacente a v do
        if visitado[w] = FALSE then
            bep(w);
end;
```

[†] Éste es pseudocódigo porque *eliminar_mín* no puede devolver un tipo complejo en Pascal.

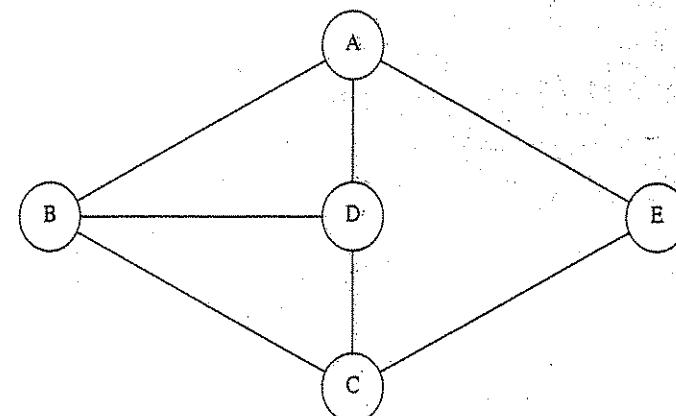
es dirigido y no fuertemente conexo, esta estrategia puede no visitar algunos nodos. Entonces buscamos un nodo no marcado, aplicamos un recorrido en profundidad allí y continuamos este proceso hasta que no queden nodos sin marcar.[†] Puesto que esta estrategia garantiza que cada arista se encuentra sólo una vez, el tiempo total para realizar el recorrido es $O(|A| + |V|)$, en tanto se usen listas de adyacencia.

9.6.1. Grafos no dirigidos

Un grafo no dirigido es conexo si y sólo si una búsqueda en profundidad que comienza en cualquier nodo visita todos los nodos. Como es tan fácil aplicar esta prueba, supondremos que los grafos que aquí nos ocupan son conexos. Si no lo son, podemos encontrar todos los componentes conectados y aplicar el algoritmo en cada uno de ellos.

Como ejemplo de la búsqueda en profundidad, supongamos que en el grafo de la figura 9.60 empezamos en el vértice A. Entonces marcamos A como visitado y llamamos recursivamente a *bep(B)*. Éste marca B como visitado y llama recursivamente a *bep(C)*. *bep(C)* marca C como visitado y llama recursivamente a *bep(D)*. Éste ve A y B, pero ambos están marcados, así que no hace llamadas recursivas. *bep(D)* también ve que C es adyacente pero está marcado, así que allí no se hace llamada recursiva, y *bep(D)* vuelve a *bep(C)*. *bep(C)* ve a B adyacente, lo ignora, encuentra un vértice adyacente, E, no visto antes y llama a *bep(E)*. Éste marca E, ignora A y C y vuelve a *bep(C)*. *bep(C)* vuelve a *bep(B)*. *bep(B)* ignora A y D y vuelve a *bep(A)* ignorando

Figura 9.60 Grafo no dirigido



[†] Una forma eficiente de implantar esto es iniciar la búsqueda en profundidad en v_1 . Si se necesita reiniciar la búsqueda en profundidad, se examina la secuencia v_{k_1}, v_{k+1}, \dots para un vértice no marcado, donde v_{k_1} es el vértice donde se inició la última búsqueda en profundidad. Esto garantiza que a través del algoritmo sólo se consume $O(|V|)$ en la búsqueda de vértice donde puedan iniciarse nuevos árboles de búsqueda en profundidad.

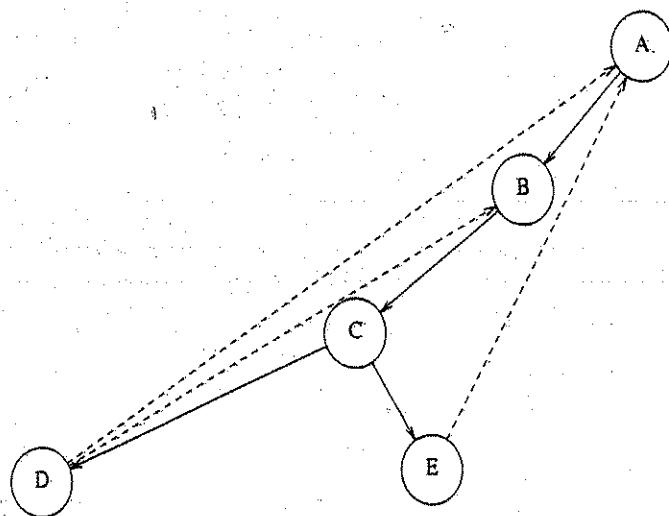


Figura 9.61 Búsqueda en profundidad en el grafo anterior

a D y E y vuelve. (De hecho hemos tocado todas las aristas dos veces, una como (v, w) y otra como (w, v) , pero esto es en realidad una por entrada en la lista de adyacencia.)

Estos pasos los ilustramos gráficamente con un *árbol de extensión en profundidad*. La raíz del árbol es A , el primer vértice visitado. Cada arista (v, w) del grafo está presente en el árbol. Si al procesar (v, w) se encuentra que w no está marcado, o si cuando se procesa (w, v) encontramos que v no está marcado, esto lo indicamos con una arista en el árbol. Si cuando procesamos (v, w) encontramos que w ya está marcado, y cuando se procesa (w, v) resulta que v ya está marcado, se traza una línea punteada, que llamaremos *arista de regreso*, para indicar que esta "arista" no es realmente parte del árbol. La búsqueda en profundidad en el grafo de la figura 9.60 se muestra en la figura 9.61.

El árbol simulará el recorrido realizado. Una numeración en orden previo del árbol, usando sólo aristas de árbol, indica el orden en el cual se marcaron los vértices. Si el grafo no es conexo, el procesamiento de todos los nodos (y aristas) requiere varias llamadas a *bep*, y cada una genera un árbol. Esta colección completa es un *bosque de extensión en profundidad*, cuyo nombre se da por razones obvias.

9.6.2. Biconectividad

Un grafo no dirigido conexo es *biconexo* (o "biconectado") si no hay vértices cuya eliminación desconecta el resto del grafo. El grafo en el ejemplo anterior es biconexo. Si los nodos son computadores y las aristas son enlaces, entonces si cualquier computador se apaga, el correo de la red no se ve afectado, excepto, por supuesto, el del computador apagado. De manera similar, si un sistema de tránsito masivo es

biconexo, los usuarios siempre tienen una ruta alternativa si no funciona alguna terminal.

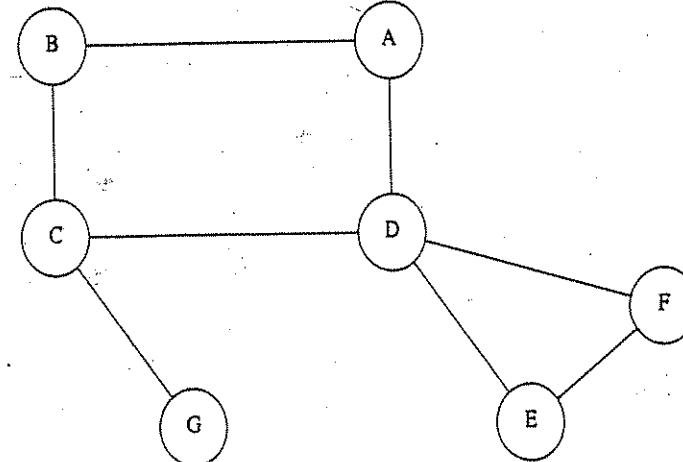
Si un grafo no es biconexo, los vértices cuya eliminación desconectaría el grafo se denominan *puntos de articulación*. Estos nodos son críticos en muchas aplicaciones. El grafo de la figura 9.62 no es biconexo: C y D son puntos de articulación. La eliminación de C desconectaría a G , y la eliminación de D desconectaría a E y F del resto del grafo.

La búsqueda en profundidad proporciona un algoritmo lineal para encontrar todos los puntos de articulación en un grafo conexo. Primero, empezando en cualquier vértice, efectuamos una búsqueda en profundidad y numeramos los nodos al visitarlos. Para cada vértice v , a este número asignado en orden previo se le llama $núm(v)$. Después, para todo vértice v en el árbol de extensión de la búsqueda en profundidad, calculamos el vértice con el menor número, al que llamamos *inferior(v)*, que es alcanzable desde v tomando cero o más aristas de árbol y después posiblemente una de regreso (en ese orden). El árbol de búsqueda en profundidad de la figura 9.63 muestra primero el número en orden previo y después el vértice con el menor número, alcanzable por la regla antes descrita.

El vértice con el menor número alcanzable por A , B y C , es el vértice 1 (A), porque pueden tomar tres aristas a D y después una de regreso a A . Podemos calcular *inferior* eficientemente realizando un recorrido en orden posterior del árbol de extensión en profundidad. Por la definición de *inferior*, *inferior(v)* es el mínimo de

1. $núm(v)$,
2. el menor $núm(w)$ entre todas las aristas de regreso (v, w) ,
3. el menor *inferior(w)* entre todas las aristas del árbol (v, w) .

La primera condición es la opción de no tomar aristas, la segunda forma es escoger aristas que no sean de árbol y una arista de regreso, y la tercera forma es escoger

Figura 9.62 Grafo con puntos de articulación C y D 

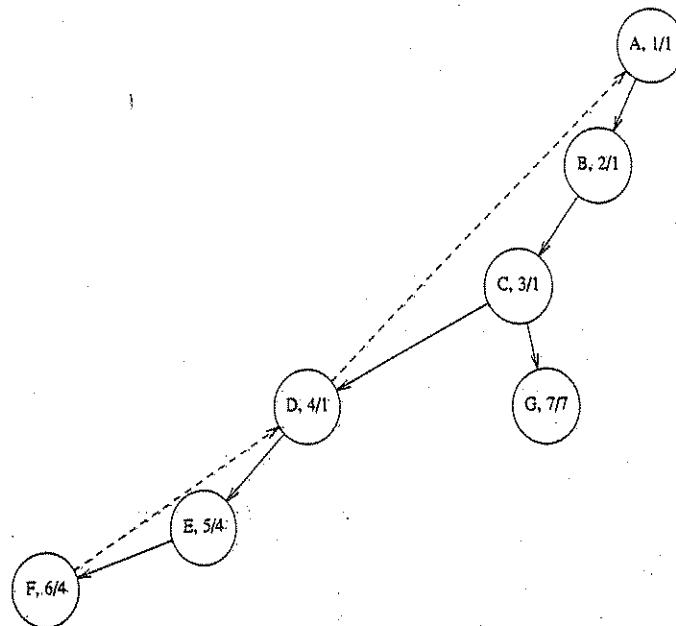


Figura 9.63: Árbol en profundidad del grafo anterior, con *númer* e *inferior*.

algunas aristas de árbol y posiblemente una de regreso. Este tercer método se describe sucintamente con una llamada recursiva. Puesto que necesitamos evaluar *inferior* para todos los hijos de v antes de poder evaluar *inferior*(v), se trata de un recorrido posterior. Para cualquier arista (v, w) , podemos decir si es de árbol o de regreso con sólo probar *númer*(v) y *númer*(w). Así, es fácil calcular *inferior*(v): sólo recorremos hacia abajo la lista de adyacencia de v , aplicamos la regla adecuada y devolvemos el mínimo. Todo el cálculo lleva un tiempo $O(|A| + |V|)$.

Todo lo que queda por hacer es usar esta información para encontrar puntos de articulación. La raíz es un punto de articulación si y sólo si tiene un hijo, porque si tiene dos hijos, la eliminación de la raíz desconecta nodos en subárboles diferentes, y si sólo tiene un hijo la eliminación sólo desconecta la raíz. Cualquier otro vértice v es un punto de articulación si y sólo si v tiene algún hijo w tal que *inferior*(w) \geq *númer*(v). Observe que esta condición siempre se satisface en la raíz; de ahí la necesidad de una prueba especial.

La parte si (if) de la demostración es evidente si examinamos los puntos de articulación que el algoritmo determina, a saber C y D . D tiene un hijo E , e *inferior*(E) \geq *númer*(D), ya que ambos son 4. Así, sólo hay una forma para que E llegue a cualquier nodo por encima de D , y es que pase por D . De manera similar, C es un punto de articulación porque *inferior*(G) \geq *númer*(C). Para probar que este algoritmo es correcto se debe demostrar que la parte sólo si de la aserción es verdadera (esto

es, encuentra *todos* los puntos de articulación). Dejamos esto como ejercicio. Como segundo ejemplo, se muestra en la figura 9.64 el resultado de aplicar este algoritmo sobre el mismo grafo, iniciando la búsqueda en profundidad a partir de C .

Concluimos dando el pseudocódigo para implantar este algoritmo. Supondremos que los arreglos *visitado*[] (con valores iniciales de falso), *númer*[], *inferior*[] y *padre*[] son globales, para mantener el código sencillo. También mantendremos una variable global llamada *contador*, la cual es iniciada a 1 para asignar los números en orden previo, *númer*[]. En condiciones normales, ésta no es una buena práctica de programación, pero incluir todas las declaraciones y pasar los parámetros adicionales podría oscurecer la lógica.

Como ya hemos afirmado, este algoritmo se puede implantar efectuando un recorrido en orden previo para calcular *númer* y después un recorrido en orden posterior para calcular *inferior*. Se puede usar un tercer recorrido para evaluar qué

Figura 9.64: Árbol en profundidad que resulta si la búsqueda en profundidad comienza en C .

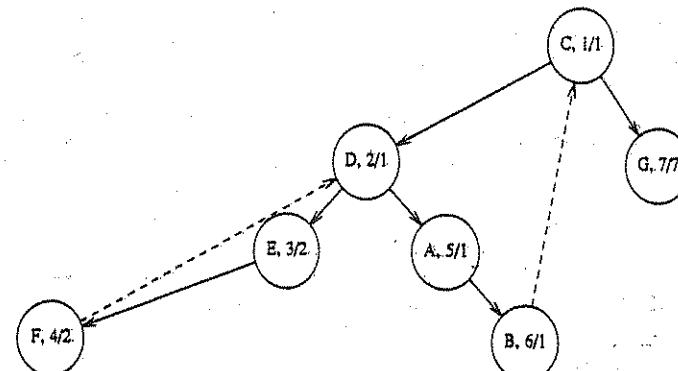


Figura 9.65: Rutina para asignar *númer* a los vértices

```
{asigna númer y calcula los padres}
procedure asignar_númer(v: vértice;
    vértice w;
begin
  1) númer[v] := contador;
  2) contador := contador + 1;
  3) visitado[v] := TRUE;
  4) for cada w adyacente a v do
      if visitado[w] = false then begin
        5) padre[w] := v;
        6) asignar_númer(w);
        7) end;
  end;
```

vértices satisfacen los criterios de punto de articulación. Efectuar tres recorridos, no obstante, puede ser inútil. La primera pasada se muestra en la figura 9.65.

Las segunda y tercera pasadas, que son recorridos en orden posterior, pueden implantarse con la codificación de la figura 9.66. La línea (8) maneja un caso especial. Si w es adyacente a v , entonces la llamada recursiva a w encontrará v adyacente a w . Esta no es una arista de regreso, sólo es una arista que ya se ha considerado y que debe ser ignorada. Si no, el procedimiento calcula el mínimo de las diferentes entradas $inferior[]$ y $núm[]$, como está especificado en el algoritmo.

No hay regla que indique si un recorrido debe ser en orden previo o posterior. Es posible hacer el proceso tanto antes como después de las llamadas recursivas. El procedimiento de la figura 9.67 combina las dos rutinas *asignar_núm* y *asignar_inferior* en una forma directa para producir el procedimiento *buscar_art*.

9.6.3. Circuitos de Euler

Consideremos los tres dibujos de la figura 9.68. Un juego popular es reconstruir esas figuras con un bolígrafo, dibujando cada línea exactamente una vez. El bolígrafo no se puede levantar del papel mientras se esté dibujando. Un desafío adicional consiste en intentar que el bolígrafo termine en el punto en que empezó. Este juego tiene una solución sorprendentemente sencilla. Deje de leer si desea intentar resolverlo.

La primera figura se puede dibujar sólo si el punto de inicio es la esquina inferior izquierda o la inferior derecha, y no es posible terminar en el punto de inicio. La

Figura 9.66 Codificación para calcular *inferior* y comprobar si hay puntos de articulación

```
(asigna inferior. También comprueba si hay puntos de articulación)

procedure asignar_inferior(v: vértice);
    var w;
begin
    inferior[v] := núm[v]; {regla 1}
    for cada w adyacente a v do begin
        if núm[w] > núm[v] then {arista directa}
            begin
                asignar_inferior(w);
                if inferior[w] >= núm[v] then
                    writeln(v, 'es un punto de articulación');
                inferior[v] := min(inferior[v], inferior[w]); {regla 3}
            end
        else
            if padre[v] <> w then {arista de regreso}
                inferior[v] := min(inferior[v], núm[w]); {regla 2}
    end;
end;
```

```
procedure buscar_art(v: vértice);
begin
    visitado[v] := TRUE;
    núm[v] := contador;
    contador := contador + 1;
    inferior[v] := núm[v]; {regla 1}

    for cada w adyacente a v do begin
        if visitado[w] = false then {arista directa}
            begin
                padre[w] := v;
                buscar_art(w);
                if inferior[w] >= núm[v] then
                    writeln(v, 'es un punto de articulación');
                inferior[v] := min(inferior[v], inferior[w]); {regla 3}
            end
        else
            if padre[v] <> w then {arista de regreso}
                inferior[v] := min(inferior[v], núm[w]); {regla 2}
    end;
end;
```

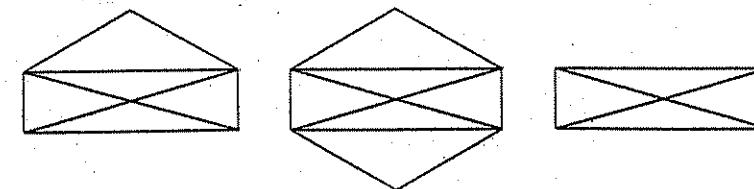
Figura 9.67 Búsqueda de los puntos de articulación en una búsqueda en profundidad

segunda figura se puede dibujar fácilmente con el punto final igual al inicial, pero la tercera no puede dibujarse con los parámetros del juego.

Podemos convertir este problema en un problema de teoría de grafos asignando un vértice a cada intersección. Entonces las aristas se pueden asignar en una forma natural, como en la figura 9.69.

Después de esta conversión, debemos encontrar un camino en el grafo que visite todas las aristas exactamente una vez. Si queremos resolver el "desafío adicional", entonces tenemos que encontrar un ciclo que visite todas las aristas exactamente una vez. Este problema de grafos fue resuelto en 1736 por Euler y marcó el inicio de la teoría de grafos. Así, es común referirse a este problema como el *camino de Euler* (algunas veces *viaje de Euler*) o *problema del circuito de Euler*, dependiendo del planteamiento específico del problema. Los problemas del viaje de Euler y del circuito

Figura 9.68 Tres dibujos



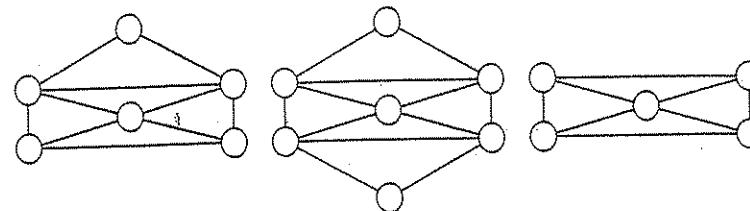


Figura 9.69 Conversión del juego a grafos

de Euler, aunque ligeramente diferentes, tienen la misma solución básica. Por ello consideraremos el problema del circuito de Euler en esta sección.

La primera observación que se puede hacer es que un circuito de Euler, que debe terminar en su vértice inicial, es posible sólo si el grafo es conexo y cada vértice tiene un grado par (número de aristas). Esto es así porque, en el circuito de Euler, se entra y se sale por un vértice. Si cualquier vértice v tiene un grado impar, tarde o temprano se alcanzará el punto en el que sólo una arista de v esté sin visitar, y tomarla nos dejará en un callejón sin salida en v . Si exactamente dos vértices tienen grado impar, un viaje de Euler, que debe visitar todas las aristas pero no necesita regresar a su vértice inicial todavía, es posible si se inicia en uno de los vértices de grado impar y termina en el otro. Si más de dos vértices tienen grado impar, entonces un viaje de Euler es imposible.

Las observaciones del párrafo anterior nos indican una condición necesaria para la existencia de un circuito de Euler. No obstante, no afirman que todos los grafos conexos que satisfacen esta propiedad deben tener un circuito de Euler, ni dan una guía de cómo encontrarlo. La condición necesaria también es suficiente. Es decir, cualquier grafo conexo, en el que todos sus vértices tienen un grado par, debe tener un circuito de Euler. Más aún, puede encontrarse un circuito en tiempo lineal.

Podemos suponer que sabemos que existe un circuito de Euler, pues se puede evaluar la condición suficiente y necesaria en un tiempo lineal. Entonces el algoritmo básico consiste en realizar una búsqueda en profundidad. Hay un número sorprendentemente grande de soluciones "obvias" que no funcionan. Algunas de ellas se presentan en los ejercicios.

El problema principal es que podríamos visitar una porción del grafo y regresar al punto de inicio prematuramente. Si se han usado todas las aristas que salen del vértice inicial, entonces parte del grafo permanece sin recorrerse. La forma más sencilla de arreglar esto es encontrar el primer vértice de este camino que tenga una arista sin visitar, y realizar otra búsqueda en profundidad. Con ello obtenemos otro circuito, que se puede empalmar con el original. Esto se continúa hasta haber recorrido todas las aristas.

Por ejemplo, consideremos el grafo de la figura 9.70. Es fácil descubrir que tiene un circuito de Euler. Supongamos que empezamos en el vértice 5 y recorremos el circuito 5, 4, 10, 5. Entonces estaremos varados, y la mayor parte del grafo permanecerá sin recorrerse. La situación se muestra en la figura 9.71.

Luego continuamos a partir del vértice 4, el cual todavía tiene aristas sin explorar. Una búsqueda en profundidad puede seguir el camino 4, 1, 3, 7, 4, 11, 10,

7, 9, 3, 4. Si empalmamos este camino con el anterior de 5, 4, 10, 5, obtendremos un camino nuevo de 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

El grafo que queda después de esto se muestra en la figura 9.72. Observe que en este grafo todos los vértices deben tener grado par, de forma que estemos seguros de encontrar un ciclo que agregar. El grafo restante puede no ser conexo, pero ello no importa. El siguiente vértice del camino que tiene aristas no recorridas es el 3. Un circuito posible sería entonces 3, 2, 8, 9, 6, 3. Al empalmar, esto da el camino 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

Figura 9.70. Grafo para el problema del circuito de Euler

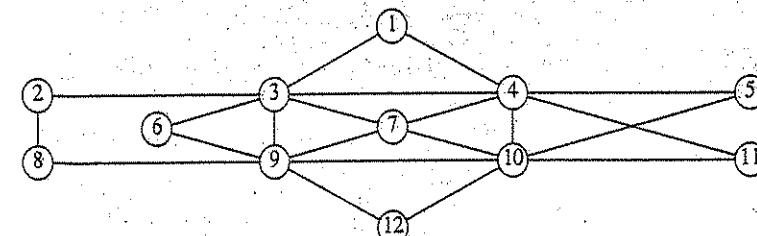


Figura 9.71. Grafo que queda después de 5, 4, 10, 5

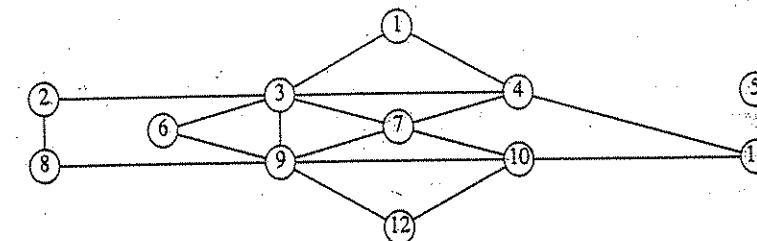
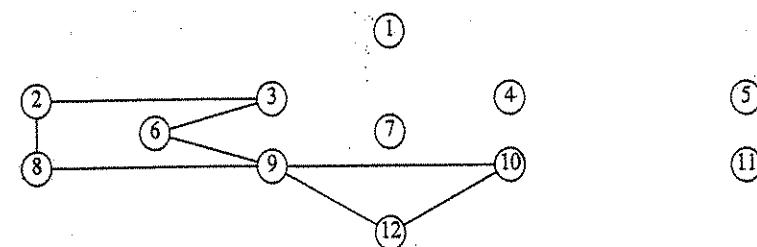


Figura 9.72. Grafo después del camino 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5



El grafo que queda está en la figura 9.73. En este camino, el siguiente vértice con una arista sin visitar es 9, y el algoritmo encuentra el circuito 9, 12, 10, 9. Cuando se agrega al camino actual, se obtiene un circuito de 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5. Como se recorren todas las aristas, el algoritmo termina con un circuito de Euler.

Para hacer eficiente este algoritmo, debemos usar las estructuras de datos adecuadas. Bosquejaremos algunas ideas, dejando como ejercicio la implantación. Para hacer simple el empalme, el camino se debe mantener como una lista enlazada. Para no hacer recorridos repetidos de las listas de adyacencia, debemos mantener, para cada lista, un apuntador a la última arista recorrida. Cuando un camino se empalma, la búsqueda de un vértice nuevo desde el cual realizar la siguiente *step* debe empezar al inicio del punto de empalme. Esto garantiza que el trabajo total efectuado en la fase de búsqueda de vértices es $O(|A|)$ durante la vida completa del algoritmo. Con las estructuras de datos adecuadas, el tiempo de ejecución del algoritmo es $O(|A| + |V|)$.

Un problema semejante consiste en encontrar un ciclo simple, en un grafo no dirigido, que visite todos los vértices. A éste se le conoce como *problema del ciclo hamiltoniano*. Aunque parece casi idéntico al problema del circuito de Euler, no se conoce ningún algoritmo eficiente para él. Veremos este problema de nuevo en la sección 9.7.

9.6.4. Grafos dirigidos

Usando la misma estrategia que con los grafos no dirigidos, los grafos dirigidos se pueden recorrer en tiempo lineal, con la búsqueda en profundidad. Si el grafo no es fuertemente conexo, es posible que una búsqueda en profundidad que comience en algún nodo no visite todos los nodos. En este caso se hacen búsquedas repetidas en profundidad, empezando en algún nodo no marcado, hasta que todos los vértices hayan sido visitados. Como ejemplo, considere el grafo dirigido de la figura 9.74.

Arbitrariamente comenzamos la búsqueda en profundidad a partir del vértice *B*; se visitan los vértices *B*, *C*, *A*, *D*, *E* y *F*. Después se recomienza en algún vértice no visitado. Arbitrariamente, comenzamos en *H*, con lo cual visitamos *I* y *J*. Por

Figura 9.73 Grafo que queda después del camino 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

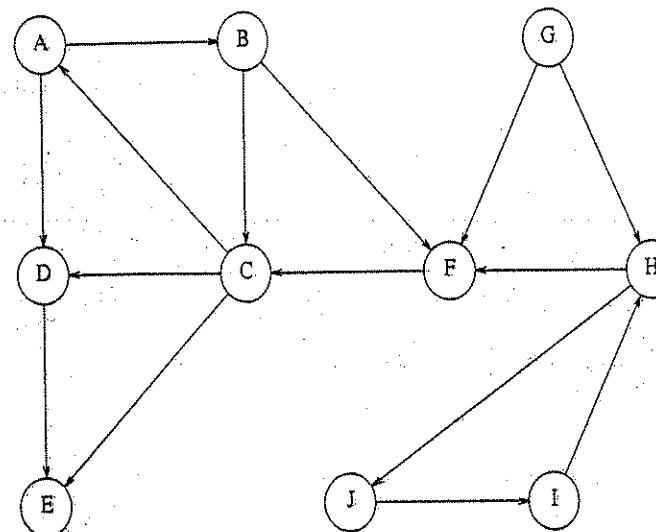
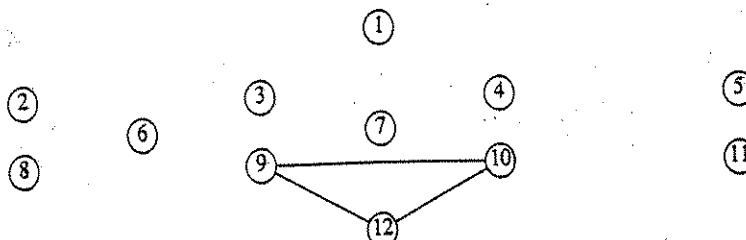
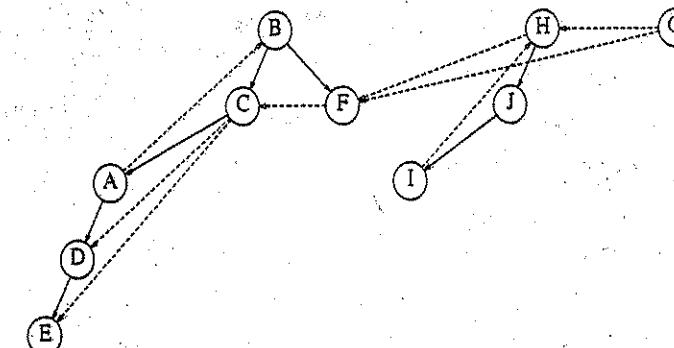


Figura 9.74. Grafo dirigido

último, comenzamos en *G*, que es el último vértice que necesita ser visitado. El árbol correspondiente de búsqueda en profundidad se muestra en la figura 9.75.

Las flechas punteadas en el bosque de extensión en profundidad son aristas (*v*, *w*) para las cuales *w* ya ha sido marcado en el momento de la consideración. En los grafos no dirigidos, siempre se trata de aristas de regreso, pero, como se puede ver, hay tres tipos de aristas que no llevan a vértices nuevos. Primero, hay *aristas de regreso*, como (*A*, *B*) e (*I*, *H*). También hay *aristas directas*, como (*C*, *D*) y (*C*, *E*), que salen de un nodo de árbol a un descendiente. Por último, hay *aristas cruzadas*, como (*F*, *C*) y (*G*, *F*), las cuales conectan dos nodos de árbol que no están relacionados directamente. Los bosques de búsqueda en profundidad se suelen dibujar con hijos y árboles nuevos que se agregan de izquierda a derecha. En una búsqueda en

Figura 9.75 Búsqueda en profundidad en el grafo anterior



profundidad en un grafo dirigido dibujado de esta manera, las aristas cruzadas siempre van de derecha a izquierda.

En algunos algoritmos que usan la búsqueda en profundidad es preciso distinguir entre los tres tipos de aristas. Esto es fácil de evaluar cuando se está efectuando la búsqueda en profundidad, por lo cual se deja como ejercicio.

La búsqueda en profundidad tiene una de sus aplicaciones en la determinación de si un grafo dirigido es o no acíclico. La regla es que un grafo dirigido es acíclico si y sólo si no tiene aristas de regreso. (El grafo anterior tiene aristas de regreso y, por tanto, no es acíclico.) El lector alerta puede recordar que también se puede usar una ordenación topológica para determinar si un grafo es acíclico. Otra forma de realizar la ordenación topológica es asignando números topológicos a los vértices: $n, n-1, \dots, 1$ por recorrido en orden posterior del bosque de extensión en profundidad. En tanto que el grafo sea acíclico, esta ordenación será consistente.

9.6.5. Localización de componentes fuertes

Es posible determinar si un grafo dirigido es fuertemente conexo realizando dos búsquedas en profundidad, y si no lo es, de hecho se pueden producir los subconjuntos de vértices que son fuertemente conexos entre ellos mismos. Esto también se puede hacer en una sola búsqueda en profundidad, pero el método usado aquí es mucho más sencillo de entender.

Primero, se efectúa una búsqueda en profundidad sobre el grafo de entrada G . Los vértices de G se numeran mediante un recorrido posterior del bosque de extensión en profundidad, y después se invierten todas las aristas de G , formando G_r . El grafo de la figura 9.76 representa G_r para el grafo G que se muestra en la figura 9.74; los vértices aparecen con sus números.

El algoritmo termina con una búsqueda en profundidad sobre G_r , iniciando siempre una búsqueda en profundidad en el vértice con el número más alto. Así, empezamos la búsqueda en profundidad en G_r en el vértice G , que tiene el número 10. Esto no nos lleva a ningún lado, así que la siguiente búsqueda se inicia en H . Esta llamada visita I y J . La siguiente llamada comienza en B y visita A , C y F . Las siguientes llamadas después de esto son $bep(D)$ y por último $bep(E)$. El bosque de extensión en profundidad que se obtiene aparece en la figura 9.77.

Cada uno de los árboles (es más fácil ver esto si ignoramos todas las aristas que no son de árbol) en este bosque de extensión en profundidad forma un componente fuertemente conexo. Así, para nuestro ejemplo, los componentes fuertemente conexos son $\{G\}$, $\{H, I, J\}$, $\{B, A, C, F\}$, $\{D\}$ y $\{E\}$.

Para ver por qué funciona este algoritmo, primero observe que si dos vértices v y w están en el mismo componente fuertemente conexo, entonces hay caminos de v a w y de w a v en el grafo original G , y por lo tanto, también en G_r . Ahora, si dos vértices v y w no están en el mismo árbol de extensión en profundidad de G_r , está claro que no pueden estar en el mismo componente fuertemente conexo.

Para probar que este algoritmo funciona, debemos demostrar que si dos vértices v y w están en el mismo árbol de extensión en profundidad de G_r , debe haber caminos de v a w y de w a v . Lo que es lo mismo, podemos demostrar que si x es la raíz del árbol de extensión en profundidad de G_r que contiene a v , entonces hay un

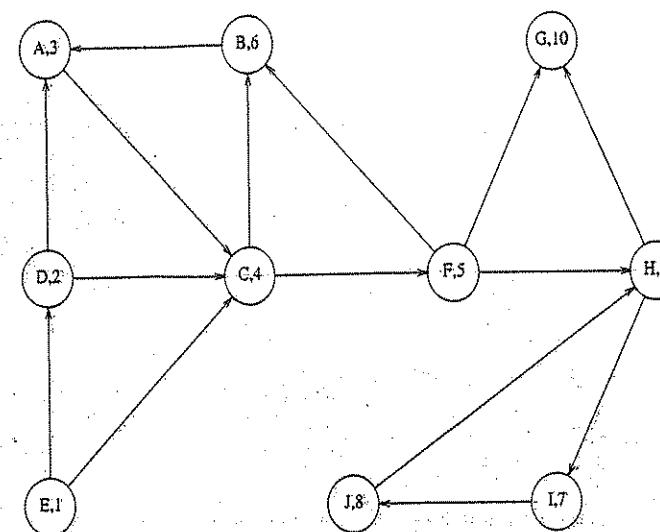
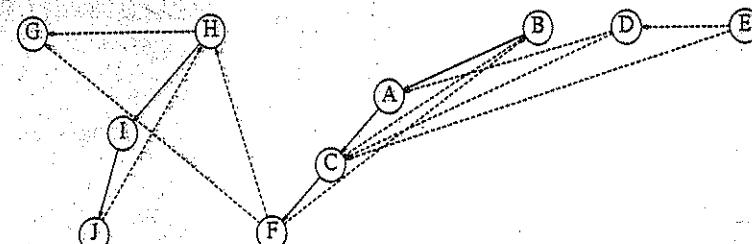


Figura 9.76. G_r numerado por recorrido en orden posterior de G

camino de x a v y de v a x . Aplicando la misma lógica a w se obtendría entonces un camino de x a w y de w a x . Estos caminos implicarían caminos de v a y y de w a v (yendo a través de x).

Puesto que v es un descendiente de x en el árbol de extensión en profundidad de G_r , hay un camino de x a v en G_r , y, por ello, un camino de v a x en G . Además, como x es la raíz, x tiene el mayor número de orden posterior de la primera búsqueda en profundidad. Por lo tanto, durante la primera búsqueda, todo el trabajo de procesar v se terminó antes de que terminara el trabajo en x . Puesto que hay un camino de v a x , se sigue que v debe ser un descendiente de x en el árbol de extensión de G ; si no, v terminaría después de x . Esto implica un camino de x a v en G , con lo que termina la demostración.

Figura 9.77. Búsqueda en profundidad de G_r . Los componentes fuertes son $\{G\}$, $\{H, I, J\}$, $\{B, A, C, F\}$, $\{D\}$ y $\{E\}$.



9.7. Introducción a la compleción NP

En este capítulo, hemos visto soluciones a una amplia variedad de problemas de la teoría de grafos. Todos ellos tienen tiempos de ejecución polinómicos, y con excepción del problema del flujo en una red, el tiempo de ejecución es lineal o sólo ligeramente más que lineal ($O(|A| \log |A|)$). También hemos mencionado, de pasada, que para algunos problemas ciertas variaciones parecen más difíciles que el original.

Recordaremos que el problema del circuito de Euler, el cual encuentra un camino que toca todas las aristas exactamente una vez, se puede resolver en tiempo lineal. El problema del ciclo hamiltoniano busca un ciclo sencillo que contenga todos los vértices. Para este problema no se conoce ningún algoritmo lineal.

El problema del camino más corto no ponderado con origen único para grafos dirigidos también puede resolverse en tiempo lineal. No se conoce ningún algoritmo de tiempo lineal para el problema correspondiente del camino simple más largo.

La situación de estas variaciones del problema es realmente mucho peor de lo que hemos descrito. No sólo no se conocen algoritmos lineales para esas variaciones, sino que tampoco se conocen algoritmos que garanticen una ejecución en tiempo polinómico. Los mejores algoritmos para estos problemas podrían tomar tiempo exponencial en algunas entradas.

En esta sección haremos un rápido examen de este problema. Este tema es más bien complejo, así que sólo lo estudiaremos de una manera rápida e informal. Por ello nuestro análisis puede ser (necesariamente) algo impreciso en ocasiones.

Veremos que hay una variedad de problemas importantes que casi son equivalentes en cuanto a su complejidad. Estos problemas forman una clase llamada problemas *NP completos*. La complejidad exacta de esos problemas *NP* completos todavía tiene que ser determinada y sigue siendo el problema abierto más notable en informática. Hay sólo dos posibilidades: o bien todos estos problemas tienen soluciones en tiempo polinómico o ninguno de ellos la tiene.

9.7.1. Fácil vs. difícil

Al clasificar problemas, el primer paso es examinar las fronteras. Ya hemos visto que muchos problemas pueden resolverse en tiempo lineal. También hemos visto algunos tiempos de ejecución $O(\log n)$, pero ellos suponen algún procesamiento previo (como puede ser que ya se haya leído la entrada o que ya se haya construido la estructura de datos) u ocurren en ejemplos aritméticos. Por ejemplo, el algoritmo *mcd*, cuando se aplica a dos números m y n , tarda un tiempo $O(\log n)$. Puesto que los números constan de $\log m$ y $\log n$ bits, respectivamente, en realidad el algoritmo *mcd* está tomando tiempo lineal sobre la *cantidad* o *tamaño* de la entrada. Así, al medir el tiempo de ejecución, lo consideraremos como una función de la cantidad de entrada. Por lo general, no se puede esperar nada mejor que el tiempo lineal.

En el otro extremo del espectro caen algunos problemas verdaderamente difíciles. Son difíciles porque son *impossibles*. Esto no implica la típica queja exasperada de que se necesitaría un genio para resolver el problema. Así como los números

reales no son suficientes para expresar una solución para $x^2 < 0$, se puede demostrar que los computadores no pueden resolver todos los problemas que vayan apareciendo. Estos problemas "imposibles" se llaman *indecidibles*.

Un problema indecidible particular es el *problema de la parada*. ¿Es posible que nuestro compilador de Pascal tenga una capacidad adicional que no sólo detecte errores de sintaxis sino también ciclos infinitos? Esto parece un problema difícil, pero uno podría esperar que si algunos programadores muy listos le dedicaran el tiempo suficiente a esta tarea podrían producir este avance.

La razón intuitiva de que este problema sea indecidible es que tal programa puede tardar un tiempo excesivo en revisarse a sí mismo. Por esta razón, a estos problemas se les llama en ocasiones *recursivamente indecidibles*.

Si se pudiera escribir un programa para revisión de ciclos infinitos, seguramente se podría usar para examinarse a sí mismo. Entonces podríamos producir un programa llamado *CICLO*, que tome como entrada un programa *P* y ejecute *P* sobre sí mismo. Éste imprime la palabra *Sí* si *P* entra en un ciclo infinito cuando se ejecuta sobre sí mismo. Si *P* termina al ejecutarse sobre sí mismo, una cosa natural a hacer sería visualizar *NO*. En vez de esto, haremos que *CICLO* entre en un ciclo infinito.

¿Qué sucede cuando *CICLO* se proporciona como entrada a sí mismo? O bien *CICLO* se detiene, o no se detiene. El problema es que ambas posibilidades conducen a contradicciones, casi del mismo modo que la paradójica frase: "Este enunciado es falso."

Según nuestra definición, *CICLO(P)* entra en un ciclo infinito si *P(P)* termina. Supongamos que cuando *P = CICLO*, *P(P)* termina. Entonces, de acuerdo con el programa *CICLO*, *CICLO(P)* está obligado a entrar en un ciclo infinito. Así, *CICLO(CICLO)* debería terminar y entrar en un ciclo infinito a la vez, lo cual es claramente imposible. Por el contrario, supongamos que cuando *P = CICLO*, *P(P)* entra en un ciclo infinito. Entonces *CICLO(P)* debe terminar, y se llega al mismo conjunto de contradicciones. Así, no es posible que el programa *CICLO* pueda existir.

9.7.2. La clase NP

Un poco por debajo de los horrores de los problemas indecidibles se encuentra la clase *NP*. *NP* significa *tiempo polinómico no determinista* (por sus siglas en inglés: *non-deterministic polynomial-time*). Una máquina determinista, en cada instante está ejecutando una instrucción. Dependiendo de la instrucción, después pasa a alguna otra instrucción, la cual es única. Una máquina no determinista tiene una variedad de pasos siguientes alternativos. Es libre de escoger el que quiera, y si uno de esos pasos lleva a una solución, siempre elegirá el correcto. Así, una máquina no determinista tiene un poder de predicción extremadamente bueno (óptimo). Es probable que éste parezca un modelo ridículo, ya que no es posible que nadie pueda construir una máquina no determinista, y porque parecería una increíble mejora a nuestro computador estándar (todo problema podría entonces parecer trivial). Veremos que el no determinismo es una construcción teórica muy útil. Además, el no determinismo no es tan potente como uno podría pensar. Por ejemplo, los problemas indecidibles siguen siendo indecidibles, aun permitiendo el no determinismo.

Una forma sencilla de comprobar si un problema está en la clase *NP* es expresar el problema como una pregunta de sí/no. El problema está en *NP* si, en tiempo polinómico, se puede demostrar que cualquier ocurrencia "sí" es correcta. No tenemos que preocuparnos de las ocurrencias "no", ya que el programa siempre hace la elección correcta. Así, para el problema del ciclo hamiltoniano, una ocurrencia "sí" sería cualquier circuito sencillo en el grafo que incluye todos los vértices. Éste está en *NP*, ya que, dado el camino, es sencillo comprobar que realmente es un ciclo hamiltoniano. Preguntas apropiadamente enunciadas, como "¿Hay un camino sencillo de longitud K ? ", también se pueden comprobar fácilmente y están en *NP*. Es trivial evaluar cualquier camino que satisfaga esta propiedad.

La clase *NP* incluye todos los problemas que tienen soluciones en tiempos polinomiales, ya que es obvio que la solución da una prueba. Se podría esperar que, como es mucho más fácil comprobar una respuesta que obtener una desde el principio, habría problemas en *NP* que no tienen soluciones en tiempo polinómico. Hasta la fecha no se ha encontrado tal problema, así que es totalmente posible, aunque los expertos no lo consideran probable, que el no determinismo no sea una mejora tan importante. El problema es que la demostración de cotas inferiores exponenciales es una tarea extremadamente difícil. La técnica de acotado de la teoría de la información, con la que demostramos que la ordenación requiere $\Omega(n \log n)$ comparaciones, no parece adecuada para esta tarea porque los árboles de decisión no son suficientemente grandes.

Hay que señalar, también, que no todos los problemas decidibles están en *NP*. Consideremos el problema de determinar si un grafo *no* tiene un ciclo hamiltoniano. Demostrar que un grafo tenga un ciclo hamiltoniano es un asunto relativamente simple: sólo se necesita exhibir uno. Nadie sabe cómo demostrar, en tiempo polinómico, que un grafo no tiene un ciclo hamiltoniano. Parece que tenemos que enumerar todos los ciclos y revisarlos uno por uno. Así, no se sabe que el problema del ciclo no hamiltoniano esté en *NP*.

9.7.3. Problemas *NP* completos

Entre todos los problemas de los que se sabe que están en *NP*, hay un subconjunto, llamado de los problemas *NP* completos, que contiene los más difíciles. Un problema *NP* completo tiene la propiedad de que cualquier problema en *NP* puede ser reducido polinómicamente a él.

Un problema P_1 puede ser reducido a P_2 como sigue: se hace una correspondencia tal que cualquier ocurrencia de P_1 pueda transformarse en una ocurrencia de P_2 . Se resuelve P_2 y después se hace una correspondencia de la respuesta de nuevo con el original. Como ejemplo, los números entran en una calculadora de bolsillo en decimal. Los números decimales se convierten a binario, y en binario se realizan todos los cálculos. Después la respuesta final se convierte de nuevo a decimal para su visualización. Para que P_1 sea reducible polinómicamente a P_2 , todo el trabajo asociado a las transformaciones debe efectuarse en tiempo polinómico.

La razón de que los problemas *NP* completos sean los problemas *NP* más difíciles es que un problema *NP* completo se puede usar esencialmente como una subrutina de cualquier problema en *NP*, con sólo una cantidad polinómica de trabajo

adicional. Así, si cualquier problema *NP* completo tiene una solución en tiempo polinómico, entonces todo problema en *NP* debe tener una solución en tiempo polinómico. Esto hace de los problemas *NP* completos los más difíciles de todos los *NP*.

Suponga que tenemos un problema *NP* completo P_1 y que sabemos que P_2 está en *NP*. Además supongamos que P_1 se reduce polinómicamente a P_2 , de modo que podemos resolver P_1 usando P_2 con sólo una penalización en tiempo polinómico. Puesto que P_1 es *NP* completo, todo problema en *NP* se reduce polinómicamente a P_1 . Aplicando la propiedad de la cerradura de los polinomios, vemos que todo problema en *NP* es reducible polinómicamente a P_2 ; reducimos el problema a P_1 y después reducimos de P_1 a P_2 . Así, P_2 es *NP* completo.

Por ejemplo, supongamos que ya sabemos que el problema del ciclo hamiltoniano es *NP* completo. El problema del agente viajero es como sigue.

PROBLEMA DEL AGENTE VIAJERO:

Dado un grafo completo $G = (V, A)$, con costos de aristas, y un entero K , ¿hay un ciclo simple que visite todos los vértices y tenga un costo total $\leq K$?

El problema es diferente del ciclo hamiltoniano, porque todas las $|V|(|V| - 1)/2$ aristas están presentes y el grafo es ponderado. Este problema tiene muchas aplicaciones importantes. Por ejemplo, las tarjetas de circuitos impresos necesitan tener agujeros para poder insertar los circuitos integrados, las resistencias y otros componentes electrónicos. Esto se hace por medios mecánicos. La perforación del agujero es una operación rápida; el paso que consume tiempo es el posicionamiento del taladro. El tiempo requerido para ello depende de la distancia recorrida de agujero a agujero. Como quisieramos perforar todos los agujeros (y después regresar al inicio para la siguiente tarjeta) y minimizar el tiempo consumido en el recorrido, lo que tenemos es el problema del agente viajero.

El problema del agente viajero es *NP* completo. Es fácil ver que una solución se puede comprobar en tiempo polinómico, así que ciertamente es *NP*. Para demostrar que es *NP* completo, reducimos polinómicamente el problema del ciclo hamiltoniano a éste. Para hacerlo construimos un grafo nuevo G' . G' tiene los mismos vértices que G . Para G' , cada arista (v, w) tiene un peso de 1 si $(v, w) \in G$, y 2 en cualquier otro caso. Escogemos que $K = |V|$. Véase la figura 9.78.

Es fácil verificar que G tiene un ciclo hamiltoniano si y sólo si G' tiene un recorrido del agente viajero de peso total $|V|$.

Ahora hay una larga lista de problemas que sabemos que son *NP* completos. Para demostrar que algún problema nuevo es *NP* completo, tenemos que demostrar que está en *NP*, y después se debe transformar un problema *NP* completo adecuado en éste. Aunque la transformación a un problema del agente viajero fue más bien directa, de hecho la mayoría de las transformaciones son muy complicadas y requieren algunas construcciones intrincadas. En general, se consideran varios problemas *NP* completos diferentes antes del problema que realmente da la reducción. Como sólo nos interesan las ideas generales, no mostraremos más transformaciones; el lector interesado puede consultar las referencias.

El lector alerta puede preguntarse cómo es que realmente se demostró que el primer problema *NP* completo lo era. Puesto que demostrar que un problema es *NP*

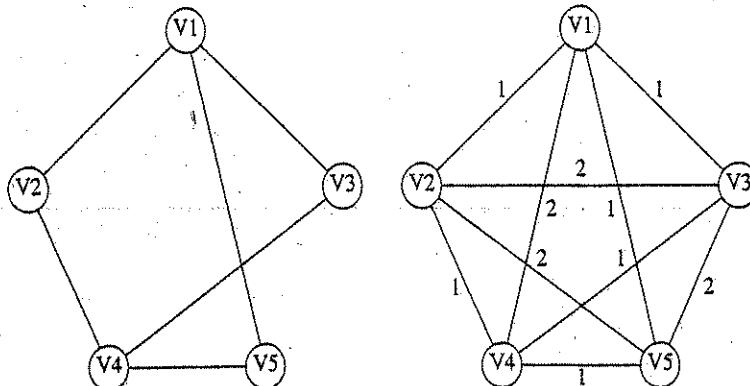


Figura 9.78 Transformación de un problema de ciclo hamiltoniano en un problema del agente viajero

completo requiere la transformación de éste en otro *NP* completo, debe haber algún problema *NP* completo para el cual esta estrategia no funcionará. El primer problema que se demostró que era *NP* completo fue el de la *satisfacibilidad*. El problema de la satisfacibilidad toma como entrada una expresión booleana y pregunta si la expresión tiene una asignación a las variables que dan un valor de 1.

La satisfacibilidad es ciertamente *NP*, ya que es fácil evaluar una expresión booleana y comprobar si el resultado es 1. En 1971 Cook demostró que la satisfacibilidad era *NP* completa demostrando directamente que todos los problemas que están en *NP* se pueden transformar a la satisfacibilidad. Para ello se valió del hecho conocido acerca de todo problema en *NP*: todo problema en *NP* puede resolverse en tiempo polinómico por medio de un computador no determinista. El modelo formal de un computador se denomina *máquina de Turing*. Cook mostró cómo se pueden simular las acciones de esta máquina con una fórmula booleana, extremadamente larga y compleja, pero polinómica al fin y al cabo. Esta fórmula booleana sería verdadera si y sólo si el programa que ejecutara la máquina de Turing produjera una respuesta "sí" a su entrada.

Una vez demostrado que la satisfacibilidad es *NP* completa, se comprobó que un grupo de nuevos problemas *NP* completos, incluyendo los más clásicos, eran *NP* completos.

Además de la satisfacibilidad, los problemas del circuito hamiltoniano, del agente viajero y del camino más largo, que ya hemos examinado, algunos de los problemas *NP* completos más conocidos que no hemos estudiado son el del *empaquetamiento en recipientes*, el de la *mochila*, el de la *coloración de grafos* y el de las *cuadrillas*. La lista es bastante amplia e incluye problemas de sistemas operativos (planificación y seguridad), sistemas de bases de datos, investigación operativa, lógica y especialmente teoría de grafos.

Resumen

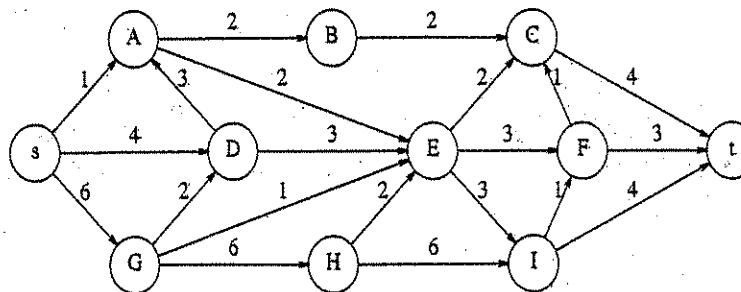
En este capítulo hemos visto cómo usar los grafos para modelar un gran número de problemas de la vida real. Muchos de los grafos que se presentan son por lo regular dispersos, así que es importante poner atención a las estructuras de datos con que se implantan.

También hemos visto una clase de problemas que no parecen tener soluciones eficientes. En el capítulo 10, estudiaremos algunas técnicas para tratar esos problemas.

Ejercicios

- 9.1 Encuentre una ordenación topológica para el grafo de la figura 9.79.
- 9.2 Si se usa una pila en vez de una cola para el algoritmo de la ordenación topológica de la sección 9.1, ¿se obtiene una ordenación topológica diferente? ¿Por qué una estructura de datos puede dar una respuesta "mejor"?
- 9.3 Escriba un programa para efectuar una ordenación topológica sobre un grafo.
- 9.4 Una matriz de adyacencia requiere $O(|V|^2)$ sólo para la asignación de valores iniciales usando un ciclo doble estándar. Proponga un método que almacene un grafo en una matriz de adyacencia (tal que la prueba de la existencia de una arista sea $O(1)$), pero que evite el tiempo de ejecución cuadrático.
- 9.5 a. Encuentre el camino más corto desde A a todos los demás vértices del grafo de la figura 9.80.
b. Encuentre el camino más corto no ponderado desde B a todos los demás vértices del grafo de la figura 9.80.
- 9.6 ¿Cuál es el tiempo de ejecución para el peor caso del algoritmo de Dijkstra cuando se implanta con montículos *d* (sección 6.5)?

Figura 9.79



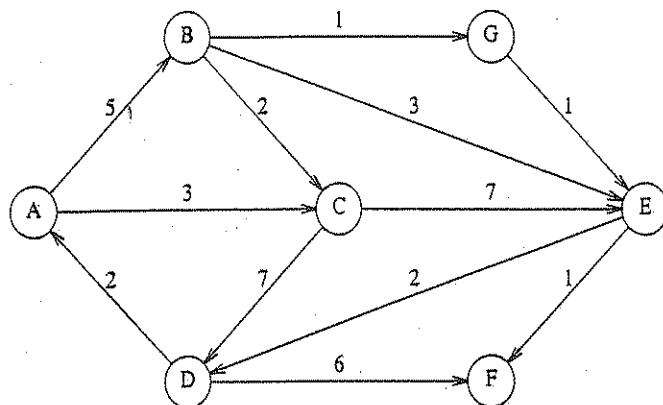


Figura 9.80

- 9.7 a. Proporcione un ejemplo donde el algoritmo de Dijkstra dé la respuesta incorrecta en presencia de una arista negativa pero no de un ciclo de costo negativo.
- **b. Demuestre que el algoritmo del camino más corto ponderado que sugerimos en la sección 9.3.3 funciona si hay aristas de costo negativo, pero no ciclos de costo negativo, y que el tiempo de ejecución de este algoritmo es $O(|A||V|)$.
- 9.8 Suponga que todos los pesos de las aristas de un grafo son enteros entre 1 y $|A|$. ¿Cuán rápido se puede implantar el algoritmo de Dijkstra?
- 9.9 Escriba un programa para resolver el problema del camino más corto con origen único.
- 9.10 a. Explique cómo modificar el algoritmo de Dijkstra para contar el número de diferentes caminos mínimos de v a w .
- b. Explique cómo modificar el algoritmo de Dijkstra de modo tal que si hay más de un camino mínimo de v a w , se escoga el camino con el menor número de aristas.
- 9.11 Encuentre el flujo máximo en la red de la figura 9.79.
- 9.12 Suponga que $G = (V, A)$ es un árbol, s es la raíz y que agregamos un vértice t y aristas de capacidad infinita desde todas las hojas en G a t . Proporcione un algoritmo en tiempo lineal para encontrar el flujo máximo de s a t .
- 9.13 Un grafo bipartido, $G = (V, A)$, es un grafo tal que V se puede partitionar en dos subconjuntos V_1 y V_2 y ninguna arista tiene ambos vértices en el mismo subconjunto.
- a. Proporcione un algoritmo en tiempo lineal para determinar si un grafo es bipartido.

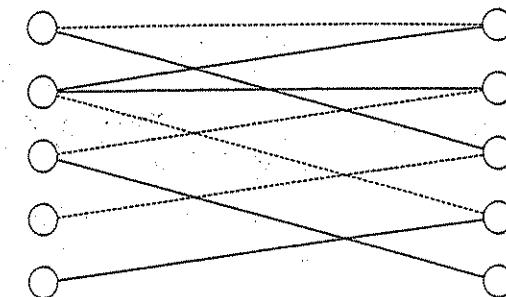


Figura 9.81 Grafo bipartido

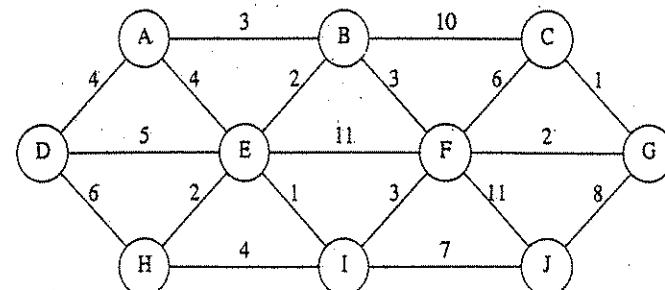
- b. El problema de la correspondencia bipartida consiste en encontrar el mayor subconjunto A' de A tal que ningún vértice esté incluido en más de una arista. Una correspondencia de cuatro aristas (indicada por medio de aristas punteadas) se muestra en la figura 9.81. Hay una correspondencia de cinco aristas, la cual es máxima.

Muestre cómo usar el problema de la correspondencia bipartida para resolver el siguiente problema. Tenemos un conjunto de instructores, un conjunto de cursos y una lista de cursos que cada instructor puede impartir. Si ningún instructor debe impartir más de un curso, y sólo un instructor puede impartir un curso dado, ¿cuál es el número máximo de cursos que se pueden ofrecer?

- c. Demuestre que el problema del flujo en la red puede servir para resolver el problema de la correspondencia bipartida.
- d. ¿Cuál es la complejidad en tiempo de la solución de la parte (b)?

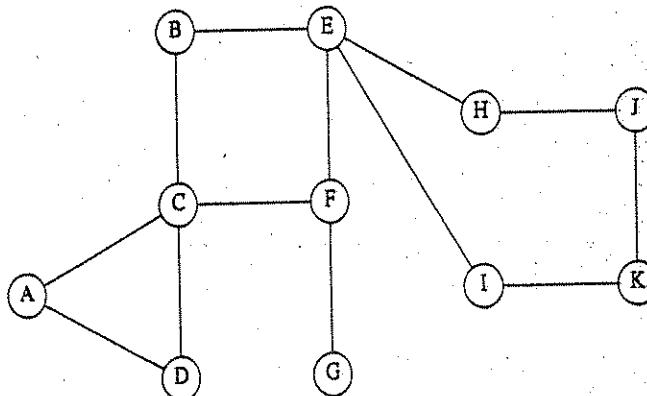
- 9.14 Proporcione un algoritmo para encontrar un camino creciente que permita el flujo máximo.

Figura 9.82



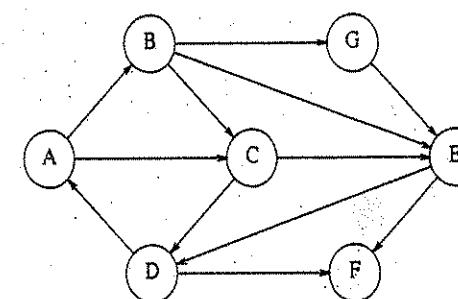
- 9.15 a. Encuentre un árbol de extensión mínimo para el grafo de la figura 9.82 usando tanto el algoritmo de Prim como el de Kruskal.
 b. ¿Es único este árbol de extensión mínimo? ¿Por qué?
- 9.16 ¿Funciona el algoritmo de Prim o el de Kruskal si hay pesos de arista negativos?
- 9.17 Demuestre que un grafo de V vértices puede tener V^{V-2} árboles de extensiones mínimos.
- 9.18 Escriba un programa para implantar el algoritmo de Kruskal.
- 9.19 Si todas las aristas de un grafo tienen pesos entre 1 y $|A|$, ¿con qué rapidez se puede obtener el árbol de extensión mínimo?
- 9.20 Proporcione un algoritmo para encontrar un árbol de extensión *máximo*. ¿Es más difícil que encontrar un árbol de extensión mínimo?
- 9.21 Determine todos los puntos de articulación del grafo de la figura 9.83. Muestre el árbol de extensión en profundidad y los valores de *núm* e *inferior* para cada vértice.
- 9.22 Compruebe que el algoritmo para encontrar los puntos de articulación funciona.
- 9.23 a. Proporcione un algoritmo para encontrar el número mínimo de aristas que hay que eliminar de un grafo no dirigido para que el grafo resultante sea acíclico.
 *b. Demuestre que este problema es *NP* completo para grafos dirigidos.
- 9.24 Demuestre que en un bosque de extensión en profundidad de un grafo dirigido, todas las aristas cruzadas van de derecha a izquierda.
- 9.25 Cree un algoritmo para decidir si una arista (v, w) en un bosque de extensión en profundidad de un grafo dirigido es una arista de árbol, de regreso, cruzada o directa.

Figura 9.83



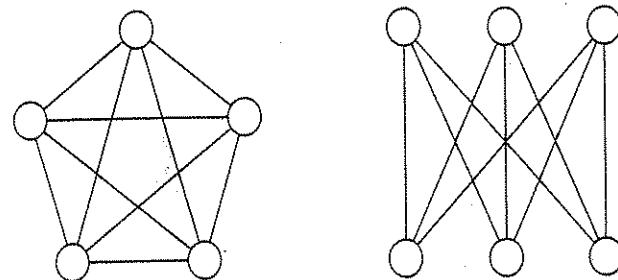
- 9.26 Encuentre los componentes fuertemente conexos del grafo de la figura 9.84.
- 9.27 Escriba un programa para encontrar los componentes fuertemente conexos de un digrafo.
- *9.28 Proporcione un algoritmo que encuentre los componentes fuertemente conexos en sólo una búsqueda en profundidad. Use un algoritmo semejante al algoritmo de la biconectividad.
- 9.29 Los *componentes biconexos* de un grafo G son una partición de las aristas en conjuntos tales que el grafo formado por cada conjunto de aristas es biconexo. Modifique el algoritmo de la figura 9.67 para encontrar los componentes biconexos en vez de los puntos de articulación.
- 9.30 Suponga que efectuamos una búsqueda en amplitud de un grafo no dirigido y construimos un árbol de extensión en amplitud. Demuestre que todas las aristas del árbol son de árbol o cruzadas.
- 9.31 Presente un algoritmo para encontrar en un grafo no dirigido (conexo) un camino que vaya a través de todas las aristas exactamente una vez en cada dirección.
- 9.32 a. Escriba un programa para encontrar un circuito de Euler en un grafo si es que existe.
 b. Escriba un programa para encontrar un viaje de Euler en un grafo si es que existe.
- 9.33 Un circuito de Euler en un grafo dirigido es un ciclo en el cual toda arista es visitada exactamente una vez.
- *a. Demuestre que un grafo dirigido tiene un circuito de Euler si y sólo si es fuertemente conexo y todo vértice tiene iguales sus grados de entrada y de salida.
 *b. Proporcione un algoritmo en tiempo lineal para encontrar un circuito de Euler en un grafo dirigido, cuando existe.

Figura 9.84



- 9.34 a. Considere la siguiente solución al problema del circuito de Euler. Suponga que el grafo es biconexo. Efectúe una búsqueda en profundidad, tomando aristas de regreso sólo como último recurso. Si el grafo no es biconexo, aplique recursivamente el algoritmo sobre los componentes biconexos. ¿Funciona este algoritmo?
- b. Suponga que al tomar aristas de regreso, tomamos la arista de regreso al ancestro más cercano. ¿Funciona el algoritmo?
- 9.35 Un grafo plano es un grafo que se puede dibujar sin que se corte ningún par de aristas.
- *a. Demuestre que ninguno de los grafos de la figura 9.85 es plano.
 - **b. Demuestre que un grafo es plano si y sólo si no contiene alguno de los grafos de la parte (a) como subgrafos.
 - **c. Demuestre que en un grafo plano, $|A| \leq 3|V| - 6$.
- 9.36 Un *multigrafo* es un grafo en el cual se permiten aristas múltiples entre pares de vértices. ¿Cuáles de los algoritmos del capítulo funcionan sin modificaciones para los multigráficos? ¿Qué modificaciones deben hacerse para los demás?
- *9.37 Sea $G = (V, A)$ un grafo no dirigido. Use la búsqueda en profundidad para diseñar un algoritmo lineal para convertir cada arista de G en una arista dirigida tal que el grafo resultante sea fuertemente conexo, o determine que esto no es posible.
- 9.38 Sea un conjunto de n palos que reposan uno sobre otro en alguna configuración. Cada palo está especificado por sus dos extremos; cada extremo es una triplete ordenada que da sus coordenadas x , y y z ; ningún palo está vertical. Se puede tomar un palo sólo si ningún otro está sobre él.
- a. Explique cómo escribir una rutina que tome dos palos a y b e informe si a está arriba, abajo o no guarda ninguna relación con b . (Esto no tiene nada que ver con la teoría de grafos.)
 - b. Proporcione un algoritmo que determine si es posible tomar todos los palos, y si es así, que dé una secuencia de selecciones que lo haga.

Figura 9.85



- 9.39 El problema de las *cuadrillas* se puede enunciar como sigue: dado un grafo no dirigido $G = (V, A)$ y un entero K , ¿contiene G un subgrafo completo de al menos K vértices?

El problema de la *cobertura de vértices* se puede enunciar como sigue: dado un grafo no dirigido $G = (V, A)$ y un entero K , ¿contiene G un subconjunto $V' \subset V$ tal que $|V'| \leq K$ y toda arista de G tenga un vértice en V' ? Demuestre que el problema de las cuadrillas es polinómicamente reducible a la cobertura de vértices.

- 9.40 Suponga que el problema del ciclo hamiltoniano es *NP* completo para grafos no dirigidos.

- a. Demuestre que el problema del ciclo hamiltoniano es *NP* completo para grafos dirigidos.
- b. Demuestre que el problema del camino simple más largo no ponderado es *NP* completo para grafos dirigidos.

- 9.41 El problema del *coleccionista de tarjetas de béisbol* es como sigue: dados los paquetes P_1, P_2, \dots, P_m , cada uno de los cuales contiene un subconjunto de las tarjetas de béisbol del año, y un entero K , ¿es posible colectar todas las tarjetas de béisbol escogiendo un número de paquetes $\leq K$? Demuestre que el problema del colecciónista de tarjetas de béisbol es *NP* completo.

Referencias

Entre los buenos textos de teoría de grafos se incluyen [7], [12], [21] y [34]. En [36], [38] y [45] se cubren temas más avanzados, prestándose ahí mayor atención a los tiempos de ejecución.

El uso de las listas de adyacencia fue presentado en [23]. El algoritmo de la ordenación topológica proviene de [28], como se describió en [31]. El algoritmo de Dijkstra apareció en [8]. Las mejoras con montículos-d y montículos de Fibonacci se describieron en [27] y [14], respectivamente. El algoritmo del camino más corto con pesos negativos en las aristas se debe a Bellman [3]; Tarjan [45] describe una forma más eficiente de garantizar la terminación.

En [13] Ford y Fulkerson proporcionaron un trabajo original y fecundo sobre flujos en redes. La idea del aumento a lo largo de los caminos más cortos o en los caminos que admiten el mayor incremento del flujo proviene de [11]. Otros tratamientos del problema se pueden encontrar en [9], [30] y [20]. Un algoritmo para el problema del flujo de costo mínimo aparece en [18].

Un algoritmo antiguo para los árboles de extensión mínimos se puede encontrar en [4]. El algoritmo de Prim es de [39]; el algoritmo de Kruskal aparece en [32]. Dos algoritmos $O(|A| \log \log |V|)$ son [5] y [46]. Los algoritmos teóricamente mejor conocidos aparecen en [14] y [16]. Un estudio empírico de esos algoritmos indica que el algoritmo de Prim, implantado con *decrementar_llave*, es mejor en la práctica sobre la mayoría de los grafos [37].

El algoritmo para la biconectividad proviene de [41]. El primer algoritmo en tiempo lineal para componentes fuertes (ejercicio 9.28) aparece en el mismo artículo.

El algoritmo presentado en el texto se debe a Kosaraju (inédito) y Sharir [40]. Otras aplicaciones de búsqueda en profundidad aparecen en [24], [25], [42] y [43] (como se mencionó en el capítulo 8, los resultados de [42] y [43] se han mejorado, pero el algoritmo básico no ha cambiado).

La obra clásica de la teoría de referencia sobre los problemas NP completos es [19]. Se puede encontrar material adicional en [1]. La complejidad NP de la satisfactoriedad se demuestra en [6]. El otro artículo precursor es [29], el cual demostró la complejidad NP de 21 problemas. Un excelente estudio sobre la teoría de la complejidad es [44]. Un algoritmo de aproximación para el problema del agente viajero, que por lo regular da resultados casi óptimos, se puede encontrar en [35].

Una solución al ejercicio 9.8 aparece en [2]. En [22] y [33] se encuentran soluciones al problema de la correspondencia bipartida del ejercicio 9.13. El problema puede generalizarse agregando pesos a las aristas y eliminando la restricción de que el grafo es bipartido. Las soluciones eficientes para el problema de la correspondencia no ponderada para grafos generales son muy complejas. Los detalles se encuentran en [10], [15] y [17].

El ejercicio 9.35 se ocupa de los grafos planos, los cuales surgen comúnmente en la práctica. Los grafos planos son muy dispersos, y muchos problemas difíciles son más fáciles en grafos planos. Un ejemplo es el problema del isomorfismo de grafos, que se puede resolver en tiempo lineal para grafos planos [26]. No se conoce ningún algoritmo en tiempo polinómico para grafos generales.

1. A. V. Aho, J. E. Hopcroft, y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
2. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, y R. E. Tarjan, "Faster Algorithms for the Shortest Path Problem", *Journal of the ACM*, 37 (1990), págs. 213–223.
3. R. E. Bellman, "On a Routing Problem", *Quarterly of Applied Mathematics*, 16 (1958), págs. 87–90.
4. O. Boruvka, "Ojistém problému minimálním (On a Minimal Problem)", *Práca Moravské Přírodovedecké Společnosti*, 3 (1926), págs. 37–58.
5. D. Cherdon y R. E. Tarjan, "Finding Minimum Spanning Trees", *SIAM Journal on Computing*, 5 (1976), págs. 724–742.
6. S. Cook, "The Complexity of Theorem Proving Procedures", *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971), págs. 151–158.
7. N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, 1974.
8. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik*, 1 (1959), págs. 269–271.
9. E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation", *Soviet Mathematics Doklady*, 11 (1970) págs. 1277–1280.
10. J. Edmonds, "Paths, Trees, and Flowers", *Canadian Journal of Mathematics*, 17 (1965), págs. 449–467.
11. J. Edmonds y R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of the ACM*, 19 (1972), págs. 248–264.
12. S. Even, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.
13. L. R. Ford, Jr. y D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.
14. M. L. Fredman y R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms", *Journal of the ACM*, 34 (1987), págs. 596–615.

15. H. N. Gabow, "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking", *Proceedings of First Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), págs. 434–443.
16. H. N. Gabow, Z. Galil, T. H. Spencer, y R. E. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees on Directed and Undirected Graphs", *Combinatorica*, 6 (1986), págs. 109–122.
17. Z. Galil, "Efficient Algorithms for Finding Maximum Matchings in Graphs", *ACM Computing Survey*, 18 (1986) págs. 23–38.
18. Z. Galil y E. Tardos, "An $O(n^2(m+n \log n) \log n)$ Min-Cost Flow Algorithm", *Journal of the ACM*, 35 (1988), págs. 374–386.
19. M. R. Garey y D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.
20. A. V. Goldberg y R. E. Tarjan, "A New Approach to the Maximum-Flow Problem", *Journal of the ACM*, 35 (1988), págs. 921–940.
21. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
22. J. E. Hopcroft y R. M. Karp, "An $n^{3/2}$ Algorithm for Maximum Matchings in Bipartite Graphs", *SIAM Journal on Computing*, 2 (1973), págs. 225–231.
23. J. E. Hopcroft y R. E. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation", *Communications of the ACM*, 16 (1973), págs. 372–378.
24. J. E. Hopcroft y R. E. Tarjan, "Dividing a Graph into Triconnected Components", *SIAM Journal on Computing*, 2 (1973), págs. 135–158.
25. J. E. Hopcroft y R. E. Tarjan, "Efficient Planarity Testing", *Journal of the ACM*, 21 (1974), págs. 549–568.
26. J. E. Hopcroft y J. K. Wong, "Linear Time Algorithm for Isomorphism of Planar Graphs", *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing* (1974), págs. 172–184.
27. D. B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks", *Journal of the ACM*, 24 (1977), págs. 1–13.
28. A. B. Kahn, "Topological Sorting of Large Networks", *Communications of the ACM*, 5 (1962), págs. 558–562.
29. R. M. Karp, "Reducibility among Combinatorial Problems", *Complexity of Computer Computations* (eds. R. E. Miller y J. W. Thatcher), Plenum Press, Nueva York, 1972, págs. 85–103.
30. A. V. Karazanov, "Determining the Maximal Flow in a Network by the Method of Pre-flows", *Soviet Mathematics Doklady*, 15 (1974), págs. 434–437.
31. D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, 2a. ed., Addison-Wesley, Reading, MA, 1973.
32. 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), págs. 48–50.
33. H. W. Kuhn, "The Hungarian Method for the Assignment Problem", *Naval Research Logistics Quarterly*, 2 (1955), págs. 83–97.
34. E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Reinhart, and Winston, Nueva York, NY, 1976.
35. S. Lin y B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem", *Operations Research*, 21 (1973), págs. 498–516.
36. K. Mehlhorn, *Data Structures and Algorithms 2 Graph Algorithms and NP-completeness*, Springer-Verlag, Berlín, 1984.
37. 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), págs. 400–411.

38. C. H. Papadimitriou y K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, NJ, 1982.
39. R. C. Prim, "Shortest Connection Networks and Some Generalizations", *Bell System Technical Journal*, 36 (1957), págs. 1389-1401.
40. M. Sharir, "A Strong-Connectivity Algorithm and Its Application in Data Flow Analysis", *Computers and Mathematics with Applications*, 7 (1981), págs. 67-72.
41. R. E. Tarjan, "Depth First Search and Linear Graph Algorithms", *SIAM Journal on Computing*, 1 (1972), págs. 146-160.
42. R. E. Tarjan, "Testing Flow Graph Reducibility", *Journal of Computer and System Sciences*, 9 (1974), págs. 355-365.
43. R. E. Tarjan, "Finding Dominators in Directed Graphs", *SIAM Journal on Computing*, 3 (1974), págs. 62-89.
44. R. E. Tarjan, "Complexity of Combinatorial Algorithms", *SIAM Review*, 20 (1978), págs. 457-491.
45. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
46. A. C. Yao, "An $O(|E| \log \log |V|)$ Algorithm for Finding Minimum Spanning Trees", *Information Processing Letters*, 4 (1975), págs. 21-23.

Técnicas de diseño de algoritmos

Hasta ahora nos hemos ocupado de la implantación eficiente de algoritmos. Hemos visto que al dar un algoritmo no necesitamos especificar las estructuras de datos reales. Al programador toca escoger la estructura de datos adecuada para reducir el tiempo de ejecución tanto como sea posible.

En este capítulo cambiaremos nuestra atención de la *implantación* de algoritmos al *diseño* de algoritmos. La mayoría de los algoritmos vistos hasta ahora son directos y sencillos. El capítulo 9 contiene algunos algoritmos que son mucho más sutiles, y algunos requieren una argumentación (en algunos casos larga) para demostrar que son correctos. En este capítulo nos concentraremos en cinco de los tipos comunes de algoritmos de resolución de problemas. Para muchos problemas, es muy probable que al menos funcione uno de estos métodos. En lo específico, para cada tipo de algoritmo haremos lo siguiente:

- Veremos el enfoque general.
- Estudiaremos varios ejemplos (los ejercicios al final del capítulo darán muchos ejemplos más).
- Analizaremos, en términos generales, la complejidad en tiempo y espacio, cuando sea menester.

10.1. Algoritmos ávidos

El primer tipo de algoritmo que examinaremos es el *algoritmo ávido (greedy algorithm)*. Ya hemos visto tres algoritmos ávidos en el capítulo 9: los de Dijkstra, Prim y Kruskal. Los algoritmos ávidos funcionan en fases. En cada fase, se toma una decisión que parece buena, sin considerar las consecuencias futuras. En general, esto significa que se escoge algún *óptimo local*. Esta estrategia de "tomar lo que se pueda

"ahora" es de donde proviene el nombre de esta clase de algoritmos. Cuando el algoritmo termina, esperamos que el óptimo local sea igual al *óptimo global*. Si éste es el caso, el algoritmo es correcto; si no, el algoritmo ha producido una solución subóptima. Si no se requiere la mejor respuesta absoluta, se usan a veces algoritmos ávidos simples para generar respuestas aproximadas, en vez de emplear los algoritmos complejos que suelen requerirse para generar una respuesta exacta.

Hay varios ejemplos reales de algoritmos ávidos. El más obvio es el problema del cambio de monedas. Por ejemplo, para cambiar moneda estadounidense, *repetidamente* dispensamos la de mayor valor. Así, para dar 17 dólares y 61 centavos en cambio, damos un billete de diez dólares, uno de cinco, dos de un dólar, dos monedas de 25 centavos, una de 10 y una moneda de un centavo. Al hacer esto, estamos seguros de que el número de billetes y monedas es mínimo. Este algoritmo no funciona en todos los sistemas monetarios pero, por fortuna, podemos probar que funciona en el sistema monetario de Estados Unidos. En efecto, funcionaría incluso si hubiera billetes de dos dólares y monedas de cincuenta centavos.

Los problemas de tráfico dan un ejemplo en el que las elecciones óptimas locales no siempre funcionan. Por ejemplo, a ciertas horas punta en Miami es mejor no circular en las calles principales aunque parezcan vacías porque el tráfico se saturará una milla más adelante. Lo que es aún más molesto, en algunos casos es mejor hacer una desviación temporal en la dirección opuesta a nuestro destino a fin de evitar los cuellos de botella del tráfico.

En lo que resta de esta sección, veremos varias aplicaciones que se valen de algoritmos ávidos. La primera es un problema simple de planificación. Virtualmente todos los problemas de este tipo son *NP* completos (o de una complejidad semejante) o se pueden resolver con un algoritmo ávido. La segunda aplicación se ocupa de la compresión de archivos y es uno de los más antiguos resultados de la informática. Para concluir, veremos un ejemplo de un algoritmo ávido de aproximación.

10.1.1. Un problema simple de planificación

Desde los trabajos j_1, j_2, \dots, j_n , con tiempos de ejecución conocidos t_1, t_2, \dots, t_n , respectivamente, y un solo procesador. ¿Cuál es la mejor forma de planificar esos trabajos a fin de minimizar el tiempo medio de terminación? En toda esta sección, supondremos la planificación *no priorizante*: una vez iniciado un trabajo, se debe ejecutar hasta terminar.

Por ejemplo, supongamos que se tienen los cuatro trabajos y tiempos de ejecución asociados de la figura 10.1. Una planificación posible se muestra en la figura 10.2. Debido a que j_1 termina en 15 (unidades de tiempo), j_2 en 23, j_3 en 26 y j_4 en 36, el tiempo medio de terminación es 25. Una planificación mejor, que produce un tiempo de terminación medio de 17.75, se muestra en la figura 10.3.

La planificación dada en la figura 10.3 se ajusta de acuerdo con el trabajo más pequeño primero. Podemos demostrar que esto siempre da una planificación óptima. Sean los trabajos en la planificación j_1, j_2, \dots, j_n . El primer trabajo termina en un tiempo t_1 . El segundo termina después de $t_1 + t_2$, y el tercero termina después de $t_1 + t_2 + t_3$. A partir de esto, se ve que el costo total C de la planificación es:

$$C = \sum_{k=1}^n (n - k + 1)t_{i_k} \quad (10.1)$$

$$C = (n + 1) \sum_{k=1}^n t_{i_k} - \sum_{k=1}^n k \cdot t_{i_k} \quad (10.2)$$

Trabajo	Tiempo
j_1	15
j_2	8
j_3	3
j_4	10

Figura 10.1 Trabajos y tiempos

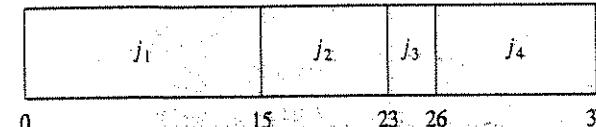


Figura 10.2 Planificación #1

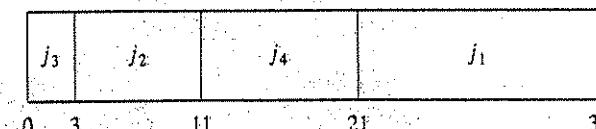


Figura 10.3 Planificación #2 (óptima)

Cabe indicar que en la ecuación (10.2), la primera suma es independiente del orden de los trabajos, así que sólo la segunda suma afecta el costo total. Supongamos que en una ordenación existe alguna $x > y$ tal que $t_x < t_y$. Entonces un cálculo muestra que intercambiando j_x y j_y , la segunda suma se incrementa, disminuyendo el costo total. Así, cualquier planificación de trabajos en la cual los tiempos son (no monótonamente) no crecientes debe ser subóptima. La única planificación que queda es aquélla en la cual los trabajos se ordenan según el que requiera menor tiempo de ejecución, haciendo un desempate arbitrario.

Este resultado indica la razón de por qué el planificador de procesos del sistema operativo por lo general le da precedencia a los trabajos más pequeños.

El caso del multiprocesador

Es posible extender este problema al caso de varios procesadores. De nuevo tenemos los trabajos j_1, j_2, \dots, j_n , con tiempos de ejecución asociados t_1, t_2, \dots, t_n , y un número

P de procesadores. Sin pérdida de generalidad, supondremos que los trabajos están ordenados, empezando por los que requieran el menor tiempo de ejecución. Como ejemplo, supongamos que $P = 3$ y los trabajos son como se muestra en la figura 10.4.

La figura 10.5 exhibe un arreglo óptimo para minimizar el tiempo de terminación promedio. Los trabajos j_1, j_4 y j_7 se ejecutan en el procesador 1. El procesador 2 maneja j_2, j_5 y j_8 , y el procesador 3 ejecuta los trabajos restantes. El tiempo total para la terminación es 40, para un promedio de $\frac{165}{9} = 18.33$.

El algoritmo para resolver el caso del multiprocesador es empezar con los trabajos en orden, iterando a través de los procesadores. No es difícil demostrar que ninguna otra ordenación puede ser mejor, aunque si el número de procesadores P divide uniformemente el número de trabajos n , hay muchas ordenaciones óptimas. Esto se obtiene, para cada $0 \leq i < n/P$, colocando cada uno de los trabajos de j_{iP+1} hasta $j_{(i+1)P}$ en un procesador diferente. En este caso, la figura 10.6 muestra una segunda solución óptima.

Trabajo	Tiempo
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

Figura 10.4 Trabajos y tiempos

Figura 10.5 Una solución óptima para el caso del multiprocesador

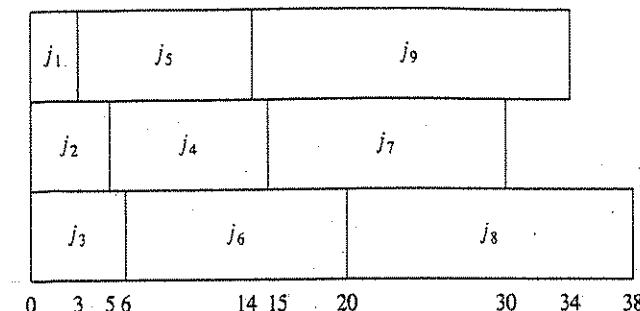
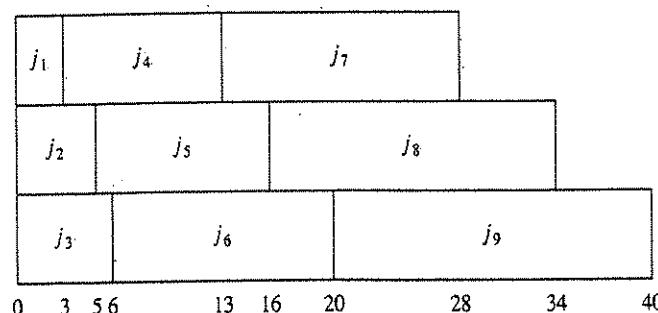


Figura 10.6 Una segunda solución óptima para el caso del multiprocesador

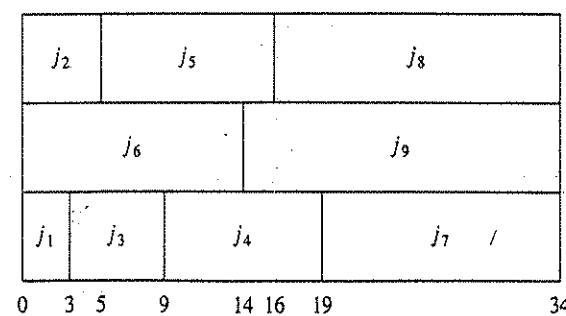
Aun si P no divide exactamente a n , pueden persistir muchas soluciones óptimas, incluso si todos los tiempos de los trabajos son diferentes. Dejamos como ejercicio una mayor investigación de esto.

Minimización del tiempo de terminación final

Cerramos esta sección considerando un problema muy similar. Supongamos que sólo nos interesa el momento en que termina el último trabajo. En los dos ejemplos anteriores, esos tiempos de terminación son 40 y 38, respectivamente. La figura 10.7 muestra que el tiempo mínimo de terminación final es 34, y claramente no puede mejorarse porque todo procesador siempre está ocupado.

Aunque esta planificación no tiene tiempo de terminación medio mínimo, tiene el mérito de que el tiempo de terminación de la secuencia completa es más temprano. Si el mismo usuario posee todos los trabajos, éste es el método preferible. Aunque esos problemas son muy semejantes, este nuevo problema parece NP completo; es sólo otra forma de expresar el problema de la mochila o del empaque-

Figura 10.7 Minimización del tiempo de terminación final



tamiento binario, los cuales estudiaremos más adelante en esta sección. Así, al parecer, la minimización del tiempo de terminación final es mucho más difícil que minimizar el tiempo de terminación medio.

10.1.2. Códigos de Huffman

En esta sección consideraremos una segunda aplicación de los algoritmos ávidos, conocida como *compresión de archivos*.

El conjunto normal de caracteres ASCII consta de unos 100 caracteres "visualizables". Para diferenciarlos se requieren $\lceil \log 100 \rceil = 7$ bits. Siete bits permiten la representación de 128 caracteres, así que el conjunto de caracteres ASCII agrega algunos otros caracteres "no visualizables". Un octavo bit se añade como comprobación de paridad. El punto importante, no obstante, es que si el tamaño del conjunto de caracteres es C , entonces se necesitan $\lceil \log C \rceil$ bits en una codificación estándar.

Supongamos que se tiene un archivo que contiene sólo los caracteres *a*, *e*, *i*, *s*, *t*, más espacios en blanco y *cambios de línea*. Supongamos, además, que el archivo tiene diez *a*, quince *e*, doce *i*, tres *s*, cuatro *t*, trece blancos y un *cambio de línea*. Como lo muestra la tabla de la figura 10.8, este archivo requiere 174 bits para representarse, ya que son 58 caracteres y cada carácter requiere tres bits.

En la realidad, los archivos pueden ser bastante grandes. Muchos de esos archivos tan grandes son salidas de algún programa, y suele haber una gran disparidad entre los caracteres más y menos frecuentes. Por ejemplo, muchos archivos de datos grandes tienen una gran cantidad de dígitos, blancos y *cambios de línea*, pero pocas *q* y *x*. Tal vez nos interese reducir el tamaño del archivo en el caso de transmitirlo por una línea telefónica lenta. También, como prácticamente en toda máquina el espacio en disco es precioso, uno se podría preguntar si es posible dar un código mejor y reducir el número total de bits requeridos.

La respuesta es que sí es posible, y una estrategia simple alcanza ahorros del 25% en archivos grandes regulares y hasta del 50 y 60% en muchos archivos de datos grandes. La estrategia general es permitir que la longitud del código varíe de

Carácter	Código	Frecuencia	Bits totales
<i>a</i>	000	10	30
<i>e</i>	001	15	45
<i>i</i>	010	12	36
<i>s</i>	011	3	9
<i>t</i>	100	4	12
espacio	101	13	39
cambio de línea	110	1	3
Total			174

Figura 10.8 Uso de un esquema de codificación estándar

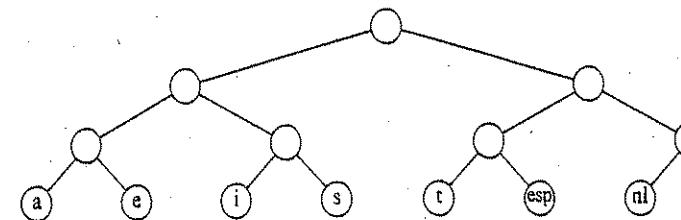


Figura 10.9 Representación del código original en un árbol

carácter a carácter y asegurar que los caracteres que ocurren con frecuencia tengan códigos cortos. Cabe mencionar que si todos los caracteres se presentan con la misma frecuencia, no habrá probabilidades de ningún ahorro.

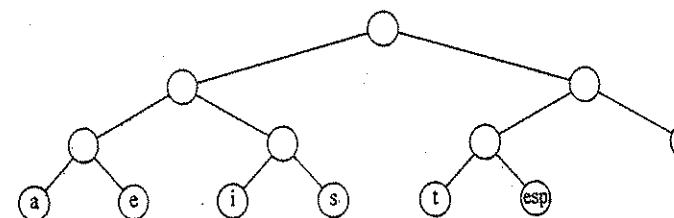
El código binario que representa el alfabeto se puede representar por medio del árbol binario de la figura 10.9. Este árbol tiene datos sólo en las hojas. La representación de cada carácter se puede encontrar empezando en la raíz y guardando el camino, usando un 0 para indicar la rama izquierda y un 1 para indicar la rama derecha. Por ejemplo, *s* se alcanza yendo a la izquierda, a la derecha y una vez más a la derecha. Esto se codifica como 011. A esta estructura de datos a veces se le llama *trie*. Si el carácter *c* está a la profundidad d_i y ocurre f_i veces, entonces el costo del código es igual a $\sum d_i f_i$.

Se puede obtener un mejor código que el de la figura 10.9 notando que *cambio de línea* es hijo único. Colocando el símbolo *cambio de línea* un nivel arriba, obtenemos el árbol nuevo de la figura 10.10. Este árbol nuevo tiene un costo de 173, pero aún está lejos de ser óptimo.

Observe que el árbol de la figura 10.10 es un *árbol completo*: todos los nodos son hojas o tienen dos hijos. Un código óptimo siempre tendrá esta propiedad, ya que de otra forma, como hemos visto, los nodos con sólo un hijo podrían subirse un nivel.

Si los caracteres se colocan sólo en las hojas, cualquier secuencia de bits puede decodificarse siempre sin ambigüedades. Por ejemplo, supongamos que la cadena codificada es 010011100010110001000111. 0 no es un código de carácter, 01 no es un código de carácter, pero 010 representa *t*, así que el primer carácter es *i*. Después sigue 011, queda una *t*. A continuación está 11, que es un *cambio de línea*. El resto del

Figura 10.10 Árbol ligeramente mejor



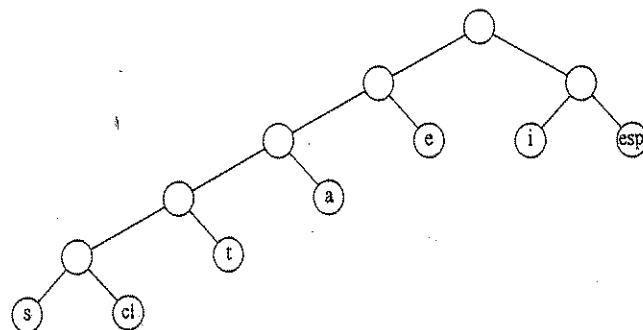


Figura 10.11 Código prefijo óptimo.

código es *a*, *espacio*, *t*, *i*, *e*, y *cambio de línea*. Así, no importa si los códigos de carácter son de longitud diferente, siempre que ningún código sea prefijo de otro código. Tal codificación se conoce como *codificación prefija*. A la inversa, si un carácter está contenido en un nodo no hoja, ya no es posible garantizar que la decodificación no será ambigua.

Considerando todo lo anterior vemos que el problema básico es encontrar el árbol binario completo de costo total mínimo (definido antes), donde todos los caracteres estén contenidos en las hojas. El árbol de la figura 10.11 muestra el árbol óptimo para nuestro alfabeto del ejemplo. Como puede verse en la figura 10.12, este código usa sólo 146 bits.

Observe que existen muchos códigos óptimos. Se puede obtener intercambiando los hijos en el árbol de codificación. La principal cuestión no resuelta, entonces, es cómo construir el árbol de codificación. El algoritmo para hacer esto fue creado por Huffman en 1952. Por ello a este sistema de codificación suele llamársele código de Huffman.

Figura 10.12 Código prefijo óptimo

Carácter	Código	Frecuencia	Bits totales
<i>a</i>	001	10	30
<i>e</i>	01	15	30
<i>i</i>	10	12	24
<i>s</i>	00000	3	15
<i>t</i>	0001	4	16
<i>espacio</i>	11	13	26
<i>cambio de línea</i>	00001	1	5
Total			146

Algoritmo de Huffman

En esta sección supondremos que el número de caracteres es C . El algoritmo de Huffman se puede describir como sigue. Mantenemos un bosque de árboles. El *peso* de un árbol es igual a la suma de las frecuencias de sus hojas. $C - 1$ veces, se seleccionan los dos árboles, A_1 y A_2 , de menor peso, eligiendo arbitrariamente entre iguales, y se forma un árbol nuevo con los subárboles A_1 y A_2 . Al inicio del algoritmo, hay C árboles de un solo nodo, uno por cada carácter. Al final del algoritmo hay un árbol, y este es el árbol óptimo de codificación de Huffman.

El desarrollo de un ejemplo aclarará la operación del algoritmo. La figura 10.13 muestra el bosque inicial; el peso de cada árbol se indica con números pequeños en la raíz. Se fusionan los dos árboles de menor peso para crear el bosque de la figura 10.14. A la raíz nueva se le llamará A_1 , para que las uniones futuras se puedan establecer sin ambigüedad. Arbitrariamente, hemos hecho de *s* el hijo izquierdo; se puede usar cualquier procedimiento para elegir en la igualdad. El peso total del árbol nuevo es sólo la suma de los pesos de los árboles anteriores, y por ello es fácil calcularlo. También es sencillo crear el árbol nuevo, ya que sólo se necesita obtener un nodo nuevo, poner los apuntadores izquierdo y derecho y guardar el peso.

Ahora se tienen seis árboles, y nuevamente seleccionamos los dos de menor peso. Estos son A_1 y *t*, los cuales se unen en un nuevo árbol con raíz A_2 y peso 8. (figura 10.15). El tercer paso fusiona A_2 y *a*, creando A_3 , con peso $10 + 8 = 18$. La figura 10.16 muestra el resultado de esta operación.

Después de completar la tercera fusión, los dos árboles de menor peso son los de sólo un nodo que representan *a* y *espacio blanco*. La figura 10.17 muestra cómo se fusionan estos árboles en el árbol nuevo con raíz A_4 . El quinto paso consiste en unir los árboles con raíces *e* y A_3 , ya que esos árboles tienen los pesos menores. El resultado de este paso se muestra en la figura 10.18.

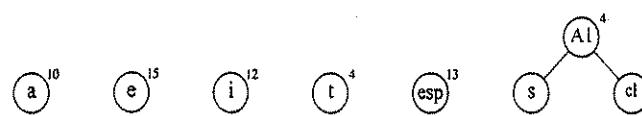
Por último, el árbol óptimo (figura 10.11), se obtiene al combinar los dos árboles restantes. La figura 10.19 muestra este árbol óptimo, con raíz A_6 .

Esbozaremos las ideas que intervienen en la demostración de que el algoritmo de Huffman produce un código óptimo y dejaremos los detalles como ejercicio.

Figura 10.13 Etapa inicial del algoritmo de Huffman



Figura 10.14 Algoritmo de Huffman después de la primera fusión



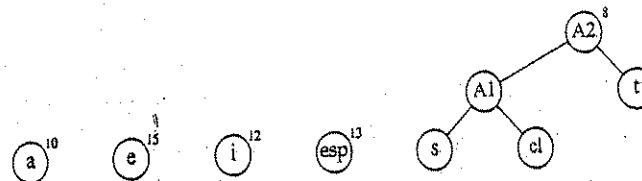


Figura 10.15 Algoritmo de Huffman después de la segunda fusión

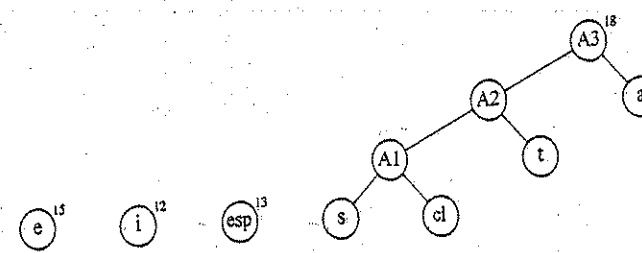


Figura 10.16 Algoritmo de Huffman después de la tercera fusión

Primero, no es difícil demostrar por contradicción que el árbol debe ser completo, pues ya hemos visto cómo se puede mejorar un árbol que no sea completo.

Después, hay que demostrar que los dos caracteres menos frecuentes α o β deben ser los dos nodos más profundos (aunque otros nodos pueden ser tan profundos). De nuevo, esto es fácil de demostrar por contradicción, ya que si α o β no es un nodo más profundo, entonces debe haber algún γ que lo sea (recuerde que el árbol es completo). Si α es menos frecuente que γ , entonces se puede mejorar el costo intercambiándolos en el árbol.

Entonces podemos argüir que los caracteres en cualesquiera dos nodos a la misma profundidad se pueden intercambiar sin afectar la optimalidad. Esto demuestra que siempre se puede encontrar un árbol óptimo que contenga los dos símbolos menos frecuentes como hermanos; así el primer paso no es un error.

La demostración se puede completar mediante un argumento por inducción. Conforme se fusionan los árboles, consideraremos que el nuevo conjunto de caracteres consiste en los caracteres en las raíces. Así, en nuestro ejemplo, después de cuatro fusiones, podemos ver que el conjunto de caracteres consta de e y los metacaracteres A3 y A4. Es probable que esta parte sea la más complicada de la demostración; queda al lector cubrir todos los detalles.

Figura 10.17 Algoritmo de Huffman después de la cuarta fusión

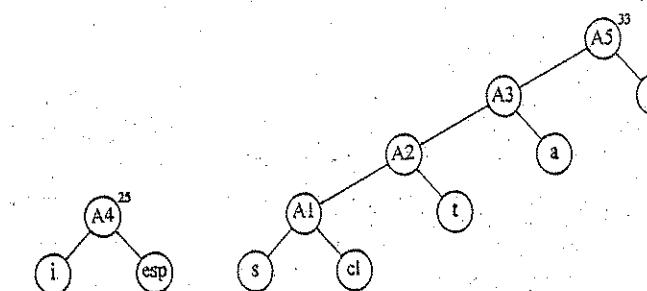
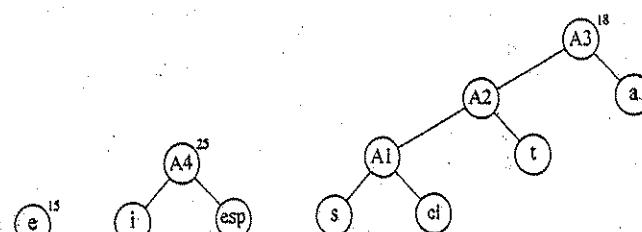


Figura 10.18 Algoritmo de Huffman después de la quinta fusión

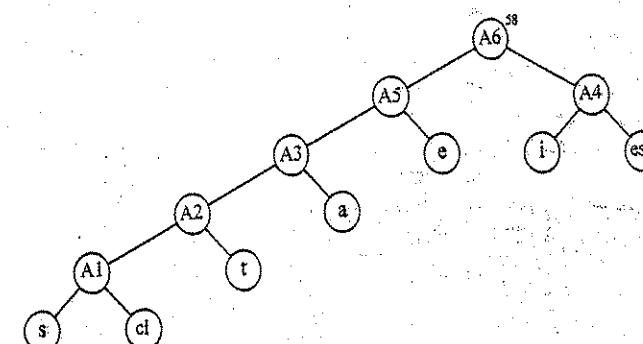


Figura 10.19 Algoritmo de Huffman después de la fusión final

La razón para considerar ávido este algoritmo es que en cada etapa efectuamos la fusión sin reparar en consideraciones globales. Simplemente seleccionamos los dos árboles más pequeños.

Si mantenemos los árboles en una cola de prioridad, ordenados por pesos, el tiempo de ejecución es $O(C \log C)$, ya que habrá un *construir_montículo*, $2C - 2$ *eliminar_mín*, y $C - 2$ *insertar*, en una cola de prioridad que nunca tiene más de C elementos. Una implantación simple de la cola de prioridad, usando una lista enlazada, daría un algoritmo $O(C^2)$. La elección de la implantación por colas de prioridad depende de lo grande que sea C . En el caso típico de un conjunto de caracteres ASCII, C es suficientemente pequeño para que el tiempo de ejecución cuadrático sea aceptable. En tal aplicación, prácticamente todo el tiempo de ejecución se consumirá en los accesos a disco requeridos para leer el archivo de entrada y escribir la versión comprimida.

Hay dos detalles por considerar. Primero, la información de codificación debe transmitirse al inicio del archivo comprimido, ya que de otra forma será imposible decodificar. Hay varias formas de hacer esto; véase el ejercicio 10.4. Para archivos pequeños, el costo de transmitir esta tabla anulará cualquier ahorro posible en la compresión, y el resultado probable será la expansión del archivo. Por supuesto,

esto puede detectarse para dejar el archivo intacto. Para archivos grandes, el tamaño de la tabla no es significativo.

El segundo problema es que, tal como se describió, este algoritmo es de dos pasadas. La primera recolecta los datos de frecuencia y la segunda hace la codificación. Obviamente, ésta no es una propiedad deseable para un programa que maneja archivos grandes. En las referencias se describen algunas alternativas.

10.1.3. Empaquetamiento aproximado en recipientes

En esta sección consideraremos algunos algoritmos para resolver el problema del *empaquetamiento* en recipientes. Estos algoritmos se ejecutan rápido, pero no necesariamente producen soluciones óptimas. Demostraremos, no obstante, que las soluciones producidas no están demasiado lejos de ser óptimas.

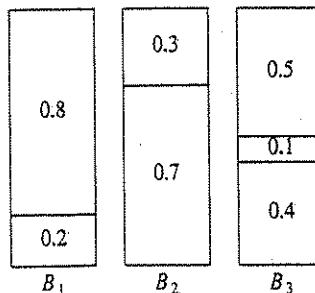
Sean n elementos de tamaños t_1, t_2, \dots, t_n . Todos los tamaños satisfacen $0 < t_i \leq 1$. El problema consiste en empaquetar esos elementos en el menor número de recipientes, dado que cada recipiente tiene una capacidad unitaria. A manera de ejemplo, la figura 10.20 muestra un empaquetamiento óptimo de una lista de elementos con tamaños 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

Hay dos versiones del problema del empaquetamiento en recipientes. La primera versión es el empaquetamiento *en línea*. En esta versión, cada elemento debe colocarse en un recipiente antes de procesar el siguiente elemento. La segunda versión es el problema del empaquetamiento *fuerza de línea*. En un algoritmo fuera de línea, no necesitamos hacer nada hasta leer toda la entrada. La diferencia entre los algoritmos en línea y fuera de línea se estudió en la sección 8.2.

Algoritmos en línea

El primer aspecto a considerar es si un algoritmo en línea puede o no realmente dar siempre una respuesta óptima, aun si se permite el cálculo ilimitado. Recordaremos que aun cuando se permita el cálculo ilimitado, un algoritmo en línea debe colocar un elemento antes de procesar el siguiente elemento y no puede cambiar su decisión.

Figura 10.20 Empaquetamiento óptimo de 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8



Para demostrar que un algoritmo en línea no siempre puede dar una solución óptima, proporcionaremos unos datos particularmente difíciles para trabajar. Consideré una secuencia de entrada E_1 de m elementos pequeños de peso $\frac{1}{2} - \epsilon$ seguida de m elementos grandes de peso $\frac{1}{2} + \epsilon$, con $0 < \epsilon < 0.01$. Es obvio que estos elementos se pueden empaquetar en m recipientes si colocamos un elemento pequeño y uno grande en cada recipiente. Supongamos que hay un algoritmo en línea óptimo A que podría efectuar este empaquetamiento. Consideremos la operación del algoritmo A sobre la secuencia E_2 , que consta sólo de m elementos pequeños de peso $\frac{1}{2} - \epsilon$. E_2 puede empaquetarse en $\lceil m/2 \rceil$ recipientes. No obstante, A colocará cada elemento en un recipiente separado, ya que A debe producir los mismos resultados sobre E_2 que con la primera mitad de E_1 , ya que la primera mitad de E_1 es exactamente la misma entrada que E_2 . Esto significa que A usará el doble de recipientes que una solución óptima para E_2 . Lo que hemos demostrado es que no hay algoritmo óptimo para el empaquetamiento en línea.

Lo que el argumento anterior muestra es que un algoritmo en línea nunca sabe cuándo termina la entrada, así cualquier garantía de rendimiento que ofrezca se debe cumplir en cualquier instante durante el algoritmo. Si seguimos esta estrategia podemos demostrar lo siguiente.

TEOREMA 10.1.

Hay entradas que fuerzan cualquier algoritmo en línea de empaquetamiento en recipientes a usar al menos $\frac{4}{3}$ del número óptimo de recipientes.

DEMOSTRACIÓN:

Supongamos que no es así y, por simplicidad, que m es par. Consideremos cualquier algoritmo en línea A ejecutándose con la secuencia de entrada E_1 , anterior. Recordaremos que esta secuencia consta de m elementos pequeños seguidos de m elementos grandes. Consideremos lo que el algoritmo A ha hecho después de procesar el m -ésimo elemento. Supongamos que A ya ha usado b recipientes. En este momento del algoritmo, el número óptimo de recipientes es $m/2$, porque podemos colocar dos elementos en cada recipiente. Así, sabemos que $2b/m < \frac{4}{3}$ por la suposición de un rendimiento garantizado mejor que $\frac{4}{3}$.

Ahora consideremos el rendimiento del algoritmo A después de haber empaquetado todos los elementos. Todos los recipientes creados después del b -ésimo recipiente deben contener exactamente un elemento, ya que todos los elementos pequeños están colocados en los primeros b recipientes, y dos elementos grandes no cabrán en un recipiente. Puesto que cada uno de los primeros b recipientes puede contener dos elementos como máximo, y los recipientes restantes tienen un elemento cada uno, vemos que el empaquetamiento de $2m$ elementos requerirá al menos $2m - b$ recipientes. Puesto que $2m$ elementos pueden ser empaquetados óptimamente usando m recipientes, la garantía de rendimiento asegura que $(2m - b)/m < \frac{4}{3}$.

La primera desigualdad implica que $b/m < \frac{2}{3}$ y la segunda desigualdad implica que $b/m > \frac{2}{3}$, lo cual es una contradicción. Así, ningún algoritmo en línea puede garantizar que producirá un empaquetamiento con menos de $\frac{4}{3}$ del número óptimo de recipientes.

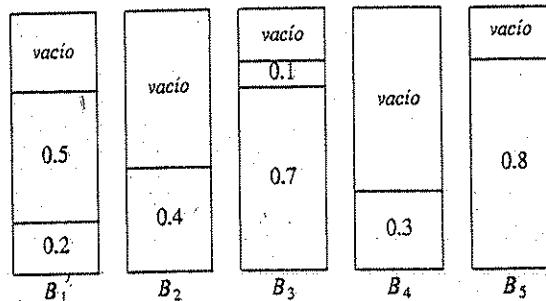


Figura 10.21 Próximo ajuste para 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Hay tres algoritmos sencillos que garantizan que el número de recipientes en uso no es más del doble del óptimo. También hay unos cuantos algoritmos más complejos con mejores garantías.

Próximo ajuste

Probablemente el algoritmo más sencillo sea el de *próximo ajuste*. Al procesar cada elemento, revisamos si cabe en el mismo recipiente que el último elemento. Si es así, se coloca allí; si no, se crea un recipiente. Es increíblemente sencillo implantar este algoritmo y se ejecuta en tiempo lineal. La figura 10.21 muestra el empaquetamiento producido para la misma entrada de la figura 10.20.

Próximo ajuste no sólo es fácil de programar, también es fácil analizar su comportamiento para el peor caso.

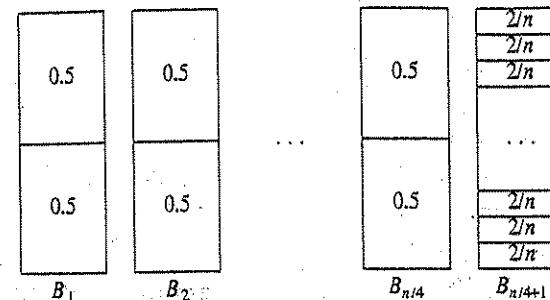
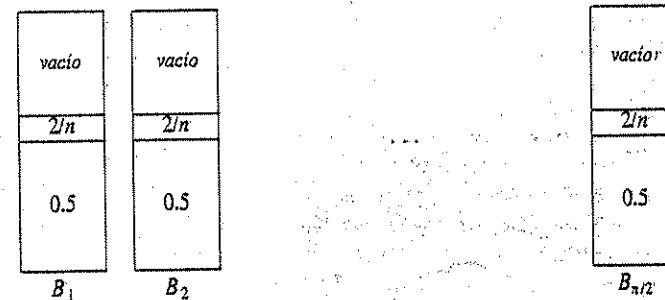
TEOREMA 10.2

Sea m el número óptimo de recipientes requeridos para empaquetar una lista E de elementos. Entonces el próximo ajuste nunca usa más de $2m$ recipientes. Existen secuencias tales que el próximo ajuste usa $2m - 2$ recipientes.

DEMOSTRACIÓN:

Consideremos cualesquiera recipientes adyacentes B_j y B_{j+1} . La suma de los tamaños de todos los elementos en B_j y B_{j+1} debe ser mayor que 1, ya que en otro caso todos esos elementos se habrían colocado en B_j . Si aplicamos este resultado a todos los pares de recipientes adyacentes veremos que se ha malgastado como máximo la mitad del espacio. Así, el próximo ajuste usa a lo más el doble de recipientes.

Para ver que esta cota es ajustada, supongamos que los n elementos tienen tamaño $t_i = 0.5$ si i es impar y $t_i = 2/n$ si i es par. Supongamos que n es divisible entre 4. El empaquetamiento óptimo, mostrado en la figura 10.22, consta de $n/4$ recipientes, cada uno con 2 elementos de tamaño 0.5, y un recipiente más con $n/2$ elementos de tamaño $2/n$, para un total de $(n/4) + 1$. La figura 10.23 muestra que el próximo ajuste usa $n/2$ recipientes. Así, el próximo ajuste puede ser forzado a usar a lo más el doble del número óptimo de recipientes.

Figura 10.22 Empaquetamiento óptimo para 0.5, $2/n$, 0.5, $2/n$, $0.5, 2/n, \dots$ Figura 10.23 Empaquetamiento de próximo ajuste para 0.5, $2/n$, $0.5, 2/n, 0.5, 2/n, \dots$

Primer ajuste

Aunque el próximo ajuste tiene una garantía razonable, en la práctica es ineficiente porque crea recipientes nuevos cuando no los necesita. En la ejecución, ejemplo, se podría haber colocado el elemento de tamaño 0.3 en B_1 o en B_2 , en vez de crear un recipiente nuevo.

La estrategia del *primer ajuste* consiste en recorrer los recipientes en orden y colocar el elemento nuevo en el primero en que haya suficiente espacio para contenerlo. Así se crea sólo un recipiente nuevo cuando los resultados de colocaciones previas no hayan dejado más alternativa. La figura 10.24 muestra el empaquetamiento que resulta del primer ajuste sobre nuestra entrada estándar.

Un método simple de implantación del primer ajuste procesaría cada elemento recorriendo secuencialmente la lista de recipientes. Esto podría tardar $O(n^2)$. Es posible implantar el primer ajuste para que se ejecute en $O(n \log n)$; dejamos esto como ejercicio.

Un breve razonamiento nos convencerá de que, en cualquier momento, cuando mucho un recipiente puede estar vacío en más de la mitad, ya que si un segundo recipiente también estuviera medio vacío, su contenido cabría en el primero. Así, podemos concluir inmediatamente que el primer ajuste garantiza una solución con a lo más el doble del número óptimo de recipientes.

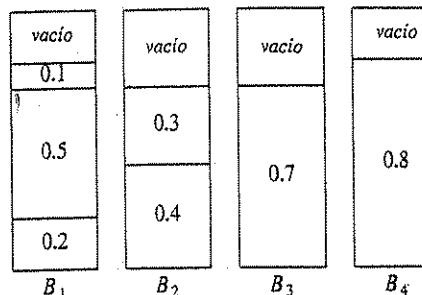


Figura 10.24 Primer ajuste para 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Por otro lado, el caso malo usado en la demostración de la cota del rendimiento del próximo ajuste no se aplica al primer ajuste. Así, uno puede preguntarse si se puede demostrar que hay una cota mejor. La respuesta es afirmativa, pero la demostración es compleja.

TEOREMA 10.3.

Sea m el número óptimo de recipientes requeridos para empaquetar una lista E de elementos. Entonces el primer ajuste nunca usa más de $\lceil \frac{17}{10}m \rceil$ recipientes. Existen secuencias tales que el primer ajuste usa $\lceil \frac{17}{10}(m - 1) \rceil$ recipientes.

DEMOSTRACIÓN:

Véanse las referencias al final del capítulo.

En la figura 10.25 se muestra un ejemplo donde el primer ajuste actúa casi tan deficientemente como indica en el teorema anterior. La entrada consta de $6m$ elementos de tamaño $\frac{1}{7} + \epsilon$, seguidos de $6m$ elementos de tamaño $\frac{1}{3} + \epsilon$, seguidos de $6m$ elementos de tamaño $\frac{1}{2} + \epsilon$. Un empaquetamiento simple coloca un elemento de cada tamaño en un recipiente y requiere $6m$ recipientes. El primer ajuste requiere $10m$ recipientes.

Cuando se ejecuta un primer ajuste en un número grande de elementos con tamaños uniformemente distribuidos entre 0 y 1, hay resultados empíricos que demuestran que el primer ajuste usa casi un 2% más de recipientes que el óptimo. En muchos casos, esto es bastante aceptable.

Mejor ajuste

La tercera estrategia en línea que examinaremos es el *mejor ajuste*. En vez de colocar un elemento nuevo en el primer espacio que se encuentre, se pone en el espacio más ajustado de entre todos los recipientes. Un empaquetamiento característico se muestra en la figura 10.26.

Observe que el elemento de tamaño 0.3 se coloca en B_3 , donde cabe perfectamente, en vez de B_2 . Uno podría esperar que el rendimiento mejore, pues ahora

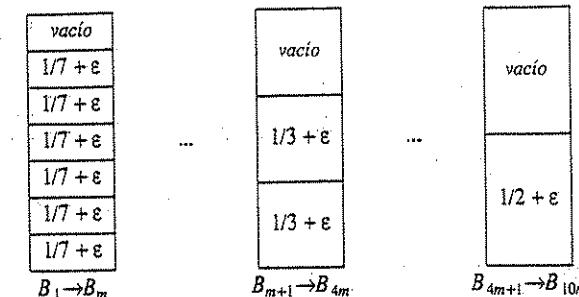


Figura 10.25: Un caso donde el primer ajuste es 60% peor que el óptimo

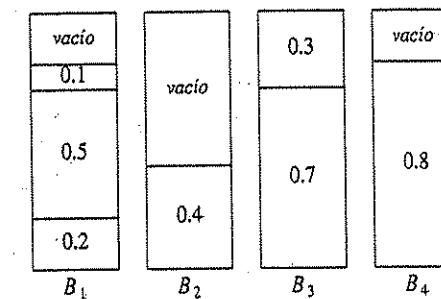
estamos haciendo una elección más refinada de los recipientes. Éste no es el caso, porque los malos casos genéricos son los mismos. El mejor ajuste nunca es más de aproximadamente 1.7 veces peor que el óptimo, y hay entradas para las cuales (casi) se alcanza esta cota. Sin embargo, es sencillo codificar el mejor ajuste, en especial si se requiere un algoritmo $O(n \log n)$.

Algoritmos fuera de línea

Si podemos ver la lista de elementos completa antes de producir una respuesta, entonces deberíamos esperar un resultado mejor. En efecto, como al fin y al cabo podemos encontrar el empaquetamiento óptimo mediante una búsqueda exhaustiva, ya tenemos una mejora teórica sobre el caso en línea.

El principal problema con todos los algoritmos en línea es la dificultad de empaquetar elementos grandes, en especial cuando ocurren tardíamente en la entrada. La forma natural para hacer esto es ordenar los elementos, colocando primero los más grandes. Entonces, podemos aplicar el primer o mejor ajuste, obteniendo los algoritmos *primer ajuste decreciente* y *mejor ajuste decreciente*, respec-

Figura 10.26 Mejor ajuste para 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8



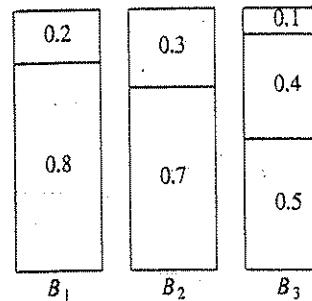


Figura 10.27 Primer ajuste para 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1

tivamente. La figura 10.27 muestra que en nuestro caso esto lleva a una solución óptima (aunque, por supuesto, esto no es cierto en general).

En esta sección, nos ocuparemos del primer ajuste decreciente. Los resultados del mejor ajuste decreciente son casi idénticos. Puesto que es posible que los tamaños de los elementos no sean diferentes, algunos autores prefieren llamar al algoritmo *primer ajuste no creciente*. Conservaremos el nombre original. Supondremos, sin pérdida de generalidad, que los tamaños de las entradas ya están ordenados.

Lo primero a remarcar es que el peor caso, que mostró que el primer ajuste usa $10m$ recipientes en vez de $6m$ recipientes, no se aplica cuando los elementos están ordenados. Demostraremos que si un empaquetamiento óptimo usa m recipientes, el primer ajuste decreciente nunca usa más de $(4m + 1)/3$ recipientes.

El resultado depende de dos observaciones. Primero, todos los elementos con peso mayor que $1/3$ se colocarán en los primeros m recipientes. Esto implica que todos los elementos en los recipientes adicionales tienen peso de $\frac{1}{3}$ como máximo. La segunda observación es que el número de elementos en los recipientes adicionales puede ser $(m - 1)$. Combinando los dos resultados, encontraremos que se pueden requerir a lo más $\lceil (m - 1)/3 \rceil$ recipientes adicionales. Ahora probaremos ambas observaciones.

LEMA 10.1

Sean los n elementos (ordenados en orden decreciente) con tamaños de entrada t_1, t_2, \dots, t_n , respectivamente, y supongamos que el empaquetamiento óptimo es de m recipientes. Entonces todos los elementos que el primer ajuste decreciente coloca en recipientes adicionales tienen un tamaño de $\frac{1}{3}$ como máximo.

DEMOSTRACIÓN:

Supongamos que el i -ésimo elemento es el primero colocado en el recipiente $m + 1$. Necesitamos demostrar que $t_i \leq \frac{1}{3}$. Comprobaremos esto por contradicción. Supongamos que $t_i > \frac{1}{3}$.

Se infiere que $t_1, t_2, \dots, t_{i-1} > \frac{1}{3}$, ya que los tamaños están ordenados. De esto se infiere que todos los recipientes B_1, B_2, \dots, B_m tienen cuando mucho dos elementos cada uno.

Consideremos el estado del sistema después de que el $i-1$ -ésimo elemento se coloca en un recipiente, pero antes de que se coloque el i -ésimo. Ahora queremos demostrar que (suponiendo que $t_i > \frac{1}{3}$) los primeros m recipientes se ordenan como sigue: primero hay algunos recipientes con exactamente un elemento, y después los recipientes restantes tienen dos elementos.

Supongamos que hubiera dos recipientes B_x y B_y , tales que $1 \leq x < y \leq m$. B_x tiene dos elementos y B_y tiene un elemento. Sean x_1 y x_2 los dos elementos en B_x , y sea y_1 el elemento en B_y . $x_1 \geq y_1$, ya que x fue colocado en el recipiente anterior. $x_2 \geq t_i$, por un razonamiento semejante. Así, $x_1 + x_2 \geq y_1 + t_i$. Esto implica que t_i se podría colocar en B_y . Pero nuestra hipótesis no es posible. Así, si $t_i > \frac{1}{3}$, entonces, en el momento en que intentamos procesar t_i , los primeros m recipientes están dispuestos de tal modo que los primeros j tienen un elemento y los siguientes $m-j$ tienen dos elementos.

Para probar el lema demostraremos que no hay forma de colocar todos los elementos en m recipientes, lo cual contradice la premisa del lema.

Desde luego, ningún par de elementos de t_1, t_2, \dots, t_i se puede colocar en un recipiente, por ningún algoritmo, porque si pudieran, el primer ajuste lo habría hecho también. Además, sabemos que el primer ajuste no ha colocado ninguno de los elementos de tamaño $t_{j+1}, t_{j+2}, \dots, t_i$ en los primeros j recipientes, así que ninguno de ellos cabe. Con esto, en cualquier empaquetamiento, específicamente el óptimo, debe haber j recipientes que no contienen estos elementos. Se infiere que los elementos de tamaño $t_{j+1}, t_{j+2}, \dots, t_{i-1}$ deben estar contenidos en algún conjunto de $m-j$ recipientes, y de las consideraciones anteriores, el número total de tales elementos es $2(m-j)$.

La demostración se completa observando que si $t_i > \frac{1}{3}$, no hay forma de que t_i se coloque en uno de esos m recipientes. Por supuesto, no puede ir en uno de los j recipientes, porque si se pudiera, el primer ajuste lo habría hecho también. Para colocarlo en uno de los restantes $m-j$ recipientes se requiere distribuir $2(m-j) + 1$ elementos en los $m-j$ recipientes. Así, algún recipiente tendría que tener tres elementos, cada uno de los cuales es mayor que $\frac{1}{3}$, una clara imposibilidad.

Esto contradice el hecho de que todos los tamaños pueden colocarse en m recipientes, así que la suposición original debe ser incorrecta. Por tanto, $t_i \leq \frac{1}{3}$.

LEMA 10.2

El número de objetos colocados en recipientes adicionales $m-1$ como máximo.

DEMOSTRACIÓN:

Supongamos que hay al menos m objetos colocados en recipientes adicionales. Sabemos que $\sum_{i=1}^n s_i \leq m$, pues todos los objetos caben en m recipientes. Supongamos que B_j se llena con un peso total W_j para $1 \leq j \leq m$. Supongamos que los primeros m objetos adicionales tienen tamaños x_1, x_2, \dots, x_m . Entonces, puesto

[†] Recuerde que el primer ajuste empaquetó estos elementos en $m-j$ recipientes y colocó dos elementos en cada recipiente. Así, hay $2(m-j)$ elementos.

que los elementos en los primeros m recipientes más los primeros m elementos adicionales son un subconjunto de todos los elementos, se infiere que

$$\sum_{i=1}^n s_i \geq \sum_{j=1}^m W_j + \sum_{j=1}^m x_j \geq \sum_{j=1}^m (W_j + x_j)$$

Ahora $W_j + x_j > 1$, porque de otra forma el elemento correspondiente a x_j habría sido colocado en B_j . Así

$$\sum_{i=1}^n s_i > \sum_{j=1}^m 1 > m$$

Pero esto es imposible si los n elementos se pueden empaquetar en m recipientes. Así, puede haber $m - 1$ elementos adicionales como máximo.

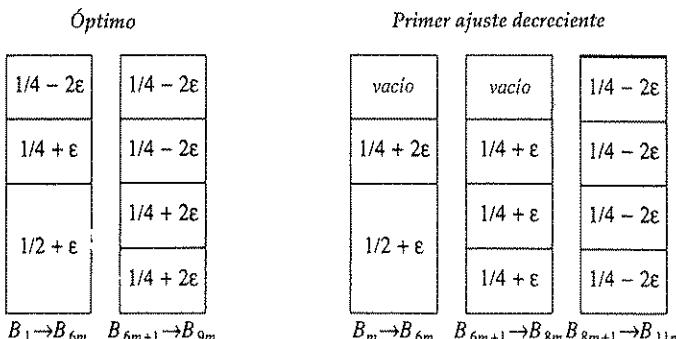
TEOREMA 10.4.

Sea m el número óptimo de recipientes requeridos para empaquetar una lista E de elementos. Entonces el primer ajuste decreciente nunca usa más de $(4m + 1)/3$ recipientes.

DEMOSTRACIÓN:

Hay $m - 1$ elementos adicionales, de tamaño máximo de $\frac{1}{3}$. Así, puede haber a lo más $\lceil (m - 1)/3 \rceil$ recipientes adicionales. Por tanto, el número total de recipientes usados por el primer ajuste decreciente es $\lceil (4m - 1)/3 \rceil \leq (4m + 1)/3$.

Figura 10.28 Ejemplo donde el primer ajuste decreciente usa $11m$ recipientes, pero sólo se requieren $9m$ recipientes



Es posible demostrar que hay una cota mucho más ajustada tanto para el primer ajuste decreciente como para el próximo ajuste decreciente.

TEOREMA 10.5.

Sea m el número óptimo de recipientes requeridos para empaquetar una lista E de elementos. Entonces el primer ajuste decreciente nunca usa más de $\frac{11}{9}m + 4$ recipientes. Existen secuencias tales que el primer ajuste decreciente usa $\frac{11}{9}m$ recipientes.

DEMOSTRACIÓN:

La cota superior requiere un análisis muy complicado. La cota inferior se exhibe como una secuencia de $6m$ elementos de tamaño $\frac{1}{2} + \epsilon$, seguidos de $6m$ elementos de tamaño $\frac{1}{4} + 2\epsilon$, seguidos de $6m$ elementos de tamaño $\frac{1}{4} + \epsilon$, seguidos de $12m$ elementos de tamaño $\frac{1}{4} - 2\epsilon$. La figura 10.28 muestra que el empaquetamiento óptimo requiere $9m$ recipientes, pero el primer ajuste decreciente usa $11m$ recipientes.

En la práctica, el primer ajuste decreciente funciona extremadamente bien. Si la elección de los puntos es uniforme sobre el intervalo unitario, entonces el número esperado de recipientes adicionales es $\Theta(\sqrt{m})$. El empaquetamiento en recipientes es un ejemplo sutil de cómo las simples heurísticas ávidas pueden dar buenos resultados.

10.2. "Divide y vencerás"

Otra técnica común usada en el diseño de algoritmos es "divide y vencerás". Los algoritmos "divide y vencerás" constan de dos partes:

Dividir: Los problemas más pequeños se resuelven recursivamente (excepto, por supuesto, los casos base).

Vencer: la solución del problema original se forma entonces a partir de las soluciones de los subproblemas.

Tradicionalmente, las rutinas en las cuales el texto contiene al menos dos llamadas recursivas se denominan algoritmos de "divide y vencerás", no así las rutinas cuyo texto sólo contiene una llamada recursiva. En general, insistimos en que los subproblemas sean ajenos (esto es, que en esencia no se traslapen). Revisemos algunos de los algoritmos recursivos que se han cubierto en este texto.

Ya hemos visto varios algoritmos de "divide y vencerás". En la sección 2.4.3 vimos una solución $O(n \log n)$ para el problema de la suma de la subsecuencia máxima. En el capítulo 4 estudiamos estrategias de recorrido de árboles en tiempo lineal. En el capítulo 7 vimos los ejemplos clásicos de "divide y vencerás", a saber, la ordenación por intercalación y la ordenación rápida, que tienen cotas $O(n \log n)$ para el peor caso y el caso medio, respectivamente.

También hemos visto varios ejemplos de algoritmos recursivos que probablemente no se clasifican como estrategia de "divide y vencerás", sino que sólo se reducen a un caso único más simple. En la sección 1.3 examinamos una rutina simple

para visualizar un número. En el capítulo 2 efectuamos exponentiación eficiente mediante la recursión. En el capítulo 4 examinamos rutinas de búsqueda sencillas para árboles binarios de búsqueda. En la sección 6.6 vimos la recursión sencilla para fusionar montículos a izquierda. En la sección 7.7 proporcionamos un algoritmo para la selección en un tiempo medio lineal. En el capítulo 8 se escribió recursivamente la operación *buscar* sobre conjuntos ajenos. En el capítulo 9 se mostraron rutinas para recuperar el camino más corto en el algoritmo de Dijkstra y otros procedimientos para efectuar la búsqueda en profundidad en grafos. Ninguno de esos algoritmos realmente son algoritmos de "divide y vencerás", porque sólo realizan una llamada recursiva.

También hemos visto, en la sección 2.4, una rutina recursiva muy deficiente para calcular los números de Fibonacci. Esta podría identificarse como algoritmo de "divide y vencerás", pero su ineficiencia es muy importante, porque en realidad el problema no se divide del todo.

En esta sección, veremos más ejemplos del paradigma de "divide y vencerás". Nuestra primera aplicación es un problema de *geometría computacional*. Dados n puntos en un plano, demostraremos que el par de puntos más cercanos se puede encontrar en un tiempo $O(n \log n)$. Los ejercicios describen algunos otros problemas de geometría computacional que pueden resolverse con la estrategia de "divide y vencerás". El resto de la sección muestra algunos resultados extremadamente interesantes, pero sobre todo teóricos. Proporcionamos un algoritmo que resuelve el problema de la selección en un tiempo $O(n)$ para el peor caso. También mostramos que 2 números de n bits se pueden multiplicar en $O(n^2)$ operaciones y que dos matrices $n \times n$ se pueden multiplicar en $O(n^3)$. Desafortunadamente, aun cuando estos algoritmos tienen mejores cotas para el peor caso que los algoritmos convencionales, ninguno es práctico, excepto en entradas muy grandes.

10.2.1. Tiempo de ejecución de algoritmos de "divide y vencerás"

Todos los algoritmos eficientes de "divide y vencerás" que veremos dividen el problema en subproblemas, cada uno de los cuales es alguna fracción del problema original, y después efectúan algún trabajo adicional para calcular la respuesta final. Como ejemplo, hemos visto que la ordenación por intercalación opera sobre dos problemas, cada uno de los cuales tiene la mitad del tamaño del original, y después usa trabajo adicional $O(n)$. Esto da la ecuación del tiempo de ejecución (con las condiciones iniciales adecuadas).

$$T(n) = 2T(n/2) + O(n)$$

En el capítulo 7 vimos que la solución de esta ecuación es $O(n \log n)$. Mediante el siguiente teorema se puede determinar el tiempo de ejecución de la mayoría de los algoritmos de "divide y vencerás".

TEOREMA 10.6.

La solución de la ecuación $T(n) = aT(n/b) + \Theta(n^k)$, donde $a \geq 1$ y $b > 1$, es

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \log n) & \text{si } a = b^k \\ O(n^k) & \text{si } a < b^k \end{cases}$$

DEMOSTRACIÓN:

Siguiendo el análisis de la ordenación por intercalación del capítulo 7, supondremos que n es una potencia de b ; así, sea $n = b^m$. Entonces $n/b = b^{m-1}$ y $n^k = (b^m)^k = b^{mk} = b^{km} = (b^k)^m$. Supongamos que $T(1) = 1$, e ignoremos el factor constante en $\Theta(n^k)$. Entonces se tiene

$$T(b^m) = aT(b^{m-1}) + (b^k)^m$$

Si dividimos todos entre a^m , se obtiene la ecuación

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \left\{ \frac{b^k}{a} \right\}^m \quad (10.3)$$

Se puede aplicar esta ecuación a otros valores de m , obteniendo

$$\frac{T(b^{m-1})}{a^{m-1}} = \frac{T(b^{m-2})}{a^{m-2}} + \left\{ \frac{b^k}{a} \right\}^{m-1} \quad (10.4)$$

$$\frac{T(b^{m-2})}{a^{m-2}} = \frac{T(b^{m-3})}{a^{m-3}} + \left\{ \frac{b^k}{a} \right\}^{m-2} \quad \dots \quad (10.5)$$

$$\frac{T(b^1)}{a^1} = \frac{T(b^0)}{a^0} + \left\{ \frac{b^k}{a} \right\}^1 \quad (10.6)$$

Nos valemos de nuestra treta convencional de sumar las ecuaciones proyectivas (10.3) a la (10.6). Prácticamente todos los términos de la izquierda cancelan a los principales términos de la derecha, dando

$$\frac{T(b^m)}{a^m} = 1 + \sum_{i=1}^m \left\{ \frac{b^k}{a} \right\}^i \quad (10.7)$$

$$= \sum_{i=0}^m \left\{ \frac{b^k}{a} \right\}^i \quad (10.8)$$

Así

$$T(n) = T(b^m) = a^m \sum_{i=0}^m \left\{ \frac{b^k}{a} \right\}^i \quad (10.9)$$

Sia $a > b^k$, entonces la suma es una serie geométrica con razón menor que 1. Puesto que la suma de series infinitas convergería a una constante, esta suma finita está acotada también por una constante, y así se aplica la ecuación (10.10):

$$T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a}) \quad (10.10)$$

Si $a = b^k$, entonces cada término en la suma es 1. Puesto que la suma contiene $1 + \log_b n$ términos y $a = b^k$ implica que $\log_b a = k$,

$$\begin{aligned} T(n) &= O(a^m \log_b n) = O(n^{\log_b a} \log_b n) = O(n^k \log_b n) \\ &= O(n^k \log n) \end{aligned} \quad (10.11)$$

Por último, si $a < b^k$, entonces los términos de la serie geométrica son mayores que 1, y se aplica la segunda fórmula de la sección 1.2.3. Obtenemos

$$T(n) = a^m \frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1} = O(a^m (b^k/a)^m) = O((b^k)^m) = O(n^k) \quad (10.12)$$

con lo que se demuestra el último caso del teorema.

A manera de ejemplo, la ordenación por intercalación tiene $a = b = 2$ y $k = 1$. Se aplica el segundo caso, con la respuesta $O(n \log n)$. Si se resuelven tres problemas, cada uno de los cuales es la mitad del tamaño original, y se combinan las soluciones con trabajo adicional $O(n)$, entonces $a = 3$, $b = 2$ y $k = 1$. Aquí se aplica el caso 1, dando una cota de $O(n^{\log_3 3}) = O(n^{1.59})$. Un algoritmo que resuelva tres problemas reducidos a la mitad, pero que requiera trabajo $O(n^2)$ para conseguir la solución, tendría un tiempo de ejecución $O(n^3)$, pues se aplicaría el tercer caso.

Hay dos casos importantes que no cubre el teorema 10.6. Enunciaremos dos teoremas más, dejando las demostraciones como ejercicios. El teorema 10.7 generaliza el teorema anterior.

TEOREMA 10.7.

La solución de la ecuación $T(n) = aT(n/b) + \Theta(n^k \log^p n)$, donde $a \geq 1$, $b > 1$ y $p \geq 0$ es

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \log^{p+1} n) & \text{si } a = b^k \\ O(n^k \log^p n) & \text{si } a < b^k \end{cases}$$

TEOREMA 10.8.

Si $\sum_{i=1}^k \alpha_i < 1$, la solución de la ecuación es $T(n) = \sum_{i=1}^k T(\alpha_i n) + O(n)$ es $T(n) = O(n)$.

10.2.2. El problema de los puntos más cercanos

La entrada a nuestro primer problema es una lista P de puntos en un plano. Si $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$, entonces la distancia euclídea entre p_1 y p_2 es $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Necesitamos encontrar el par de puntos más cercanos. Es posible que dos puntos tengan la misma posición; en tal caso ese par es el más cercano, con distancia cero.

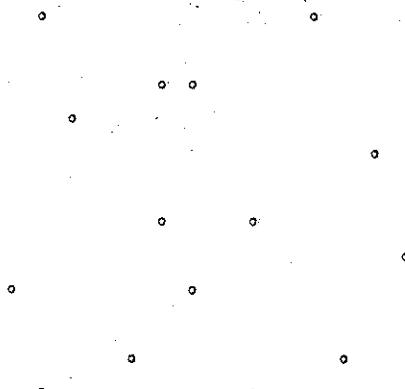
Si hay n puntos, entonces hay $n(n+1)/2$ pares de distancias. Podemos revisar todas ellas, obteniendo un programa muy corto, pero al costo de un algoritmo $O(n^2)$. Puesto que este tratamiento es sólo una búsqueda exhaustiva, deberíamos esperar hacerlo mejor.

Supongamos que los puntos han sido ordenados de acuerdo con la coordenada x . En el peor caso, se agrega $O(n \log n)$ a la cota de tiempo final. Entendiendo que se mostrará una cota $O(n \log n)$ para el algoritmo completo, esta ordenación es esencialmente libre, desde el punto de vista de la complejidad.

La figura 10.29 muestra como ejemplo un pequeño conjunto de puntos P . Como los puntos se ordenan de acuerdo con la coordenada x , es posible dibujar una línea vertical imaginaria que divide el conjunto de puntos en dos mitades, P_L y P_R . Ciertamente esto es fácil. Ahora tenemos casi exactamente la misma situación que en el problema de la suma de la subsecuencia máxima de la sección 2.4.3. O bien, ambos puntos más cercanos están en P_L , o ambos están en P_R , o uno está en P_L y el otro en P_R . Llamemos a esas distancias d_L , d_M y d_R . La figura 10.30 muestra la partición del conjunto de puntos y las tres distancias.

Podemos calcular d_L y d_R recursivamente. Entonces el problema es calcular d_M . Puesto que es de desear una solución $O(n \log n)$, debemos ser capaces de calcular d_M con sólo un trabajo $O(n)$ adicional. Ya hemos visto que si un procedimiento consta de dos llamadas recursivas de la mitad del tamaño y un trabajo adicional $O(n)$, entonces el tiempo total será $O(n \log n)$.

Figura 10.29 Pequeño conjunto de puntos



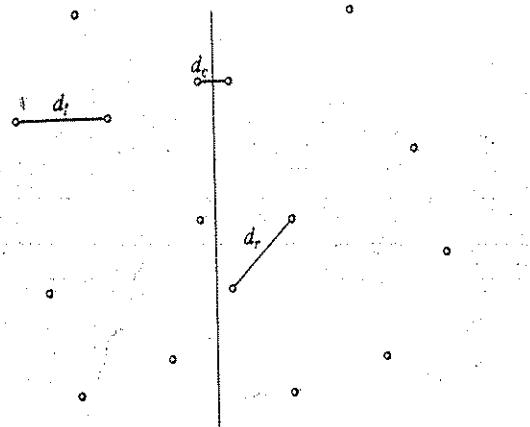


Figura 10.30 P dividido en P_1, P_2 ; se muestran las distancias más cortas

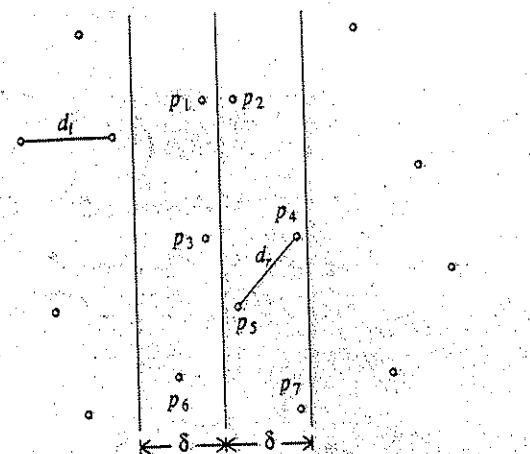


Figura 10.31 Franja de dos sendas que contiene todos los puntos considerados para la franja d_c .

(Todos los puntos están en la franja)

```
for i := 1 to NÚM_PUNTOS_EN_FRANJA do
    for j := i + 1 to NÚM_PUNTOS_EN_FRANJA do
        if dist( $p_i, p_j$ ) <  $\delta$  then
             $\delta := \min(\delta, d_c)$ 
```

Figura 10.32 Cálculo de fuerza bruta de $\min(\delta, d_c)$.

(Todos los puntos están en la franja y ordenados de acuerdo con la coordenada y)

```
for i := 1 to NÚM_PUNTOS_EN_FRANJA do
    for j := i + 1 to NÚM_PUNTOS_EN_FRANJA do
        if las coordenadas  $y$  de  $p_i$  y  $p_j$  difieren en más de  $\delta$  then
            ir al siguiente  $p_i$  (salir del ciclo interior)
        else
            if  $\text{dist}(p_i, p_j) < \delta$  then
                 $\delta := \text{dist}(p_i, p_j)$ 
```

Figura 10.33 Cálculo refinado de $\min(\delta, d_c)$.

Sea $\delta = \min(d_i, d_d)$. La primera observación es que sólo necesitamos calcular d_c si d_c mejora en δ . Si d_c es tal distancia, entonces los dos puntos que definen d_c deben estar dentro de δ de la línea divisoria; nos referiremos a esta área como *franja*. Como se muestra en la figura 10.31, esta observación limita el número de puntos que debemos considerar (en nuestro caso, $\delta = d_d$).

Hay dos estrategias que pueden intentarse para calcular d_c . Para grandes conjuntos de puntos que están distribuidos uniformemente, el número de puntos que se espera estén en la franja es muy pequeño. En efecto, es fácil argüir que sólo $O(\sqrt{n})$ puntos en medio están en la franja. Así, podríamos efectuar un cálculo de fuerza bruta sobre estos puntos en un tiempo $O(n)$. El pseudocódigo de la figura 10.32 implanta esta estrategia.

En el peor caso, todos los puntos pueden estar en la franja, así que esta estrategia no siempre funciona en tiempo lineal. Este algoritmo se puede mejorar con la siguiente observación: las coordenadas y de los dos puntos que definen d_c pueden diferir a lo más en δ . En otro caso, $d_c > \delta$. Supongamos que los puntos de la franja están ordenados de acuerdo con sus coordenadas y . Por lo tanto, si las coordenadas y de p_i y p_j difieren por más de δ entonces podemos proceder con p_{i+1} . Esta sencilla modificación se implanta en la figura 10.33.

Esta prueba adicional tiene un efecto significativo sobre el tiempo de ejecución porque para cada p_i sólo unos cuantos puntos p_j se examinan antes que las coordenadas y de p_i y p_j difieran en más de δ y fuercen una salida del ciclo *for* interior. La figura 10.34 muestra, por ejemplo, que para el punto p_3 , sólo los dos puntos p_4 y p_5 caen en la franja con distancia vertical δ .

En el peor caso, para cualquier punto p_i , a lo más se consideran 7 puntos p_j . Esto es porque esos puntos deben caer ya sea en el cuadrado δ por δ de la mitad izquierda de la franja o en el cuadrado δ por δ de la mitad derecha de la franja. Por otro lado, todos los puntos en cada cuadrado δ por δ están separados por al menos δ . En el peor caso, cada cuadrado contiene cuatro puntos, uno en cada esquina. Uno de esos puntos es p_i , dejando a lo más siete puntos por considerar. Esta situación del peor caso se muestra en la figura 10.35. Observe que aun cuando p_{12} y p_{21} tienen las mismas coordenadas, podrían ser puntos diferentes. Para el análisis real, sólo es importante que el número de puntos en el rectángulo λ por 2λ sea $O(1)$, y esto es ciertamente claro.

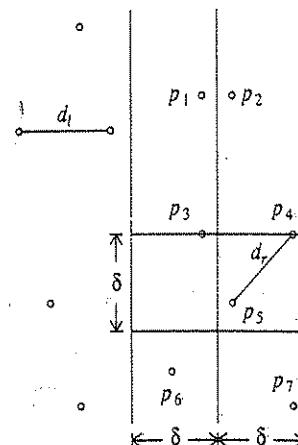
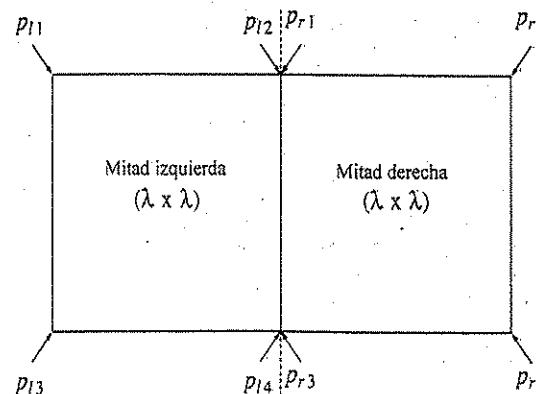


Figura 10.34 Sólo p_4 y p_5 se consideran en el segundo ciclo *for*

Puesto que como máximo se consideran siete puntos en cada p_i , el tiempo para calcular un d_i que sea mejor que δ es $O(n)$. Así, parece que tenemos una solución recursiva del problema de los puntos más cercanos, basada en las dos llamadas recursivas de mitad de tamaño más el trabajo lineal adicional para combinar los dos resultados. No obstante, todavía no tenemos una solución $O(n \log n)$.

El problema es que hemos supuesto que está disponible una lista de puntos ordenada por la coordenada y . Si efectuamos esta ordenación para cada llamada recursiva, tendremos $O(n \log n)$ trabajo adicional: esto da un algoritmo $O(n \log^2 n)$.

Figura 10.35 A lo más ocho puntos caben en el rectángulo; hay dos coordenadas compartidas por cada dos puntos



lo cual no es del todo malo, en especial cuando se compara con el $O(n^2)$ de fuerza bruta. Sin embargo, no es difícil reducir el trabajo para cada llamada recursiva a $O(n)$, asegurando así un algoritmo $O(n \log n)$.

Mantendremos dos listas. Una es la lista de puntos ordenada según la coordenada x , y la otra es la lista de puntos ordenada según la coordenada y . Llamaremos a estas listas P y Q , respectivamente. Se pueden obtener por medio de un paso de preprocessamiento de ordenación a un costo de $O(n \log n)$ y así no se afecta la cota de tiempo. P_l y Q_l son las listas pasadas a la llamada recursiva de la mitad izquierda, y P_d y Q_d son las listas pasadas a la llamada recursiva de la mitad derecha. Ya hemos visto que P se parte a la mitad fácilmente. Una vez que se conoce la línea divisoria, se va secuencialmente sobre Q , colocando cada elemento en Q_l o Q_d , según sea adecuado. Es fácil ver que Q_l y Q_d automáticamente se ordenarán de acuerdo con la coordenada y . Cuando regresen las llamadas recursivas, recorreremos la lista Q y descartaremos todos los puntos cuyas coordenadas x no estén en la franja. Ahora Q sólo contiene puntos en la franja, y se garantiza que esos puntos están ordenados según sus coordenadas y .

Esta estrategia asegura que el algoritmo completo es $O(n \log n)$, porque sólo se efectúa $O(n)$ trabajo adicional.

10.2.3. El problema de la selección

El problema de la selección requiere encontrar el k -ésimo menor elemento en una lista S de n números. El caso especial de encontrar la mediana es de interés particular. Esto ocurre cuando $k = \lceil n/2 \rceil$.

En los capítulos 1, 6 y 7 vimos varias soluciones para el problema de la selección. La del capítulo 7 usa una variación de la ordenación rápida y se ejecuta en tiempo medio $O(n)$. De hecho, está descrita en el artículo original de Hoare sobre la ordenación rápida.

Aunque este algoritmo se ejecuta en tiempo medio lineal, tiene un peor caso $O(n^2)$. Para el peor caso la selección puede resolverse en tiempo $O(n \log n)$ fácilmente, ordenando los elementos, pero por mucho tiempo no se supo si la selección se podría lograr en un tiempo $O(n)$ para el peor caso. El algoritmo *selección_rápida* planteado en la sección 7.7.6 es bastante eficiente en la práctica, así que era una cuestión de interés teórico, sobre todo.

Recordaremos que el algoritmo básico es una simple estrategia recursiva. Suponiendo que n es mayor que el punto de corte donde los elementos están ordenados simplemente, se elige un elemento v , denominado pivote. Los elementos restantes se colocan en dos conjuntos, S_1 y S_2 . S_1 contiene elementos que es seguro que no son mayores que v , y S_2 contiene elementos que no son menores que v . Por último, si $k \leq |S_1|$, entonces el k -ésimo elemento menor en S puede encontrarse recursivamente calculando el k -ésimo elemento menor en S_1 . Si $k = |S_1| + 1$, el pivote es el k -ésimo elemento menor. Si no es así, el k -ésimo elemento menor en S es el $(k - |S_1| - 1)$ -ésimo elemento menor en S_2 . La diferencia principal entre este algoritmo y la ordenación rápida es que sólo hay un subproblema por resolver, en vez de dos.

A fin de obtener un algoritmo lineal, debemos asegurar que el subproblema es sólo una fracción del original y no sólo unos pocos elementos más pequeño que el

original. Por supuesto, siempre podemos encontrar un elemento si estamos dispuestos a perder algo de tiempo en ello. La dificultad del problema es que no se puede consumir demasiado tiempo en encontrar el pivote.

Para la ordenación rápida, vimos que una buena elección del pivote era tomar tres elementos y usar su mediana. Esto da alguna expectativa de que el pivote no sea demasiado malo, pero no es una garantía. Podríamos elegir 21 elementos al azar, ordenarlos en un tiempo constante, usar el décimoprimer como pivote y obtener un pivote que muy probablemente será mejor. Sin embargo, si esos 21 elementos fueran los 21 mayores, entonces el pivote seguiría siendo el peor. Extendiendo esto, podríamos usar hasta $O(n/\log n)$ elementos, ordenarlos usando una ordenación por montículo en un tiempo total de $O(n)$, y estar casi seguros, desde el punto de vista estadístico, de obtener un buen pivote. En el peor caso, sin embargo, esto no funciona porque podríamos elegir los $O(n/\log n)$ elementos mayores, y entonces el pivote sería el $[n - O(n/\log n)]$ -ésimo mayor, que no es una fracción constante de n .

La idea básica aún es útil. En efecto, veremos que podemos usar esto para mejorar el número esperado de comparaciones que hace la selección rápida. Para obtener un buen peor caso, no obstante, la idea clave es emplear un nivel de indirección más. En vez de buscar la mediana de una muestra de elementos aleatorios, buscaremos la mediana de una *muestra de medianas*.

El algoritmo básico de selección del pivote es como sigue:

1. Acomodar los n elementos en $\lfloor n/5 \rfloor$ grupos de 5 elementos, ignorando los (como máximo cuatro) elementos adicionales.
2. Encontrar la mediana de cada grupo. Esto da una lista M de $\lfloor n/5 \rfloor$ medianas.
3. Encontrar la mediana de M . Devolver esto como el pivote v .

Nos valdremos del término *partición con base en la mediana de la mediana de cinco* para describir el algoritmo de selección rápida que usa la regla de selección de pivote antes dada. Ahora demostraremos que la partición con base en la mediana de la mediana de cinco garantiza que cada subproblema sea como máximo casi un 70% tan grande como el problema original. También demostraremos que el pivote se puede calcular con la suficiente rapidez como para garantizar un tiempo de ejecución $O(n)$ para todo el algoritmo de la selección.

Por el momento, supongamos que n es divisible entre 5, así que no hay elementos adicionales. Supongamos también que $n/5$ es impar para que el conjunto M contenga un número impar de elementos. Como veremos, esto proporciona alguna simetría. Así es que estamos suponiendo, por conveniencia, que n es de la forma $10k + 5$. También supondremos que todos los elementos son diferentes. El algoritmo real debe asegurar el manejo del caso en que esto no sea cierto. La figura 10.36 muestra cómo elegir el pivote cuando $n = 45$.

En la figura 10.36, v representa el elemento elegido como pivote por el algoritmo. Puesto que v es la mediana de nueve elementos, y estamos suponiendo que todos los elementos son diferentes, debe haber cuatro medianas mayores que v y cuatro menores. Se les denota con G y P , respectivamente. Consideremos un grupo de cinco elementos con una mediana grande (tipo G). La mediana del grupo es menor que dos elementos del grupo y mayor que dos elementos del grupo.

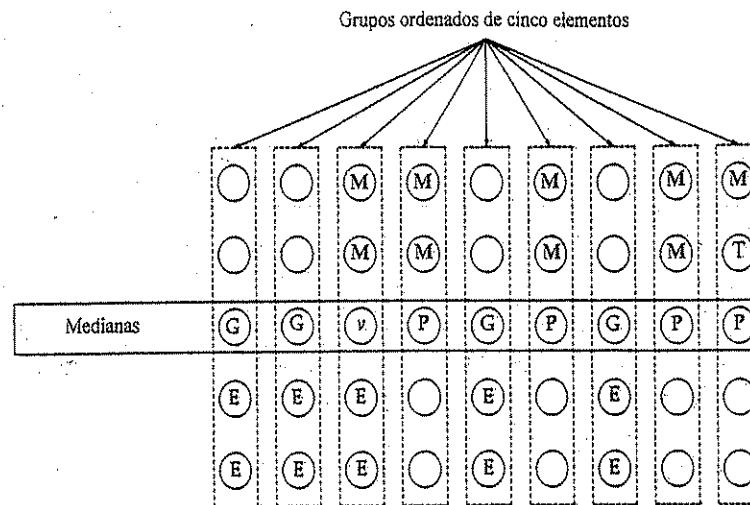


Figura 10.36 Cómo se elige el pivote

E representará a los elementos *enormes*. Estos son elementos de los que sabemos que son mayores que una mediana grande. De manera semejante, M representa los elementos *minúsculos*, los cuales son menores que una mediana pequeña (P). Hay 10 elementos de tipo E : dos están en cada uno de los grupos con una mediana tipo G , y dos están en el mismo grupo que v . Similarmente, hay 10 elementos del tipo M .

Con seguridad los elementos de tipo G o E son mayores que v , y los elementos de tipo P o M son menores que v . Así, es seguro que hay 14 elementos grandes y 14 pequeños en nuestro problema. Por lo tanto, una llamada recursiva podría hacerse en $45 - 14 - 1 = 30$ elementos, como máximo.

Extendamos este análisis a una n general de la forma $10k + 5$. En este caso, hay k elementos de tipo G y k elementos de tipo P . Hay $2k + 2$ elementos de tipo E , y también $2k + 2$ elementos de tipo M . Así, hay $3k + 2$ elementos que se garantiza que son mayores que v y $3k + 2$ elementos que es seguro que son menores. Así, en este caso, la llamada recursiva puede contener a lo más $7k + 2 < 0.7n$ elementos. Si n no es de la forma $10k + 5$, se pueden utilizar argumentos semejantes sin afectar el resultado básico.

Queda por acotar el tiempo de ejecución para obtener el elemento pivote. Hay dos pasos básicos. Podemos encontrar la mediana de cinco elementos en tiempo constante. Por ejemplo, no es difícil clasificar cinco elementos en ocho comparaciones. Debemos hacer esto $\lfloor n/5 \rfloor$ veces, así que este paso tarda un tiempo $O(n)$. Luego tenemos que calcular la mediana de un grupo de $\lfloor n/5 \rfloor$ elementos. La forma obvia de hacerlo es ordenar el grupo y devolver el elemento del medio. Pero esto lleva un tiempo $O(\lfloor n/5 \rfloor \log \lfloor n/5 \rfloor) = O(n \log n)$, así que no funciona. La solución es llamar recursivamente al algoritmo de selección sobre los $\lfloor n/5 \rfloor$ elementos.

Esto completa la descripción del algoritmo básico. Aún hay algunos detalles que necesitan cumplirse si se desea una implantación real. Por ejemplo, los duplicados deben ser manejados adecuadamente, y el algoritmo necesita un corte sufi-

cientemente grande para asegurar que las llamadas recursivas progresen. Hay una gran cantidad de sobrecarga implicada, y este algoritmo no es práctico en absoluto, así que no describiremos nada más que los detalles que hay que considerar. Incluso, desde un punto de vista teórico, el algoritmo es un gran avance, porque, como demuestra el siguiente teorema, el tiempo de ejecución es lineal para el peor caso.

TEOREMA 10.9.

El tiempo de ejecución de la selección rápida usando la partición con base en la mediana de la mediana de cinco es $O(n)$.

DEMOSTRACIÓN:

El algoritmo consta de dos llamadas recursivas de tamaño $0.7n$ y $0.2n$, más el trabajo lineal adicional. Por el teorema 10.8, el tiempo de ejecución es lineal.

Reducción del número promedio de comparaciones

La estrategia de "divide y vencerás" puede usarse también para reducir el número esperado de comparaciones requeridas por el algoritmo de selección. Veamos un ejemplo concreto. Supongamos que se tiene un grupo S de 1000 números y se busca el centésimo número menor, al que llamaremos x . escogemos un subconjunto S' de S consistente en 100 números. Esperaríamos que el valor de x sea semejante en tamaño al décimo elemento de S' . Más específicamente, el quinto número menor en S' es casi seguramente menor que x , y el décimoquinto número menor de S' es casi seguramente mayor que x .

Más generalmente, se escoge una muestra S' de s elementos de los n elementos. Sea algún número, el cual elegiremos para minimizar el número medio de comparaciones usadas por el procedimiento. Buscamos los elementos ($v_1 = ks/n - \delta$ -ésimo y ($v_2 = ks/n + \delta$ -ésimo en S'). Casi seguramente, el k -ésimo elemento menor en S caerá entre v_1 y v_2 , de tal modo que nos quedará un problema de selección sobre 2 elementos. Con poca probabilidad, el k -ésimo elemento menor no cae en este intervalo, y se tiene un trabajo considerable por hacer. No obstante, con una buena elección de s y δ , se puede asegurar, por las leyes de la probabilidad, que el segundo caso no afecta adversamente el trabajo total.

Si se efectúa un análisis, encontramos que si $s = n^{2/3} \log^{1/3} n$ y $\delta = n^{1/3} \log^{2/3} n$, entonces el número esperado de comparaciones es $n + k + O(n^{2/3} \log^{1/3} n)$, que es óptimo excepto para los términos de orden menor. (Si $k > n/2$, podemos considerar el problema simétrico de encontrar el $(n-k)$ -ésimo elemento mayor.)

La mayor parte del análisis es fácil de hacer. El último término representa el costo de realizar las dos selecciones para determinar v_1 y v_2 . El costo medio de la partición, suponiendo una estrategia razonablemente inteligente, es igual a n más el rango esperado de v_2 en S , el cual es $n + k + O(n\delta/s)$. Si el k -ésimo elemento cae en S' , el costo de terminar el algoritmo es igual al costo de la selección sobre S' , a saber, $O(s)$. Si el k -ésimo elemento menor no cae en S' , el costo es $O(n)$. Sin embargo, s y δ se eligieron para garantizar que esto ocurra con una probabilidad muy escasa $o(1/n)$, así que el costo esperado de esta posibilidad es $o(1)$, que es un término que tiende a cero conforme crece n . El cálculo exacto se deja como ejercicio (10.21).

Este análisis muestra que para encontrar la mediana se requieren cerca de $1.5n$ comparaciones en promedio. Por supuesto, este algoritmo requiere algo de aritmética de punto flotante para calcular s , lo cual en algunas máquinas puede reducir la velocidad del algoritmo. Aun así, los experimentos han demostrado que, si se implementa correctamente, este algoritmo se compara favorablemente con la implantación de la selección rápida del capítulo 7.

10.2.4. Mejoras teóricas para problemas de aritmética

En esta sección describiremos un algoritmo de "divide y vencerás" que multiplica dos números de n dígitos. Nuestro modelo de computación previo suponía que la multiplicación se hacia en tiempo constante, porque los números eran pequeños. Para números grandes, esta suposición deja de ser válida. Si la multiplicación se mide en términos del tamaño de los números por multiplicar, el algoritmo de multiplicación natural toma un tiempo cuadrático. El algoritmo de "divide y vencerás" se ejecuta en tiempo subcuadrático. También presentaremos el algoritmo clásico de "divide y vencerás" que multiplica dos matrices n por n en tiempo subcúbico.

Multiplicación de enteros

Supongamos que se quiere multiplicar dos números x y y de n dígitos. Si exactamente uno de x y y es negativo, entonces la respuesta es negativa; en otro caso es positiva. Así, podemos hacer esta comprobación y después suponer que $x, y \geq 0$. El algoritmo que casi todos usan al multiplicar manualmente requiere $\Theta(n^2)$ operaciones porque cada dígito de x se multiplica por cada dígito de y .

Si $x = 61\ 438\ 521$ y $y = 94\ 736\ 407$, $xy = 5\ 820\ 464\ 730\ 934\ 047$. Partamos x y y en dos mitades, que contengan los dígitos más y menos significativos, respectivamente. Entonces $x_i = 6\ 143$, $x_d = 8\ 521$, $y_i = 9\ 473$ y $y_d = 6\ 407$. También tenemos $x = x_i \cdot 10^4 + x_d$ y $y = y_i \cdot 10^4 + y_d$. Se infiere que

$$xy = x_i y_i \cdot 10^8 + (x_i x_d + x_d y_i) \cdot 10^4 + x_d y_d$$

Observe que esta ecuación consta de cuatro multiplicaciones: $x_i y_i$, $x_i y_d$, $x_d y_i$ y $x_d y_d$, cada una de las cuales tiene la mitad del tamaño del problema original ($n/2$ dígitos). Las multiplicaciones por 10^8 y 10^4 se reducen a la colocación de ceros. Esto y las sumas subsecuentes agregan sólo un trabajo adicional $O(n)$. Si efectuamos estas multiplicaciones recursivamente con este algoritmo, parando en un caso base adecuado, entonces obtenemos la recurrencia

$$T(n) = 4T(n/2) + O(n)$$

Del teorema 10.6 veremos que $T(n) = O(n^2)$, así que desafortunadamente no hemos mejorado el algoritmo. Para obtener un algoritmo subcuadrático debemos usar menos de cuatro llamadas recursivas. La observación clave es que

$$x_i y_d + x_d y_i = (x_i - x_d)(y_d - y_i) + x_i y_i + x_d y_d$$

Así, en vez de usar dos multiplicaciones para calcular el coeficiente de 10^4 , podemos utilizar una multiplicación, más el resultado de dos multiplicaciones que ya se han realizado. La figura 10.37 muestra cómo sólo hace falta resolver tres subproblemas recursivos.

Es fácil ver que ahora la ecuación de recurrencia satisface

$$T(n) = 3T(n/2) + O(n)$$

y obtenemos $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$. Para completar el algoritmo, se debe tener un caso base que se pueda resolver sin recursión.

Cuando ambos números son de un dígito, podemos hacer la multiplicación usando una tabla. Si un número tiene cero dígitos, se devuelve cero. En la práctica, si fuéramos a usar este algoritmo, escogeríamos el caso base más conveniente para la máquina.

Aunque este algoritmo tiene mejor comportamiento asintótico que el algoritmo cuadrático estándar, rara vez se usa porque para n pequeña la sobrecarga es significativa, y para n grandes hay mejores algoritmos. Estos algoritmos también hacen uso amplio de la estrategia de "divide y vencerás".

Multiplicación de matrices

Un problema numérico fundamental es la multiplicación de dos matrices. La figura 10.38 da un algoritmo simple $O(n^3)$ para calcular $C = AB$, donde A , B y C son

Figura 10.37 El algoritmo "divide y vencerás" en acción

Función	Valor	Complejidad computacional
x_i	6 143	dada
x_d	8 521	dada
y_i	9 473	dada
y_d	6 407	dada
$d_1 = x_i \cdot x_d$	-2 378	$O(n)$
$d_2 = y_i \cdot y_d$	-3 066	$O(n)$
$x_i y_i$	58 192 639	$T(n/2)$
$x_d y_d$	54 594 047	$T(n/2)$
$d_1 d_2$	7 290 948	$T(n/2)$
$d_3 = d_1 d_2 + x_i y_i + x_d y_d$	120 077 634	$O(n)$
$x_d y_d$	54 594 047	Calculado arriba
$d_3 10^4$	1 200 776 340 000	$O(n)$
$x_i y_i 10^8$	5 819 263 900 000 000	$O(n)$
$x_i y_i 10^8 + d_3 10^4 + x_d y_d$	5 820 464 730 934 047	$O(n)$

matrices $n \times n$. El algoritmo se infiere directamente de definición de la multiplicación de matrices. Para calcular $C_{i,j}$, se obtiene el producto escalar de la i -ésima fila de A con la j -ésima columna de B .

Por largo tiempo se creyó que para multiplicar matrices se requería $\Omega(n^3)$. Sin embargo, a finales de los años sesenta Strassen mostró cómo romper la barrera de $\Omega(n^3)$. La idea básica del algoritmo de Strassen consiste en dividir cada matriz en cuatro cuadrantes, como se muestra en la figura 10.39. Entonces es fácil demostrar que

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Figura 10.38 Multiplicación simple de matrices $O(n^3)$

```
procedure mult_matrices(A; B: matriz; var C: matriz; n: integer);
  var i, j, k: integer;
begin
  for i:=1 to n do (iniciación)
    for j:=1 to n do
      C[i,j] := 0.0;
  for i:=1 to n do
    for j:=1 to n do
      for k:=1 to n do
        C[i,j] := C[i,j] + A[i,k]*B[k,j];
end;
```

Figura 10.39 Descomposición de $AB = C$ en cuatro cuadrantes

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Por ejemplo, para realizar la multiplicación AB

$$AB = \begin{bmatrix} 3 & 4 & 1 & 6 \\ 1 & 2 & 5 & 7 \\ 5 & 1 & 2 & 9 \\ 4 & 3 & 5 & 6 \end{bmatrix} \begin{bmatrix} 5 & 6 & 9 & 3 \\ 4 & 5 & 3 & 1 \\ 1 & 1 & 8 & 4 \\ 3 & 1 & 4 & 1 \end{bmatrix}$$

definimos las siguientes ocho matrices $n/2$ por $n/2$:

$$A_{1,1} = \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} \quad A_{1,2} = \begin{bmatrix} 1 & 6 \\ 5 & 7 \end{bmatrix} \quad B_{1,1} = \begin{bmatrix} 5 & 6 \\ 4 & 5 \end{bmatrix} \quad B_{1,2} = \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix}$$

$$A_{2,1} = \begin{bmatrix} 5 & 1 \\ 4 & 3 \end{bmatrix} \quad A_{2,2} = \begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix} \quad B_{2,1} = \begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix} \quad B_{2,2} = \begin{bmatrix} 8 & 4 \\ 4 & 1 \end{bmatrix}$$

Entonces podríamos efectuar ocho multiplicaciones de matrices $n/2$ por $n/2$ y cuatro sumas de matrices $n/2$ por $n/2$. Las sumas de matrices tardan un tiempo $O(n^2)$. Si la multiplicación de matrices se hace recursivamente, entonces el tiempo de ejecución satisface

$$T(n) = 8T(n/2) + O(n^2)$$

Del teorema 10.6, se ve que $T(n) = O(n^3)$, así que no hemos logrado ninguna mejoría. Como se vio con la multiplicación de enteros, debemos reducir el número de subproblemas por debajo de ocho. Strassen se valió de una estrategia semejante a la del algoritmo de "divide y vencerás" de la multiplicación de enteros y mostró cómo usar sólo siete llamadas recursivas arreglando con cuidado los cálculos. Las siete multiplicaciones son

$$M_1 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

$$M_2 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

$$M_3 = (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2})$$

$$M_4 = (A_{1,1} + A_{1,2})B_{2,2}$$

$$M_5 = A_{1,1}(B_{1,2} - B_{2,2})$$

$$M_6 = A_{2,2}(B_{2,1} - B_{1,1})$$

$$M_7 = (A_{2,1} + A_{2,2})B_{1,1}$$

Una vez realizadas las multiplicaciones, la respuesta final se puede obtener con ocho sumas más.

$$C_{1,1} = M_1 + M_2 - M_4 + M_6$$

$$C_{1,2} = M_4 + M_5$$

$$C_{1,3} = M_6 + M_7$$

$$C_{1,4} = M_2 - M_3 + M_5 - M_7$$

Es directo verificar que esta ingeniosa ordenación produce los valores deseados. Ahora el tiempo de ejecución satisface la recurrencia

$$T(n) = 7T(n/2) + O(n^2)$$

La solución de esta recurrencia es $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

Como es usual, hay detalles por considerar, como el caso cuando n no es una potencia de dos, pero en realidad se trata de minúsculas. El algoritmo de Strassen es

peor que el algoritmo directo hasta que n llega a ser muy grande. No se generaliza para el caso en que las matrices son muy dispersas (con muchos elementos cero), y no se paralleliza fácilmente. Cuando se ejecuta con entradas de punto flotante, es menos estable numéricamente que el algoritmo clásico. Así, sólo tiene una aplicabilidad limitada. Sin embargo, representa un importante hito teórico y ciertamente demuestra que en informática, como en muchos otros campos, aun cuando un problema parezca tener una complejidad intrínseca, no podremos estar seguros de nada hasta comprobarlo.

10.3. Programación dinámica

En la sección anterior, vimos que un problema que se puede expresar recursivamente en términos matemáticos también se puede expresar como un algoritmo recursivo, en muchos casos llegando a una mejora significativa en el rendimiento sobre una más ingenua búsqueda exhaustiva.

Cualquier fórmula matemática recursiva se podría traducir directamente a un algoritmo recursivo, pero la realidad subyacente es que con frecuencia el compilador no hará justicia al algoritmo recursivo, produciendo un programa inefficiente. Cuando se sospecha que es probable un caso así, debemos ayudar un poco al compilador reescribiendo el algoritmo recursivo como un algoritmo no recursivo que sistemáticamente registre las respuestas de los subproblemas en una tabla. Una técnica que aprovecha este enfoque se llama *programación dinámica*.

10.3.1 Uso de una tabla en vez de la recursión

En el capítulo 2 vimos que el programa recursivo natural para calcular los números de Fibonacci es muy inefficiente. Recordemos que el programa mostrado en la figura 10.40 tiene un tiempo de ejecución $T(n)$ que satisface $T(n) \geq T(n-1) + T(n-2)$. Puesto que $T(n)$ satisface la misma relación de recurrencia que los números de Fibonacci y tiene las mismas condiciones iniciales, de hecho $T(n)$ crece a la misma velocidad que los números de Fibonacci, es decir, es exponencial.

Figura 10.40 Algoritmo inefficiente para calcular los números de Fibonacci

{Calcula los números de Fibonacci como se describieron en el capítulo 1}

{Se supone $n \geq 0$ }

```
function fib(n: integer): integer;
begin
  (1) if (n = 0) or (n = 1) then
  (2)   fib := 1
  else
    fib := fib(n-1) + fib(n-2);
  end;
```

```

function fibonacci(n: integer): integer;
var i, últ, penúlt, respuesta: integer;
begin
  if (n = 0) or (n = 1) then
    respuesta := 1
  else
    begin
      últ := 1; penúlt := 1;
      for i := 2 to n do
        begin
          respuesta := últ + penúlt;
          penúlt := últ;
          últ := respuesta;
        end;
      end;
      fibonacci := respuesta;
    end;
end;

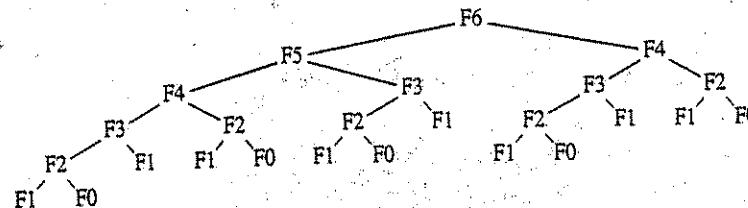
```

Figura 10.41 Algoritmo lineal para calcular números de Fibonacci

Por otro lado, como para calcular F_n , todo lo que se necesita es F_{n-1} y F_{n-2} , sólo hay que registrar los dos números de Fibonacci calculados más recientemente. Esto produce el algoritmo $O(n)$ de la figura 10.41.

La razón de que el algoritmo recursivo sea tan lento radica en el algoritmo usado para simular la recursión. Para calcular F_n , hay una llamada a F_{n-1} y F_{n-2} . No obstante, puesto que F_{n-1} hace una llamada recursiva a F_{n-2} y F_{n-3} , hay realmente dos llamadas distintas para calcular F_{n-2} . Si hacemos un seguimiento del algoritmo completo, podemos ver que F_{n-3} se calcula tres veces, F_{n-4} cinco, F_{n-5} ocho, y así sucesivamente. Como se muestra en la figura 10.42, el crecimiento de cálculos redundantes es explosivo. Si el algoritmo de simulación de la recursión fuera capaz de mantener una lista de todos los valores calculados previamente y de no hacer llamadas recursivas para un subproblema ya resuelto, se evitaría esta explosión exponencial. Ésta es la razón de que el programa de la figura 10.41 sea mucho más eficiente.

Figura 10.42 Seguimiento del cálculo recursivo de los números de Fibonacci



```

function eval(n: integer): real;
var i: integer; suma: real;
begin
  if n = 0 then
    eval := 1.0
  else
    begin
      suma := 0.0;
      for i:=0 to n-1 do
        suma := suma + eval(i);
      eval := 2.0 * suma / n + n;
    end;
end;

```

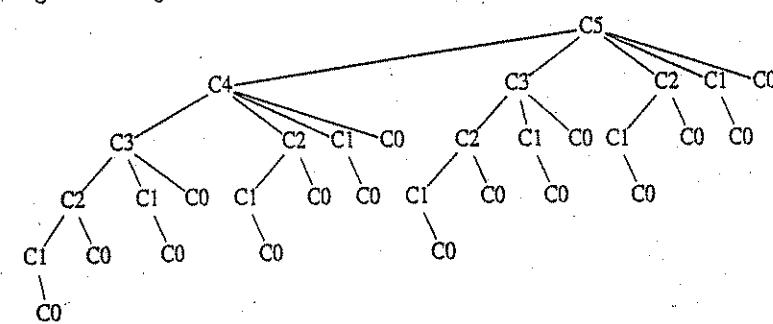
Figura 10.43 Programa recursivo para evaluar

$$C(n) = 2/n \sum_{i=0}^{n-1} C(i) + n$$

Como segundo ejemplo, en el capítulo 7 vimos cómo resolver la recurrencia $C(n) = 2/n \sum_{i=0}^{n-1} C(i) + n$, con $C(0) = 1$. Supongamos que se quiere comprobar, numéricamente, si la solución obtenida es correcta. Podríamos escribir el sencillo programa de la figura 10.43 para evaluar la recursión.

Una vez más, las llamadas recursivas duplican el trabajo. En este caso, el tiempo de ejecución $T(n)$ satisface $T(n) = \sum_{i=0}^{n-1} T(i) + n$ porque, como se mostró en la figura 10.44, hay una llamada recursiva (directa) de cada tamaño desde 0 hasta $n - 1$, más $O(n)$ trabajo adicional (¿dónde más hemos visto el árbol que aparece en la figura 10.44?). Resolviendo $T(n)$ encontramos que crece exponencialmente. Usando una tabla, obtenemos el programa de la figura 10.45. Este programa evita las llamadas recursivas redundantes y se ejecuta en tiempo $O(n^2)$. No es un programa perfecto; como ejercicio, haga el cambio sencillo que reduce su tiempo de ejecución a $O(n)$.

Figura 10.44 Seguimiento del cálculo recursivo en eval



```

function eval(n: integer): real; {se supone n <= MÁX}
var c: array [0..MÁX] of real;
    suma: real;
    i, j: integer;
begin
  if n = 0 then
    c[0] := 1.0
  else
    begin
      for i:=1 to n do {evalúa Ci, 1 ≤ i ≤ n}
      begin
        suma := 0.0;
        for j := 0 to i - 1 do {evalúa  $\sum_{j=0}^{i-1} C_j$ }
          suma := suma + c[j];
        c[i] := 2.0 * suma / i + i;
      end;
    end;
  eval := c[n];
end;

```

Figura 10.45 Evaluación de $C(n) = 2/n \sum_{i=0}^{n-1} C(i) + n$
con una tabla

10.3.2. Ordenación de multiplicaciones de matrices

Supongamos cuatro matrices, A, B, C y D, de dimensiones $A = 50 \times 10$, $B = 10 \times 40$, $C = 40 \times 30$ y $D = 30 \times 5$. Aunque la multiplicación de matrices no es conmutativa, es asociativa, lo cual significa que el producto de las matrices ABCD se puede poner entre paréntesis, y por tanto evaluar, en cualquier orden. La forma obvia de multiplicar dos matrices de dimensiones $p \times q$ y $q \times r$, respectivamente, usa multiplicaciones escalares pqr . (El uso de un algoritmo teórico superior como el algoritmo de Strassen no altera significativamente el problema que vamos a considerar, así que supondremos esta cota en el rendimiento.) ¿Cuál es la mejor forma de realizar las tres multiplicaciones de matrices requeridas para obtener ABCD?

En el caso de cuatro matrices, es sencillo resolver el problema por medio de una búsqueda exhaustiva, ya que hay sólo cinco formas de ordenar las multiplicaciones. A continuación evaluamos cada caso:

- $((AB)CD)$: La evaluación de BC requiere $10 \times 40 \times 30 = 12\,000$ multiplicaciones. La evaluación de $(BC)D$ requiere las 12 000 multiplicaciones para calcular BC, más $10 \times 30 \times 5 = 1\,500$ multiplicaciones adicionales, para un total de 13 500. La evaluación de $((AB)CD)$ requiere 13 500 multiplicaciones para $(BC)D$, más $50 \times 10 \times 5 = 2\,500$ multiplicaciones adicionales, para un total de 16 000 multiplicaciones.
- $(A(B(CD)))$: La evaluación de CD requiere $40 \times 30 \times 5 = 6\,000$ multiplicaciones. La evaluación de $B(CD)$ requiere las 6 000 multiplicaciones para calcular

CD, más $10 \times 40 \times 5 = 2\,000$ multiplicaciones adicionales, para un total de 8 000. La evaluación de $(A(B(CD)))$ requiere 8 000 multiplicaciones de B(CD), más $50 \times 10 \times 5 = 2\,500$ multiplicaciones adicionales, para un gran total de 10 500 multiplicaciones.

- $((AB)(CD))$: La evaluación de CD requiere $40 \times 30 \times 5 = 6\,000$ multiplicaciones. La evaluación de AB requiere $50 \times 10 \times 40 = 20\,000$ multiplicaciones. La evaluación de $((AB)(CD))$ requiere 6 000 multiplicaciones para CD, 20 000 para AB, más $50 \times 40 \times 5 = 10\,000$ multiplicaciones adicionales, para un gran total de 36 000 multiplicaciones.
- $((AB)C)D$: La evaluación de AB requiere $50 \times 10 \times 40 = 20\,000$ multiplicaciones. La evaluación de $(AB)C$ requiere 20 000 multiplicaciones para calcular AB, más $50 \times 40 \times 30 = 60\,000$ multiplicaciones adicionales, para un total de 80 000. La evaluación de $((AB)C)D$ requiere 80 000 multiplicaciones para $(AB)C$, más $50 \times 30 \times 5 = 7\,500$ multiplicaciones adicionales, para un total de 87 500 multiplicaciones.
- $((A(BC))D)$: La evaluación de BC requiere $10 \times 40 \times 30 = 12\,000$ multiplicaciones. La evaluación de $A(BC)$ requiere las 12 000 multiplicaciones para calcular BC, más $50 \times 10 \times 30 = 15\,000$ multiplicaciones adicionales, para un total de 27 000. La evaluación de $((A(BC))D)$ requiere 27 000 multiplicaciones para $A(BC)$, más $50 \times 30 \times 5 = 7\,500$ multiplicaciones adicionales, para un total de 34 500 multiplicaciones.

Los cálculos muestran que la mejor ordenación usa aproximadamente un noveno del número de multiplicaciones de la peor ordenación. Así, podría valer la pena efectuar unos cuantos cálculos para determinar la ordenación óptima. Desafortunadamente, ninguna de las estrategias ávidas obvias parece funcionar. Además, el número de ordenaciones posibles crece con rapidez. Supongamos que se define $T(n)$ como este número. Entonces $T(1) = T(2) = 1$, $T(3) = 2$ y $T(4) = 5$, como hemos visto. En general,

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i).$$

Para ver esto, supongamos que las matrices son A_1, A_2, \dots, A_n , y la última multiplicación realizada es $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$. Entonces hay $T(i)$ formas de calcular $(A_1 A_2 \cdots A_i)$ y $T(n-i)$ formas de calcular $(A_{i+1} A_{i+2} \cdots A_n)$. Así, hay $T(i)T(n-i)$ formas de calcular $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$ para cada posible i .

La solución de esta recurrencia son los bien conocidos números catalanes, los cuales crecen en forma exponencial. Así, para n grande, una búsqueda exhaustiva a través de todas las ordenaciones posibles es inútil. Sin embargo, este argumento de conteo da la base para una solución sustancialmente mejor que la exponencial. Sea c_i el número de columnas en la matriz A_i para $1 \leq i \leq n$. Entonces A_i tiene c_{i-1} filas, ya que de otro modo las multiplicaciones no son válidas. Definiremos c_0 como el número de filas en la primera matriz, A_1 .

Supongamos que m_{i_2, D_r} es el número de multiplicaciones requeridas para multiplicar $A_{i_2} A_{i_2+1} \cdots A_{D_r-1} A_{D_r}$. Por consistencia, $m_{i_2, i_2} = 0$. Supóngase que la última

multiplicación es $(A_{l_{izq}} \cdots A_i)(A_{i+1} \cdots A_{Der})$, donde $l_{izq} \leq i \leq Der$. Entonces el número de multiplicaciones usadas es $m_{l_{izq},i} + m_{i+1,Der} + c_{l_{izq}-1}c_ic_{Der}$. Estos tres términos representan las multiplicaciones requeridas para calcular $(A_{l_{izq}} \cdots A_i)(A_{i+1} \cdots A_{Der})$, y su producto, respectivamente.

Si definimos $M_{l_{izq},Der}$ como el número de multiplicaciones requeridas en una ordenación óptima, entonces, si $l_{izq} < Der$,

$$M_{l_{izq},Der} = \min_{l_{izq} \leq i < Der} \{ M_{l_{izq},i} + M_{i+1,Der} + c_{l_{izq}-1}c_ic_{Der} \}$$

Esta ecuación implica que si se tiene un arreglo óptimo de multiplicaciones de $A_{l_{izq}} \cdots A_{Der}$, los subproblemas $A_{l_{izq}} \cdots A_i$ y $A_{i+1} \cdots A_{Der}$ no pueden realizarse subóptimamente. Esto debe estar claro, ya que de otra forma podríamos mejorar el resultado completo sustituyendo el cálculo subóptimo por uno óptimo.

La fórmula se traduce directamente en un programa recursivo, pero, como vimos en la última sección, tal programa sería evidentemente ineficiente. Sin embargo, como sólo hay aproximadamente $n^2/2$ valores de $M_{l_{izq},Der}$ que siempre necesitan calcularse, está claro que para almacenar esos valores se puede usar una tabla. Un examen posterior muestra que si $Der - l_{izq} = k$, entonces sólo los valores $M_{x,y}$ que son necesarios en el cálculo de $M_{l_{izq},Der}$ satisfacen $y - x < k$. Esto no indica el orden en el cual necesitamos calcular la tabla.

Si queremos visualizar la ordenación real de las multiplicaciones además de la respuesta final $M_{1,n}$, podemos usar las ideas de los algoritmos sobre los caminos más cortos del capítulo 9. Siempre que se haga un cambio a $M_{l_{izq},Der}$, registramos el valor de i que es responsable. Esto da el sencillo programa de la figura 10.46.

Aunque el énfasis de este capítulo no está en la codificación, cabe señalar que muchos programadores tienden a abbreviar los nombres de las variables a una sola letra. c, i , y k se usan como variables de una sola letra porque esto concuerda con los nombres con que describimos el algoritmo, descripción que es muy matemática. Sin embargo, por lo regular es mejor evitar usar l como nombre de variable, porque "l" se parece en mucho a 1 y puede dificultar la depuración si se comete un error de transcripción.

Regresando a los aspectos algorítmicos, este programa contiene un triple ciclo anidado y es fácil ver que se ejecuta en un tiempo $O(n^3)$. Las referencias describen un algoritmo más rápido, pero como el tiempo para multiplicar matrices es probablemente mucho más grande que el requerido para obtener la ordenación óptima, este algoritmo todavía es bastante práctico.

10.3.3. Árbol binario de búsqueda óptimo

Nuestro segundo ejemplo de programación dinámica considera la siguiente entrada: se tiene una lista de palabras, w_1, w_2, \dots, w_n , y las probabilidades fijas p_1, p_2, \dots, p_n de su ocurrencia. El problema es acomodar esas palabras en un árbol binario de búsqueda a fin de minimizar el tiempo de acceso total esperado. En un árbol binario de búsqueda, el número de comparaciones necesarias para tener acceso a un elemento a profundidad d es $d + 1$, así que si w_i se encuentra a la profundidad d_i , entonces queremos minimizar $\sum_{i=1}^n p_i(1 + d_i)$.

{Calcula la ordenación óptima de la multiplicación de matrices}
 {c contiene el número de columnas de cada una de las n matrices}
 {c[0] es el número de filas de la matriz 1}
 {El número mínimo de multiplicaciones se deja en M[1,n]}
 {La ordenación real se puede obtener vía otro procedimiento usando últ_cambio}

```
procedure matriz_opt(c: arreglo_de_enteros, n: integer; var M,  

                     últ_cambio: arreglo_dos_d);  

  var i, k, Izq, Der, esta_M: integer;  

begin  

  for Izq := 1 to n do  

    M[Izq, Izq] := 0;  

  for k := 1 to n - 1 do {k es Der - Izq}  

    for Izq := 1 to n - k do {para cada posición}  

    begin  

      Der := Izq + k;  

      M[Izq, Der] := MAXINT;  

      for i := Izq to Der - 1 do {calcula min}  

      begin  

        esta_M := M[Izq, i] + M[i+1,Der] + c[Izq - 1]*c[i]*c[Der];  

        if esta_M < M[Izq, Der] then {actualiza min}  

        begin  

          M[Izq, Der] := esta_M;  

          últ_cambio[Izq, Der] := i;  

        end;  

      end;  

    end;  

  end;
```

Figura 10.46 Programa para encontrar la ordenación óptima de las multiplicaciones de matrices

Palabra	Probabilidad
a	0.22
al	0.18
ama	0.20
eso	0.05
si	0.25
sin	0.02
su	0.08

Figura 10.47 Entrada de ejemplo para el problema del árbol binario de búsqueda óptimo

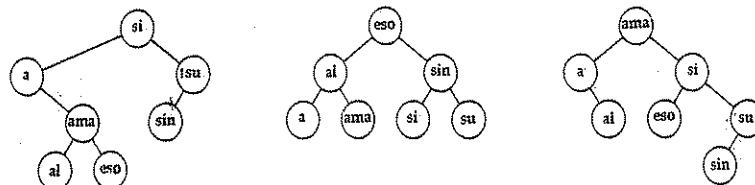


Figura 10.48 Tres posibles árboles binarios de búsqueda para los datos de la tabla anterior

Por ejemplo, la figura 10.47 muestra siete palabras y su probabilidad de ocurrencia en algún contexto. La figura 10.48 presenta tres posibles árboles binarios de búsqueda. Sus costos de búsqueda se muestran en la figura 10.49.

El primer árbol se formó con una estrategia ávida. La palabra con la mayor probabilidad de acceso se colocó en la raíz. Los subárboles izquierdo y derecho se formaron entonces recursivamente. El segundo árbol es un árbol de búsqueda perfectamente equilibrado. Ninguno de ellos es óptimo, según demuestra la existencia de un tercer árbol. De esto podemos ver que ninguna de las soluciones obvias funciona.

Al principio esto es sorprendente porque el problema parece muy semejante a la construcción del árbol de codificación de Huffman, el cual, como ya vimos, se puede resolver por medio de un algoritmo ávido. La construcción de un árbol binario de búsqueda óptimo es más difícil, porque los datos no están restringidos a aparecer sólo en la hojas, y también porque el árbol debe satisfacer la propiedad de árbol binario de búsqueda.

Figura 10.49 Comparación de los tres árboles binarios de búsqueda

Entrada	Árbol #1		Árbol #2		Árbol #3		
	Palabra	Probabilidad	Costo de acceso	Costo de acceso	Costo de acceso	Costo de acceso	
w_i	p_i	Una vez	En secuencia	Una vez	En secuencia	Una vez	En secuencia
a	0.22	2	0.44	3	0.66	2	0.44
al	0.18	4	0.72	2	0.36	3	0.54
ama	0.20	3	0.60	3	0.60	1	0.20
eso	0.05	4	0.20	1	0.05	3	0.15
si	0.25	1	0.25	3	0.75	2	0.50
sin	0.02	3	0.06	2	0.04	4	0.08
su	0.08	2	0.16	3	0.24	3	0.24
Totales		1.00		2.43		2.70	
							2.15

Una solución de programación dinámica se infiere de dos observaciones. Una vez más, supongamos que se intenta colocar las palabras (ordenadas) $w_{Izq}, w_{Izq+1}, \dots, w_{Der-1}, w_{Der}$ en un árbol binario de búsqueda. Supongamos que el árbol binario de búsqueda óptimo tiene a w_i como raíz, donde $Izq \leq i \leq Der$. Entonces el subárbol izquierdo debe contener w_{Izq}, \dots, w_{i-1} , y el subárbol derecho debe contener w_{i+1}, \dots, w_{Der} (por la propiedad del árbol binario de búsqueda). Además, ambos subárboles también deben ser óptimos, pues de otra forma podrían ser sustituidos por subárboles óptimos, lo cual daría una mejor solución para w_{Izq}, \dots, w_{Der} . Así, podemos escribir una fórmula para el costo $C_{Izq,Der}$ de un árbol binario de búsqueda óptimo. La figura 10.50 puede ser útil.

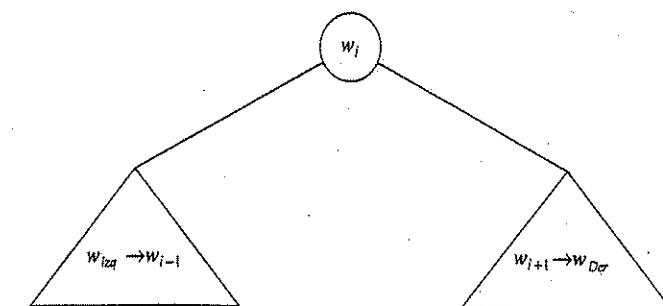
Si $Izq > Der$, el costo del árbol es 0; éste es el caso NIL, que siempre tenemos en árboles binarios de búsqueda. Si no es así, la raíz cuesta p_i . El subárbol izquierdo tiene un costo de $C_{Izq, i-1}$, y el derecho tiene un costo de $C_{i+1,Der}$, relativos a su raíz. Como lo muestra la figura 10.50, cada nodo en esos subárboles es un nivel más profundo desde w_i que desde sus respectivas raíces, así que debemos sumar $\sum_{j=Izq}^{i-1} p_j$ y $\sum_{j=i+1}^{Der} p_j$. Esto da la fórmula

$$\begin{aligned} C_{Izq,Der} &= \min_{Izq \leq i \leq Der} \left\{ p_i + C_{Izq, i-1} + C_{i+1,Der} + \sum_{j=Izq}^{i-1} p_j + \sum_{j=i+1}^{Der} p_j \right\} \\ &= \min_{Izq \leq i \leq Der} \left\{ C_{Izq, i-1} + C_{i+1,Der} + \sum_{j=Izq}^{Der} p_j \right\} \end{aligned}$$

A partir de esta ecuación, es directo escribir un programa que calcule el costo del árbol binario de búsqueda óptimo. Como es usual, el árbol de búsqueda real se puede mantener guardando el valor de i que minimiza $C_{Izq,Der}$. Se puede usar la rutina recursiva estándar para visualizar el árbol real.

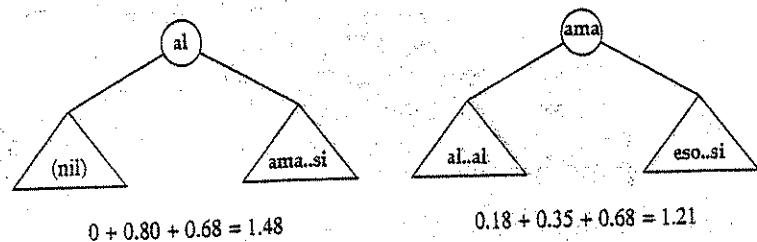
La figura 10.51 muestra la tabla que se producirá con el algoritmo. De cada subintervalo de palabras, se conserva el costo y la raíz del árbol binario de búsqueda óptimo. La entrada de la parte más baja, por supuesto, calcula el árbol binario de

Figura 10.50 Estructura de un árbol binario de búsqueda óptimo



	Izq=1	Izq=2	Izq=3	Izq=4	Izq=5	Izq=6	Izq=7
Iteración = 1	a..a	al..al	ama..ama	eso..eso	si..si	sin..sin	su..su
	.22 a	.18 am	.20 ama	.05 eso	.25 si	.02 sin	.08 su
Iteración = 2	a..al	al..ama	ama..eso	eso..si	si..sin	sin..su	
	.58 a	.56 am	.30 ama	.35 si	.29 si	.12 al	
Iteración = 3	a..ama	al..eso	ama..si	eso..sin	si..su		
	1.02 al	.66 am	.80 si	.39 si	.47 si		
Iteración = 4	a..eso	al..si	ama..sin	eso..su			
	1.17 al	1.21 am	.84 si	.57 si			
Iteración = 5	a..si	al..sin	ama..su				
	1.83 ama	1.27 am	1.02 si				
Iteración = 6	a..sin	al..su					
	1.89 ama	1.53 am					
Iteración = 7	a..su						
	2.15 am						

Figura 10.51 Cálculo del árbol binario de búsqueda óptimo para la entrada de ejemplo



búsqueda óptimo para el conjunto completo de palabras de la entrada. El árbol óptimo es el tercer árbol mostrado en la figura 10.48.

El cálculo preciso del árbol binario de búsqueda óptimo para un subintervalo particular, a saber, *al..si*, se muestra en la figura 10.52. Se obtiene calculando el árbol de costo mínimo resultante de colocar *al*, *ama*, *eso* y *si* en la raíz. Por ejemplo, cuando *ama* se coloca en la raíz, el subárbol izquierdo contiene *al..al* (de costo 0.18, vía un cálculo previo), el subárbol derecho contiene *eso..si* (de costo 0.35), y $p_{al} + p_{ama} + p_{eso} + p_{si} = 0.68$, para un costo total de 1.21.

El tiempo de ejecución de este algoritmo es $O(n^3)$, porque al implantarse se obtiene un triple ciclo. En los ejercicios se bosqueja un algoritmo $O(n^3)$ para el problema.

10.3.4. Camino más corto entre todos los pares

Nuestra tercera y última aplicación de la programación dinámica es un algoritmo para calcular los caminos más cortos ponderados entre cada par de puntos en un grafo dirigido $G = (V, E)$. En el capítulo 9, vimos un algoritmo para el problema del *camino más corto con origen único*, el cual busca el camino más corto de algún vértice arbitrario s a todos los demás. Ese algoritmo (de Dijkstra) se ejecuta en un tiempo $O(|V|^2)$ para grafos densos, pero es sustancialmente más rápido para grafos dispersos. Proporcionaremos un algoritmo corto para resolver el problema de todos los pares para grafos densos. El tiempo de ejecución del algoritmo es $O(|V|^3)$, que no es una mejoría asintótica sobre las $|V|$ iteraciones del algoritmo de Dijkstra, pero podría ser más rápido sobre un grafo muy denso, porque sus ciclos son más ajustados. El algoritmo también se ejecuta correctamente si hay costos de aristas negativos, pero no ciclos de costo negativo; el algoritmo de Dijkstra falla en este caso.

Recordemos los detalles importantes del algoritmo de Dijkstra (el lector puede revisar la sección 9.3). El algoritmo de Dijkstra empieza en un vértice s y funciona en etapas. Tarde o temprano cada vértice del grafo se elige como vértice intermedio. Si el vértice elegido actual es v , entonces para cada $w \in V$, se pone $d_{vw} = \min(d_{vw}, d_v + c_{vw})$. Esta fórmula dice que la mejor distancia a w (desde s) es la distancia antes conocida a w desde s , o bien el resultado de ir de s a v (óptimamente) y entonces directamente de v a w .

El algoritmo de Dijkstra da la idea del algoritmo de programación dinámica: se eligen los vértices en orden secuencial. Definiremos $D_{k,ij}$ como el peso del camino más corto de v_i a v_j que sólo usa a v_1, v_2, \dots, v_k como intermedio. De acuerdo con esta definición, $D_{0,ij} = c_{ij}$, donde $c_{ij} = \infty$ si (v_i, v_j) no es una arista en el grafo. También por la definición, $D_{1,1,1,j}$ es el camino más corto de v_1 a v_j en el grafo.

Como se muestra en la figura 10.53, cuando $k > 0$, podemos escribir una fórmula simple de $D_{k,ij}$. El camino más corto de v_i a v_j que sólo usa v_1, v_2, \dots, v_k como intermedio es el camino más corto que no usa v_k como intermedio, o consiste en la unión de los dos caminos $v_i \rightarrow v_k$ y $v_k \rightarrow v_j$, cada uno de los cuales usa sólo los primeros $k-1$ vértices como intermedios. Esto lleva a la fórmula

$$D_{k,ij} = \min(D_{k-1,ij}, D_{k-1,ik} + D_{k-1,kj})$$

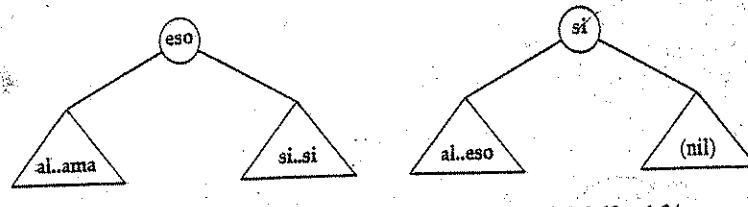


Figura 10.52 Cálculo de la entrada de la tabla (1.21, *ama*) para *al..si*

El requerimiento de tiempo es una vez más $O(|V|^3)$. A diferencia de los dos ejemplos anteriores de programación dinámica, esta cota de tiempo no ha sido reducida sustancialmente por otro enfoque. Puesto que la k -ésima etapa depende sólo de la $(k-1)$ -ésima etapa, parece que sólo se necesita mantener dos matrices $|V| \times |V|$.

No obstante, el uso de k como vértice *intermedio* en un camino que se inicia o finaliza con k no mejora el resultado a menos que haya un ciclo negativo. Así, sólo se necesita una matriz, porque $D_{k-1,i,k} = D_{i,k}$ y $D_{k-1,k,j} = D_{k,k,j}$, lo cual implica que ninguno de los términos de la derecha cambia sus valores ni necesita guardarse. Esta observación conduce al simple programa de la figura 10.53.

En un grafo completo, donde todo par de vértices está conectado (en ambas direcciones), es casi seguro que este algoritmo será más rápido que $|V|$ iteraciones del algoritmo de Dijkstra, porque los ciclos son muy ajustados. Las líneas [1] a [4] se pueden ejecutar en paralelo, igual que las líneas [6] a [10]. Así, este algoritmo parece ser adecuado para cómputo paralelo.

La programación dinámica es una potente técnica de diseño de algoritmos que proporciona un punto de inicio para una solución. En esencia, éste es el paradigma de "divide y vencerás" de resolución de problemas más sencillos primero, con la importante diferencia de que los problemas más sencillos no son una división clara del original. Como los subproblemas se resuelven repetidamente, es importante registrar sus soluciones en una tabla en vez de volver a calcularlos. En algunos casos,

Figura 10.53 Camino más corto entre todos los pares

```

(Calcula los caminos más cortos entre todos los pares)
(A[ ] contiene la matriz de adyacencia)
(D[ ] contiene los valores del camino más corto)
(El camino real se puede calcular vía otro procedimiento usando el camino)

procedure todos_los_pares(A: arreglo_dos; var D, camino: arreglo_dos_d);
  var i, j, k: integer;
begin
  for i := 1 to |V| do {inicia D y camino}
    for j := 1 to |V| do
      begin
        D[i, j] := A[i, j];
        camino[i, j] := 0;
      end;
  for k := 1 to |V| do {considera cada  $v_k$  como un intermedio en orden}
    for i := 1 to |V| do
      for j := 1 to |V| do
        if d[i, k] + d[k, j] < d[i, j] then {actualiza mÍn}
          begin
            d[i, j] := d[i, k] + d[k, j];
            camino[i, j] := k;
          end;
    end;
end;

```

la solución puede mejorarse (aunque ello no siempre es obvio y con frecuencia resulta difícil); en otros casos, la técnica de programación dinámica es el mejor enfoque conocido.

De algún modo, si se ha visto un problema de programación dinámica, se han visto todos. Más ejemplos de programación dinámica aparecen en los ejercicios y las referencias.

10.4. Algoritmos aleatorizados

Pongamos el caso de que usted es un profesor que cada semana propone prácticas de programación. Se quiere asegurar que los estudiantes estén haciendo sus propios programas o, al menos, que entiendan el código que proponen. Una solución es realizar un examen el día que entregan cada programa. Por otro lado, tales exámenes llevan tiempo de clase adicional, así que sería práctico hacerlos sólo para la mitad de los programas. El problema es decidir cuándo hacer los exámenes.

Por supuesto, si los exámenes se anuncian por adelantado, ello se podría interpretar como una licencia implícita de no aplicarse en el 50% de los programas que no se examinarán. Se podría adoptar la estrategia no anunciada de realizar exámenes en programas alternos, pero los estudiantes no tardarían en descubrir la estrategia. Otra posibilidad sería hacer los exámenes en los programas que parecen importantes, pero esto generaría similares patrones de examen cada semestre. Siendo tan comunicativos como son los estudiantes, es probable que esta estrategia carecería de sorpresa después de un semestre.

Un método que parece eliminar esos problemas es usar una moneda. Se prepara un examen para cada programa (elaborar los exámenes no es, ni cercanamente, una tarea tan costosa en tiempo como calificarlos), y al inicio de la clase, el profesor lanzará una moneda para decidir si el examen se hace o no. De esta forma, es imposible conocer antes de la clase si habrá examen o no, y estos patrones no se repiten de semestre a semestre. Así, los estudiantes tendrán que esperar que haya un examen con una probabilidad de 50%; sin importar los patrones anteriores. La desventaja es que es posible que no se haga ningún examen en todo el semestre. Esta no es una ocurrencia probable, a menos que sospechemos de la moneda. Cada semestre, el número esperado de exámenes es la mitad del número de programas, y con una alta probabilidad, el número de exámenes no se desviará mucho de esto.

Este ejemplo ilustra lo que se conoce como *algoritmos aleatorizados*. Al menos una vez durante el algoritmo, se usa un número aleatorio para tomar una decisión. El tiempo de ejecución del algoritmo depende no sólo de la entrada particular, sino también de los números aleatorios que ocurren.

El tiempo de ejecución de un algoritmo aleatorizado para el peor caso es casi siempre igual al tiempo de ejecución de un algoritmo no aleatorizado para el peor caso. La diferencia importante es que un buen algoritmo aleatorizado no tiene entradas malas, sino sólo malos números aleatorios (en relación con la entrada específica). Parecería que ésta es una diferencia filosófica, pero de hecho es muy importante, como lo muestra el siguiente ejemplo.

Consideremos dos variantes de la ordenación rápida. el pivote de la variante A es el primer elemento, mientras que la variante B tiene como pivote un elemento elegido al azar. En ambos casos, el tiempo de ejecución para el peor caso es $\Theta(n^2)$, porque es posible que en cada paso se elija como pivote al elemento más grande. La diferencia entre los peores casos es que hay una entrada particular que siempre puede presentarse en la variante A y que causa el mal tiempo de ejecución. La variante A se ejecutará en tiempo $\Theta(n^2)$ cada vez que reciba una lista ya ordenada. Si la variante B se presenta con la misma entrada dos veces, tendrá dos tiempos de ejecución distintos, dependiendo de lo que ocurra con los números aleatorios.

En todo el texto, en nuestros cálculos de los tiempos de ejecución, hemos supuesto que todas las entradas son igualmente probables. Esto no es cierto, porque una entrada casi ordenada, por ejemplo, ocurre con mucha más frecuencia de lo esperado estadísticamente, y esto ocasiona problemas, en particular para la ordenación rápida y para los árboles binarios de búsqueda. Con un algoritmo aleatorizado, la entrada específica pierde importancia: Los números aleatorios son importantes, y podemos obtener un tiempo de ejecución *esperado*, donde ahora tenemos una media de todos los números aleatorios posibles en vez de todas las entradas posibles. Usando la ordenación rápida con un pivote aleatorio se tiene un algoritmo con tiempo esperado $O(n \log n)$. Esto significa que para cualquier entrada, aun la entrada ya ordenada, se espera que el tiempo de ejecución sea $O(n \log n)$, con base en las estadísticas de los números aleatorios. Una cota de tiempo de ejecución esperada es algo más fuerte que una cota del caso medio pero, por supuesto, es más débil que la correspondiente cota para el peor caso. Por otro lado, como vimos en el problema de la selección, las soluciones que obtienen la cota del peor caso no siempre son tan prácticas como sus contrapartes del caso medio. Y por lo regular, los algoritmos aleatorizados lo son.

Aquí examinaremos dos usos de la aleatorización. Primero, veremos un esquema nuevo en el cual las operaciones de los árboles binarios de búsqueda tengan un tiempo esperado de $O(\log n)$. De nuevo, esto significa que no hay malas entradas, sólo malos números aleatorios. Desde un punto de vista teórico, esto no es lo más emocionante del mundo, ya que los árboles de búsqueda equilibrados alcanzan esta cota en el peor caso. No obstante, el uso de la aleatorización produce algoritmos relativamente sencillos de búsqueda, inserción y, en especial, de eliminación.

Nuestra segunda aplicación es un algoritmo aleatorizado para comprobar la primalidad de números grandes. Que se sepa, no hay algoritmos eficientes no aleatorizados en tiempo polinómico para este problema. El algoritmo presentado se ejecuta rápidamente pero en ocasiones comete un error. No obstante, la probabilidad del error puede hacerse tan pequeña, que resulta despreciable.

10.4.1. Generadores de números aleatorios

Puesto que nuestros algoritmos requieren números aleatorios, debemos contar con un método para generarlo. En realidad, es virtualmente imposible lograr la verdadera aleatoriedad en un computador, ya que estos números dependerán del algoritmo, y por tanto, es imposible que sean aleatorios. En general, es suficiente

producir números *seudoaleatorios*, que son números que parecen aleatorios. Los números aleatorios tienen muchas propiedades estadísticas conocidas, y los números seudoaleatorios satisfacen la mayoría de ellas. Sorprendentemente, esto es mucho más fácil de decir que de hacer.

Supongamos que sólo se necesita lanzar una moneda; así, se debe generar un 0 o un 1 al azar. Una forma de hacer esto es examinar el reloj del sistema. El reloj puede almacenar la hora como un entero que cuenta el número de segundos a partir del 1º de enero de 1970.^t Entonces podríamos usar el bit de menor peso. El problema es que esto no funciona bien si necesitamos una secuencia de números aleatorios. Un segundo es mucho tiempo, y el reloj puede no cambiar en absoluto mientras el programa se está ejecutando. Aun si el tiempo estuviera almacenado en unidades de microsegundos, si el programa se estuviera ejecutando por sí mismo, la secuencia de números que se generaría estaría lejos de ser aleatoria, ya que el tiempo entre llamadas al generador sería esencialmente idéntico en toda invocación al programa. Vemos, entonces, que lo realmente necesario es una *secuencia* de números aleatorios.^f Estos números deben parecer independientes. Si se lanza una moneda y aparece cara, en el siguiente lanzamiento de moneda debe ser igualmente probable que caiga cara o cruz.

El método estándar para generar números aleatorios es el generador congruente lineal, descrito por primera vez por Lehmer en 1951. Los números x_1, x_2, \dots se generan satisfaceiendo

$$x_{i+1} = ax_i \bmod m.$$

Para iniciar la secuencia, se debe dar algún valor de x_0 . Este valor se conoce como *semilla*. Si $x_0 = 0$, entonces la secuencia está lejos de ser aleatoria, pero si a y m se eligen correctamente, cualquier otro $1 \leq x_0 < m$ es igualmente válido. Si m es prima, x_i nunca es 0. Por ejemplo, si $m = 11$, $a = 7$ y $x_0 = 1$, entonces los números generados son

$$7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2, \dots$$

Observe que después de $m - 1 = 10$ números, la secuencia se repite. Así, esta secuencia tiene un periodo de $m - 1$, que es tan grande como es posible (por el principio de la casilla). Si m es prima, siempre hay elecciones de a que dan un periodo completo de $m - 1$. Algunas elecciones de a no lo dan; si $a = 5$ y $x_0 = 1$, la secuencia tiene un periodo corto de 4.

$$5, 4, 9, 1, 5, 4, \dots$$

Obviamente, si se elige una m grande, un primo de 31 bits, el periodo debe ser significativamente grande para la mayoría de las aplicaciones. Lehmer sugirió el uso del primo de 31 bits $m = 2^{31} - 1 = 2\,147\,483\,647$. Para este primo, $a = 7^5 = 16\,807$ es uno de los muchos valores que da un generador de periodo completo. Su uso se ha estudiado bien y es recomendado por expertos en el campo. Veremos después

^t UNIX hace esto.

^f Usaremos el término aleatorio en vez de seudoaleatorio en lo que queda de la sección.

que con generadores de números aleatorios, experimentar significa estropear, así que respetemos la advertencia de seguir esta fórmula hasta que no se diga otra cosa.

Esta rutina parece fácil de implantar. En general, se usa una variable global para mantener el valor actual en la secuencia de x . Éste es el caso raro en que es útil una variable global. Esta variable global es iniciada por alguna rutina. Cuando se depura un programa que usa números aleatorios, es probable que sea mejor poner $x_0 = 1$, para que siempre ocurra la misma secuencia. Cuando el programa esté corregido, se podrá usar el reloj del sistema o pedir al usuario un valor de entrada para la semilla.

Es común devolver un número aleatorio real en el intervalo abierto $(0, 1)$ (0 y 1 no son valores posibles) lo que puede hacerse dividiendo entre m . De esto, se puede calcular por normalización un número aleatorio en cualquier intervalo $[a, b]$. Esto da la rutina "obvia" de la figura 10.54, la cual desafortunadamente funciona en pocas máquinas.

El problema con esta rutina es que la multiplicación puede producir desbordamiento, y no sólo es éste un error, sino que afecta el resultado incluso si se permite tal desbordamiento. Schrage dio un procedimiento en el cual todos los cálculos se pueden obtener en una máquina de 32 bits sin desbordamiento. Calculamos el cociente y el resto de m/a y los definimos como q y r , respectivamente. En este caso, $q = 127773$, $r = 2836$ y $r < q$. Se tiene

$$\begin{aligned}x_{i+1} &= ax_i \bmod m \\&= ax_i - m(ax_i \bmod m) \\&= ax_i - m(x_i \bmod q) + m(x_i \bmod q) - m(ax_i \bmod m) \\&= ax_i - m(x_i \bmod q) + m(x_i \bmod q - ax_i \bmod m)\end{aligned}$$

Puesto que $x_i = q(x_i \bmod q) + x_i \bmod q$, podemos sustituir el primer ax_i y obtener

$$\begin{aligned}x_{i+1} &= a(q(x_i \bmod q) + x_i \bmod q) - m(x_i \bmod q) + m(x_i \bmod q - ax_i \bmod m) \\&= (aq - m)(x_i \bmod q) + a(x_i \bmod q) + m(x_i \bmod q - ax_i \bmod m)\end{aligned}$$

Figura 10.54 Generador de números aleatorios que no funciona

```
var semilla: integer; {variable global}

const
  a = 16807;           {7^5}
  m = 2147483647;     {2^31 - 1}

  function aleatorio: real;
  begin
    semilla := (a * semilla) mod m;
    aleatorio := semilla / m;
  end;
```

Como $m = aq + r$, se infiere que $aq - m = -r$. Así, se obtiene

$$x_{i+1} = a(x_i \bmod q) - r(x_i \bmod q) + m(x_i \bmod q - ax_i \bmod m)$$

El término $\delta(x_i) = (x_i \bmod q - ax_i \bmod m)$ es 0 o 1, porque ambos términos son enteros y su diferencia cae entre 0 y 1. Así tenemos

$$x_{i+1} = a(x_i \bmod q) - r(x_i \bmod q) + m\delta(x_i)$$

Una rápida revisión muestra que, como $r < q$, todos los términos restantes se pueden calcular sin desbordamiento (ésta es una de las razones para escoger $a = 7^5$). Además, $\delta(x_i) = 1$ sólo si los términos restantes se evalúan como menores que cero. Así no es necesario calcular explícitamente $\delta(x_i)$, sino que puede ser determinada por una simple resta. Esto lleva al programa de la figura 10.55.

Este programa funciona siempre que $\text{MAXINT} \geq 2^{31} - 1$. Se puede caer en la tentación de suponer que todas las máquinas tienen, en su acervo estándar, un generador de números aleatorios al menos tan bueno como el de la figura 10.55. Desafortunadamente, esto no es cierto. Muchos acervos tienen generadores basados en la función:

$$x_{i+1} = (ax_i + c) \bmod 2^b$$

donde b se elige en correspondencia al número de bits de los enteros de la máquina, y c es impar. Estos acervos también devuelven x_i en vez de un valor entre 0 y 1. Desafortunadamente, esos generadores siempre producen valores de x_i que alternan entre pares e impares, propiedad difícilmente deseable. En efecto, los k bits de

Figura 10.55 Generador de números aleatorios que funciona en máquinas de 32 bits

```
var semilla: integer; {variable global}

const
  a = 16807;           {7^5}
  m = 2147483647;     {2^31 - 1}
  q = 127773;           {m div a}
  r = 2836;             {m mod a}

  function aleatorio: real;
  var semilla_temp: integer;
  begin
    semilla_temp := a * (semilla mod q) - r * (semilla div q);
    if semilla_temp >= 0 then
      semilla := semilla_temp
    else
      semilla := semilla_temp + m;
    aleatorio := semilla / m;
  end;
```

menor peso se repiten en un ciclo con periodo 2^k (en el mejor caso). Muchos otros generadores de números aleatorios tienen ciclos mucho menores que el dado en la figura 10.55. No son adecuados para el caso donde se necesitan secuencias largas de números aleatorios. Por último, parece que podemos obtener un mejor generador de números aleatorios sumando una constante a la ecuación. Por ejemplo, parece que

$$x_{i+1} = [16807x_i + 1] \bmod (2^{31} - 1)$$

sería de algún modo un poco más aleatoria. Esto ilustra cuán frágiles son estos generadores.

$$[16807(1319592028) + 1] \bmod (2^{31} - 1) = 1319592028,$$

así que si la semilla es 1319592028, el generador se atasca en un ciclo de periodo 1.

10.4.2. Listas con saltos

Nuestro primer uso de la aleatorización es una estructura de datos que permite la búsqueda y la inserción en un tiempo esperado $O(\log n)$. Como se mencionó al inicio de esta sección, esto significa que el tiempo de ejecución de cada operación en cualquier secuencia de entrada tiene un valor esperado de $O(\log n)$, donde la expectativa se basa en el generador de números aleatorios. Es posible agregar la eliminación y todas las operaciones que implican ordenación y obtener cotas de tiempo esperadas que correspondan con las cotas de tiempo medio de los árboles binarios de búsqueda.

La estructura de datos más simple posible para permitir la búsqueda es la lista enlazada. La figura 10.56 muestra una lista enlazada simple. El tiempo para realizar una búsqueda es proporcional al número de nodos que se han de examinar, que es n como máximo.

La figura 10.57 muestra una lista enlazada en la cual cada segundo nodo tiene un apuntador adicional al nodo que está dos lugares más adelante en la lista. Por ello, en el peor caso se examinan a lo más $\lceil n/2 \rceil + 1$ nodos.

Podemos extender esta idea y obtener la figura 10.58. Aquí, cada cuarto nodo tiene un apuntador al nodo que está cuatro nodos más adelante. Sólo se examinan $\lceil n/4 \rceil + 2$ nodos.

Figura 10.56 Lista enlazada simple

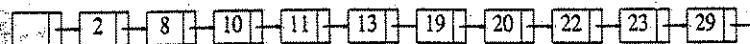


Figura 10.57 Lista enlazada con apuntadores a dos celdas más adelante

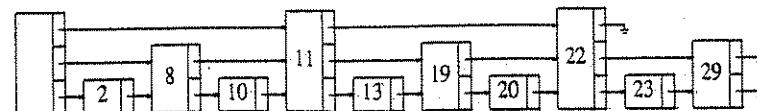
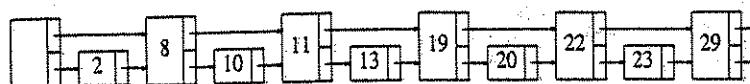


Figura 10.58 Lista enlazada con apuntadores a cuatro celdas más adelante

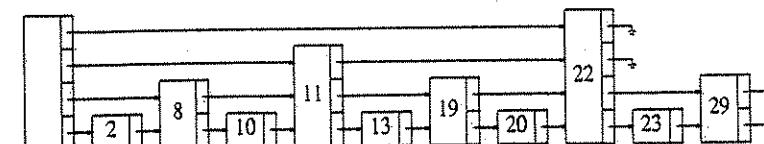
El caso límite de este argumento se muestra en la figura 10.59. Cada 2^i -ésimo nodo tiene un apuntador al nodo 2^i adelante de él. El número total de apuntadores sólo se ha duplicado, pero ahora se examinan cuando mucho $\lceil \log n \rceil$ nodos durante una búsqueda. No es difícil ver que el tiempo total consumido en una búsqueda es $O(\log n)$, porque la búsqueda consiste en avanzar a un nodo nuevo o bajar a un apuntador inferior en el mismo nodo. Cada paso consume a lo más un tiempo total $O(\log n)$ durante una búsqueda. Observe que la búsqueda en esta estructura de datos es esencialmente búsqueda binaria.

El problema con esta estructura de datos es su excesiva rigidez para permitir la inserción eficiente. La clave para hacer esta estructura de datos utilizable es relajar un poco las condiciones de la estructura. Definimos como *nodo de nivel k* al nodo que tiene k apuntadores. Como lo muestra la figura 10.59, el i -ésimo apuntador en cualquier nodo de nivel k ($k \geq i$) apunta al siguiente nodo con al menos i niveles. Esta es una propiedad fácil de sostener; no obstante, la figura 10.59 muestra una propiedad más restrictiva que ésta. Así se desecha la restricción de que el i -ésimo apuntador apunte al nodo 2^i más adelante, y se sustituye por la condición anterior menos restrictiva.

En el momento de insertar un elemento nuevo, le asignamos un nodo nuevo. En este punto, debemos decidir a qué nivel debe estar el nodo. Examinando la figura 10.59, encontramos que casi la mitad de los nodos está al nivel 1, casi un cuarto al nivel 2 y, en general, aproximadamente $1/2^i$ nodos están en el nivel i . Escogemos al azar el nivel del nodo, de acuerdo con su distribución de probabilidad. La forma más sencilla de hacer esto es lanzar una moneda hasta que caiga cara y usar el número total de lanzamientos como el nivel del nodo. La figura 10.60 muestra una lista con saltos representativa.

Dado esto, es sencillo describir los algoritmos de la lista con saltos. Para realizar un *buscar*, empezamos en el apuntador más alto en la cabecera. Recorremos este nivel hasta encontrar el siguiente nodo que es mayor que el buscado (o nil). Cuando

Figura 10.59 Lista enlazada con apuntadores a 2^i celdas más adelante



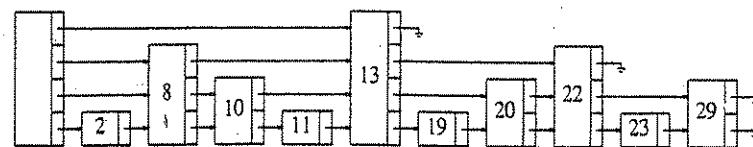


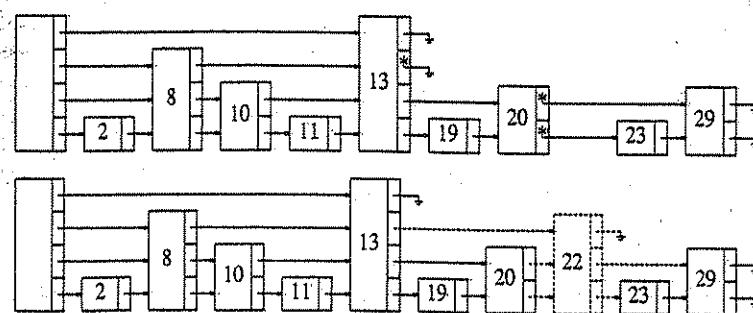
Figura 10.60 Una lista con saltos

esto ocurre, vamos al siguiente nivel inferior y continuamos la estrategia. Cuando se detiene el progreso en el nivel 1, o bien se está al frente del nodo buscado, o éste no se encuentra en la lista. Para realizar un *insertar*, se procede como en un *buscar*, y se hace un seguimiento de cada punto en el que se cambia de nivel. El nodo nuevo, cuyo nivel se determina al azar, se empalma entonces en la lista. Esta operación se muestra en la figura 10.61.

Una análisis superficial muestra que, como el número esperado de nodos en cada nivel permanece sin cambio con respecto al algoritmo original (no aleatorizado), la cantidad total de trabajo esperado para recorrer los nodos del mismo nivel tampoco cambia. Esto indica que esas operaciones tienen costos esperados $O(\log n)$. Por supuesto, se requiere una demostración más formal, pero no es muy diferente de ésta.

Las listas con saltos son semejantes a las tablas de dispersión en las que se requiere una aproximación del número de elementos que estarán en la lista (para que se pueda determinar el número de niveles). Si no está disponible una aproximación, podemos suponer un número grande o usar una técnica semejante a la redispersión. También es incómodo implantar listas con saltos en Pascal, porque los nodos del nivel k y del nivel j requieren diferentes declaraciones de tipos si $k \neq j$, a menos que estemos dispuestos a malgastar un espacio considerable. También, los problemas usuales con el caso especial de nil hacen el código menos elegante. Aun así, los experimentos han demostrado que las listas con saltos son tan eficientes como muchas implantaciones de árboles de búsqueda equilibrados y ciertamente mucho más sencillas de implantar en muchos lenguajes.

Figura 10.61 Antes y después de una inserción



10.4.3. Comprobación de primalidad

En esta sección examinaremos el problema de determinar si un número grande es primo o no. Como se mencionó al final del capítulo 2, algunos esquemas de criptografía dependen de la dificultad de factorizar un número grande, de 200 dígitos, en dos primos de 100 dígitos. A fin de implantar este esquema, necesitamos un método para generar estos dos primos. El problema es de gran interés teórico, porque nadie sabe ahora cómo comprobar si un número n con d dígitos es primo, usando un tiempo polinómico con respecto a d . Por ejemplo, el método obvio de comprobar la divisibilidad entre números impares del 3 a \sqrt{n} requiere cerca de $\frac{1}{2}\sqrt{n}$ divisiones, lo cual es casi $2^{d/2}$. Por otro lado, no se piensa este que problema sea NP completo; así, es uno de los pocos problemas que quedan por resolver; en el momento de escribir esto se ignora aún su complejidad.

En este capítulo proporcionaremos un algoritmo en tiempo polinómico que puede comprobar la primalidad. Si el algoritmo declara que el número no es primo, podemos estar seguros de que no lo es. Si el algoritmo declara que es primo, entonces con alta probabilidad, pero no el 100% de certeza, el número es primo. La probabilidad de error no depende del número particular que se está evaluando; sino de las elecciones aleatorias hechas por el algoritmo. Así, en ocasiones este algoritmo comete errores, pero veremos que la tasa de error se puede hacer arbitrariamente despreciable.

La clave del algoritmo es un famoso teorema enunciado por Fermat.

TEOREMA 10.10.

Teorema menor de Fermat: Si p es primo, y $0 < a < p$, entonces $a^{p-1} \equiv 1 \pmod{p}$.

DEMOSTRACIÓN:

Una demostración de este teorema se puede encontrar en cualquier libro sobre teoría de números.

Por ejemplo, puesto que 67 es primo, $2^{66} \equiv 1 \pmod{67}$. Esto sugiere un algoritmo para determinar si un número n es primo. Sólo se revisa si $2^{n-1} \equiv 1 \pmod{n}$. Si $2^{n-1} \equiv 1 \pmod{n}$, entonces podemos estar seguros de que n no es primo. Por otro lado, si la igualdad se mantiene, entonces probablemente n es primo. Por ejemplo, el menor n que satisface $2^{n-1} \equiv 1 \pmod{n}$ pero no es primo es $n = 341$.

En ocasiones este algoritmo cometerá errores, pero el problema es que siempre serán los mismos errores. En otras palabras, hay un conjunto fijo de n para el cual no funciona. Podemos intentar aleatorizar el algoritmo como sigue: se toma $1 < a < n - 1$ al azar. Si $a^{n-1} \equiv 1 \pmod{n}$, se declara que n es probablemente primo, en otro caso se declara que n definitivamente no es primo. Si $n = 341$, y $a = 3$, encontramos que $3^{340} \equiv 56 \pmod{341}$. Así, si sucede que el algoritmo elige $a = 3$, obtendrá la respuesta correcta para $n = 341$.

Aunque al parecer esto funciona, hay números que engañan aun a este algoritmo para la mayoría de las elecciones de a . A tal conjunto de números se le conoce como *números de Carmichael*. No son primos, pero satisfacen $a^{n-1} \equiv 1 \pmod{n}$ para toda $0 < a < n$ que sea relativamente prima con respecto a n . El menor número así es 561.

Entonces, necesitamos una comprobación adicional para mejorar las oportunidades de no cometer un error.

En el capítulo 7, demostramos un teorema relacionado con el reconocimiento cuadrático. Un caso especial de este teorema es el siguiente:

TEOREMA 10.11.

Si p es primo y $0 < x < p$, las únicas soluciones de $a x^2 \equiv 1 \pmod{p}$ son $x = 1, p - 1$.

DEMOSTRACIÓN:

$x^2 \equiv 1 \pmod{p}$ implica que $x^2 - 1 \equiv 0 \pmod{p}$. Esto implica que $(x-1)(x+1) \equiv 0 \pmod{p}$. Puesto que p es primo, $0 \leq x < p$, y p debe dividir $(x-1)$ o $(x+1)$, se demuestra el teorema.

Por lo tanto, si en cualquier momento en el cálculo de la computación de $a^{p-1} \pmod{n}$ descubrimos una violación a este teorema, podemos concluir que n definitivamente es no primo. Si se usa *potencia*, de la sección 2.4.4, vemos que habrá varias oportunidades de aplicar esta comprobación. Modificamos esta rutina para realizar las operaciones \pmod{n} , y aplicamos la prueba del teorema 10.11. Esta estrategia se implanta en la figura 10.62.

Recordemos que si *comprobar_primo* devuelve *DEFINITIVAMENTE_COMPUUESTO*, se ha probado que n no puede ser primo. La demostración no es constructiva porque no da un método para encontrar los factores. Se ha demostrado que para cualquier n (suficientemente grande), cuando mucho $(n-9)/4$ valores de a engañan al algoritmo. Así, si a se elige al azar, y el algoritmo responde *PROBABLEMENTE_PRIMO*, entonces el algoritmo es correcto al menos el 75% del tiempo. Supongamos que *comprobar_primo* se ejecuta 50 veces. La probabilidad de que el algoritmo sea engañado una vez es $\frac{1}{4}$ como máximo. Así, la probabilidad de que 50 ensayos independientes engañen al algoritmo nunca es mayor que $1/4^{50} = 2^{-100}$. Ésta es una aproximación realmente muy conservadora, que se cumple en pocas elecciones de n . Aun así, es más probable ver un error de hardware que una indicación incorrecta de la primalidad.

10.5. Algoritmos con retroceso

La última técnica de diseño de algoritmos que examinaremos es el *retroceso* (*backtracking*). En muchos casos, un algoritmo con retroceso es equivalente a una implantación inteligente de la búsqueda exhaustiva, con un rendimiento por lo general desfavorable. Sin embargo, no siempre es así y aunque lo sea, en algunos casos, los ahorros sobre la búsqueda exhaustiva a base de fuerza bruta pueden ser significativos. Por supuesto, el rendimiento es relativo: un algoritmo $O(n^k)$ para la ordenación es ciertamente malo, pero un algoritmo $O(n^k)$ para el problema del agente viajero (o cualquier problema NP completo) sería un resultado histórico.

Un ejemplo práctico de algoritmo con retroceso es el problema de acomodar el mobiliario en una casa nueva. Hay muchas posibilidades que intentar, pero de hecho sólo se suelen considerar unas cuantas. Iniciando sin acomodo alguno, cada

```

type resultado_comprobación = (PROBABLEMENTE_PRIMO,
                                 DEFINITIVAMENTE_COMPUUESTO);

{Calcula resultado =  $a^{p-1} \pmod{n}$ 
{Si en cualquier punto  $x^2 \equiv 1 \pmod{n}$  se detecta con  $x \neq 1, x \neq n-1$ ,
{entonces se pone qué_es_n a DEFINITIVAMENTE_COMPUUESTO}
{Se suponen enteros muy grandes, así que esto es pseudocódigo.}

procedure potencia(a, p, n: integer; var resultado: integer;
                    var qué_es_n: resultado_comprobación);
begin
  var x: integer;
  begin
    [1] if p = 0 then      {caso base}
        resultado := 1;
    else
      begin
        [2] qué_es_n := DEFINITIVAMENTE_COMPUUESTO;
        [3] potencia(a, p div 2, n, x, qué_es_n);
        [4] resultado := (x * x) mod n;

        {Comprueba si  $x^2 \equiv 1 \pmod{n}, x \neq 1, x \neq n-1$ 
        if (resultado = 1) and (x <> 1) and (x <> n-1) then
          qué_es_n := DEFINITIVAMENTE_COMPUUESTO;
      end;
    end;
  end;

  {comprobar_primo: determina si  $n \geq 3$  es primo usando un valor de  $a$ 
  {repite este procedimiento tantas veces como se necesite para
  {una tasa de error deseada.}

  function comprobar_primo(n: integer): resultado_comprobación;
  var
    a, resultado: integer;
    qué_es_n: resultado_comprobación;
  begin
    escoger a aleatoriamente en 2..n-2;
    qué_es_n := PROBABLEMENTE_PRIMO;
    [9] potencia(a, n-1, n, resultado, qué_es_n); {calcula  $a^{n-1} \pmod{n}$ 
    if (resultado <> 1) or (qué_es_n = DEFINITIVAMENTE_COMPUUESTO) then
      comprobar_primo := DEFINITIVAMENTE_COMPUUESTO
    else
      comprobar_primo := PROBABLEMENTE_PRIMO;
    end;
  end;

```

Figura 10.62 Algoritmo probabilista para la comprobación de la primalidad

pieza del mobiliario se coloca en alguna parte de la habitación. Si se colocan todos los muebles y el propietario queda contento, el algoritmo termina. Si se alcanza un punto donde toda colocación subsecuente de muebles es indeseable, debemos deshacer el último paso e intentar otra alternativa. Por supuesto, esto puede forzar otra descolocación, y así sucesivamente. Si encontramos que deshacemos todos los primeros pasos posibles, entonces no hay ninguna colocación de muebles que sea satisfactoria. Si no es así, tarde o temprano se termina con un acomodo satisfactorio. Observe que aunque este algoritmo es esencialmente a base de fuerza bruta, no ensaya directamente todas las posibilidades. Por ejemplo, nunca se intenta colocar el sofá en la cocina. Pronto se descartan muchas otras colocaciones malas, porque se detecta un subconjunto indeseable del acomodo. La eliminación de un grupo grande de posibilidades en un paso se denomina *poda*.

Veremos dos ejemplos de algoritmos con retroceso. El primero es un problema de geometría computacional. El segundo muestra cómo los computadores eligen movimientos en juegos como el ajedrez y las damas.

10.5.1. El problema de la reconstrucción del camino de cuota

Supongamos que se tienen n puntos, p_1, p_2, \dots, p_n , localizados en el eje x . x_i es la coordenada x de p_i . Supongamos además que $x_1 = 0$ y que los puntos se dan de izquierda a derecha. Estos n puntos determinan $n(n - 1)/2$ distancias (no necesariamente únicas) d_1, d_2, \dots, d_n entre cada par de puntos de la forma $|x_i - x_j|$ ($i \neq j$). Está claro que si tenemos el conjunto de puntos, es fácil construir el conjunto de distancias en tiempo $O(n^3)$. Este conjunto no estará ordenado, pero si queremos establecer una cota de tiempo $O(n^2 \log n)$, las distancias se pueden ordenar también. El problema de la reconstrucción del camino de cuota consiste en reconstruir un conjunto de puntos a partir de las distancias. Esto tiene aplicaciones en física y biología molecular (véanse las referencias para información más específica). El nombre se deriva de la analogía de los puntos con las salidas de cuota en las autopistas de la costa este de Estados Unidos. Así como la factorización parece más difícil que la multiplicación, el problema de la reconstrucción parece ofrecer más dificultad que el de la construcción. Nadie ha podido dar un algoritmo que garantice un funcionamiento en tiempo polinómico. El algoritmo que presentaremos parece ejecutarse en $O(n^2 \log n)$; no se conoce ningún contraejemplo para esta conjectura, pero sólo es eso: una conjectura.

Por supuesto, dada una solución al problema, se puede construir un número infinito de otras soluciones, añadiendo un desplazamiento a todos los puntos. Es por ello que insistiremos que el primer punto esté anclado en 0 y que el conjunto de puntos que constituye una solución se presente en orden no decreciente.

Sea D el conjunto de las distancias, y supongamos que $|D| = m = n(n - 1)/2$. Como ejemplo, supongamos que

$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$$

Puesto que $D = 15$, sabemos que $n = 6$. Se inicia el algoritmo poniendo $x_1 = 0$. Claramente, $x_6 = 10$, porque 10 es el mayor elemento en D . Eliminaremos 10 de D . Los puntos colocados y las distancias restantes se muestran en la figura siguiente.

$$\begin{array}{c} | \\ x_1 = 0 \\ | \\ x_6 = 10 \end{array}$$

$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}.$$

La distancia mayor restante es 8, lo cual significa que $x_2 = 2$ o $x_5 = 8$. Por simetría, podemos concluir que la elección no es importante porque ambas dan una solución (las cuales son imágenes espectaculares una de la otra), o ninguna lo hace, así que se puede poner $x_5 = 8$ sin afectar la solución. Entonces se remueven las distancias $x_6 - x_5 = 2$ y $x_5 - x_1 = 8$ de D , obteniendo

$$\begin{array}{c} | \\ x_1 = 0 \\ | \\ x_5 = 8 \\ | \\ x_6 = 10 \end{array}$$

$$D = \{1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 7\}.$$

El siguiente paso no es obvio. Puesto que 7 es el valor más grande en D , $x_4 = 7$ o bien $x_2 = 3$. Si $x_4 = 7$, las distancias $x_6 - 7 = 3$ y $x_5 - 7 = 1$ también deben estar presentes en D . Una rápida revisión muestra que de hecho lo están. Por otro lado, si se pone $x_2 = 3$, entonces $3 - x_1 = 3$ y $x_5 - 3 = 5$ deben estar presentes en D . También estas distancias están en D , por lo que no hay una guía sobre qué decisión tomar. Por ello, intentamos una y vemos si lleva a una solución. Si ocurre que no, podemos regresar e intentar la otra. Al intentar la primera opción, se pone $x_4 = 7$, lo cual deja

$$\begin{array}{c} | \\ x_1 = 0 \\ | \\ x_4 = 7 \\ | \\ x_5 = 8 \\ | \\ x_6 = 10 \end{array}$$

$$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}.$$

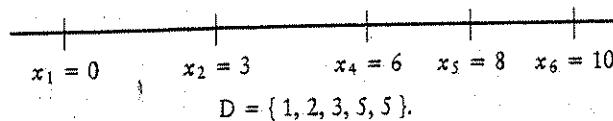
En este punto, tenemos que $x_1 = 0$, $x_4 = 7$, $x_5 = 8$ y $x_6 = 10$. Ahora la distancia mayor es 6, así que o bien $x_3 = 6$ o $x_2 = 4$. Pero si $x_3 = 6$, entonces $x_4 - x_3 = 1$, lo cual es imposible, pues 1 ya no está en D . Por otro lado, si $x_2 = 4$ entonces $x_2 - x_1 = 4$, $x_5 - x_2 = 4$. Esto también es imposible porque 4 sólo aparece una vez en D . Así, esta línea de razonamiento no lleva a ninguna solución, por lo cual hacemos un retroceso.

Puesto que $x_4 = 7$ no produjo una solución, intentamos $x_2 = 3$. Si esto falla también, desistimos y no se informa ninguna solución. Ahora tenemos

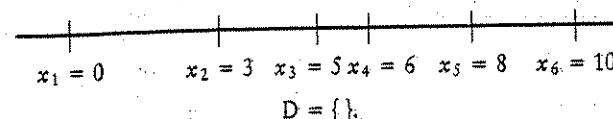
$$\begin{array}{c} | \\ x_1 = 0 \\ | \\ x_2 = 3 \\ | \\ x_5 = 8 \\ | \\ x_6 = 10 \end{array}$$

$$D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}.$$

Una vez más hay que escoger entre $x_4 = 6$ y $x_3 = 4$. $x_3 = 4$ es imposible porque D sólo tiene una ocurrencia de 4, y en esta elección estarían implicadas dos. $x_4 = 6$ es posible, así que obtenemos



La única elección que queda es asignar $x_3 = 5$; esto funciona porque deja a D vacío, y tenemos una solución.



La figura 10.63 muestra un árbol de decisión que representa las acciones tomadas para llegar a la solución. En vez de etiquetar las ramas, hemos puesto las etiquetas en los nodos de destino de las ramas. Un nodo con un asterisco indica que los puntos elegidos son inconsistentes con las distancias dadas; los nodos con dos asteriscos sólo tienen nodos imposibles como hijos, así que representan un camino incorrecto.

El pseudocódigo para implantar este algoritmo es casi directo. La rutina conductora, *camino_cuota*, se muestra en la figura 10.64. Recibe el arreglo de puntos x (que no necesita valores iniciales), el arreglo de distancias D y n .^t Si se descubre una solución, *encontrado* se pondrá a verdadero, y la respuesta se colocará en x . Si no, *encontrado* será falso y x estará indefinida. El arreglo de distancias D permanecerá sin cambio. La rutina pone x_1, x_{n-1} y x_n , como se describió antes, altera (la copia de) D y llama al algoritmo con retroceso *colocar* para colocar los otros puntos. Suponemos que ya se ha hecho un comprobación para asegurar que $|D| = n(n - 1)/2$.

La parte más difícil es el algoritmo con retroceso, que se muestra en la figura 10.65. Como la mayoría de los algoritmos con retroceso, la implantación más conveniente es recursiva. Pasamos los mismos argumentos más los límites *Izq* y *Der*; x_{Izq}, \dots, x_{Der} son las coordenadas x de los puntos que estamos intentando colocar. D se pasa como *var*, pues hacer copias puede ser costoso, en especial en un procedimiento recursivo. Si D está vacío (o $Izq > Der$), entonces se ha encontrado una solución, y se puede regresar. Si no es así, primero intentamos colocar $x_{Der} = D_{\max}$. Si todas las distancias adecuadas están presentes (en la cantidad correcta), entonces provisionalmente colocamos este punto, eliminamos esas distancias e intentamos llenar de *Izq* a *Der* - 1. Si las distancias no están presentes, o falla el intento de llenar de *Izq* a *Der* - 1, entonces tratamos de poner $x_{Izq} = x_i - D_{\max}$, usando una estrategia semejante. Si esto no funciona, entonces no hay solución; en otro caso, se ha encontrado una solución, y esta información al fin y al cabo se pasa de regreso a *camino_cuota* por medio de las variables *encontrado* y x .

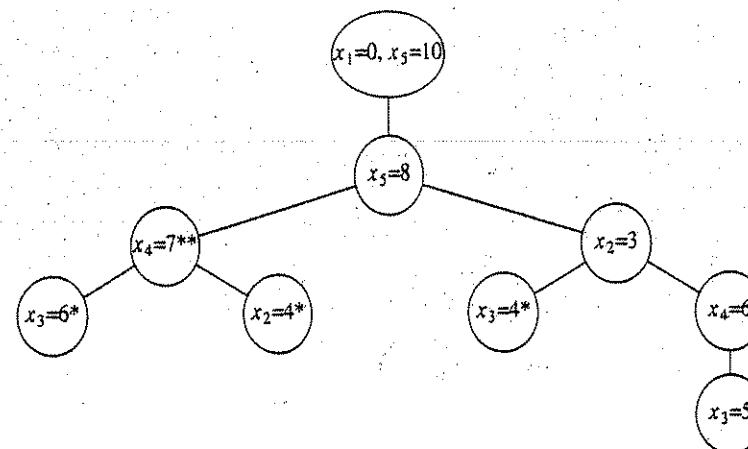


Figura 10.63 Árbol de decisión del ejemplo trabajado para la reconstrucción de caminos de cuota

```

procedure camino_cuota(var x; D; n; var encontrado);
begin
  [1] encontrado := false
  [2] x[1] := 0;
  [3] x[n] := eliminar_máx(D);
  [4] x[n-1] := eliminar_máx(D);
  [5] if x[n] - x[n-1] ∈ D then
      begin
        [6]   eliminar(x[n] - x[n-1], D);
        [7]   colocar(x, D, n, 2, n-2, encontrado);
      end
    else
      [8]   encontrado := false;
  end;

```

Figura 10.64 Algoritmo de reconstrucción de caminos de cuota:
rutina conductora

^t Se han usado variables de una letra, que en general es un mal estilo, por consistencia con el ejemplo trabajado. También, por simplicidad, no se dan los tipos de las variables.

{Algoritmo con retroceso para colocar los puntos}
 $\{x[Izq] \dots x[Der]\}$
 $\{x[1] \dots x[Izq-1] \text{ y } x[Der+1] \dots x[n] \text{ ya se han colocado provisionalmente}\}$
 $\{\text{Si encontrado es cierto, entonces } x[Izq] \dots x[Der] \text{ tendrá valores.}\}$

```

procedure colocar(var x; var D; n; Izq, Der; var encontrado);
    var d_máx;
begin
    [1] if D está vacío then
        encontrado := true
    else
        begin
            [3] d_máx := buscar_máx(D);

            {Comprueba si es factible poner  $x[Der] = d_máx$ }
            [4] if  $|x[j] - d_máx| \in D$  para toda  $1 \leq j < Izq$  y  $Der < j \leq n$  then
                begin
                    [5]  $x[Der] := d_máx$ ; {intenta  $x[Der] = d_máx$ }
                    [6] for  $1 \leq j < Izq, Der < j \leq n$  do
                        eliminar( $|x[j] - d_máx|$ , D);
                    [8] colocar(x, D, n, Izq, Der - 1, encontrado);

                    if encontrado = false then {retroceso}
                        for  $1 \leq j < Izq, Der < j \leq n$  do {deshace la eliminación}
                            insertar( $|x[j] - d_máx|$ , D);
                end;

                [12] if (encontrado = false) and (( $|x[n] - d_máx - x[j]| \in D$ )
                    para toda  $1 \leq j < Izq$  y  $Der < j \leq n$ ) then
                    begin
                        [14]  $x[Izq] := x[n] - d_máx$ ; {la misma lógica que antes}
                        [15] for  $1 \leq j < Izq$  y  $Der < j \leq n$  do
                            eliminar( $|x[n] - d_máx - x[j]|$ , D);
                        [17] colocar(x, D, n, Izq + 1, Der, encontrado);

                        if encontrado = false then {retroceso}
                            for  $1 \leq j < Izq, Der < j \leq n$  do {deshace la eliminación}
                            insertar( $|x[n] - d_máx - x[j]|$ , D);
                    end;
                end;
            end;
        end;

```

Figura 10.65 Algoritmo de reconstrucción de caminos de cuota:
pasos con retroceso

El análisis del algoritmo implica dos factores. Supongamos que las líneas 9 a 11 y 18 a 20 nunca se ejecutan. Podemos mantener D como un árbol binario equilibrado (o desplegado). Si nunca retrocedemos, hay a lo más $O(n^2)$ operaciones que implican a D, como la eliminación y los *buscar* de las líneas 4 y 12 a 13. Esta pretensión es obvia para las eliminaciones, ya que D tiene $O(n^2)$ elementos y ningún elemento se reinserta nunca. Cada llamada a *colocar* usa a lo más $2n$ buscar, y como *colocar* nunca retrocede en este análisis, puede haber $2n^2$ buscar como máximo. Así, si no hay retroceso, el tiempo de ejecución es $O(n^2 \log n)$.

Por supuesto, los retrocesos ocurren, y si son repetidos, entonces el rendimiento del algoritmo resulta afectado. No se conoce ninguna cota polinómica en la cantidad de retrocesos, pero, por otro lado, no hay ejemplos patológicos que muestren que el retroceso debe suceder más de $O(1)$ veces. Así, es completamente posible que este algoritmo sea $O(n^2 \log n)$. Los experimentos que han mostrado que si los puntos tienen coordenadas enteras distribuidas uniforme y aleatoriamente a partir de $[0, D_{\max}]$, donde $D_{\max} = \Theta(n^2)$, entonces, casi seguramente, cuando mucho se efectúa un retroceso durante el algoritmo completo.

10.5.2. Juegos

Como última aplicación, consideraremos la estrategia que un computador puede usar para jugar un juego de estrategia como las damas o el ajedrez. Usaremos como ejemplo el mucho más sencillo juego de tres en raya (o "gato"), porque facilita la ilustración de las cosas.

El tres en raya es, por supuesto, un empate si ambos lados juegan óptimamente. Realizando un cuidadoso análisis caso por caso, no es difícil construir un algoritmo que nunca pierda y siempre gane cuando se presente la oportunidad. Esto puede hacerse porque ciertas posiciones son trampas conocidas y pueden manejarse por medio de una tabla de consulta. Otras estrategias, como ocupar la casilla del centro cuando está disponible, hacen más simple el análisis. Si se hace esto, entonces usando una tabla siempre podemos escoger un movimiento con base sólo en la posición actual. Por supuesto, esta estrategia requiere que, en materia de razonamiento, sea el programador, y no el computador, el que realice la mayoría del trabajo.

Estrategia minimax

La estrategia general es cuantificar la "bondad" de una posición por medio de una función de evaluación. Una posición que es un triunfo para el computador puede tener el valor +1; un empate podría tener 0, y una posición perdida por el computador obtendría -1. Una posición para la cual se puede determinar esta asignación examinando el tablero se conoce como posición *terminal*.

Si una posición no es terminal, el valor de la posición está determinado por la suposición recursiva del juego óptimo por ambos lados. A ésta se le llama estrategia *minimax*, porque un jugador (el humano) intenta minimizar el valor de la posición, mientras que el otro jugador (el computador) trata de maximizarla.

Una *posición sucesora* de P es cualquier posición P_s alcanzable desde P haciendo una jugada. Si el computador ha de mover cuando está en una posición P , recursivamente evalúa el valor de todas las posiciones sucesoras. El computador escoge el movimiento con el valor más alto; éste es el valor de P . Para evaluar cualquier posición sucesora P_s , todos los sucesores de P_s se evalúan recursivamente, y se escoge el valor más pequeño. Este valor menor representa la réplica más favorable para el jugador humano.

El código de la figura 10.66 hace más clara la estrategia del computador. Las líneas 1 a 4 evalúan triunfos o empates inmediatos. Si ninguno de estos casos se aplica, la posición es no terminal. Recordando que *valor* debe contener el máximo de todas las posiciones sucesoras posibles, la línea 5 lo inicia al menor valor posible y el ciclo de las líneas 6 a 13 busca mejoras. Cada posición sucesora se evalúa recursivamente por medio de las líneas 8 a 10. Esto es recursivo, porque, como veremos, el procedimiento *buscar_mov_humano* llama a *buscar_mov_computador*. Si la respuesta del humano a un movimiento deja al computador con una posición más favorable que la obtenida con el mejor movimiento anterior del computador, entonces se actualizan *valor* y *mejor_mov*. La figura 10.67 muestra el procedimiento para la selección del movimiento del jugador humano. La lógica es prácticamente idéntica, excepto que el jugador humano elige el movimiento que conduce a la posición con el valor más bajo. En efecto, no es difícil combinar estos dos procedimientos en uno pasando una variable adicional, que indica a quién toca mover. Esto dificulta un tanto leer el código, por lo que nos hemos quedado con las rutinas separadas.

Dejamos como ejercicio las rutinas de soporte. La computación más costosa es el caso en el que toca al computador abrir el juego. Puesto que en esta etapa el juego es un empate forzoso, el computador selecciona la casilla 1.[†] Se examinaron un total de 97 162 posiciones, y el cálculo tardó 2.5 segundos en un VAX 8800. No se hizo ningún intento por optimizar el código. Cuando el computador juega en segundo término, el número de posiciones examinadas es 5185 si el jugador humano elige la casilla del centro, 9761 con una casilla en esquina y 13 233 con una casilla de la orilla que no es esquina.

Para juegos más complejos, como las damas o el ajedrez, obviamente no es factible buscar todos los caminos hasta los nodos terminales.[‡] En este caso tenemos que parar la búsqueda después de alcanzar cierta profundidad de recursión. Los nodos donde se detiene la recursión se vuelven nodos terminales. Estos nodos terminales se evalúan con una función que calcula 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 fuerza de piezas, y factores posicionales. La función de evaluación es crucial para el éxito porque la selección del movimiento del computador se basa en la maximización de esta función. Los mejores programas de ajedrez tienen funciones de evaluación sorprendentemente elaborados.

[†] Se numeraron las casillas empezando arriba a la izquierda y hacia la derecha. No obstante, esto sólo importa para las rutinas de soporte.

[‡] Se estima que si se hiciera esta búsqueda para el caso del ajedrez, al menos se examinarían 10^{100} posiciones para el primer movimiento. Aun si las mejoras descritas más adelante en esta sección se incorporaran, no se podría reducir este número a un nivel práctico.

{Procedimiento recursivo para buscar el mejor movimiento del computador}
{mejor_mov es un número de 1 a 9 que indica la casilla.}

{Los valores posibles satisfacen}

{COMP_PIERDE < EMPATE < COMP_GANA}

{El procedimiento complementario buscar_mov_humano está abajo}

```
procedure buscar_mov_comp(var tablero: tipo_tablero;
                           var mejor_mov, valor: integer);
var ni, i, respuesta: integer; {"ni" significa "no importa"}
begin
  if tablero_lleno(tablero) then
    valor := EMPATE
  else
    if comp_gana_inmediato(tablero, mejor_mov) then
      valor := COMP_GANA
    else
      begin
        valor := COMP_PIERDE;
        for i := 1 to 9 do      {intenta en cada casilla}
        begin
          if está_vacio(tablero, i) then
            begin
              colocar(tablero, i, COMP);
              buscar_mov_humano(tablero, ni, respuesta);
              descolocar(tablero, i); {restaura tablero}

              if respuesta > valor then  {actualiza mejor movimiento}
                begin
                  valor := respuesta;
                  mejor_mov := i;
                end;
            end;
        end;
      end;
    end;
end;
```

Figura 10.66 Algoritmo minimax del tres en raya: selección del computador

Sin embargo, para el ajedrez por computador, el factor individual más importante parece ser el número de movimientos anticipados que puede prever el programa. En ocasiones a esto se le llama *capa*; es igual a la profundidad de la recursión. Para implantar esto, a las rutinas de búsqueda se les pasa un parámetro adicional.

El método básico para incrementar el factor de anticipación en programas de juego consiste en tener métodos que evalúen menos nodos sin perder información. Un método que ya hemos visto usa una tabla para mantener la pista de todas las posiciones ya evaluadas. Por ejemplo, en el curso de la búsqueda del primer

```

procedure buscar_mov_humano(var tablero: tipo_tablero;
    var mejor_mov, valor: integer);
    var ni, i, respuesta: integer;      {ni significa no importa}
begin
    if tablero_lleno(tablero) then
        valor := EMPATE
    else
        if humano_gana_inmediato(tablero, mejor_mov) then
            valor := COMP_PIERDE
        else
            begin
                valor := COMP_GANA;
                for i:= 1 to 9 do
                begin
                    if está_yacío(tablero, i) then
                    begin
                        colocar(tablero, i, HUMANO);
                        buscar_mov_comp(tablero, ni, respuesta);
                        descolocar(tablero, i);
                        if respuesta < valor then
                            begin
                                valor := respuesta;
                                mejor_mov := i;
                            end;
                    end;
                end;
            end;
        end;
end;

```

Figura 10.67 Algoritmo minimax para tres en raya: selección humana

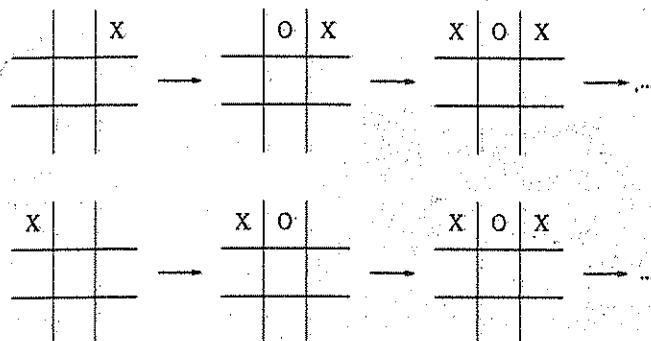


Figura 10.68 Dos búsquedas que llegan a una posición idéntica

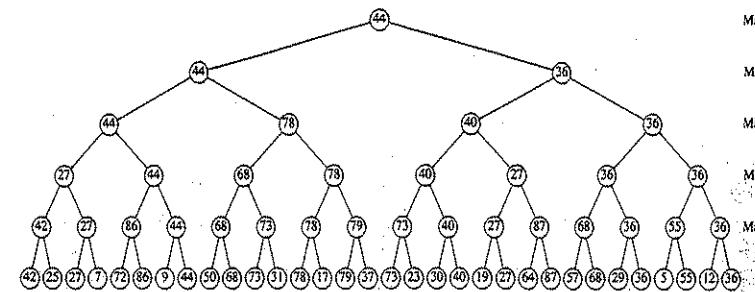


Figura 10.69 Árbol de juego hipotético

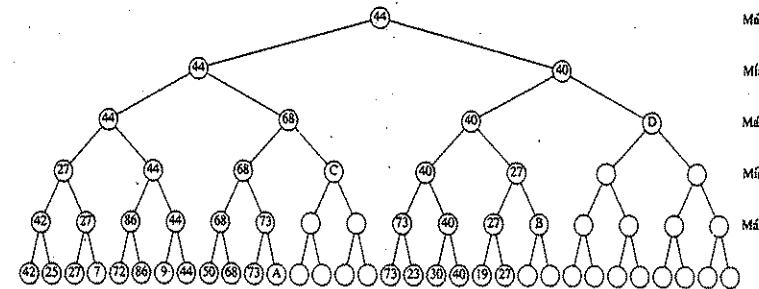
movimiento, el programa examinará las posiciones de la figura 10.68. Si se guardan los valores de las posiciones, no hace falta volver calcular la segunda ocurrencia de una posición; en esencia se convierte en una posición terminal. La estructura de datos que registra esto se llama *tabla de transposición*; ésta casi siempre se implanta por medio de dispersión. En muchos casos, esto puede ahorrar muchos cálculos. Por ejemplo, en los finales de ajedrez, cuando hay relativamente pocas piezas, los ahorros en tiempo permiten que las búsquedas desciendan varios niveles más abajo.

Poda α - β

Es probable que la mejora más significativa que en general podemos obtener sea la que se conoce como poda α - β . La figura 10.69 muestra el seguimiento de las llamadas recursivas con que se evalúa alguna posición hipotética en un juego hipotético. A esto se le suele llamar *árbol de juego*. (Hasta ahora hemos evitado el uso de este término, porque es algo confuso: en realidad no se construye ningún árbol en el algoritmo. El árbol de juego sólo es un concepto abstracto.) El valor del árbol de juego es 44.

La figura 10.70 muestra la evaluación del mismo árbol de juego, con varios nodos no evaluados. Casi la mitad de los nodos terminales no ha sido revisada. Demostraremos que su evaluación no cambiaría el valor en la raíz.

Figura 10.70 Árbol de juego podado



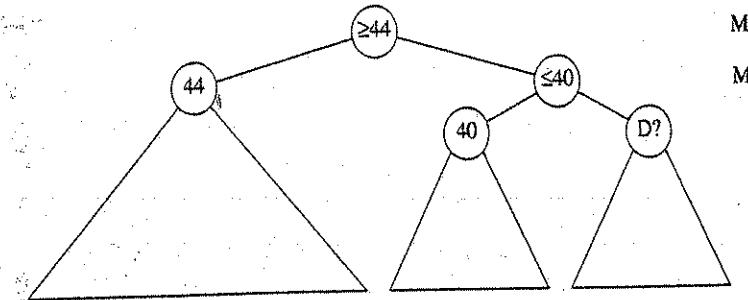
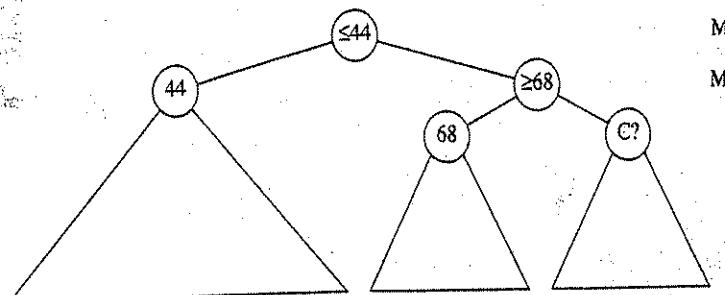


Figura 10.71 El nodo marcado con ? carece de importancia

Primero, consideremos el nodo D. La figura 10.71 muestra la información recolectada en el momento de evaluar D. En este punto, aún estamos en *buscar_mov_humano* y estamos contemplando una llamada a *buscar_mov_comp* sobre D. No obstante, ya sabemos que *buscar_mov_humano* devolverá 40 como máximo, pues es un nodo *mín*. Por otro lado, su nodo padre, *máx*, ya encontró una secuencia que asegura 44. Y no es posible que nada que haga D mejore este valor. Por lo tanto, no se necesita evaluar D. Esta poda al árbol se denomina poda α . Una situación idéntica se da en el nodo B. Para implantar la poda α , *tomar_mov_comp* pasa su tentativa máxima (α) a *tomar_mov_humano*. Si el mínimo provisional de *tomar_mov_humano* cae abajo de este valor, entonces *tomar_mov_humano* regresa de inmediato.

Algo semejante ocurre en los nodos A y C. Esta vez, estamos a la mitad de un *buscar_mov_comp* y a punto de llamar a *buscar_mov_humano* para evaluar C. La figura 10.72 muestra la situación en el nodo C. Sin embargo, *buscar_mov_humano*, al nivel *mín*, que ha llamado a *buscar_mov_comp*, ya determinó que puede forzar un valor de 44 como máximo (recordemos que los valores bajos son buenos desde el punto de vista humano). Puesto que *buscar_mov_comp* tiene un máximo provisional de 68, nada que haga C afectará el resultado en el nivel *mín*. Por lo tanto, no debe evaluarse C. Este tipo de podas se llama poda β , y es la versión simétrica de la poda α . Al combinar ambas técnicas tenemos una poda $\alpha\text{-}\beta$.

Figura 10.72 El nodo marcado con ? carece de importancia



[Igual que antes, pero realizando una poda $\alpha\text{-}\beta$.
[La rutina principal debe hacer la llamada con $\alpha = \text{COMP_PIERDE}$,
 $\beta = \text{COMP_GANAN}$]

```

procedure buscar_mov_comp(var tablero; var mejor_mov; var valor;  $\alpha$ ;  $\beta$ );
  var ni, i, respuesta: integer; {ni quiere decir no importa}
begin
  [1] if tablero_lleno(tablero) then
    valor := EMPATE
  else
    [2] if comp_gana_inmediato(tablero, mejor_mov) then
      valor := COMP_GANA
    else
      begin
        [3] begin
          valor :=  $\alpha$ ; i := 1;
          while (i <= 9) and (valor <  $\beta$ ) do
            begin
              [4] if está_vacio(tablero, i) then
                begin
                  colocar(tablero, i, COMP);
                  buscar_mov_humano(tablero, ni, respuesta, valor,  $\beta$ );
                  descolocar(tablero, i);
                end;
              end;
              [5] if respuesta > valor then
                begin
                  [6] valor := respuesta;
                  mejor_mov := i;
                end;
              end;
              [7] i := i + 1;
            end;
          end;
        end;
      end;
    end;
end;
```

Figura 10.73 Algoritmo minimax para el juego de tres en raya con una poda $\alpha\text{-}\beta$: selección del computador

La implantación de la poda $\alpha\text{-}\beta$ requiere un código sorprendentemente pequeño. No es tan difícil como podría pensarse, aunque muchos programadores tardan mucho tiempo en hacerlo sin buscar en un libro de referencia. La figura 10.73 muestra la mitad del esquema de la poda $\alpha\text{-}\beta$ (sin las declaraciones de tipos); no debe ser problemático codificar la otra mitad.

Para aprovechar al máximo la poda $\alpha\text{-}\beta$, los programas para juegos suelen intentar aplicar una función de evaluación en los nodos no terminales, buscando colocar los mejores movimientos en algún momento temprano de la búsqueda. El resultado es que se poda más de lo que uno esperaría en una ordenación aleatoria de los nodos. También se emplean otras técnicas, como la búsqueda más profunda en las líneas de juego más activas.

En la práctica, la poda α - β restringe la búsqueda a sólo $O(\sqrt{n})$ nodos, donde n es el tamaño del árbol de juego completo. Este ahorro es enorme y significa que las búsquedas que usan la poda α - β pueden ir al doble de la profundidad en comparación con un árbol no podado. Nuestro ejemplo del tres en raya no es ideal porque existen muchos valores idénticos, pero aun así, la búsqueda inicial de 97 162 nodos se reduce a 4493. (La cuenta incluye nodos no terminales.)

En muchos juegos, los computadores se encuentran entre los mejores jugadores del mundo. Las técnicas usadas son muy interesantes, y pueden aplicarse a otros problemas más serios. Se pueden encontrar más detalles en las referencias.

Resumen

Este capítulo ilustra cinco de las técnicas más comunes en el diseño de algoritmos. Al confrontar algún problema, vale la pena ver si se aplica cualquiera de estos métodos. A menudo, una elección adecuada del algoritmo, en combinación con el uso juicioso de las estructuras de datos, puede conducir rápidamente a soluciones eficientes.

Ejercicios

- 10.1 Demuestre que funciona el algoritmo ávido para minimizar el tiempo medio de terminación de la planificación de tareas con procesadores múltiples.
- 10.2 La entrada es un conjunto de trabajos j_1, j_2, \dots, j_n , cada uno de los cuales tarda una unidad de tiempo en terminar. Cada trabajo j_i gana d_i dólares si se termina en el límite de tiempo t_i , y no gana nada si termina después del límite.
 - a. Proporcione un algoritmo ávido $O(n^2)$ para resolver el problema.
 - **b. Modifique su algoritmo para obtener una cota de tiempo $O(n \log n)$. *Sugerencia: La cota de tiempo se debe por completo a la ordenación de los trabajos de acuerdo con el dinero. El resto del algoritmo se puede implantar usando la estructura de datos de conjunto ajeno, en $O(n \log n)$.*
- 10.3 Un archivo contiene sólo puntos, espacios, cambios de línea, comas y dígitos con las frecuencias siguientes: punto (100), espacio (605), cambio de línea (100), coma (705), 0 (431), 1 (242), 2 (176), 3 (59), 4 (185), 5 (250), 6 (174), 7 (199), 8 (205), 9 (217). Construya el código de Huffman.
- 10.4 Parte del archivo codificado debe ser una cabecera que indique el código de Huffman. Proporcione un método para construir la cabecera de tamaño máximo $O(n)$ (además de los símbolos), donde n es el número de símbolos.
- 10.5 Complete la demostración de que el algoritmo de Huffman genera un código prefijo óptimo.
- 10.6 Demuestre que si los símbolos se ordenan de acuerdo con la frecuencia, el algoritmo de Huffman se puede implantar en tiempo lineal.

- 10.7 Escriba un programa para implantar la compresión (y descompresión) de archivos, usando el algoritmo de Huffman.
- *10.8 Demuestre que cualquier algoritmo en línea de empaquetamiento en recipientes se puede forzar a usar al menos $\frac{3}{2}$ del número óptimo de recipientes, considerando la siguiente secuencia de elementos: n elementos de tamaño $\frac{1}{6} - 2\epsilon$, n elementos de tamaño $\frac{1}{3} + \epsilon$, n elementos de tamaño $\frac{1}{2} + \epsilon$.
- 10.9 Explique cómo implantar primer ajuste y mejor ajuste en tiempo $O(n \log n)$.
- 10.10 Muestre la operación de todas las estrategias de empaquetamiento en recipientes presentadas en la sección 10.1.3 sobre la entrada 0.42, 0.25, 0.27, 0.07, 0.72, 0.86, 0.09, 0.44, 0.50; 0.68, 0.73, 0.31, 0.78, 0.17, 0.79, 0.37, 0.73, 0.23, 0.30.
- 10.11 Escriba un programa que compare el rendimiento (tanto en tiempo como en el número de recipientes usados) de las diferentes heurísticas de empaquetamiento en recipientes.
- 10.12 Demuestre el teorema 10.7.
- 10.13 Demuestre el teorema 10.8.
- *10.14 En una unidad cuadrada se colocan n puntos. Demuestre que la distancia entre el par más cercano es $O(n^{-1/2})$.
- *10.15 Justifique que, para el algoritmo de los puntos más cercanos, el número medio de puntos en la franja es $O(\sqrt{n})$. *Sugerencia: Utilice el resultado del ejercicio anterior.*
- 10.16 Escriba un programa para implantar el algoritmo del par más cercano.
- 10.17 ¿Cuál es el tiempo de ejecución asintótico de la selección rápida, si se usa una estrategia de partición con base en la mediana de la mediana de tres?
- 10.18 Demuestre que la ordenación rápida con la partición con base en la mediana de la mediana de siete es lineal. ¿Por qué no se usa la partición con base en la mediana de la mediana de siete en la demostración?
- 10.19 Implemente el algoritmo de selección rápida del capítulo 7, la selección rápida que usa una partición de la mediana de la mediana de cinco y el algoritmo de muestreo del final de la sección 10.2.3. Compare los tiempos de ejecución.
- 10.20 Buena parte de la información empleada para calcular la mediana de la mediana de cinco se desecha. Muestre cómo se puede reducir el número de comparaciones a partir de un uso más cuidadoso de la información.
- *10.21 Termine el análisis del algoritmo de muestreo descrito al final de la sección 10.2.3, y explique cómo se eligen los valores de δ y s .
- 10.22 Muestre cómo es que el algoritmo de la multiplicación recursiva calcula xy , donde $x = 1234$ y $y = 4321$. Incluya todos los cálculos recursivos.
- 10.23 Muestre cómo multiplicar dos números complejos $x = a + bi$ y $y = c + di$ con sólo tres multiplicaciones.
- 10.24 a. Demuestre que

$$x_1y_r + x_ry_1 = (x_1 + x_r)(y_1 + y_r) - x_1y_1 - x_ry_r$$

- b. Esto da un algoritmo $O(n^{1.5})$ para multiplicar números de n bits. Compare este método con la solución del texto.
- 10.25 *a. Muestre cómo multiplicar dos números resolviendo cinco problemas que sean casi la tercera parte del tamaño original.
 *b. Generalice este problema para obtener un algoritmo $O(n^{1+\epsilon})$ para cualquier constante $\epsilon > 0$.
 c. ¿Es mejor el algoritmo de la parte (b) que $O(n \log n)$?
- 10.26 ¿Por qué es importante que el algoritmo de Strassen no use la commutatividad en la multiplicación de dos matrices?
- 10.27 Se pueden multiplicar dos matrices de 70×70 con 143 640 multiplicaciones. Muestre cómo puede aprovecharse esto para mejorar la cota dada por el algoritmo de Strassen.
- 10.28 ¿Cuál es la forma óptima de calcular $A_1 A_2 A_3 A_4 A_5 A_6$, donde las dimensiones de las matrices son: $A_1: 10 \times 20$, $A_2: 20 \times 1$, $A_3: 1 \times 40$, $A_4: 40 \times 5$, $A_5: 5 \times 30$, $A_6: 30 \times 15$?
- 10.29 Demuestre que no funciona ninguno de los siguientes algoritmos ávidos para la multiplicación enlazada de matrices. En cada paso:
 a. obtenga la multiplicación más económica;
 b. calcule la multiplicación más costosa;
 c. calcule la multiplicación entre las dos matrices M_i y M_{i+1} , tal que el número de columnas en M_i sea minimizado (deshaciendo igualdades por medio de alguna de las reglas anteriores).
- 10.30 Escriba un programa para calcular la mejor ordenación de la multiplicación de matrices. Incluye la rutina para visualizar la ordenación real.
- 10.31 Muestre el árbol binario de búsqueda óptimo para las siguientes palabras, donde la frecuencia de aparición se indica entre paréntesis: *a* (0.18), *antes* (0.19), *el* (0.23), *es* (0.21), *o* (0.19).
- *10.32 Extienda el algoritmo para árboles binarios de búsqueda óptimos, a fin de que permita búsquedas no exitosas. En este caso, q_j , para $1 \leq j < n$, es la probabilidad de que una búsqueda se ejecute para cualquier palabra W que satisfaga $w_i < W < w_{j+1}$. q_0 es la probabilidad de realizar una búsqueda para $W < w_0$, y q_n es la probabilidad de realizar una búsqueda para $W > w_n$. Nótese que $\sum_{i=1}^n p_i + \sum_{j=0}^n q_j = 1$.
- *10.33 Supóngase que $C_{ii} = 0$ y que en otro caso

$$C_{ij} = W_{ij} + \min_{i < k < j} (C_{ik-1} + C_{kj})$$

Suponga que W satisface la *desigualdad del cuadrángulo*, a saber, para toda $i \leq i' \leq j \leq j'$,

$$W_{ij} + W_{i'j'} \leq W_{i'j} + W_{ij'}$$

Suponga además que W es *monótona*: si $i \leq i'$ y $j \leq j'$, entonces $W_{ij} \leq W_{i'j'}$.
 a. Compruebe que C satisface la desigualdad del cuadrángulo.

- b. Sea R_{ij} la k mayor que alcanza la mínima $C_{ik-1} + C_{kj}$. (Esto es, en caso de igualdades, escoja la k mayor). Demuestre que

$$R_{ij} \leq R_{ij-1} \leq R_{i+1,j+1}$$

- c. Demuestre que R es no decreciente sobre cada fila y columna.
 d. Use esto para demostrar que todas las entradas en C se pueden calcular en un tiempo $O(n^3)$.
 e. ¿Cuál de los algoritmos de programación dinámica se puede resolver en $O(n^2)$ con esas técnicas?
- 10.34 Escriba una rutina para reconstruir los caminos más cortos a partir del algoritmo de la sección 10.3.4.
- 10.35 Examine el generador de números aleatorios en su sistema. ¿Qué tan aleatorio es?
- 10.36 Escriba las rutinas para realizar la inserción, la eliminación y la búsqueda en listas con saltos.
- 10.37 Proporcione una demostración formal de que el tiempo esperado para las operaciones en listas con saltos es $O(\log n)$.
- 10.38 La figura 10.74 muestra una rutina para lanzar una moneda, suponiendo que *aleatorio* devuelve un entero (lo que sucede en muchos sistemas). ¿Cuál es el rendimiento esperado de los algoritmos en listas con saltos si el generador de números aleatorios usa un módulo de la forma $m = 2^b$ (lo cual desafortunadamente prevalece en muchos sistemas)?
- 10.39 a. Mediante el algoritmo de exponenciación demuestre que $2^{340} \equiv 1 \pmod{341}$.
 b. Muestre cómo funciona la comprobación aleatorizada de la primalidad para $n = 561$ con varias elecciones de a .
- 10.40 Implemente el algoritmo de reconstrucción de caminos de cuota.
- 10.41 Dos conjuntos de puntos son *homométricos* si producen el mismo conjunto de distancias y no son rotaciones uno del otro. El siguiente conjunto de distancias proporciona dos conjuntos de puntos diferentes: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 17. Encuentre los dos conjuntos de puntos.

Figura 10.74 Lanzador de monedas de calidad cuestionable

```
type
  cara_moneda = (cara, cruz);

function lanza: cara_moneda;
begin
  if (aleatorio mod 2) = 0 then (usa el bit menos significativo)
    lanza := cara
  else
    lanza := cruz;
end;
```

- 10.42 Extienda el algoritmo de reconstrucción para encontrar *todos* los conjuntos de puntos homométricos dado un conjunto de distancias.
- 10.43 Muestre el resultado de la poda α - β sobre el árbol de la figura 10.75.
- 10.44 a. El código de la figura 10.73, ¿implanta una poda α o β ?
b. Implante la rutina complementaria.
- 10.45 Escriba los procedimientos restantes para el juego de tres en raya.
- 10.46 El *problema de empaquetamiento unidimensional de círculos* es como se indica a continuación. Se tienen n círculos de radios r_1, r_2, \dots, r_n . Estos círculos se empaquetan en una caja de modo que cada círculo es tangente a la base de la caja, y se acomodan en el orden original. El problema es encontrar el ancho de la caja de tamaño mínimo. La figura 10.76 muestra un ejemplo con círculos de radios 2, 1 y 2, respectivamente. La caja de tamaño mínimo tiene un ancho de $4 + 4\sqrt{2}$.
- *10.47 Suponga que las aristas de un grafo no dirigido G satisfacen la desigualdad del triángulo: $c_{u,v} + c_{v,w} \geq c_{u,w}$. Muestre cómo calcular un recorrido del agente viajero con un costo del doble del óptimo como máximo. Sugerencia: Construya un árbol de extensión mínima.

Figura 10.75 Árbol de juego, que puede ser podado

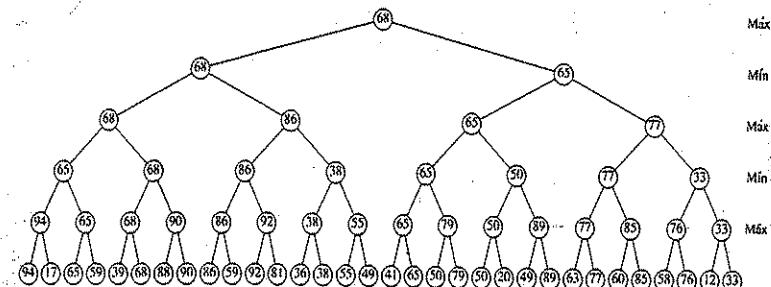
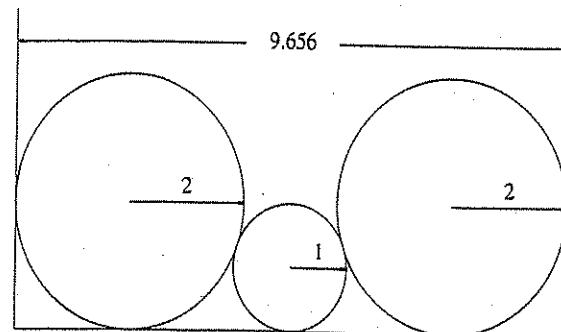
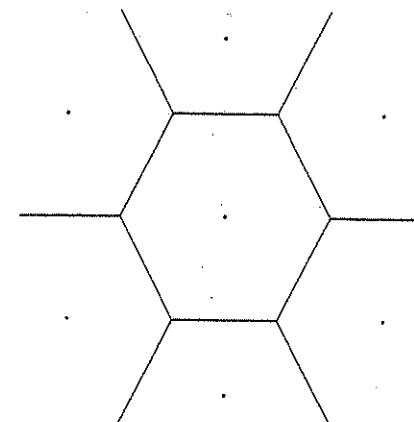


Figura 10.76 Ejemplo del problema de empaquetamiento de círculos



- *10.48 Usted es director de un torneo y necesita organizar un esquema de todos contra todos entre $n = 2^k$ jugadores. En este torneo, cada uno juega exactamente un juego diario; después de $n - 1$ días, ha tenido lugar un encuentro entre todo par de jugadores. Proporcione un algoritmo para hacer esto.
- 10.49 a. Compruebe que en un torneo de todos contra todos siempre es posible acomodar a los jugadores en un orden j_1, j_2, \dots, j_n tal que para toda $1 \leq k < n$, j_k ha ganado el encuentro contra j_{k+1} .
b. Proporcione un algoritmo $O(n \log n)$ para encontrar tal acomodo. El algoritmo debe servir como demostración para la parte (a).
- *10.50 Tenemos un conjunto $P = p_1, p_2, \dots, p_n$ de n puntos en un plano. Un *diagrama de Voronoi* es una partición del plano en n regiones R_i tales que todos los puntos en R_i están más cercanos a p_i que cualquier otro punto en P . La figura 10.77 ilustra un diagrama de Voronoi para siete puntos (en un agradable acomodo). Proporcione un algoritmo $O(n \log n)$ para construir el diagrama de Voronoi.
- *10.51 Un *polígono convexo* es un polígono con la propiedad de que cualquier segmento de recta cuyos extremos estén en el polígono cae por completo dentro del polígono. El problema del *casco convexo* consiste en encontrar el área más pequeña de un polígono convexo que encierre un conjunto de puntos en el plano. La figura 10.78 muestra el casco convexo de un conjunto de 40 puntos. Proporcione un algoritmo $O(n \log n)$ para encontrar el casco convexo.
- *10.52 Considere el problema de la justificación a la derecha de un párrafo. El párrafo contiene una secuencia de palabras p_1, p_2, \dots, p_n de longitud a_1, a_2, \dots, a_n , las cuales queremos partir en líneas de longitud L . Las palabras están separadas por espacios blancos cuya longitud ideal es b (milímetros), pero los blancos se pueden alargar o contraer tanto como sea necesario (pero debe ser > 0), de modo que una línea $p_ip_{i+1}\dots p_j$ tenga una longitud exacta L . No obstante, para cada blanco b' se cargan $|b' - b|$ puntos desagradables. La excepción es la última:

Figura 10.77 Diagrama de Voronoi



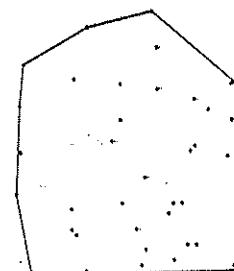
línea, para la cual sólo cargamos algo si $b' < b$ (en otras palabras, sólo se carga algo en caso de una contracción), puesto que la última línea no necesita justificación. Así, si b_i es la longitud del blanco entre a_i y a_{i+1} , entonces lo desagradable de poner cualquier línea (excepto la última) $p_ip_{i+1}...p_j$ para $j > i$ se mide como $\sum_{k=i}^{j-1} |b_k - b_l| = (j-i)(b' - b)$, donde b' es el tamaño promedio de un blanco en esta línea. Esto es cierto en la última línea sólo si $b' < b$, de otro modo la última línea no es desagradable en absoluto.

- Proporcione un algoritmo de programación dinámica para encontrar la asignación menos desagradable de p_1, p_2, \dots, p_n en líneas de longitud L . Sugerencia: Para $i = n, n-1, \dots, 1$, calcule la mejor forma de poner p_i, p_{i+1}, \dots, p_n .
- Proporcione las complejidades en tiempo y espacio para el algoritmo (como una función del número de palabras, n).
- Considere el caso especial en que se esté usando una impresora de líneas en vez de una impresora láser, y suponga que el valor óptimo de b es 1 (espacio). En este caso, no se permite la contracción de blancos, ya que el siguiente espacio más pequeño podría ser 0. Proporcione un algoritmo en tiempo lineal para generar la asignación menos desagradable en una impresora de líneas.

*1053 El problema de la *subsecuencia creciente más larga* es como sigue: Dados los números a_1, a_2, \dots, a_m , encuentre el valor máximo de k tal que $a_{i_1} < a_{i_2} < \dots < a_{i_k}$ e $i_1 < i_2 < \dots < i_k$. Por ejemplo, si la entrada es 3, 1, 4, 1, 5, 9, 2, 6, 5, la subsecuencia creciente más larga tiene longitud cuatro (1, 4, 5, 9 entre otras). Proporcione un algoritmo $O(n^2)$ para resolver el problema de la subsecuencia creciente más larga.

*1054 El problema de la *subsecuencia común más larga* es como se indica: Dadas dos secuencias $A = a_1, a_2, \dots, a_n$ y $B = b_1, b_2, \dots, b_n$, encuentre la longitud, k , de la

Figura 10.78 Ejemplo de un casco convexo



secuencia más larga $C = c_1, c_2, \dots, c_k$ de tal modo que C es una subsecuencia tanto de A como de B . Por ejemplo, si

$$A = p, r, o, g, r, a, m, a, c, i, o, n$$

y

$$B = d, i, n, a, m, i, c, a,$$

entonces la subsecuencia común más larga es a, m y tiene longitud 2. Proporcione un algoritmo para resolver el problema de la subsecuencia común más larga. El algoritmo se debe ejecutar en un tiempo $O(mn)$.

*1055 El problema de la *correspondencia de patrones* es como sigue: Dados una cadena C , y un patrón P , encuentre la primera aparición de P en C . La *correspondencia aproximada de patrones* permite k no correspondencias de tres tipos:

- Un carácter puede estar en C pero no en P ;
- Un carácter puede estar en P pero no en C ;
- P y C pueden diferir en una posición.

Por ejemplo, si buscamos el patrón "libreta" con a lo más tres no correspondencias en la cadena "librta de estructuras de datos", encontramos una correspondencia (se inserta una e y se cambia una s por una a). Proporcione un algoritmo $O(mn)$ para resolver el problema de la correspondencia aproximada de cadenas, donde $m = |P|$ y $n = |C|$.

*1056 Una forma del problema de la *mochila* es como sigue. Tenemos un conjunto de enteros $A = a_1, a_2, \dots, a_n$ y un entero K . ¿Hay un subconjunto de A cuya suma sea exactamente igual a K ?

- Proporcione un algoritmo que resuelva el problema de la mochila en un tiempo $O(nK)$.
- ¿Por qué esto no demuestra que $P = NP$?

*1057 Se tiene un sistema monetario con monedas de valor (decreciente) c_1, c_2, \dots, c_n centavos.

- Prepare un algoritmo que calcule el número mínimo de monedas que se requieren para dar K centavos en cambio.
- Proporcione un algoritmo que calcule el número de formas diferentes de dar K centavos de cambio.

*1058 Considere el problema de colocar ocho reinas en un tablero de ajedrez (de ocho por ocho). Se dice que dos reinas se atacan si están en la misma fila, columna o diagonal (no necesariamente la principal).

- Prepare un algoritmo aleatorizado para colocar ocho reinas en el tablero sin que se ataquen.
- Proporcione un algoritmo con retroceso para resolver el mismo problema.
- Implante ambos algoritmos y compare el tiempo de ejecución.

*1059 En el juego de ajedrez, un caballo en la fila r y la columna c se puede mover a la fila $1 \leq r' \leq T$ y la columna $1 \leq c' \leq T$ (donde T es el tamaño del tablero), siempre y cuando

$$|r - r'| = 2 \text{ y } |c - c'| = 1$$

o

$$|r - r'| = 1 \text{ y } |c - c'| = 2$$

Una *vuelta del caballo* es una secuencia de movimientos que visita todas las casillas exactamente una vez antes de regresar a la casilla inicial.

- a. Demuestre que la vuelta del caballo no puede existir si T es impar.
 - b. Proporcione un algoritmo con retroceso para encontrar la vuelta del caballo.
- 10.60 Considere el algoritmo recursivo de la figura 10.79 para encontrar el camino más corto ponderado en un grafo acíclico, de s a t .
- a. ¿Por qué no funciona este algoritmo en grafos generales?
 - b. Compruebe que este algoritmo termina al aplicarse sobre grafos acíclicos.
 - c. ¿Cuál es el tiempo de ejecución para el peor caso del algoritmo?

Referencias

El artículo original sobre los códigos de Huffman es [21]. En [29], [31] y [32] se estudian variaciones del algoritmo. Otro conocido esquema de compresión es la codificación de Ziv-Lempel [52], [53]. Aquí los códigos tienen una longitud fija pero representan cadenas en vez de caracteres. [3] y [34] son buenas investigaciones sobre los esquemas de compresión comunes.

Figura 10.79 Algoritmo recursivo del camino más corto

```

function más_corto(s, t: vértice; G: grafo): distancia;
  var di, temp: distancia;
begin
  if s = t then
    di := 0
  else
    begin
      di := ∞;
      for cada vértice v adyacente a s do
        begin
          temp := más_corto(v, t, G);
          if csv + temp < di then
            di := csv + temp;
        end;
      más_corto := di;
    end;
end;

```

El análisis de las heurísticas de empaquetamiento en recipientes apareció por primera vez en la tesis doctoral de Johnson y se publicó en [22]. La cota inferior mejorada del empaquetamiento en línea en recipientes dada en el ejercicio 10.8 proviene de [50]; este resultado fue mejorado después en [35]. [44] describe otro enfoque sobre el empaquetamiento en línea en recipientes.

El teorema 10.7 procede de [6]. El algoritmo de los puntos más cercanos apareció en [45]. [47] describe el problema de la reconstrucción de caminos de cuota y sus aplicaciones. [14] y [40] son dos libros sobre el relativamente nuevo campo de la geometría computacional. [2] contiene las notas de un curso sobre geometría computacional impartido en el Massachusetts Institute of Technology; incluye una extensa bibliografía.

El algoritmo de selección en tiempo lineal apareció en [8]. [17] analiza el enfoque de muestreo que encuentra la mediana en $1.5n$ comparaciones esperadas. La multiplicación $O(n^{1.5})$ procede de [23]. En [9] y [24] se estudian generalizaciones. El algoritmo de Strassen aparece en el pequeño artículo [48]. Éste establece los resultados y no mucho más. Pan [38] da varios algoritmos de "divide y vencerás", entre ellos el del ejercicio 10.27. La cota mejor conocida es $O(n^{2.37})$, la cual se debe a Coppersmith y Winograd [13].

Las referencias clásicas sobre la programación dinámica son los libros [4] y [5]. El problema de la ordenación de matrices fue estudiado por primera vez en [19]. En [20] se demostró que el problema se puede resolver en tiempo $O(n \log n)$.

Knuth [25] proporcionó un algoritmo $O(n^3)$ para la construcción de árboles binarios de búsqueda óptimos. El algoritmo del camino más corto entre todos los pares se debe a Floyd [16]. Un algoritmo $O(n^3(\log \log n / \log n)^{1/3})$ teóricamente mejor fue dado por Fredman [18], pero no es práctico, lo cual no es sorprendente. En ciertas condiciones, el tiempo de ejecución de los programas dinámicos se puede mejorar automáticamente en un factor de n o más. Esto se estudia en el ejercicio 10.33, así como en [15] y [51].

El análisis de los generadores de números aleatorios se basa en [39]. Park y Miller atribuyen la implantación transportable a Schrage [46]. Las listas con saltos se estudian en Pugh [41]. El algoritmo aleatorizado para comprobar la primalidad se debe a Miller [36] y Rabin [43]. El teorema de que cuando mucho $(n - 9)/4$ valores de a engañan al logaritmo es de Monier [37]. En [42] se analizan otros algoritmos aleatorizados.

En [1], [26] y [27] se puede encontrar más información sobre la poda $\alpha-\beta$. Los mejores programas que juegan ajedrez, damas, Otelo y backgammon han alcanzado el estatuto de clase mundial. [33] describe un programa de Otelo. El artículo aparece en un ejemplar especial sobre juegos por computador (sobre todo ajedrez); este ejemplar es una mina de oro en ideas. Uno de los artículos describe el uso de la planificación para resolver finales de ajedrez completamente cuando quedan pocas piezas sobre el tablero. En ciertos casos, investigaciones relacionadas han ocasionado el cambio de la regla de 50 movimientos.

El ejercicio 10.41 está resuelto en [7]. Éste es el único caso conocido de un conjunto de puntos homométrico sin distancias duplicadas. Está abierta la determinación de si existen otros para $n > 6$. Christofides [12] da una solución al ejercicio 10.47, así como un algoritmo que genera un viaje cuya optimidad es de $\frac{3}{2}$. El ejercicio

10.52 se estudia en [28]. El ejercicio 10.55 está resuelto en [49]. Un algoritmo $O(kn)$ está en [30]. El ejercicio 10.57 se analiza en [10], pero no hay que confundirse por el título del artículo.

1. B. Abramson, "Control Strategies for Two-Player Games", *ACM Computing Surveys*, 21, (1989), págs. 137-161.
2. A. Aggarwal y J. Wein, *Computational Geometry: Lecture Notes for 18.409*, MIT Laboratory for Computer Science, 1988.
3. T. Bell, I. H. Witten y J. G. Cleary, "Modeling for Text Compression", *ACM Computing Surveys*, 21 (1989), págs. 557-591.
4. R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, N.J., 1957.
5. R. E. Bellman y S. E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J., 1962.
6. J. L. Bentley, D. Haken y J. B. Saxe, "A General Method for Solving Divide-and-Conquer Recurrences", *SIGACT News*, 12 (1980), págs. 36-44.
7. G. S. Bloom, "A Counterexample to the Theorem of Picard", *Journal of Combinatorial Theory A* (1977), págs. 378-379.
8. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest y R. E. Tarjan, "Time Bounds for Selection", *Journal of Computer and System Sciences*, 7 (1973), págs. 448-461.
9. A. Borodin y J. I. Munro, *The Computational Complexity of Algebraic and Numerical Problems*, American Elsevier, Nueva York, 1975.
10. L. Chang y J. Korsh, "Canonical Coin Changing and Greedy Solutions", *Journal of the ACM*, 23 (1976), págs. 418-422.
11. N. Christofides, "Worst-case Analysis of a New Heuristic for the Traveling Salesman Problem", *Management Science Research Reports # 388*, Carnegie-Mellon University, Pittsburgh, PA, 1976.
12. D. Coppersmith y S. Winograd, "Matrix Multiplication via Arithmetic Progressions", *Proceedings of the Nineteenth Annual ACM Symposium of the Theory of Computing* (1987), págs. 1-6.
13. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlín, 1987.
14. D. Eppstein, Z. Galil y R. Giancarlo, "Speeding up Dynamic Programming", *Twenty-ninth Annual IEEE Symposium on the Foundations of Computer Science* (1988), págs. 488-495.
15. R. W. Floyd, "Algorithm 97: Shortest Path", *Communications of the ACM*, 5, (1962), pág. 345.
16. R. W. Floyd y R. L. Rivest, "Expected Time Bounds for Selection", *Communications of the ACM*, 18 (1975), págs. 165-172.
17. M. L. Fredman, "New Bounds on the Complexity of the Shortest Path Problem", *SIAM Journal on Computing*, 5 (1976), págs. 83-89.
18. S. Godbole, "On Efficient Computation of Matrix Chain Products", *IEEE Transactions on Computers*, 9 (1973), págs. 864-866.
19. T. C. Hu y M. R. Shing, "Computations of Matrix Chain Products, Part I", *SIAM Journal on Computing*, 11 (1982), págs. 362-373.
20. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proceedings of the IRE*, 40 (1952), págs. 1098-1101.
21. D. S. Johnson, A. Derners, J. D. Ullman, M. R. Garey y R. L. Graham, "Worst-case Performance Bounds for Simple One-Dimensional Packing Algorithms", *SIAM Journal on Computing*, 3 (1974), págs. 299-325.
22. A. Karatsuba y Y. Ofman, "Multiplication of Multi-digit Numbers on Automata", *Doklady Akademii Nauk SSSR*, 145 (1962), págs. 293-294.

23. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, 2a. ed., Addison-Wesley, Reading, MA, 1981.
24. D. E. Knuth, "Optimum Binary Search Trees", *Acta Informatica*, 1 (1971), págs. 14-25.
25. D. E. Knuth y R. W. Moore, "Estimating the Efficiency of Backtrack Programs", *Mathematics of Computation*, 29 (1975), págs. 121-136.
26. D. E. Knuth, "An Analysis of Alpha-Beta Cutoffs", *Artificial Intelligence*, 6 (1975), págs. 293-326.
27. D. E. Knuth, *TeX and Metafont, New Directions in Typesetting*, Digital Press, Bedford, MA, 1981.
28. D. E. Knuth, "Dynamic Huffman Coding", *Journal of Algorithms*, 6 (1985), págs. 163-180.
29. G. M. Landau y U. Vishkin, "Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm", *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing* (1986), págs. 220-230.
30. L. L. Larmore, "Height-Restricted Optimal Binary Trees", *SIAM Journal on Computing*, 16 (1987), págs. 1115-1123.
31. L. L. Larmore y D. S. Hirschberg, "A Fast Algorithm for Optimal Length-Limited Huffman Codes", *Journal of the ACM*, 37 (1990), págs. 464-473.
32. K. Lee y S. Mahajan, "The Development of a World Class Othello Program", *Artificial Intelligence*, 43 (1990), págs. 21-36.
33. D. A. Lelewer y D. S. Hirschberg, "Data Compression", *ACM Computing Surveys*, 19 (1987), págs. 261-296.
34. F. M. Liang, "A Lower Bound for On-line Bin Packing", *Information Processing Letters*, 10 (1980), págs. 76-79.
35. G. L. Miller, "Riemann's Hypothesis and Tests for Primality", *Journal of Computer and System Sciences*, 13 (1976), págs. 300-317.
36. L. Monier, "Evaluation and Comparison of Two Efficient Probabilistic Primality Testing Algorithms", *Theoretical Computer Science*, 12 (1980), págs. 97-108.
37. V. Pan, "Strassen's Algorithm is Not Optimal", *Proceedings of the Nineteenth Annual IEEE Symposium on the Foundations of Computer Science* (1978), págs. 166-176.
38. S. K. Park y K. W. Miller, "Random Number Generators: Good Ones are Hard to Find", *Communications of the ACM*, 31 (1988), págs. 1192-1201.
39. F. P. Preparata y M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Nueva York, NY, 1985.
40. W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Communications of the ACM*, 33 (1990), págs. 668-676.
41. M. O. Rabin, "Probabilistic Algorithms", en *Algorithms and Complexity, Recent Results and New Directions* (J. F. Traub, ed.), Academic Press, Nueva York, 1976, págs. 21-39.
42. M. O. Rabin, "Probabilistic Algorithms for Testing Primality", *Journal of Number Theory*, 12 (1980), págs. 128-138.
43. P. Ramanan, D. J. Brown, C. C. Lee y D. T. Lee, "On-line Bin Packing in Linear Time", *Journal of Algorithms*, 10 (1989), págs. 305-326.
44. M. I. Shamos y D. Hoey, "Closest-Point Problems", *Proceedings of the Sixteenth Annual IEEE Symposium on the Foundations of Computer Science* (1975), págs. 151-162.
45. L. Schrage, "A More Portable FORTRAN Random Number Generator", *ACM Transactions of Mathematics Software*, 5 (1979), págs. 132-138.
46. S. S. Skiena, W. D. Smith y P. Lemke, "Reconstructing Sets from Interpoint Distances", *Sixth Annual ACM Symposium on Computational Geometry* (1990), págs. 332-339.
47. V. Strassen, "Gaussian Elimination is Not Optimal", *Numerische Mathematik*, 13 (1969), págs. 354-356.
48. R. A. Wagner y M. J. Fischer, "The String-to String Correction Problem", *Journal of the ACM*, 21 (1974), págs. 168-173.

49. A. C. Yao, "New Algorithms for Bin Packing", *Journal of the ACM*, 27 (1980), págs. 207-227.
50. F. F. Yao, "Efficient Dynamic Programming Using Quadrangle Inequalities", *Proceedings of the Twelfth Annual ACM Symposium on the Theory of Computing* (1980), págs. 429-435.
51. J. Ziv y A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, IT23 (1977), págs. 337-343.
52. J. Ziv y A. Lempel, "Compression of Individual Sequences via Variable-rate Coding", *IEEE Transactions on Information Theory*, IT24 (1978), págs. 530-536.

Análisis amortizado

En este capítulo analizaremos el tiempo de ejecución de varias de las estructuras de datos avanzadas que presentamos en los capítulo 4 y 6. En particular, consideraremos el tiempo de ejecución para el peor caso de cualquier secuencia de m operaciones. Esto contrasta con el análisis más común, en el cual se da una cota del peor caso para cualquier operación *simple*.

Por ejemplo, hemos visto que los árboles AVL permiten las operaciones estándar sobre árboles en un tiempo por operación de $O(\log n)$ para el peor caso. Es un tanto complicado implantar árboles AVL, no sólo porque existe una multitud de casos, sino también porque hay que mantener y actualizar correctamente información acerca del equilibrio en altura. La razón de que se usen los árboles AVL es que una secuencia de $\Theta(n)$ operaciones sobre un árbol de búsqueda no equilibrado podría requerir un tiempo $\Theta(n^2)$, que sería costoso. Para árboles de búsqueda, el tiempo de ejecución $O(n)$ del peor caso de una operación no es el problema real. El principal problema es que esto podría ocurrir repetidas veces. Los árboles desplegados ofrecen una buena alternativa. Aunque cualquier operación todavía puede requerir un tiempo $\Theta(n)$, este comportamiento degenerado no puede ocurrir en repetidas ocasiones, y podemos demostrar que cualquier secuencia de m operaciones tarda un tiempo (total) para el peor caso de $O(m \log n)$. Así, a la larga, esta estructura de datos se comporta como si cada operación tomara $O(\log n)$. Esta es la llamada *cota de tiempo amortizado*.

Las cotas amortizadas son más débiles que las cotas correspondientes para el peor caso, porque no ofrecen garantía para ninguna operación simple. Como por lo regular esto no es importante, estamos dispuestos a sacrificar la cota en una operación simple, si podemos retener la misma cota para la secuencia de operaciones y al mismo tiempo simplificar la estructura de datos. Las cotas amortizadas son más fuertes que la cota del caso medio. Por ejemplo, los árboles binarios de búsqueda tienen un tiempo medio $O(\log n)$ por operación, pero aun así es posible que una secuencia de m operaciones tome un tiempo $O(mn)$.

Debido a que para obtener una cota amortizada se necesita que examinemos una secuencia completa de operaciones en vez de una sola, esperamos que el análisis sea un poco más sofisticado. Veremos que esta expectativa se cumple en general.

En este capítulo:

- analizaremos las operaciones sobre colas binomiales;
- analizaremos los montículos oblicuos;
- presentaremos y analizaremos los montículos de Fibonacci;
- estudiaremos los árboles desplegados.

11.1. Un acertijo no relacionado

Consideremos el siguiente acertijo: dos gatitos están en los extremos opuestos de un campo de fútbol americano, a 100 yardas de distancia. Caminan dirigiéndose uno al otro a una velocidad de 10 yardas por minuto. Al mismo tiempo, su madre está en un extremo del campo; y puede correr a 100 yardas por minuto. La madre corre de un gatito al otro, dando vueltas sin pérdida de velocidad, hasta que los gatitos (y con ellos su madre) se encuentran en el centro del campo. ¿Cuánto corrió la madre?

No es difícil resolver el juego con un cálculo de fuerza bruta. Se dejan los detalles al lector, pero es de esperar que este cálculo implique el cálculo de la suma de una serie geométrica infinita. Aunque este cálculo directo producirá una respuesta, ocurre que es posible llegar a una solución más simple introduciendo una variable adicional, a saber, el tiempo.

Puesto que los gatitos están a 100 yardas y se acercan a una velocidad combinada de 20 yardas por minuto, les toma cinco minutos llegar al medio campo. Como la madre corre a 100 yardas por minuto, su recorrido total es de 500 yardas.

Este acertijo ilustra el hecho de que a veces es más fácil resolver un problema por vías indirectas. Los análisis amortizados que efectuaremos se apoyarán en esta idea. Presentaremos una variable adicional, conocida como *potencial*, para poder demostrar resultados que de otro modo parecen muy difíciles de establecer.

11.2. Colas binomiales

La primera estructura de datos que veremos es la cola binomial del capítulo 6, de la que ahora haremos un breve repaso. Recordaremos que un *árbol binomial* B_0 es un árbol de un nodo y, para $k > 0$, el árbol binomial B_k se construye combinando dos árboles binomiales B_{k-1} en uno. Los árboles binomiales B_0 a B_4 se muestran en la figura 11.1.

El *rango* de un nodo en un árbol binomial es igual al número de hijos; en particular, el rango de la raíz de B_k es k . Una *cola binomial* es una colección de árboles binomiales con orden de montículo, en la cual puede haber a lo más un

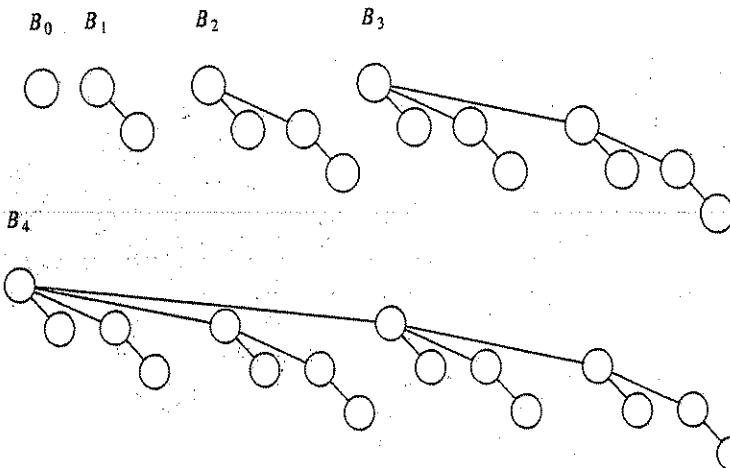


Figura 11.1 Árboles binomiales B_0, B_1, B_2, B_3 y B_4 .

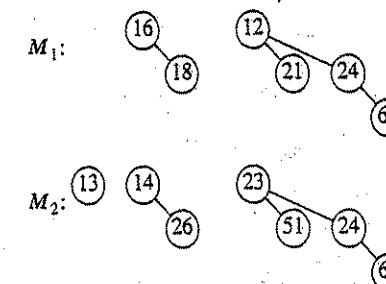
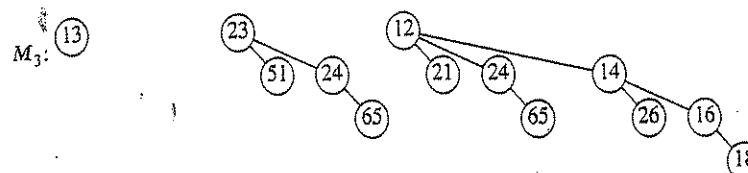


Figura 11.2 Dos colas binomiales M_1 y M_2

árbol binomial B_k para cualquier k . En la figura 11.2 se muestran dos colas binomiales, M_1 y M_2 .

La operación más importante es *fusionar*. Para fusionar dos colas binomiales se ejecuta una operación semejante a la suma de enteros binarios: en cualquier etapa se puede tener cero, uno, dos o posiblemente tres árboles B_k , dependiendo de si las dos colas de prioridad contienen un árbol B_k o no, y de si un árbol B_k es acarreado del paso anterior. Si hay cero o un árbol B_k , se coloca como árbol en la cola binomial resultante. Si hay dos árboles B_k , se unen en un árbol B_{k+1} y se acarrea; si hay tres árboles B_k , uno se coloca como árbol en la cola binomial y los otros dos se unen y acarrean. El resultado de la fusión de M_1 y M_2 se muestra en la figura 11.3.

La inserción se realiza creando una cola binomial de un nodo y efectuando un *fusionar*. El tiempo para hacer esto es $m+1$, donde m representa el tipo más pequeño de árbol binomial B_m no presente en la cola binomial. Así, la inserción en una cola binomial que tiene un árbol B_0 pero ningún árbol B_1 requiere dos pasos. La elimina-

Figura 11.3 Cola binomial M_3 ; el resultado de la fusión de M_1 y M_2

ción del mínimo se alcanza eliminando el mínimo y dividiendo la cola binomial original en dos colas binomiales, las cuales se fusionan después. En el capítulo 6 se da una explicación menos concisa de tales operaciones.

Primero consideraremos un problema muy sencillo. Supongamos que se quiere construir una cola binomial de n elementos. Sabemos que la construcción de un montículo binario de n elementos puede hacerse en $O(n)$; así, esperamos una cota semejante para colas binomiales.

PROPOSICIÓN:

Una cola binomial de n elementos se puede construir por medio de n inserciones sucesivas en tiempo $O(n)$.

La proposición, si es cierta, proporcionaría un algoritmo extremadamente simple. Puesto que el tiempo del peor caso por cada inserción es $O(\log n)$, no es obvio que la proposición sea verdadera. Recordaremos que si este algoritmo se aplicara a montículos binarios, el tiempo de ejecución sería $O(n \log n)$.

Para demostrar la proposición, podríamos hacer un cálculo directo. Para medir el tiempo de ejecución, definimos el costo de cada inserción como una unidad de tiempo más una unidad adicional para cada paso de enlazado. Sumando este costo de todas las inserciones se llega al tiempo de ejecución total. Este total es n unidades más el número total de pasos de enlazado. El primero, tercero, quinto y todos los pasos con número impar no requieren pasos de enlazado, ya que B_0 no está presente en el momento de la inserción. Así, la mitad de las inserciones no requiere pasos de enlazado. Una cuarta parte de las inserciones sólo requiere un paso de enlazado (segunda, sexta, décima, etc.). Una octava parte requiere dos, y así sucesivamente. Podríamos sumar todo esto y acotar el número de pasos de enlazado con base en n , demostrando la proposición. Este cálculo a base de fuerza bruta no ayudará cuando intentemos analizar una secuencia de operaciones que sólo contenga inserciones, por lo que utilizaremos otro enfoque para demostrar este resultado.

Consideremos el resultado de una inserción. Si en el momento de la inserción el árbol B_0 no está presente, entonces la inserción cuesta un total de una unidad de tiempo, usando la misma contabilidad que antes. El resultado de la inserción es que ahora hay un árbol B_0 , y así, se ha añadido un árbol al bosque de los árboles binomiales. Si hay un árbol B_0 pero ningún B_1 , entonces la inserción cuesta dos unidades. El bosque nuevo tendrá un árbol B_1 pero ya no tendrá árbol B_0 , así que el número de árboles en el bosque permanece sin cambio. Una inserción que cueste tres unidades creará un árbol B_2 pero destruirá los árboles B_0 y B_1 , dejando una

pérdida neta de un árbol en el bosque. De hecho, en general, es fácil ver que una inserción que cuesta c unidades produce un incremento neto de $2 - c$ árboles en el bosque, porque se crea un árbol B_{c-1} pero se eliminan todos los árboles B_i con $0 \leq i < c - 1$. Así, las inserciones costosas eliminan árboles, mientras que las inserciones económicas crean árboles.

Sea C_i el costo de la i -ésima inserción. Sea A_i el número de árboles después de la i -ésima inserción. $T_0 = 0$ es el número inicial de árboles. Entonces se tiene la invariante

$$C_i + (A_i - A_{i-1}) = 2 \quad (11.1)$$

Entonces se tiene

$$C_1 + (A_1 - A_0) = 2$$

$$C_2 + (A_2 - A_1) = 2$$

...

$$C_{n-1} + (A_{n-1} - A_{n-2}) = 2$$

$$C_n + (A_n - T_{n-1}) = 2$$

Si sumamos todas estas ecuaciones, la mayoría de los términos A_i se anulan, dejando

$$\sum_{i=1}^n C_i + A_n - A_0 = 2n$$

o, equivalentemente,

$$\sum_{i=1}^n C_i = 2n - (A_n - A_0)$$

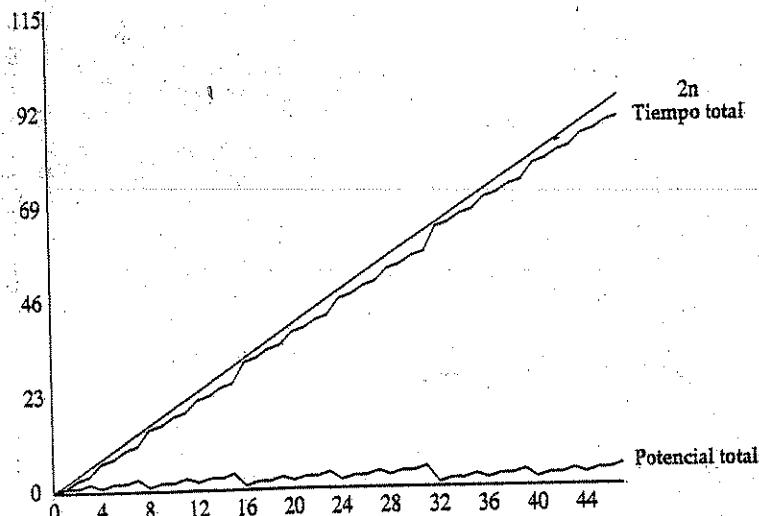
Recordaremos que $A_0 = 0$ y que con seguridad A_n , el número de árboles después de las n inserciones, es no negativo, por lo que $(A_n - A_0)$ es no negativo. Así

$$\sum_{i=1}^n C_i \leq 2n$$

lo cual demuestra la proposición.

Durante la rutina *construir_cola_binomial*, cada inserción tenía un tiempo de $O(\log n)$ para el peor caso, pero como la rutina completa usaba a lo más $2n$ unidades de tiempo, las inserciones se comportaban como si cada una usara no más de 2 unidades.

Este ejemplo ilustra la técnica general que usaremos. El estado de la estructura de datos en cualquier momento está dado por una función conocida como *potencial*. La función potencial no es mantenida por el programa, sino que más bien es un dispositivo de contabilidad que ayudará en el análisis. Cuando las operaciones toman menos tiempo que el que hemos asignado para ellas, el tiempo no usado se

Figura 11.4 Secuencia de n operaciones insertar

"ahorra" en la forma de un potencial más alto. En nuestro ejemplo, el potencial de la estructura de datos simplemente es el número de árboles. En el análisis anterior, cuando tenemos inserciones que sólo usan una unidad en vez de las dos unidades asignadas, la unidad adicional se guarda para después con un incremento en el potencial. Cuando las operaciones exceden el tiempo asignado, el exceso de tiempo es contabilizado como un decremento en el potencial. El potencial se puede ver como la representación de una cuenta de ahorros. Si una operación usa menos tiempo que el asignado, la diferencia se ahorra para usarla después en operaciones más costosas. La figura 11.4 muestra el tiempo de ejecución acumulado usado por `construir_cola_binomial` a lo largo de una secuencia de inserciones. Observe que el tiempo de ejecución nunca excede $2n$ y que el potencial en la cola binomial después de cualquier inserción mide la cantidad de ahorro.

Una vez elegida una función potencial, escribiremos la ecuación principal:

$$T_{\text{real}} + \Delta \text{Potencial} = T_{\text{amortizado}} \quad (11.2)$$

T_{real} , el *tiempo real* de una operación, representa la cantidad exacta (observada) de tiempo requerido para ejecutar una operación particular. En un árbol binario de búsqueda, por ejemplo, el tiempo real para realizar un *buscar(x)* es 1 más la profundidad del nodo que contiene a x . Si sumamos la ecuación básica sobre la secuencia completa, y si el potencial final es al menos tan grande como el potencial inicial, entonces el tiempo amortizado es una cota superior del tiempo real usado durante la ejecución de la secuencia. Observe que mientras T_{real} varía de operación a operación, $T_{\text{amortizado}}$ es estable.

Elegir una función potencial que pruebe una cota significativa es una tarea muy intrincada; no hay ningún método que se use. En general, se intenta con muchas funciones potenciales antes de encontrar la que funcione. Sin embargo, el análisis anterior sugiere unas cuantas reglas, las cuales indican las propiedades que tienen las buenas funciones potenciales. La función potencial debe cumplir lo siguiente:

- Siempre suponer su mínimo al principio de la secuencia. Un método popular de elección de la función potencial consiste en asegurar que la función potencial sea 0 inicialmente, y sea no negativa siempre. Todos los ejemplos que encontraremos aplicarán esta estrategia.
- Cancelar un término en el tiempo real. En nuestro caso, si el costo real era c , entonces el cambio del potencial era $2 - c$. Cuando se suman, se obtiene un costo amortizado de 2. Esto se muestra en la figura 11.5.

Ahora podemos realizar un análisis completo de las operaciones de colas binomiales.

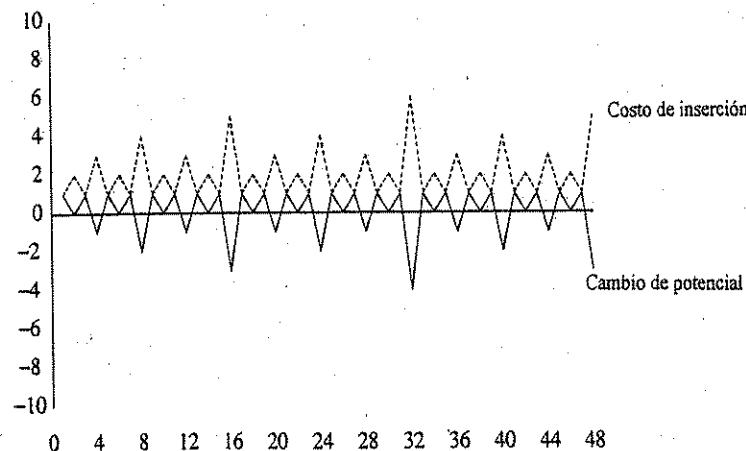
TEOREMA 11.1

Los tiempos de ejecución amortizados de insertar, eliminar_mín y fusionar son $O(1)$, $O(\log n)$ y $O(\log n)$, respectivamente, para colas binomiales.

DEMOSTRACIÓN:

La función potencial es el número de árboles. El potencial inicial es 0, y el potencial siempre es no negativo, así que el tiempo amortizado es una cota superior del tiempo real. El análisis de *insertar* se infiere del argumento anterior. Para *fusionar*, supongamos que los dos árboles tienen nodos n_1 y n_2 con árboles

Figura 11.5 El costo de inserción y el cambio de potencial para cada operación en una secuencia



A_1 y A_2 , respectivamente. Sea $n = n_1 + n_2$. El tiempo real para efectuar la fusión es $O(\log(n_1) + \log(n_2)) = O(\log n)$. Después de la fusión, puede haber a lo más $\log n$ árboles, así que el potencial se puede incrementar en $O(\log n)$ como máximo. Esto da una cota amortizada de $O(\log n)$. La cota de *eliminar_mín* se infiere de una manera semejante.

11.3. Montículos oblicuos

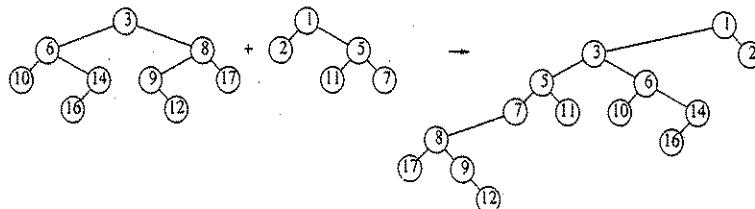
El análisis de colas binomiales es un ejemplo bastante fácil de un análisis amortizado. Ahora examinaremos los montículos oblicuos. Como es común con muchos de nuestros ejemplos, una vez encontrada la función potencial correcta, el análisis es fácil. La parte difícil es elegir una función potencial significativa.

Recordaremos que para los montículos oblicuos la operación clave es la fusión. Para fusionar dos montículos oblicuos combinamos sus caminos derechos y hacemos de éste el nuevo camino izquierdo. Para cada nodo del camino nuevo, excepto el último, el subárbol izquierdo anterior se enlaza como subárbol derecho. Se sabe que el último nodo del camino izquierdo nuevo no tiene subárbol derecho, así que es absurdo darle uno. La cota no depende de esta excepción, y si se codifica la rutina recursivamente esto sucederá de forma natural. La figura 11.6 muestra el resultado de la fusión de dos montículos oblicuos.

Supongamos que tenemos dos montículos, M_1 y M_2 , y dos nodos r_1 y r_2 en sus respectivos caminos derechos. Entonces el tiempo real para ejecutar la fusión es proporcional a $r_1 + r_2$, así que dejaremos la notación O grande y cargaremos una unidad de tiempo por cada nodo que se encuentre en los caminos. Puesto que los montículos no tienen estructura, es posible que todos los nodos en ambos montículos se encuentren en el camino derecho, y esto daría una cota para el peor caso $\Theta(n)$ en la fusión de montículos (el ejercicio 11.3 pide la construcción de un ejemplo). Demostraremos que el tiempo amortizado para fusionar dos montículos oblicuos es $O(\log n)$.

Lo que necesitamos es una especie de función potencial que capture el efecto de las operaciones sobre montículos oblicuos. Recordemos que el efecto de *fusión* es que todo nodo sobre el camino derecho se mueve al camino izquierdo, y su hijo izquierdo anterior se convierte en el nuevo hijo derecho. Una idea puede ser clasificar cada nodo como nodo derecho o izquierdo, dependiendo de si es o no un

Figura 11.6 Fusión de dos montículos oblicuos



hijo derecho, y usar el número de nodos derechos como la función potencial. Aunque al principio el potencial es 0 y siempre es no negativo, el problema es que el potencial no decrece después de una fusión y, por tanto, no refleja adecuadamente los ahorros en la estructura de datos. El resultado es que esta función potencial no se puede usar para probar la cota deseada.

Una idea semejante es clasificar los nodos como pesados o ligeros, dependiendo de si el subárbol derecho de cualquier nodo tiene o no más nodos que el subárbol izquierdo.

DEFINICIÓN:

Un nodo p es *pesado* si el número de descendientes del subárbol derecho de p es al menos la mitad del número de descendientes de p , y *ligero* en caso contrario. Observe que el número de descendientes de un nodo incluye al nodo mismo.

Por ejemplo, la figura 11.7 muestra un montículo oblicuo. Los nodos con llaves 15, 3, 6, 12 y 7 son pesados, y todos los demás son ligeros.

La función potencial que usaremos es el número de nodos pesados en (la colección de) los montículos. Esto parece una buena elección, porque un camino derecho largo contendrá un número excesivo de nodos pesados. Puesto que los hijos de los nodos de este camino son intercambiados, estos nodos se convertirán en nodos ligeros como resultado de la fusión.

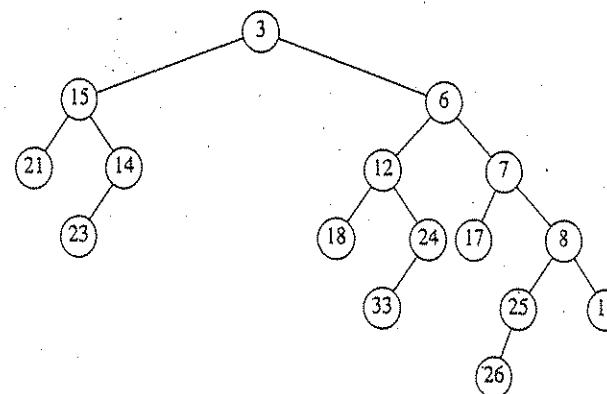
TEOREMA 11.2.

El tiempo amortizado para fusionar dos montículos oblicuos es $O(\log n)$.

DEMOSTRACIÓN:

Sean M_1 y M_2 los dos montículos, con nodos n_1 y n_2 , respectivamente. Supongamos que el camino derecho de M_1 tiene l_1 nodos ligeros y p_1 nodos pesados,

Figura 11.7 Montículo oblicuo—los nodos pesados son 3, 6, 7, 12 y 15



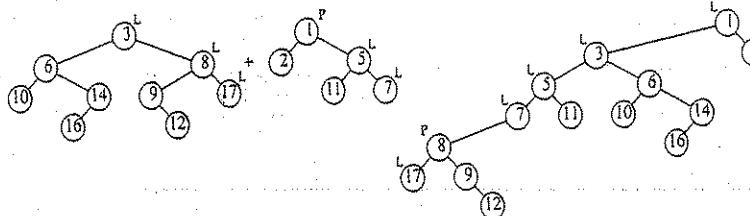


Figura 11.8 Cambio en el estado pesado/ligero después de una fusión

para un total de $l_1 + p_1$. De manera similar, M_2 tiene l_2 nodos ligeros y p_2 nodos pesados en su camino derecho, para un total de $l_2 + p_2$ nodos.

Si se adopta el convenio de que el costo de fusionar dos montículos oblicuos es el número total de nodos de sus caminos derechos, entonces el tiempo real para ejecutar la fusión es $l_1 + l_2 + p_1 + p_2$. Ahora los únicos nodos cuyo estado de pesado/ligero puede cambiar son aquellos que están inicialmente en el camino derecho (y llegan a un camino izquierdo), ya que ningún otro nodo tiene sus subárboles alterados. Esto se muestra en el ejemplo de la figura 11.8.

Si un nodo pesado está inicialmente en el camino derecho, después de la fusión debe convertirse en un nodo ligero. Los otros nodos que estaban en el camino derecho eran ligeros y pueden o no volverse pesados, pero como estamos demostrando una cota superior, tendremos que suponer lo peor, que es que se vuelvan pesados e incrementen el potencial. Entonces el cambio neto en el número de nodos pesados es, como máximo, $l_1 + l_2 - p_1 - p_2$. Sumando el tiempo real y el cambio del potencial (ecuación 11.2) se llega a una cota amortizada de $2(l_1 + l_2)$.

Ahora debemos demostrar que $l_1 + l_2 = O(\log n)$. Puesto que l_1 y l_2 son el número de nodos ligeros de los caminos derechos originales, y el subárbol derecho de un nodo ligero es de tamaño menor que la mitad del árbol con raíz en el nodo ligero, se infiere directamente que el número de nodos ligeros en el camino derecho es a lo más $\log n_1 + \log n_2$, el cual es $O(\log n)$.

La demostración se completa observando que el potencial inicial es 0 y que el potencial siempre es no negativo. Es importante verificar esto, ya que si no el tiempo amortizado no acota el tiempo real y carece de significado.

Puesto que las operaciones *insertar* y *eliminar_mín* son básicamente sólo operaciones *fusionar*, también tienen cotas amortizadas $O(\log n)$.

11.4. Montículos de Fibonacci

En la sección 9.3.2, mostramos cómo usar las colas de prioridad para mejorar el tiempo ingenuo de ejecución $O(|V|^3)$ del algoritmo del camino más corto de Dijkstra. La observación importante fue que el tiempo de ejecución estaba dominado por las $|A|$ operaciones *decrementar_llave* y $|V|$ operaciones *insertar* y *eliminar_mín*. Estas operaciones tienen lugar en un conjunto de tamaño máximo $|V|$.

Al utilizar un montículo binario, todas estas operaciones tardan un tiempo $O(\log |V|)$, así que la cota resultante del algoritmo de Dijkstra se puede reducir a $O(|A| \log |V|)$.

A fin de reducir esta cota de tiempo, debe mejorarse el tiempo requerido para efectuar la operación *decrementar_llave*. Los montículos- d , descritos en la sección 6.5, dan una cota de tiempo $O(\log_d |V|)$ para las operaciones *decrementar_llave* al igual que *insertar*, pero una cota $O(d \log_d |V|)$ para *eliminar_mín*. Eligiendo d para equilibrar los costos de $|A|$ operaciones *decrementar_llave* con $|V|$ operaciones *eliminar_mín*, y recordando que d siempre debe ser al menos 2, vemos que una buena elección de d es

$$d = \max(2, \lfloor |A| / |V| \rfloor).$$

Esto mejora la cota de tiempo del algoritmo de Dijkstra a

$$O(|A| \log_{\lceil |A| / |V| \rceil} |V|).$$

El *montículo de Fibonacci* es una estructura de datos que permite todas las operaciones básicas sobre montículos en un tiempo amortizado $O(1)$, con excepción de *eliminar_mín* y *eliminar*, las cuales toman un tiempo amortizado $O(\log n)$. De inmediato se sigue que las operaciones sobre montículos en el algoritmo de Dijkstra requerirán un tiempo total $O(|A| + |V| \log |V|)$.

Los montículos de Fibonacci[†] generalizan las colas binomiales agregando dos conceptos nuevos:

Una implantación diferente de decrementar_llave: El método visto antes consiste en filtrar el elemento hacia la raíz. No parece razonable esperar una cota amortizada de $O(1)$ para esta estrategia, así que se necesita un método nuevo.

Fusión perezosa: Dos montículos se fusionan sólo cuando es necesario. Esto es semejante a la eliminación perezosa. Para la fusión perezosa, *fusionar* es económica, pero como la fusión perezosa realmente no fusiona árboles, la operación *eliminar_mín* podría encontrar muchos árboles, elevando el costo de la operación. Cualquier *eliminar_mín* podría tardar un tiempo lineal, pero siempre es posible cargar el tiempo a las anteriores operaciones *fusionar*. En particular, una *eliminar_mín* costosa debe haber estado precedida por un gran número de *fusionar* excesivamente económicas, que han podido almacenar potencial adicional.

11.4.1. Corte de nodos en montículos a izquierda

En montículos binarios a izquierda, la operación *decrementar_llave* se implanta reduciendo el valor de un nodo y filtrándolo hacia la raíz hasta establecer el orden de montículo. En el peor caso, esto puede tomar un tiempo $O(\log n)$, que es la longitud del camino más largo hacia la raíz en un árbol equilibrado.

[†] El nombre proviene de una propiedad de esta estructura de datos, que demostramos después en la sección.

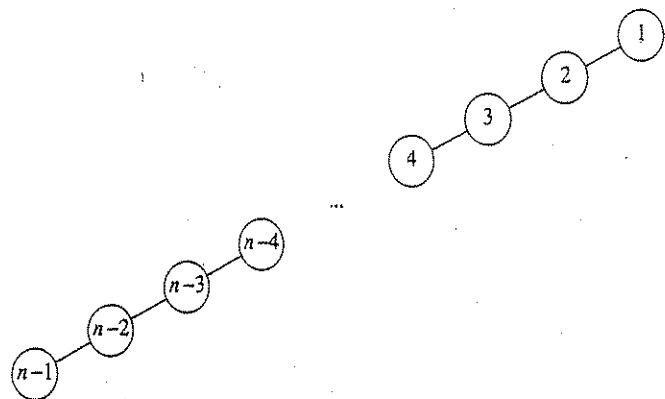


Figura 11.9 La disminución de $n - 1$ a 0 vía un filtrado tomaría un tiempo $\Theta(n)$

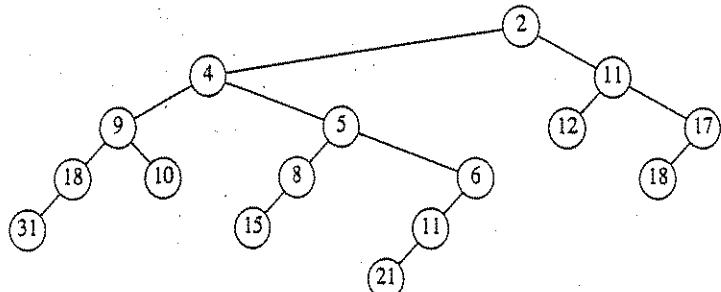


Figura 11.10 Muestra del montículo a izquierda M

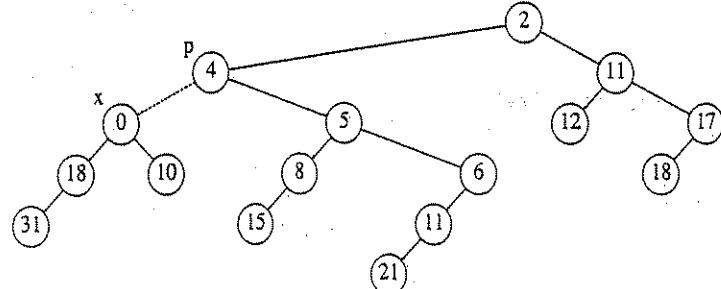


Figura 11.11 La disminución de 9 a 0 crea una violación al orden del montículo

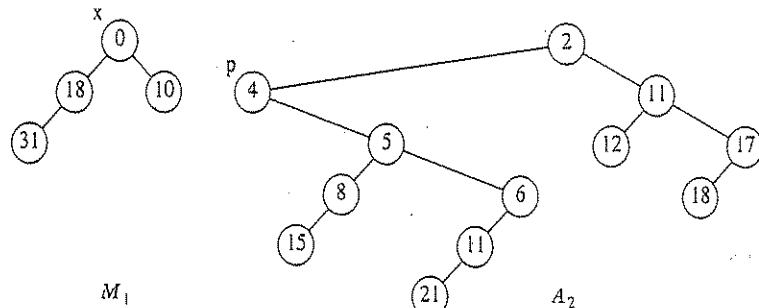


Figura 11.12 Los dos árboles después del corte

Esta estrategia no funciona si el árbol que representa la cola de prioridad no tiene profundidad $O(\log n)$. Por ejemplo, si esta estrategia se aplica a árboles a izquierda, entonces la operación *decrementar_llave* podría tardar $\Theta(n)$, como lo muestra el ejemplo de la figura 11.9.

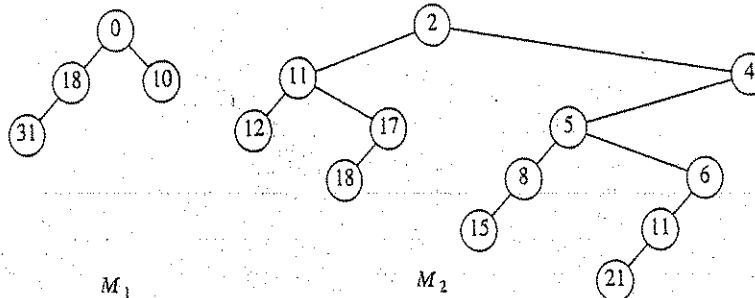
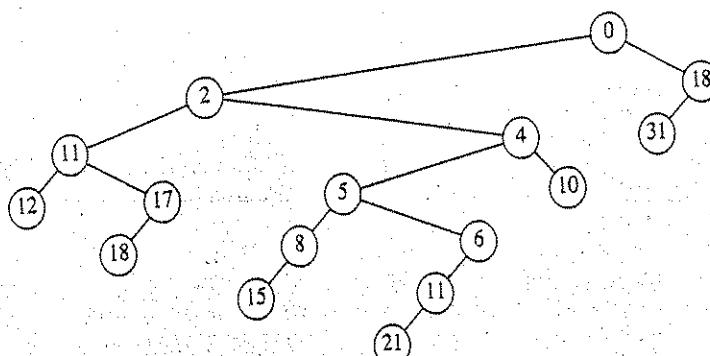
Vemos que, para montículos a izquierda, se necesita otra estrategia para la operación *decrementar_llave*. Nuestro ejemplo será el montículo a izquierda de la figura 11.10. Supongamos que queremos decrementar la llave de valor 9 hacia 0. Si hacemos el cambio, encontramos que hemos creado una violación al orden de montículo, lo cual se indica con una línea punteada en la figura 11.11.

No queremos filtrar el cero hasta la raíz porque, como hemos visto, hay casos en los que podría ser costoso. La solución es *cortar* el montículo por la línea punteada, creando así dos árboles, y después fusionar de nuevo los dos árboles en uno. Sea x el nodo al cual se le aplica la operación *decrementar_llave*, y sea p su parente. Después del corte, tenemos dos árboles, a saber, M_1 con la raíz x y A_2 , que es el árbol original con M_1 eliminado. La situación se ilustra en la figura 11.12.

Si ambos árboles fueran montículos a izquierda, podrían fusionarse en un tiempo $O(\log n)$, quedando todo resuelto. Es fácil ver que M_1 es un montículo a izquierda, pues ninguno de sus nodos ha tenido ningún cambio en sus descendientes. Así, puesto que todos sus nodos satisficieron originalmente la propiedad de árbol a izquierda, deben continuar haciéndolo.

No obstante, parece que este esquema no funcionará porque A_2 no necesariamente es a izquierda. Sin embargo, es fácil restablecer la propiedad de montículo a izquierda haciendo dos observaciones:

- Sólo aquellos nodos que se encuentren en el camino de p a la raíz de A_2 pueden violar la propiedad del montículo a izquierda; éstos se pueden corregir intercambiando los hijos.
- Puesto que la longitud máxima del camino derecho tiene a lo más $\log(n + 1)$ nodos, sólo necesitamos revisar los primeros $\log(n + 1)$ nodos, sobre el camino de p a la raíz de A_2 . La figura 11.3 muestra M_1 y A_2 después de convertir a A_2 en un montículo a izquierda.

Figura 11.13 A_2 convertido en el montículo a izquierda M_2 Figura 11.14 $\text{decrementar_llave}(M, x, 9)$ terminado después de la fusión de M_1 y M_2

Como podemos convertir A_2 en el montículo a izquierda M_2 en $O(\log n)$ pasos, y después fusionar M_1 y M_2 , tenemos un algoritmo $O(\log n)$ si realizamos la operación *decrementar_llave* en montículos a izquierda. El montículo resultante del ejemplo se muestra en la figura 11.14.

11.4.2. Fusión perezosa de colas binomiales

La segunda idea en que se basan los montículos de Fibonacci es la fusión *perezosa*. Aplicaremos esta idea a las colas binomiales y demostraremos que el tiempo amortizado para efectuar una operación *fusionar* (al igual que la inserción, que es un caso especial) es $O(1)$. El tiempo amortizado de *eliminar_mín* aún será $O(\log n)$.

La idea es como sigue: para fusionar dos colas binomiales, sólo se concatenan las dos listas de árboles binomiales para crear una cola binomial nueva. Esta cola nueva puede tener varios árboles del mismo tamaño, lo que viola la propiedad de cola binomial. Llamaremos a esto *cola binomial perezosa* a fin de mantener la consistencia.

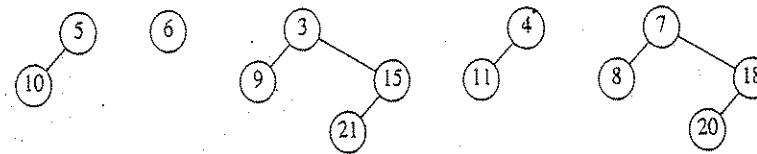


Figura 11.15 Cola binomial perezosa

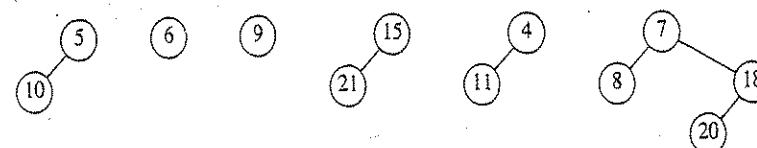


Figura 11.16 Cola binomial perezosa después de eliminar el elemento menor (3)

tencia. Ésta es una operación rápida, que siempre lleva un tiempo constante (en el peor caso). Como antes, se hace una inserción creando una cola binomial de un nodo y fusionando. La diferencia es que *fusionar* es perezosa.

La operación *eliminar_mín* es mucho más ardua, porque es donde finalmente se convierte la cola binomial perezosa en una cola binomial estándar, pero, como demostraremos, aún es un tiempo amortizado $O(\log n)$, pero no $O(\log n)$ para el peor caso, como antes. Para realizar un *eliminar_mín*, encontramos (y tarde o temprano devolvemos) el elemento mínimo. Como antes, lo eliminamos de la cola haciendo árboles nuevos de cada uno de sus hijos. Despues se fusionan todos los árboles en una cola binomial, fusionando dos árboles de igual tamaño hasta que ya no sea posible.

A manera de ejemplo, la figura 11.15 muestra una cola binomial perezosa. En una cola binomial perezosa puede haber más de un árbol del mismo tamaño. Podemos conocer el tamaño de los árboles examinando el campo *rango* de la raíz, que da el número de hijos (e implícitamente el tipo de árbol). Para efectuar *eliminar_mín* eliminamos el elemento menor, igual que antes, y se obtiene el árbol de la figura 11.16.

Ahora tenemos que fusionar todos los árboles y obtener una cola binomial estándar, la cual tiene a lo más un árbol de cada rango. A fin de hacer eficiente esto, debemos ser capaces de efectuar el *fusionar* en un tiempo proporcional al número de árboles presentes (A) ($\log n$, el que sea mayor). Para hacerlo, formamos un arreglo de listas, $L_0, L_1, \dots, L_{R_{\max}+1}$, donde R_{\max} es el rango del árbol más grande. Cada lista L_r contiene todos los árboles del rango r . Entonces se aplica el procedimiento de la figura 11.17.

Cada vez que se está en el ciclo, en las líneas [3] a [5], el número total de árboles se reduce en 1. Esto significa que esta parte del código, que tardaría un tiempo

```

(1) for r := 0 to  $\lfloor \log n \rfloor$  do
(2)   while  $|L_r| \geq 2$  do
begin
(3)   elimina dos árboles de  $L_r$ ;
(4)   fusiona los dos árboles en un árbol nuevo;
(5)   suma el árbol nuevo a  $L_{r+1}$ ;
end;
  
```

Figura 11.17 Procedimiento para restablecer una cola binomial

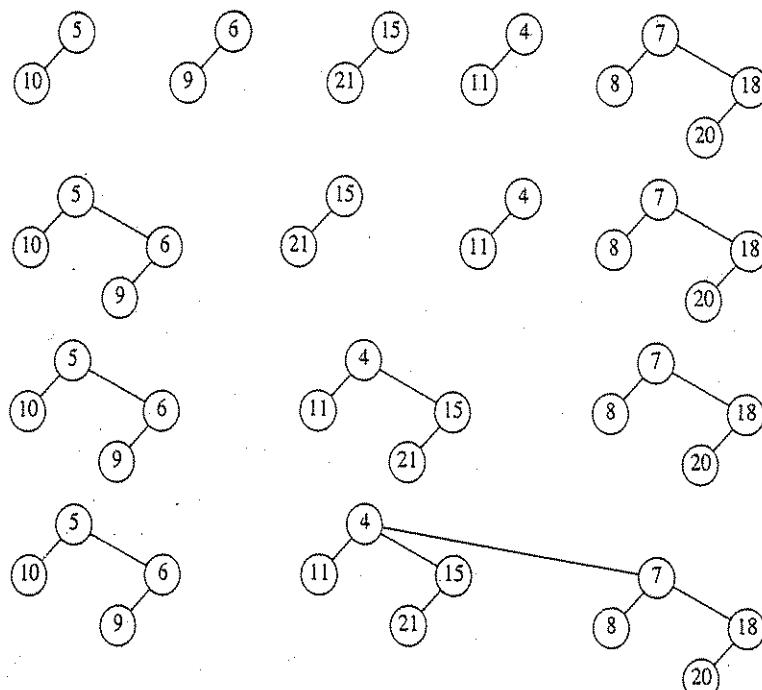


Figura 11.18 Fusión de árboles binomiales en una cola binomial

constante por ejecución, sólo puede ejecutarse $A-1$ veces, donde A es el número de árboles. Los contadores del ciclo *for* y las condiciones al final del ciclo *while* tardan un tiempo $O(\log n)$, así que el tiempo de ejecución es $O(A + \log n)$, tal como se

requería. La figura 11.18 muestra la ejecución de este algoritmo en la colección anterior de árboles binomiales.

Análisis amortizado de las colas binomiales perezosas

Para llevar a cabo el análisis amortizado de las colas binomiales perezosas, usaremos la misma función potencial que se empleó para las colas binomiales estándar. Así, el potencial de una cola binomial perezosa es el número de árboles.

TEOREMA 11.3.

Los tiempos de ejecución amortizados de fusionar e insertar son ambos $O(1)$ para las colas binomiales perezosas. El tiempo de ejecución amortizado de eliminar_mín es $O(\log n)$.

DEMOSTRACIÓN:

La función potencial es el número de árboles de la colección de colas binomiales. El potencial inicial es 0, y el potencial siempre es no negativo. Así, para una secuencia de operaciones, el tiempo total amortizado es una cota superior del tiempo total real.

Para la operación *fusionar*, el tiempo real es constante, y el número de árboles de la colección de colas binomiales permanece sin cambio, así que, por la ecuación (11.2), el tiempo amortizado es $O(1)$.

Para la operación *insertar*, el tiempo real es constante, y el número de árboles puede incrementarse en a lo más 1, así que el tiempo amortizado es $O(1)$.

La operación *eliminar_mín* es más compleja. Sea r el rango del árbol que contiene el elemento mínimo, y sea A el número de árboles. Así, el potencial al inicio de la operación *eliminar_mín* es A . Para realizar un *eliminar_mín*, los hijos del nodo más pequeño se parten en árboles separados. Esto crea $A + r$ árboles, los cuales pueden combinarse en una cola binomial estándar. El tiempo real para efectuar esto es $A + r + \log n$ si ignoramos la constante en la notación O grande, por el argumento anterior.[†] Por otro lado, una vez hecho esto, pueden quedar cuando mucho $\log n$ árboles, por lo que la función potencial puede incrementarse en $(\log n) - A$ como máximo. Sumando el tiempo real y el cambio en el potencial se obtiene una cota amortizada de $2 \log n + r$. Puesto que todos los árboles son binomiales, sabemos que $r \leq \log n$. Con esto se llega a una cota amortizada de tiempo de $O(\log n)$ para la operación *eliminar_mín*.

11.4.3. Las operaciones del montículo de Fibonacci.

Como se mencionó antes, el montículo de Fibonacci combina la operación *decrementar_llave* de montículos a izquierda con la operación *fusionar* de colas binomiales perezosas. Desafortunadamente, no es posible usar ambas operaciones sin una

[†] Podemos hacer esto porque la constante implícita en la notación O grande se puede colocar en la función potencial y aun alcanzar la cancelación de los términos, lo cual se necesita para la demostración.

ligera modificación. El problema es que si se hacen cortes arbitrarios en los árboles binomiales, el bosque resultante ya no será una colección de árboles binomiales. Por ello, ya no será cierto que el rango de todo árbol es a lo más $\lfloor \log n \rfloor$. Puesto que se demostró que la cota amortizada de *eliminar_min* en colas binomiales perezosas es $2 \log n + r$, necesitamos $r = O(\log n)$ para mantener la cota de *eliminar_min*.

A fin de asegurar que $r = O(\log n)$, aplicamos las siguientes reglas a todos los nodos que no son raíces:

- Marcar un nodo (no raíz) la primera vez que pierda un hijo (debido a un corte).
- Si un nodo marcado pierde otro hijo, entonces se corta de su padre. Este nodo se convierte ahora en la raíz de un árbol separado y deja de estar marcado. Esto se conoce como *corte en cascada* porque varios de ellos pueden ocurrir en una operación *decrementar_llave*.

La figura 11.19 muestra un árbol en un montículo de Fibonacci antes de una operación *decrementar_llave*.

Cuando el nodo con llave 39 cambia a 12, se viola el orden del montículo. Por lo tanto, el nodo es cortado de su padre, convirtiéndose en la raíz de un árbol nuevo. Como el nodo que contiene 33 está marcado, éste es su segundo hijo perdido. Como el nodo que contiene 33 está marcado, éste es su segundo hijo perdido, cortándose por ello de su padre (10). Ahora 10 ha perdido su segundo hijo, así que

Figura 11.19 Árbol en el montículo de Fibonacci antes de decrementar 39 a 12

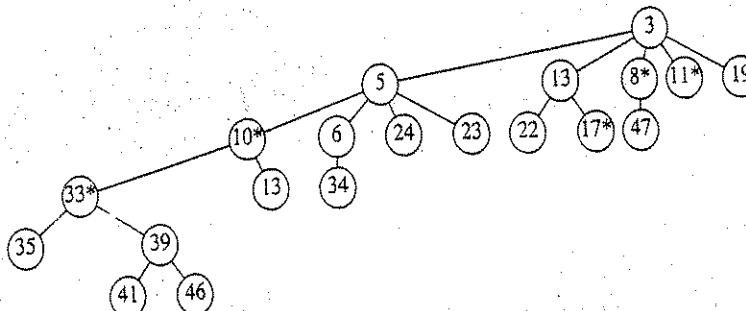
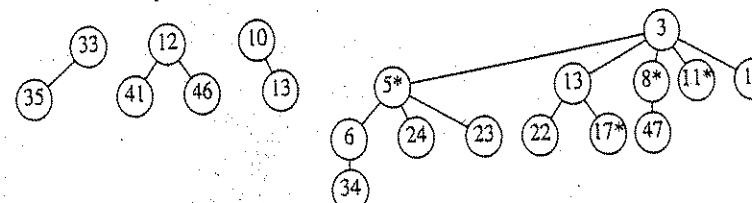


Figura 11.20 El segmento resultante del montículo de Fibonacci después de la operación *decrementar_llave*



es cortado de 5. El proceso se detiene aquí, pues 5 estaba sin marcar. Ahora se marca 5. El resultado se muestra en la figura 11.20.

Observe que 10 y 33, que fueron nodos marcados, dejan de serlo, porque ahora son nodos raíz. Ésta será una observación crucial en la demostración de la cota de tiempo.

11.4.4. Demostración de la cota de tiempo.

Recordaremos que la razón de marcar los nodos es que necesitamos acotar el rango (número de hijos) r de cualquier nodo. Ahora demostraremos que cualquier nodo con n descendientes tiene rango $O(\log n)$.

LEMMA 11.1.

Sea x un nodo de un montículo de Fibonacci. Sea h_i el i -ésimo hijo más joven de x . Entonces el rango de h_i es $i - 2$ como mínimo.

DEMOSTRACIÓN:

En el momento en que h_i fue enlazado a x , x ya tenía hijos (mayores) h_1, h_2, \dots, h_{i-1} . Así, x tenía al menos $i - 1$ hijos cuando se enlazó con h_i . Puesto que los nodos sólo se enlazan si tienen el mismo rango, se infiere que, cuando h_i fue enlazado a x , h_i tenía al menos $i - 1$ hijos. Desde ese momento, podría haber perdido a lo más un hijo, o bien habría sido cortado de x . Así, h_i tiene al menos $i - 2$ hijos.

A partir del lema 11.1, es fácil demostrar que cualquier nodo de rango r debe tener muchos descendientes.

LEMMA 11.2.

Sea F_x el número de Fibonacci definido (en la sección 1.2) por $F_0 = 1$, $F_1 = 1$ y $F_k = F_{k-1} + F_{k-2}$. Cualquier nodo de rango $r \geq 1$ tiene al menos F_{r+1} descendientes (incluyéndose a sí mismo).

DEMOSTRACIÓN:

Sea M_r el menor árbol de rango r . Claramente, $M_0 = 1$ y $M_1 = 2$. Por el lema 11.1, un árbol de rango r debe tener subárboles de rango al menos $r - 2, r - 3, \dots, 1$ y 0, más otro subárbol, el cual tiene al menos un nodo. Junto con la raíz de M_r misma, esto da un valor mínimo para M_{r+1} de $M_r = 2 + \sum_{i=0}^{r-1} M_i$. Es fácil demostrar que $M_r = F_{r+1}$ (ejercicio 1.9a).

Puesto que es bien sabido que los números de Fibonacci crecen exponencialmente, se infiere inmediatamente que cualquier nodo con s descendientes tiene rango de $O(\log s)$ como máximo. Así, tenemos

LEMMA 11.3.

El rango de cualquier nodo en un montículo de Fibonacci es $O(\log n)$.

DEMOSTRACIÓN:

Es inmediato a partir del análisis anterior.

Si todo lo que nos interesa fuera las cotas de tiempo para las operaciones *fusionar*, *insertar* y *eliminar_mín*, podríamos parar aquí y probar las cotas deseadas de tiempo amortizado. Por supuesto, el punto principal de los montículos de Fibonacci es obtener también una cota de tiempo $O(1)$ para *decrementar_llave*.

El tiempo real requerido para una operación *decrementar_llave* es 1 más el número de cortes en cascada efectuados durante la operación. Puesto que el número de cortes en cascada podría ser mucho más que $O(1)$, debemos pagar por esto con una pérdida en el potencial. Si revisamos en la figura 11.20, vemos que el número de árboles de hecho se incrementa con cada corte en cascada, así que tendremos que mejorar la función potencial para incluir algo que disminuya durante los cortes. Observe que no podemos simplemente eliminar el número de árboles de la función potencial, porque entonces no nos será posible demostrar la cota de tiempo para la operación *fusionar*. Observando de nuevo la figura 11.20 vemos que un corte en cascada ocasiona una disminución del número de nodos marcados, porque cada nodo que es víctima de un corte en cascada se convierte en una raíz no marcada. Puesto que cada corte en cascada cuesta 1 unidad de tiempo real e incrementa en 1 el potencial del árbol, contaremos cada nodo marcado como 2 unidades de potencial. De esta forma, tenemos una oportunidad de cancelar el número de cortes en cascada.

TEOREMA 11.4.

Las cotas de tiempo amortizado para los montículos de Fibonacci son $O(1)$ para insertar, fusionar y decrementar_llave y $O(\log n)$ para eliminar_mín.

DEMOSTRACIÓN:

El potencial es el número de árboles de la colección de montículos de Fibonacci más el doble del número de nodos marcados. Como es usual, el potencial inicial es 0 y siempre es no negativo. Así, para una secuencia de operaciones, el tiempo total amortizado es una cota superior del tiempo real total.

Para la operación *fusionar* el tiempo real es constante, y el número de árboles y nodos marcados permanece sin cambio; así, por la ecuación (11.2), el tiempo amortizado es $O(1)$.

Para la operación *insertar*, el tiempo real es constante, el número de árboles se incrementa en 1 y el número de nodos marcados permanece sin cambio. Así, el potencial se incrementa en 1 como máximo, y el tiempo amortizado es $O(1)$.

Para la operación *eliminar_mín*, sea r el rango del árbol que contiene el elemento mínimo, y sea A el número de árboles antes de la operación. Para realizar *eliminar_mín*, una vez más partimos los hijos de un árbol, creando r árboles adicionales nuevos. Observe que, aunque esto puede eliminar nodos marcados (convirtiéndolos en raíces no marcadas), no puede crear nodos adicionales marcados. Estos r árboles nuevos, junto con los otros A árboles, ahora deben fusionarse, a un costo de $A + r + \log n = A + O(\log n)$, por el lema 11.3. Puesto que puede haber cuando mucho $O(\log n)$ árboles, y el número de nodos marcados no puede aumentar, el cambio de potencial es $O(\log n) - A$ como máximo. Sumando el tiempo real y el cambio del potencial se obtiene la cota amortizada $O(\log n)$ para *eliminar_mín*.

Por último, para la operación *decrementar_llave*, sea C el número de cortes en cascada. El costo real de *decrementar_llave* es $C + 1$, que es el número total de cortes realizados. El primer corte (no en cascada) crea un árbol nuevo e incrementa el potencial en 1. Cada corte en cascada crea un árbol nuevo, pero convierte un nodo marcado en un nodo (raíz) no marcado, resultando una pérdida neta de una unidad por corte en cascada. El último corte también puede convertir un nodo no marcado (en la figura 11.20 este nodo es el 5) en uno marcado, incrementando así el potencial en 2. Por tanto, el cambio total en el potencial es $3 - C$. La suma del tiempo real y el cambio de potencial da un total de cuatro, que es $O(1)$.

11.5 Árboles desplegados

Como ejemplo final, analizaremos el tiempo de ejecución de los árboles desplegados. Recordemos, del capítulo 4, que después de realizar un acceso a algún elemento x , un paso de despliegue mueve x a la raíz por medio de una serie de tres operaciones: zig, zig-zag y zig-zig. Estas tres rotaciones se muestran en la figura 11.21. Adoptamos el convención de que si se están realizando tres rotaciones en el nodo x , entonces antes de la rotación p es su padre y a es su abuelo (si x no es el hijo de la raíz).

Recordaremos que el tiempo requerido para cualquier operación de árbol sobre el nodo x es proporcional al número de nodos en el camino de la raíz a x . Si contamos cada operación zig como una rotación y cada zig-zag o zig-zig como dos rotaciones, entonces el costo de cualquier acceso es igual a 1 más el número de rotaciones.

A fin de demostrar que hay una cota amortizada $O(\log n)$ para el paso de despliegue, necesitamos una función potencial que pueda incrementarse en $O(\log n)$, como máximo, en todo el paso de despliegue, pero que también cancele el número de rotaciones efectuadas durante el paso. No es fácil en lo absoluto encontrar una función potencial que satisfaga estos criterios. Un primer intento sencillo para la función potencial podría ser sumar las profundidades de todos los nodos del árbol. Pero esto no funciona porque el potencial se puede incrementar en $\Theta(n)$ durante un acceso. Un ejemplo canónico de esto ocurre cuando se insertan los elementos en un orden secuencial.

Una función potencial Φ , que sí funciona, se define como

$$\Phi(A) = \sum_{i \in A} \log S(i).$$

$S(i)$ representa el número de descendientes de i (incluyendo a i mismo). La función potencial es la suma, sobre todos los nodos i del árbol A , del logaritmo de $S(i)$.

Para simplificar la notación, definiremos

$$R(i) = \log S(i).$$

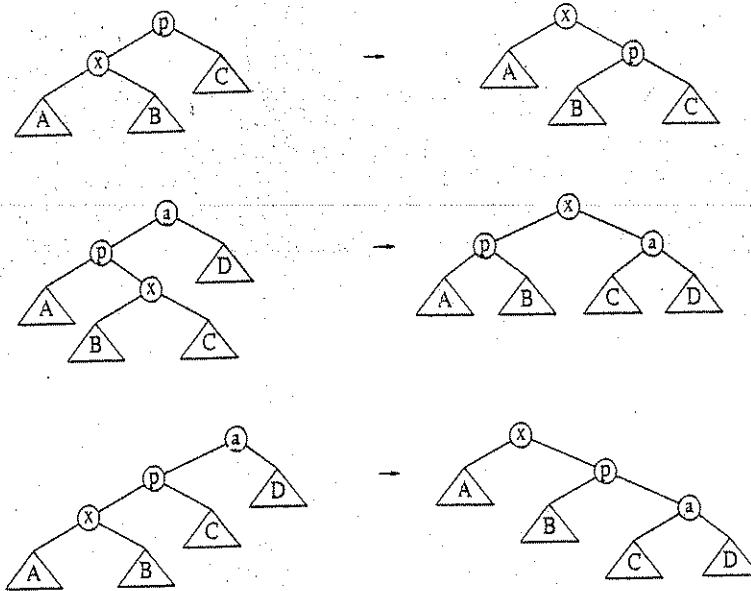


Figura 11.21 Operaciones zig, zig-zag y zig-zig; cada una tiene un caso simétrico (no mostrado)

Esto hace

$$\Phi(A) = \sum_{i \in A} R(i)$$

$R(i)$ representa el *rango* del nodo i . La terminología es semejante a la que usamos en el análisis del algoritmo de conjuntos ajenos, colas binomiales y montículos de Fibonacci. En todas estas estructuras de datos, el significado de *rango* es algo diferente, pero en general es función (de magnitud) del logaritmo del tamaño del árbol. Para un árbol A con n nodos, el rango de la raíz es simplemente $R(A) = \log n$. Utilizar la suma de los rangos como una función potencial es similar a usar la suma de las alturas como una función potencial. La diferencia importante es que mientras una rotación puede cambiar las alturas de muchos nodos del árbol, sólo pueden cambiar los rangos de x, p y a .

Antes de demostrar el teorema principal, necesitaremos el siguiente lema.

LEMA 11.4.

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 media aritmética-geométrica,

$$\sqrt{ab} \leq (a + b)/2.$$

Así

$$\sqrt{ab} \leq c/2$$

Elevando al cuadrado ambos lados

$$ab \leq c^2/4$$

Tomando los logaritmos de ambos lados se demuestra el lema.

Si tenemos cuidado con los preliminares, estamos listos para demostrar el teorema principal.

TEOREMA 11.3

El tiempo amortizado para desplegar un árbol con raíz A en el nodo x es $3(R(A) - R(x)) + 1 = O(\log n)$, como máximo.

DEMOSTRACIÓN:

La función potencial es la suma de los rangos de los nodos de A .

Si x es la raíz de A , entonces no hay rotaciones, así que no hay cambio de potencial. El tiempo real es 1 para tener acceso al nodo, por lo que el tiempo amortizado es 1 y el teorema se cumple. Así, podemos suponer que al menos hay una rotación.

Para cualquier paso de despliegue, sean $R_f(x)$ y $T_f(x)$ el rango y el tamaño de x antes del paso, y sean $R_i(x)$ y $T_i(x)$ el rango y el tamaño de x inmediatamente después del paso de despliegue. Demostraremos que el tiempo amortizado requerido para un zig es a lo más $3(R_f(x) - R_i(x)) + 1$ y que el tiempo amortizado para cualquiera de zig-zag y zig-zig es a lo más $3(R_f(x) - R_i(x))$ como máximo. Demostraremos que cuando se suma sobre todos los pasos, la suma proyecta la cota de tiempo deseada.

Paso zig: Para el paso zig, el tiempo real es 1 (para la rotación sencilla), y el cambio de potencial es $R_f(x) + R_f(p) - R_i(x) - R_i(p)$. Observe que el cambio de potencial es fácil de calcular, porque sólo los árboles de x y p cambian de tamaño. Así

$$TA_{\text{zig}} = 1 + R_f(x) + R_f(p) - R_i(x) - R_i(p)$$

De la figura 11.21 se ve que $T_f(p) \geq T_i(p)$; así, se infiere que $R_i(p) \geq R_f(p)$. Por lo tanto,

$$TA_{\text{zig}} \leq 1 + R_f(x) - R_i(x).$$

Puesto que $T_f(x) \geq T_i(x)$, se infiere que $R_f(x) - R_i(x) \geq 0$, por lo que podemos incrementar el lado izquierdo, obteniendo

$$TA_{\text{zig}} \leq 1 + 3(R_f(x) - R_i(x)).$$

Paso zig-zag: para el caso del zig-zag, el costo real es 2, y el cambio de potencial es $R_f(x) + R_f(p) + R_f(a) - R_i(x) - R_i(p) - R_i(a)$. Esto da una cota de tiempo amortizado de

$$TA_{\text{zig-zag}} = 2 + R_f(x) + R_f(p) + R_f(a) - R_i(x) - R_i(p) - R_i(a).$$

A partir de la figura 11.21, vemos que $T_f(x) = T_i(a)$, por lo que sus rangos deben ser iguales. Así, se obtiene

$$TA_{\text{zig-zag}} = 2 + R_f(p) + R_f(a) - R_i(x) - R_i(p).$$

También se ve que $T_i(p) \geq T_i(x)$. En consecuencia, $R_i(x) \leq R_i(p)$. Haciendo esta sustitución se obtiene

$$TA_{\text{zig-zag}} \leq 2 + R_f(p) + R_f(a) - 2R_i(x).$$

De la figura 11.21 vemos que $T_f(p) + T_f(a) \leq T_f(x)$. Si aplicamos el lema 11.4, obtenemos

$$\log T_f(p) + \log T_f(a) \leq 2 \log T_f(x) - 2.$$

Por la definición de *rango*, esto da

$$R_f(p) + R_f(a) \leq 2R_f(x) - 2.$$

Sustituyendo, se obtiene

$$TA_{\text{zig-zag}} \leq 2R_f(x) - 2R_i(x).$$

$$\leq 2(R_f(x) - R_i(x))$$

Puesto que $R_f(x) \geq R_i(x)$, obtenemos

$$TA_{\text{zig-zag}} \leq 3(R_f(x) - R_i(x)).$$

Paso zig-zig: El tercer caso es el zig-zig. La demostración de este caso es muy similar al caso zig-zag. Las desigualdades importantes son $R_f(x) = R_i(a)$, $R_f(x) \geq R_f(p)$, $R_i(x) \leq R_i(p)$ y $T_i(x) + T_f(a) \leq T_f(x)$. Los detalles se dejan para el ejercicio 11.8.

El costo amortizado de un despliegue completo es la suma de los costos amortizados de cada paso de despliegue. La figura 11.22 muestra los pasos efectuados en un despliegue en el nodo 2. Sea $R_1(2)$, $R_2(2)$, $R_3(2)$ y $R_4(2)$ el rango del nodo 2 en cada uno de los cuatro árboles. El costo del primer paso, que es un zig-zag, es a lo más $3(R_2(2) - R_1(2))$. El costo del segundo paso, que es un zig-zig, es $3(R_3(2) - R_2(2))$. El último paso es un zig y tiene un costo no mayor que $3(R_4(2) - R_3(2)) + 1$. El costo total se proyecta hasta $3(R_4(2) - R_1(2)) + 1$.

En general, sumando los costos amortizados de todas las rotaciones, de las cuales a lo más una puede ser un zig, vemos que el costo amortizado total para desplegar en el nodo x cuando mucho es $3(R_i(x) - R_i(x)) + 1$, donde $R_i(x)$ es el rango de x antes del primer paso de despliegue y $R_f(x)$ es el rango de x después del último paso de despliegue. Como el último paso de despliegue deja x en la raíz, obtenemos una cota amortizada de $3(R_f(A) - R_i(x)) + 1$, que es $O(\log n)$.

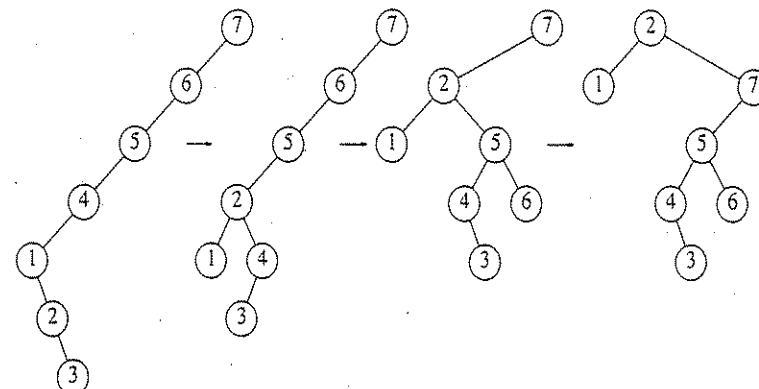


Figura 11.22 Pasos para desplegar en el nodo 2

Debido a que en un árbol desplegado cada operación requiere un despliegue, el costo amortizado de cualquier operación está dentro de un factor constante del costo amortizado de un despliegado. Con esto, todas las operaciones sobre árboles desplegados toman un tiempo amortizado de $O(\log n)$. Usando una función potencial más general es posible demostrar que los árboles desplegados tienen varias propiedades sobresalientes. Esto se estudia con mayor detalle en los ejercicios.

Resumen

En este capítulo hemos visto cómo usar un análisis amortizado para distribuir cargas entre operaciones. Para llevar a cabo el análisis, inventamos una función potencial ficticia. La función potencial mide el estado del sistema. Una estructura de datos con un potencial alto es volátil al haber sido construida sobre operaciones relativamente baratas. El costo excesivo de una operación se compensa con los ahorros en las operaciones anteriores. Se puede considerar el potencial como si fuera *potencial para desastre*, en que toda operación costosa sólo puede ocurrir cuando la estructura de datos tiene un potencial alto y ha usado considerablemente menos tiempo del asignado.

Un potencial bajo en una estructura de datos significa que el costo de cada operación ha sido casi igual a la cantidad asignada para ella. Un potencial negativo significa deuda: se consume más tiempo del asignado. Por ello el tiempo asignado (o amortizado) no es una cota significativa.

Como se expresó en la ecuación 11.2, el tiempo amortizado de una operación es igual a la suma del tiempo real y el cambio de potencial. Tomando para una secuencia completa de operaciones, el tiempo amortizado de la secuencia es igual al tiempo total de la secuencia más el cambio neto en el potencial. En tanto este

cambio sea positivo, la cota amortizada da una *cota superior* para el tiempo real consumido y es significativa.

Las claves para elegir una función potencial son garantizar que el potencial mínimo ocurra al inicio del algoritmo y que el potencial se incremente en operaciones económicas y que disminuya en operaciones costosas. Es importante que el tiempo excesivo o ahorrado se mida como un cambio opuesto en potencial. Desafortunadamente, en ocasiones es más fácil decir esto que hacerlo.

Ejercicios

11.1 ¿En qué condiciones m inserciones consecutivas en una cola binomial toman menos de $2m$ unidades de tiempo?

11.2 Suponga una cola binomial de $n = 2^{k-1}$ elementos. Alternativamente, efectúe m pares *insertar* y *eliminar_mín*. Desde luego, cada operación toma un tiempo $O(\log n)$. ¿Por qué esto no contradice la cota amortizada de tiempo $O(1)$ para la inserción?

*11.3 Demuestre que la cota amortizada de $O(\log n)$ de las operaciones sobre montículos oblicuos descritas en el texto no puede convertirse en una cota del peor caso; para ello, proporcione una secuencia de operaciones que hagan que *fusionar* requiera un tiempo $\Theta(n)$.

*11.4 Demuestre cómo fusionar dos montículos oblicuos con una pasada de arriba a abajo y reducir el costo de *fusionar* a un tiempo amortizado $O(1)$.

11.5 Extienda los montículos oblicuos para permitir la operación *decrementar_llave* en un tiempo amortizado $O(\log n)$.

11.6 Implante montículos de Fibonacci y compare su rendimiento con los montículos binarios cuando se usan en el algoritmo de Dijkstra.

11.7 Una implantación estándar de los montículos de Fibonacci requiere cuatro apuntadores por nodo (padre, hijo y dos hermanos). Muestre cómo reducir el número de apuntadores, a costo de como máximo un factor constante en el tiempo de ejecución.

11.8 Demuestre que el tiempo amortizado de un despliegue zig-zig cuando mucho es $3(R_f(x) - R_i(x))$.

11.9 Cambiando la función potencial, es posible probar diferentes cotas para el despliegue. Sea la *función peso* $P(i)$ alguna función asignada a cada nodo del árbol, y sea $S(i)$ la suma de los pesos de todos los nodos del subárbol con raíz en i ; incluido i . El caso especial $P(i) = 1$ para todos los nodos corresponde a la función usada en la demostración de la cota del despliegue. Sean n el número de nodos del árbol y m el número de accesos. Demuestre los siguientes dos teoremas:

a. El tiempo de acceso total es $O(m + (m+n)\log n)$.

*b. Si q_i es el número de veces que se tiene acceso al elemento i , y $q_i > 0$ para toda i , entonces el tiempo de acceso total es

$$O\left(m + \sum_{i=1}^n q_i \log(m/q_i)\right)$$

- 11.10 a. Muestre cómo implantar la operación *fusionar* sobre árboles desplegados de modo que una secuencia de $n - 1$ operaciones *fusionar*, que comienza con n árboles de un solo elemento, tome un tiempo $O(n \log^2 n)$.
- *b. Mejorar la cota a $O(n \log n)$.

11.11 En el capítulo 5 describimos la *redispersión*: Cuando una tabla se llena a más de la mitad, se construye una tabla nueva el doble de grande, y se redispersa la tabla anterior completa. Haga un análisis formal amortizado, con función potencial, para demostrar que el costo amortizado de una inserción es aún $O(1)$.

11.12 Demuestre que si no se permiten las eliminaciones, entonces cualquier secuencia de m inserciones en un árbol 2-3 de n nodos produce $O(m + n)$ particiones de nodos.

11.13 Una *doble cola* con *orden de montículo* es una estructura de datos consistente en una lista de elementos, en los cuales son posibles las siguientes operaciones:
encolar(x, d): inserta el elemento x en el extremo frontal de la doble cola d .
desencolar(d): retira el elemento del frente de la doble cola d y lo devuelve.
inyectar(x, d): inserta el elemento x al final de la doble cola d .
eyectar(d): elimina el elemento del final de la doble cola d y lo devuelve.
buscar_mín(d): devuelve el elemento más pequeño de la doble cola d (desaciendiendo los empates).

- a. Describa cómo permitir estas operaciones en un tiempo amortizado constante por operación.
- **b. Describa cómo permitir esas operaciones en un tiempo constante para el peor caso por operación.

Referencias

Un estudio excelente del análisis amortizado se da en [9].

La mayoría de las referencias que se incluyen en seguida duplican las citas de los capítulos anteriores. Se citan nuevamente por conveniencia y complección. Las colas binomiales fueron descritas por primera vez en [10] y analizadas en [1]. Las soluciones a 11.3 y 11.4 aparecen en [8]. Los montículos de Fibonacci se describen en [3]. El ejercicio 11.9a muestra que los árboles desplegados son óptimos, en un factor constante de los mejores árboles de búsqueda estáticos. 11.9b demuestra que los árboles desplegados son óptimos, en un factor constante de los mejores árboles de búsqueda óptimos. Éstos, al igual que otros dos resultados fuertes, se prueban en el artículo original sobre árboles desplegados [6].

La operación *fusionar* de árboles desplegados es descrita por [5]. El ejercicio 11.12 está resuelto, con un uso implícito de la amortización, en [2]. El artículo

demuestra también cómo combinar árboles 2-3 eficientemente. Una solución a 11.13 se encuentra en [4].

El análisis amortizado es utilizado en [7] para diseñar un algoritmo en línea que procesa una serie de consultas en un tiempo sólo un factor constante mayor que cualquier algoritmo fuera de línea de su clase.

1. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms", *SIAM Journal on Computing*, 7 (1978), págs. 298-319.
2. M. R. Brown y R. E. Tarjan, "Design and Analysis of a Data Structure for Representing Sorted Lists", *SIAM Journal on Computing*, 9 (1980), págs. 594-614.
3. M. L. Fredman y R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms", *Journal of the ACM*, 34 (1987), págs. 596-615.
4. M. Gajewski y R. E. Tarjan, "Deques with Heap Order", *Information Processing Letters*, 22 (1986), págs. 197-200.
5. G. Port y A. Moffat "A Fast Algorithm for Melding Splay Trees", *Proceedings of First Workshop on Algorithms and Data Structures*, 1989, 450-459.
6. D. D. Sleator y R. E. Tarjan, "Self-adjusting Binary Search Trees", *Journal of the ACM*, 32 (1985), págs. 652-686.
7. D. D. Sleator y R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules", *Communications of the ACM*, 28 (1985), págs. 202-208.
8. D. D. Sleator y R. E. Tarjan, "Self-adjusting Heaps", *SIAM Journal on Computing*, 15 (1986), págs. 52-69.
9. R. E. Tarjan, "Amortized Computational Complexity", *SIAM Journal on Algebraic and Discrete Methods*, 6 (1985), págs. 306-318.
10. J. Vuillemin, "A Data Structure for Manipulating Priority Queues", *Communications of the ACM*, 21 (1978), págs. 309-314.

Vocabulario técnico bilingüe*

agrupamiento	clustering
algoritmo aleatorizado	randomized algorithm
algoritmo de aproximación	approximation algorithm
algoritmo minimax	minimax algorithm
algoritmo unión/búsqueda	merge/find algorithm
algoritmo unión/búsqueda	union/find algorithm
algoritmos ávidos	greedy algorithms
algoritmos con retroceso	backtracking algorithms
algoritmos del camino más corto	shortest-path algorithms
algoritmos del camino más largo	longest-path algorithms
algoritmos en línea	on-line algorithms
algoritmos fuera de línea	off-line algorithms
análisis amortizado	amortized analysis
análisis de algoritmos	algorithm analysis
análisis del camino crítico	critical path analysis
árbol 2-3	2-3 tree

* Ante la falta de un lenguaje estandarizado en castellano para las ciencias de la computación, se ha elaborado el presente vocabulario con la traducción que hemos dado en este libro a los principales términos de la versión original en inglés. Esta labor se verá compensada por el servicio que pueda prestar al lector. (N. del E.)

árbol(es)
 tree(s)
 árbol-B
 B-tree
 árbol-B*
 B*-tree
 aritmética modular
 modular arithmetic
 biconectividad
 biconnectivity
 búsqueda binaria
 binary search
 búsqueda en amplitud
 breadth-first search
 búsqueda en profundidad
 depth-first search
 casco convexo
 convex hull
 centinela
 sentinel
 ciclo hamiltoniano
 Hamiltonian cycle
 ciclos de costo negativo
 negative cost cycles
 ciclos de costo positivo
 positive cost cycles
 circuito de Euler
 Euler circuit
 clase de equivalencia
 equivalence class
 código prefijo
 prefix code
 códigos de Huffman
 Huffman codes
 cola
 queue
 cola binomial
 binomial queue
 cola binomial perezosa
 lazy binomial queue
 cola de doble extremo
 double-ended queue
 cola de impresión
 line printer queue
 cola de prioridad
 priority queue

colas de prioridad de doble extremo
 deap
 coloración
 coloring
 compleción NP
 NP-completeness
 componentes fuertes
 strong components
 compresión
 compression
 compresión de archivos
 file compression
 compresión de caminos
 path compression
 comprobación de primalidad
 primality test
 condición de equilibrio
 balance condition
 conectividad
 connectivity
 conjuntos ajenos (o disjuntos)
 disjoint sets
 construcción de series
 run construction
 conversión de infija a posfija
 infix to prefix conversion
 correspondencia de patrones
 pattern matching
 cota inferior
 lower bound
 criba de Eratóstenes
 sieve of Eratosthenes
 cuadrillar
 clique
 diagrama de Voronoi
 Voronoi diagram
 diseño de algoritmos
 algorithm design
 dispersión
 hashing
 dispersión extensible
 extendible hash
 división de caminos en mitades
 path halving
 doble cola
 deque

eliminación perezosa
 lazy deletion
 empaquetamiento en recipientes
 bin packing
 estructura de datos autoajustable
 self-adjusting data structure
 exploración
 probing
 exploración lineal
 linear probing
 exponentiación
 exponentiation
 expresión infija
 infix expression
 expresión posfija
 postfix expression
 factor de carga
 load factor
 flujo en red
 network flow
 forma prefija
 prefix form
 función de Ackerman
 Ackerman function
 fusión perezosa
 lazy merging
 generador congruente lineal
 linear congruential generator
 generador de números
 aleatorios
 random number generator
 generador de permutaciones
 aleatorias
 random permutation generator
 geometría computacional
 computational geometry
 grafo
 graph
 grafo acíclico
 acyclic graph
 grafo denso
 dense graph
 grafo dirigido
 directed graph
 grafo dirigido acíclico (GDA)
 directed acyclic graph (DAG)

montículo binario
 binary heap
 montículo de Fibonacci
 Fibonacci heap
 montículo oblicuo
 skew heap
 montículo-*d*
 d-heap
 multigrafo
 multigraph
 multiplicación de matrices
 matrix multiplication
 notación O grande
 big-Oh notation
 notación omega grande
 big-omega notation
 notación polaca inversa
 reverse polish notation
 notación theta
 theta notation
 números catalanes
 Catalan numbers
 números de Carmichael
 Carmichael numbers
 operación en cortocircuito
 short-circuit operation
 ordenación
 sorting
 ordenación de Shell
 Shellsort
 ordenación externa
 external sorting
 ordenación indirecta
 indirect sorting
 ordenación por cubetas
 bucket sort
 ordenación por incrementos
 decrecientes (o de Shell)
 diminishing increment sort
 ordenación por inserciones
 insertion sort
 ordenación por intercalación
 mergesort
 ordenación por montículos
 heapsort
 ordenación por restos
 radix sort

ordenación rápida
 quicksort
 ordenación topológica
 topological sort
 partición de la mediana de tres
 median of three partitioning
 pila
 stack
 planificación
 scheduling
 planificación no priorizante
 nonpreemptive scheduling
 planificador, sistema operativo
 scheduler, operating system
 poda
 pruning
 poda α-β
 α-β pruning
 posición sucesora
 successor position
 problema cuadrático
 quadratic problem
 problema de cobertura de vértices
 vertex cover problem
 problema de empaquetamiento unidimensional de círculos
 one dimensional circle packing problem
 problema de equivalencia
 equivalence problem
 problema de equivalencia dinámica
 dynamic equivalence problem
 problema de Josefo
 Josephus problem
 problema de la mayoría
 majority problem
 problema de la mochila
 knapsack problem
 problema de la parada
 halting problem
 problema de la subsecuencia común más larga
 longest common subsequence problem

problema de la subsecuencia creciente más larga
 longest increasing subsequence problem
 problema de la suma de la subsecuencia máxima
 maximum subsequence sum problem
 problema de los párrafos
 paragraphing problem
 problema de selección
 selection problem
 problemas NP
 NP problems
 problemas recursivamente indecidibles
 recursively undecidable problems
 programación dinámica
 dynamic programming
 propiedad de orden d
 montículo
 heap order property
 próximo ajuste
 next fit
 proyección
 telescoping
 punto de articulación
 articulation point
 reconstrucción de caminos de cuota
 turnpike reconstruction
 recorrido en orden de nivel
 level-order traversal
 recorrido en orden posterior
 postorder traversal
 recorrido en orden previo
 preorder traversal
 recorrido simétrico
 inorder traversal
 recursión
 recursion
 recursión por la cola
 tail recursion
 redispersión
 rehashing
 registro de activación
 activation record
 regla de Horner
 Horner's rule
 relación reflexiva
 reflexive relation
 relación simétrica
 symmetric relation
 relación transitiva
 transitive relation
 relaciones de equivalencia
 equivalence relations
 relaciones de recurrencia
 recurrence relations
 resolución de colisiones
 collision resolution
 revisor de ortografía
 spelling checker
 selección de sustitución
 replacement selection
 selección rápida
 quickselect
 simulación de eventos
 event simulation
 sistema de archivos
 file system
 tabla de dispersión
 hash table
 tabla de símbolos
 symbol table
 tabla de transposición
 transposition table
 tasa de crecimiento
 growth rate
 TDA polinomio
 polynomial ADT
 teorema menor de Fermat
 Fermat's lesser theorem
 teoría de colas
 queueing theory
 tiempo de ejecución
 running time
 tipos de datos abstractos (TDA)
 abstract data types (ADTs)
 trie
 trie

Índice de materias

A

- Abramson, B., 442
Ackerman, función de, 282 n
Adelson-Velskii, G. M., 153
agente viajero, 349
Aggarwal, A., 442
agrupamiento, 162-168
primario, 163
secundario, 168
Aho, A. V., 44, 153, 291, 358
Ahuja, R. K., 358
ajedrez (*véase* algoritmos con retroceso, juegos)
Albertson, M. O., 15
algoritmo aleatorizado, 409-420
comprobación de primalidad, 317-318
lista con saltos, 414-416
ordenación rápida, 240
principios de, 409-410
selección, 369-392-393
algoritmo de aproximación:
empaquetamiento en recipientes,
350, 372-381
problema del agente viajero, 348-349
algoritmo minimax, 425-429
algoritmo unión/búsqueda (*véase* conjuntos ajenos)
algoritmos ávidos, 362-381
árbol de extensión mínimo, 325-332
cambio de monedas, 362
camino más corto, 306-316
códigos de Huffman, 366-372
empaquetamiento aproximado en recipientes, 372-381
planificación, 362-366
algoritmos con retroceso, 418-432
juegos, 425-432
algoritmo minimax, 425-429
poda α - β , 429-432
principios de, 418-420
reconstrucción de caminos de cuota, 420-425
algoritmos del camino más corto, 300-320
algoritmos del camino más largo, 328, 350
algoritmos en línea:
algoritmo de conjuntos ajenos, 271-289
conectividad en grafos, 272
empaquetamiento en recipientes, 372-377
equilibrio de símbolos, 272
problema de la suma de la subsecuencia máxima, 32
algoritmos fuera de línea, 258, 377-381
Allen, B., 153
análisis amortizado, 126-127, 282-288, 446-469
algoritmo de conjuntos ajenos, 282-288

árboles despelgados, 465-470
 colas binomiales, 446-452
 cola binomial perezosa, 461
 función potencial, 445-452
 montículo de Fibonacci, 461-465
 montículo oblicuo, 452-454
 análisis de algoritmos, 17-39
 análisis amortizado, 282-288, 445-470
 análisis del caso promedio, 111-114, 249-251
 demostraciones de la cota inferior, 38-39, 225-227, 253-255
 procedimientos recursivos, 25-32, 236-240, 247-251, 381-384
 reglas básicas, 20-22
 tiempos de ejecución logarítmicos, 32-37
 análisis del camino crítico, 317-320
 árbol 2-3, 139-144, 145, 471
 árbol 2-d, 152
 árbol AVL, 114-126
 eliminación, 125
 inserción, 115, 117-118, 120-123
 propiedades, 114-115
 rotación doble, 119-126
 rotación sencilla, 116-118, 125
 árbol-B, 139-145, 172
 árbol binario, 100-104
 código de Huffman, 366-372
 árbol binario de búsqueda, 93, 105-145
 árbol k-d, 152
 eliminación, 109-111
 operaciones básicas, 105-111
 óptimo, 402-407
 tiempo de ejecución medio, 111-114
 vs. tabla de dispersión, 170-172
 árbol binario de búsqueda óptimo, 402-407
 árbol binomial, 208, 213, 446-448
 grafo bipartido, 353
 árbol-B*, 150
 árboles de decisión, 253-255
 árboles de expresión, 101-104
 árbol de extensión mínimo, 325-332, 436
 árbol de juego, 429
 árboles despelgados, 126-137, 465-469

análisis amortizado de, 465-469
 análisis de, 136
 autoajuste, 119-130
 eliminación, 136
 intercalación, 442
 pasos de despliegado: implantación de, 134-137
 zig, 134, 465, 467
 zig-zag, 129-130, 465, 467-468
 zig-zig, 129-130, 135, 465, 468
 árbol(es), 93-145
 2-3, 139-144, 145, 470
 2-d, 152
 árbol-B, 139-144, 172
 árbol-B*, 150
 árbol despelgado, 126-137, 465-469
 AVL, 114-126, 445
 binario, 100-104, 366-371
 binario completo, 183-184
 binario de búsqueda, 93, 105-145
 de decisión, 253-255
 de extensión mínimo, 325-332; 436
 de juego, 429
 definiciones, 93-94
 enhebrado, 151
 equilibrado por peso, 153
 implantación hijo izquierdo/hermano derecho, 95, 213
 k-d, 152
 recorridos, 95-100, 137-139
 trie, 367-372
 árbol enhebrado, 151
 árbol k-d, 152
 árbol sintáctico, 144
 aritmética modular, 6, 417-419, 420-421
 Atkinson, M. D., 221
 auf der Heide, F. Meyer, 180

B

Baeza-Yates, R. A., 153, 180, 267, 268
 Banachowski, L., 291
 Bavel, Z., 15
 Bayer, R., 153
 Bell, T., 171, 442
 Bellman, R.E., 358, 442

Bentley, J. L., 44, 153, 442
 biconectividad, 334-338
 Bitner, J. R., 153
 Bloom, G. S., 442
 Blum, M., 442
 Blum, N., 291
 Borodin, A., 442
 Boruvka, O., 315
 Boyer, R. S., 180
 Brown, D. J., 443
 Brown, M. R., 221, 443
 Brualdi, R. A., 15
 Burge, W. H., 15
 búsqueda binaria, 33-34
 búsqueda en amplitud, 304-308
 búsqueda en profundidad, 332-345
 grafo dirigido, 342-344
 grafo no dirigido, 333-342

C

Carlsson, S., 221, 268
 Carmichael, números de, 417
 Carter, J. L., 180
 casco convexo, 437
 centinela, 179, 224, 278
 Chang, L., 442
 Chen, J., 221
 Cheriton, D., 221, 358
 Christofides, N., 442
 ciclo hamiltoniano, 342, 349-350
 ciclos de costo negativo, 301
 ciclos de costo positivo, 318
 circuito de un caballo de ajedrez, 440
 clase de equivalencia, 272-274
 Clearly, J. G., 442
 codificación prefijo, 368
 cola, 83-84, 139, 181-182, 196
 búsqueda en amplitud, 304-308
 de doble extremo, 92
 de impresión, 88
 implantación con arreglos, 83-87
 operaciones básicas, 83
 ordenación topológica, 296-300
 colas binomiales, 207-216
 análisis amortizado de, 446-452
 estructura, 208-209

fusión, 209-210, 214, 447
 fusión perezosa, 455, 458-461
 implantación de, 213-215
 inserción, 210-212
 operación *eliminar_mín*, 210, 212
 otras operaciones, 214
 cola binomial perezosa, 458-461
 cola de impresión, 82
 cola de prioridad, 181-216, 315, 330, 332, 371
 algoritmo de Dijkstra, 315, 455
 algoritmo de Kruskal, 332
 algoritmo de Prim, 330
 algoritmo de Huffman, 371
 cola binomial, 208-215
deap (colas de prioridad de doble extremo), 221
 implantaciones simples, 182-183
 montículo binario, 183-194
 montículo-d, 197-198
 montículos de Fibonacci, 454-465
 montículo a izquierda, 198-205, 455-458
 montículo mín-máx, 217, 232
 montículo oblicuo, 205-207, 452-454
 operaciones básicas, 185-189
 ordenación externa, 256-262
 ordenación por montículo, 195, 231-233, 263, 267
 simulación, 196-197
 colas de prioridad de doble extremo, 221
 coloración, 331
 Comer, D., 153
 compleción NP, 346-350, 357
 componentes fuertes, 344-345
 compresión (*véase* compresión de archivos)
 compresión de archivos, 366-372
 compresión de caminos, 280-288
 comprobación de primalidad, 417-420
 condición de equilibrio, 114
 conectividad, 272, 334-338
 conjuntos ajenos (o dispersos), 39, 271-289, 330-332
 algoritmo de Kruskal, 330-332
 algoritmos de búsqueda rápida, 273

algoritmos de unión rápida, 273-289
 compresión de caminos, 280-281, 289
 división de caminos por
 mitades, 291
 heurística de unión, 281-288
 implantación del algoritmo básico,
 274-277
 análisis, 282-288
 operación de desunión, 290
 problema de equivalencia
 dinámica, 272-274
 relaciones de equivalencia, 271-272
 construcción de series, 257-258,
 260-263
 conversión infija a posfija, 77-80
 Cook, S., 358
 Cook, teorema de, 350
 Coppersmith, D., 442
 correspondencia de patrones, 178, 439
 cota inferior:
 demostración, 226, 253-255
 empaquetamiento en línea en
 recipientes, 372-374
 información teórica, 255
 ordenación, 225-227, 240-242
 Crane, C. A., 221
 criptografía, 39, 417
 cuadrillas, 350, 357
 Culberson, J., 152, 154
 Culik, K., 154

D

deap (colas de prioridad de doble extremo), 221
 Demers, A., 442
 demostraciones:
 cota inferior, 39, 225-227, 253-255
 por contradicción, 8
 por contraejemplo, 8
 por inducción, 7-8, 12-13
 Deo, N., 358
 Dietzfelbinger, M., 180
 Dijkstra, algoritmo de, 308-316, 324,
 455
 Dijkstra, E. W., 14, 358
 Dinic, E. A., 358

diseño de algoritmos:
 algoritmos aleatorizados, 409-420
 algoritmos ávidos, 182, 308, 361-381
 algoritmos con retroceso, 418-432
 algoritmos de aproximación,
 347-350, 331, 372-381
 estrategia de "divide y vencerás",
 28, 32, 236, 381-397
 programación dinámica, 397-409
 dispersión, 155-176
 abierta (encadenamiento separado),
 159-162, 175
 árbol binario de búsqueda,
 comparada con, 175-176
 cerrada (direcccionamiento abierto),
 162-170
 agrupamiento, 163-170
 análisis de, 162-165
 eliminación, 167
 doble, 168-170
 exploración cuadrática, 165-168
 exploración lineal, 162-165
 extensible, 172-174
 factor de carga, 162
 función de dispersión, 156-165
 redispersión, 170-172, 471
 resolución de colisiones, 162-170
 resolución de colisiones aleatoria, 163
 tamaño de la tabla, 155-156, 175-176,
 416
 dispersión extensible, 172-175
 rendimiento de, 174
 "divide y vencerás", 381-397
 análisis del caso general, 381-384
 enteros, multiplicación de, 393-394
 multiplicación de matrices, 394-397
 ordenación por intercalación, 233-240
 ordenación rápida, 240-252
 principios de, 381-382
 puntos más cercanos, 385-389
 selección, 389-393
 suma de la subsecuencia máxima,
 21, 26-32
 división de caminos por mitades, 291
 Doberkat, E. E., 268
 doble cola, 92, 471
 Doyle, J., 291

Dreyfus, S. E., 442
 Driscoll, J. R., 221
 Du, H. C., 180

E

Edelsbrunner, H., 442
 Edmonds, J., 358
 Eisenbath, B., 154
 eliminación perezosa
 árbol AVL, 125
 árboles binarios de búsqueda, 105-114
 listas enlazadas, 81-82
 montículo a izquierda, 218
 tabla de dispersión cerrada, 163
 empaquetamiento en recipientes, 220,
 350, 372-381
 cota inferior para algoritmos en línea,
 373
 mejor ajuste, 376
 mejor ajuste decreciente, 377
 primer ajuste, 375-376
 primer ajuste decreciente, 377
 próximo ajuste, 374-375
 Enbody, R. J., 180
 enteros, multiplicación de, 393-394
 Eppinger, J. L., 153
 Eppstein, D., 442
 Eratóstenes, criba de, 42
 Eriksson, P., 222
 estructura de datos autoajustable:
 algoritmo de conjuntos ajenos,
 280-288
 árbol desplegado, 126-137, 465-470
 heurística de movimiento a la raíz,
 127-129
 lista, 90, 471
 montículo oblicuo, 205-207, 452-454
 Euclides, algoritmo de, 34-35
 Euler, circuito de, 338-342, 346, 355
 Euler, constante de, 6
 Even, S., 358
 exploración cuadrática, 165-168
 exploración lineal, 162-165
 exponentiación, 35-37
 exponentes, fórmulas para, 3
 expresión infija, 78

expresión posfija, 75-80, 102
 evaluación de, 76-77

F

factor de carga, 162
 Fagin, R., 180
 Fermat, teorema menor de, 417
 Fibonacci, montículo de, 454-465
 algoritmo de Dijkstra, 315
 análisis amortizado de, 461
 corte en cascada, 462
 marcado de nodos, 455-458
 operaciones básicas, 461-463
 Fibonacci, números de, 463
 intercalación polifásica, 260
 mal uso de la recursión, 26, 397
 orden k -ésimo, 260
 propiedades de, 7, 13, 463
 Fischer, M. J., 291, 292, 443
 Flaholet, P., 154, 180
 Floyd, R. W., 221, 267, 268, 441, 442
 flujo en red, 320-325
 Ford, L. R., 267, 357, 358
 forma prefija, 101
 Fredman, M. L., 180, 221, 291, 358,
 442, 472
 Friedman, J. H., 153
 Fulkerson, D. R., 357, 358
 Fuller, S. H., 154
 fusión:
 colas binomiales, 209-210, 214, 447
 montículos oblicuos, 452-454
 perezosa, 431-434, 455, 458-461
 fusión perezosa, 431-434, 455, 458-461

G

Gabow, H. N., 221, 292, 359
 Gajewska, H., 472
 Galil, Z., 359, 442
 Galler, B. A., 292
 Garey, M. R., 359, 442
 generador congruente lineal, 410-414
 generador de números aleatorios,
 410-414

generador de permutaciones aleatorias, 41
geometría computacional, 380, 382-384, 413-418, 425
árboles k -d, 152
casco convexo, 437
diagrama de Voronoi, 437
puntos más cercanos, 385-389, 437
reconstrucción de caminos de cuota, 420-425
Giancarlo, R., 442
Godbole, S., 442
Goldberg, A. V., 359
Golin, M., 268
Gonnet, G. H., 154, 180, 222, 267, 268
grafo, 293-351
 acíclico, 294, 316-321, 325
 evaluación de, 297, 344
 agente viajero, 349, 437
 árbol de extensión mínimo, 325-332
 biconectividad, 334-338
 bipartido, 352
 búsqueda en amplitud, 287, 304-306, 325
 búsqueda en profundidad, 332-345
 camino más corto, 300-320, 324, 407-409
 entre todos los pares, 382-385
 grafo acíclico, 316-320, 407-409
 grafos con aristas de costo negativo, 315
 no ponderado con origen único, 300-308
 ponderado con origen único, 300-316, 427
 camino más largo, 348, 350
ciclo, 294
 costo negativo, 301
ciclo hamiltoniano, 342, 344, 348-350
circuito de Euler, 338-342
cobertura de vértices, 357
coloración, 351
componentes fuertes, 344-345
correspondencia, 353
cuadrillas, 294, 350, 357
definiciones, 293-294
denso, 295

dirigido, 342-344
disperso, 295, 298, 314
isomorfos, 151
multígrafo, 356
nodos de actividad, 317
ordenación topológica, 296-300
plano, 356
 representación, 294-296
grafos dirigido acíclico (GDA), 294
grafos no dirigidos, 332-334
grafos dirigidos, 294
Graham, R. L., 15
Gries, D., 15
Gudes, E., 154
Guibas, L. J., 154, 180

H

Haken, D., 442
Harary, F., 359
Harries, R., 180
Hasham, A., 222
Helman, P., 15
Hibbard, T. H., 154, 255-256, 254
Hirschberg, D. S., 443
Hoare, C. A. R., 267, 268
Hoey, D., 443
Hopcroft, J. E., 44, 153, 291, 292, 358, 359
Horner, regla de, 41
Horvath, E. C., 268
Hu, T. C., 442
Huang, B., 268
Huffman, códigos de, 366-372
Huffman, D. A., 442
Hutchinson, J. P., 15

I

implantación con arreglos:
 de colas, 83-87
 de listas, 47
 de pilas, 71-74
implantación con cursores, 62-66
Incerpi, J., 268
indecidibilidad, 347
indeterminismo, 347

intercalación:
 de vías múltiples, 258-259
polifásica, 260
inversión, 225

J

Jensen, K., 15
Johnson, D. B., 222, 359
Johnson, D. S., 359, 442
Johnson, S. M., 268
Jonassen, A. T., 154
Josefo, problema de, 89
juego de tres en raya o "gato", 425-427, 428

K

Kaas, R., 222
Kaehler, E. B., 154
Kahn, A. B., 359
Karatsuba, A., 442
Karlin, A. R., 180
Karlton, P. L., 154
Karp, R. M., 180, 292, 358, 359
Karazanov, A. V., 359
Kernighan, B. W., 15, 359
Knuth, D. E. S., 44, 154, 180, 221, 222, 267, 268, 292, 359, 443
Koffman, E. B., 15
Komlos, J., 180
Korsh, J., 442
Kruskal, algoritmo de, 330-332 (véase también algoritmos ávidos)
Kruskal, J. R., 359
Kuhn, H. W., 359

L

Landau, G. M., 443
Landis, E. M., 153
Langston, M., 268
LaPoutre, J. A., 292
Larmore, L. L., 443
Lawler, E. L., 359
Lee, C. C., 43
Lee, D. T., 443

Lee, K., 443
Lehmer, D., 411
Lelewler, D. A., 443
Lemke, P., 443
Lempel, A., 444
Lewis, T. G., 180
Liang, F. M., 443
Lin, S., 359
listas con saltos, 414-416
lista de adyacencia, 295-296
lista enlazada, 47-66, 213, 295, 414-416
 adyacencia, 295
 célula de cabecera, 49-54
 errores comunes, 54
 implantación con cursores, 62-66
 implantación de, 62-66
 lista con saltos, 414-416
 lista doblemente enlazada, 56, 213
 lista enlazada circularmente, 56-57
 listas múltiples, 61-62
 pila, implantación de, 68-71
listas:
 implantación con arreglos, 47 (véase también lista enlazada)
listas múltiples, 61-62
logaritmos:
 en tiempo de ejecución, 32-37
fórmulas de, 4
Lueker, G., 180

M

Mahajan, S., 443
Manacher, G. K., 268
máquina de Turing, 350
marco de pila, 81
matriz de adyacencia, 294
Maurer, W. D., 180
mcd (véase Euclides, algoritmo de)
McDiarmid, C. J. H., 222
McGreight, E. M., 153
McKenzie, B. J., 180
Melhorn, K., 154, 180, 339, 358, 359
Miller, G. L., 443
Miller, K. W., 443
modularidad, 41

Moffat, A., 472
 Molodowitch, M., 180
 Monier, L., 443
 montículo (*véase* cola de prioridad)
 montículo a izquierda, 198-205,
 455-458
 corte de nodos, 452-454
 estructura, 198
 fusión, 200-202
 implantación de, 198-199
 inserción, 204
 operación *eliminar_min*, 204, 205
 montículo binario, 183-194
 estructura, 183-184
 inserción, 185-187
 operación *eliminar_min*, 187-189, 190
 orden de montículo, 184-185
 otras operaciones, 189-194
 montículo-d, 197, 351, 455
 montículo oblicuo, 205-207, 452-454
 análisis amortizado de, 452-454
 Moore, J. S., 180
 Moore, R. W., 443
 Moret, B. M. E., 359
 Morris, J. H., 180
 Morris, R., 180
 Munro, J. I., 152, 154, 222, 442
 multigrafo, 356
 multiplicación de matrices:
 algoritmo de Strassen, 394-397
 encadenada, 400-402

N

Nievergelt, J., 154
 notación O grande, 17-20, 23
 notación omega grande, 18-20
 notación polaca inversa, 76
 notación theta, 18-20
 números catalanes, 401

O

Odlyzko, A., 154
 Ofman, Y., 442
 operación de cortocircuito, 50
 ordenación, 223-263

algoritmos, comparados, 262
 basada en comparaciones, 224
 cotas inferiores, 225-227, 253-255
 estable, 268
 externa, 256-262, 268
 ordenación externa, 256-262, 268
 ordenación de Shell, 39, 227-231,
 262, 267
 ordenación en árboles, 145
 ordenación por cubetas, 59, 255-256
 ordenación por inserción, 225
 ordenación por intercalación, 233-240
 ordenación por montículos,
 231-233, 263, 267
 ordenación rápida, 240-252, 267
 registros grandes, 252-253
 topológica, 296-300
 ordenación de Shell, 39, 227-231,
 262-263, 267
 análisis, 228-231
 incrementos de Hibbard, 231
 incrementos de Shell, 230
 implantación, 227
 tiempo de ejecución medio, 231,
 262-263
 construcción de series, 257, 260-262
 intercalación polifásica, 260
 intercalación de vías múltiples,
 258-259
 intercalación simple, 259
 selección de sustitución, 261-262
 ordenación indirecta, 253
 ordenación por cubetas, 59, 255
 ordenación por disminución de
 incrementos, 227 (*véase* también
 ordenación de Shell)
 ordenación por inserción, 224-225
 análisis, 225
 implantación, 224
 ordenación por intercalación, 235-236
 análisis, 236-240
 implantación, 236
 intercalación, 235
 ordenación por montículos, 195,
 231-234, 263, 267
 ordenación por bases, 59-61
 ordenación rápida, 240-252, 262, 266

algoritmo básico, 240
 análisis, 247-251
 corte para archivos pequeños, 245
 dificultades, 243-245
 implantación, 245-247
 partición, 242-245
 selección del pivote, 241-242
 tratamiento de duplicados, 243
 ordenación topológica, 296-300
 Orlin, J. B., 358
 Ottman, T., 154

P

Pan, V., 443
 Papadimitriou, C. H., 360
 Papernov, A. A., 268
 Park, S. K., 443
 Patashnik, O., 15
 Perlis, A. J., 154
 Peterson, W. W., 180
 pila, 66-82, 299
 equilibrio de paréntesis, 74-75
 implantación con arreglos, 71-74
 implantación con listas, 68-71
 operaciones comunes, 67
 ordenación topológica, 298-299
 y recursión, 82
 Pippenger, N., 180
 pivoteo de la mediana de tres,
 242-247, 252
 planificación, 332, 362-365
 planificación no prioritizante, 362, 365
 planificador, sistema operativo, 182
 Plauger, P. J., 15
 Poblete, P. V., 221
 poda, 420, 429-432
 poda α - β , 429-432
 Port, G., 472
 posición sucesora, 426
 potencial, 446, 449, 469
 Pratt, V. R., 180, 268, 442
 Preparata, F. P., 443
 Prim, algoritmo de, 326-330 (*véase*
 también algoritmos ávidos)
 Prim, R. C., 360

problema de cambio de moneda, 308,
 362, 439
 problema de cobertura de vértices,
 357
 problema de empaquetamiento
 unidimensional de círculos, 436
 problema de equivalencia, 272-274
 problema de equivalencia dinámica,
 272-274
 problema de la mayoría, 43
 problema de la mochila, 350, 439
 problema de la subsecuencia común
 más larga, 438
 problema de la subsecuencia
 creciente más larga, 438
 problema de la suma de la
 subsecuencia máxima, 21, 26-32,
 44
 problema de las ocho reinas, 439
 problema de los párrafos, 437-438
 problema de selección:
 algoritmos de muestreo, 391-392
 algoritmos inefficientes de, 2
 en tiempo lineal para el peor caso,
 389-392
 en un árbol binario de búsqueda, 151
 procedimiento *selección_rápida*,
 251-252
 solución con colas de prioridad,
 194-195
 problema de la parada, 347
 problema del coleccionista de tarjetas
 de béisbol, 357
 problemas NP, 347-348
 problemas recursivamente
 indecidibles, 347
 procedimiento *selección_rápida*,
 251-252, 389
 programación dinámica, 397-409
 árbol binario de búsqueda óptimo,
 402-407
 camino más corto entre todos los
 pares, 407-409
 ordenación de multiplicaciones de
 matrices, 400-402
 principios de, 397-400