

ESTRUCTURAS DE DATOS Y ALGORITMOS

ALBERTO V. RODRÍGUEZ
JUAN E. MELCHORREÑA
ANTONI E. SÁNCHEZ



Estructura de datos y algoritmos

Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
con la colaboración de Guillermo Levine Gutiérrez;
versión en español de Américo Vargas y Jorge Lozano
PUBLICACIÓN México, DF : Addison-Wesley
Iberoamericana: Sistemas Técnicos de Edición, 1988
ISBN 968-6048-19-7

1

Diseño y análisis de algoritmos

Escribir un programa de computador para resolver un problema comprende varios pasos que van desde la formulación y especificación del problema, el diseño de la solución, su implantación, prueba y documentación, hasta la evaluación de la solución. En este capítulo se presenta el punto de vista de los autores sobre estos pasos. En los capítulos siguientes se analizan los algoritmos y estructuras de datos con los que se construye la mayor parte de los programas de computador.

1.1 De los problemas a los programas

La mitad del trabajo es saber qué problema se va a resolver. Al abordar los problemas, por lo general, éstos no tienen una especificación simple y precisa de ellos. De hecho, problemas como crear una receta digna de un gastrónomo o preservar la paz mundial pueden ser imposibles de formular de forma que admitan una solución por computador; aunque se crea que el problema puede resolverse en un computador, es usual que la distancia entre varios de sus parámetros sea considerable. A menudo sólo mediante experimentación es posible encontrar valores razonables para estos parámetros.

Si es posible expresar ciertos aspectos de un problema con un modelo formal, por lo general resulta beneficioso hacerlo, pues una vez que el problema se formaliza, pueden buscarse soluciones en función de un modelo preciso y determinar si ya existe un programa que resuelva tal problema; aun cuando no sea tal el caso, será posible averiguar lo que se sabe acerca del modelo y usar sus propiedades como ayuda para elaborar una buena solución.

Se puede recurrir casi a cualquier rama de las matemáticas y de las ciencias para obtener un modelo de cierto tipo de problemas. En el caso de problemas de naturaleza esencialmente numérica, esto puede lograrse a través de conceptos matemáticos tan familiares como las ecuaciones lineales simultáneas (por ejemplo, para determinar las corrientes en un circuito eléctrico o encontrar los puntos de tensión en estructuras de vigas conectadas) o ecuaciones diferenciales (por ejemplo, para predecir el crecimiento de una población o la velocidad de una reacción química). Tratándose de problemas de procesamiento de símbolos y textos, se pueden construir modelos con cadenas de caracteres y gramáticas formales. Entre los problemas de esta categoría están la compilación (traducción a lenguaje de máquina de programas escritos en algún lenguaje de programación) y las tareas de recuperación de infor-

mación, como el reconocimiento de ciertas palabras en el catálogo de títulos de una biblioteca.

Algoritmos

Cuando ya se cuenta con un modelo matemático adecuado del problema, puede buscarse una solución en función de ese modelo. El objetivo inicial consiste en hallar una solución en forma de *algoritmo*, que es una secuencia finita de instrucciones, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo finito. Una proposición de asignación entera como $x := y + z$ es un ejemplo de instrucción que puede ejecutarse con una cantidad finita de esfuerzo. Las instrucciones de un algoritmo pueden ejecutarse cualquier número de veces, siempre que ellas mismas indiquen la repetición. No obstante, se exige que un algoritmo termine después de ejecutar un número finito de instrucciones, sin importar cuáles fueron los valores de entrada. Así, un programa es un algoritmo mientras no entre en un ciclo infinito con ninguna entrada.

Es necesario aclarar un aspecto de esta definición de algoritmo. Se ha dicho que cada instrucción de un algoritmo debe tener un «significado preciso» y debe ser ejecutable con una «cantidad finita de esfuerzo»; pero lo que está claro para una persona, puede no estarlo para otra, y a menudo es difícil demostrar de manera rigurosa que una instrucción puede realizarse en un tiempo finito. A veces, también es difícil demostrar que una secuencia de instrucciones termina con alguna entrada, aunque esté muy claro el significado de cada instrucción. Sin embargo, al argumentar en favor y en contra, por lo general se llega a un acuerdo respecto a si una secuencia de instrucciones constituye o no un algoritmo. Quien afirme tener un algoritmo debe asumir la responsabilidad de demostrarlo. En la sección 1.5 se verá cómo estimar el tiempo de ejecución de algunas construcciones usuales con lenguajes de programación de las que se puede demostrar que requieren de un tiempo finito de ejecución.

Además de usar programas en Pascal como algoritmos, se empleará un *seudolenguaje* que es una combinación de las construcciones de un lenguaje formal de programación con proposiciones informales expresadas en español. Se utilizará Pascal como lenguaje de programación, pero para realizar los algoritmos que se estudiarán, éste puede sustituirse casi por cualquier otro lenguaje conocido. El ejemplo que se da a continuación ilustra muchos de los pasos que componen la escritura de un programa de computador, de acuerdo con el esquema de los autores.

Ejemplo 1.1. Un modelo matemático puede ayudar a diseñar un semáforo para una intersección complicada de calles. Para construir la secuencia de señales, se creará un programa que acepte como entrada el conjunto de giros permisibles en una intersección (un «giro» también será continuar en línea recta por una calle) y que divida este conjunto en el menor número posible de grupos, de manera que todos los giros de un grupo sean permisibles en forma simultánea sin colisiones. Después, se asociará una señal del semáforo a cada grupo de giros. Buscando una división que tenga el menor número posible de grupos, se obtendrá un semáforo con el menor número posible de fases.

Se tomará como ejemplo la intersección de la figura 1.1, donde las calles *C* y *E*

son de una dirección, mientras las restantes son de dos. En esta intersección es posible realizar 13 giros. Algunos pares de giros, como AB (de A hacia B) y EC , pueden ocurrir en forma simultánea, mientras que otros, como AD y EB , ocasionan el cruce entre líneas de tráfico y, por tanto, no pueden realizarse al mismo tiempo. Así, las luces del semáforo deben permitir los giros en un orden tal que AD y EB nunca ocurran al mismo tiempo, en tanto que AB y EC puedan ser simultáneos.

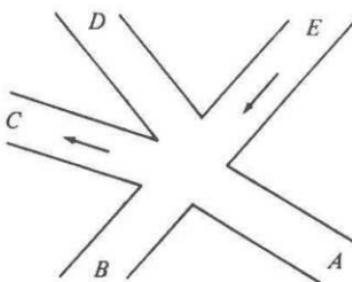


Fig. 1.1. Una intersección.

Puede obtenerse un modelo de este problema con la ayuda de una estructura matemática conocida como *grafo*. Un grafo se compone de un conjunto de puntos, llamados *vértices*, y de líneas que unen los puntos, llamadas *aristas*. Es posible construir un modelo del problema de la intersección de tráfico mediante un grafo cuyos vértices representen los giros posibles y cuyos arcos unan los pares de vértices que representan giros que no se permiten al mismo tiempo. En la figura 1.2 se muestra el grafo para la intersección de la figura 1.1. La figura 1.3 es otra representación del mismo grafo en forma de tabla, donde un 1 en la intersección de la fila i y la columna j indica que hay una arista entre los vértices i y j .

El grafo puede ayudar a solucionar el problema de diseñar el semáforo. La *coloración* de un grafo es la asignación de un color a cada vértice de éste, de tal forma que no haya dos vértices del mismo color unidos por un arco. Así pues, resulta evidente que el problema equivale a buscar una coloración del grafo de giros incompatibles que utilice el menor número posible de colores.

El problema de la coloración de grafos se ha estudiado durante varias décadas, la teoría de algoritmos ofrece mucha información. Lamentablemente, la coloración de un grafo arbitrario con el menor número posible de colores es un problema que se clasifica entre los llamados «NP-completos», para los cuales las únicas soluciones conocidas consisten, en esencia, en «intentar todas las posibilidades». En el problema de la coloración, esto significa intentar todas las asignaciones de colores a los vértices, primero con un solo color, luego con dos, con tres, y así sucesivamente, hasta encontrar una coloración válida. Con un poco de cuidado es posible acelerar este procedimiento, pero, en general, se cree que ningún algoritmo que resuelva este problema puede ser sustancialmente más eficiente que el enfoque, más obvio, recién descrito.

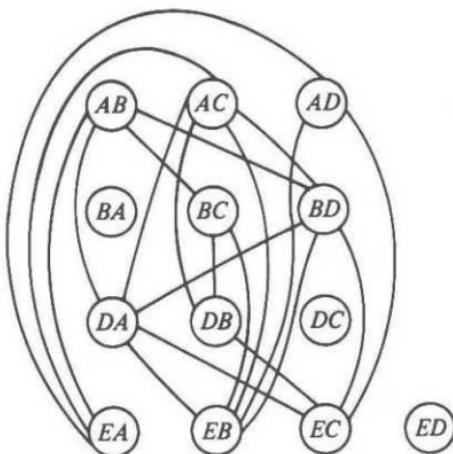


Fig. 1.2. Grafo que representa los giros incompatibles.

	<i>AB</i>	<i>AC</i>	<i>AD</i>	<i>BA</i>	<i>BC</i>	<i>BD</i>	<i>DA</i>	<i>DB</i>	<i>DC</i>	<i>EA</i>	<i>EB</i>	<i>EC</i>	<i>ED</i>
<i>AB</i>					1	1	1			1			
<i>AC</i>						1	1			1			
<i>AD</i>								1		1			
<i>BA</i>									1				
<i>BC</i>	1										1		
<i>BD</i>	1	1									1	1	
<i>DA</i>	1	1					1				1	1	
<i>DB</i>		1			1								1
<i>DC</i>													1
<i>EA</i>	1	1	1										
<i>EB</i>		1	1	1			1	1					
<i>EC</i>			1		1		1	1	1				
<i>ED</i>													

Fig. 1.3. Tabla de giros incompatibles.

Ahora, existe la posibilidad de que encontrar una solución óptima al problema en cuestión tenga un costo computacional muy elevado. Ante esto, puede elegirse entre tres enfoques. Si el grafo es pequeño se puede buscar una solución óptima a través de una búsqueda exhaustiva, intentando todas las posibilidades. Para grafos grandes, en cambio, este enfoque resulta prohibitivo por su costo, sin importar qué gra-

do de eficiencia se intente dar al programa. El segundo enfoque consiste en buscar información adicional sobre el problema. Podría suceder que el grafo tuviera propiedades especiales que hicieran innecesario probar todas las posibilidades para hallar una solución óptima. El tercer enfoque consiste en cambiar un poco el problema y buscar una solución buena aunque no necesariamente óptima; podría ser suficiente con encontrar una solución que determinara un número de colores cercano al mínimo en grafos pequeños, y que trabajara con rapidez, pues la mayor parte de las intersecciones no son tan complejas como las de la figura 1.1. Un algoritmo que produce con rapidez soluciones buenas, pero no necesariamente óptimas, se denomina *heurístico*.

El siguiente algoritmo «ávido» es un heurístico razonable para la coloración de grafos. Para empezar, se intenta colorear tantos vértices como sea posible con el primer color; después, todos los que sea posible con el segundo color, y así sucesivamente. Para dar a los vértices un nuevo color, se procede como sigue:

1. Se selecciona un vértice no coloreado y se le asigna el nuevo color.
2. Se examina la lista de vértices no coloreados. Para cada uno de ellos se determina si existe alguna arista que lo une con un vértice que ya tenga asignado el nuevo color; si tal arista no existe, se asigna el nuevo color al vértice.

Este enfoque recibe el nombre «ávido» porque le asigna color a un vértice en cuanto le es posible, sin considerar las potenciales desventajas inherentes a tal acción. Hay situaciones en las que podrían colorearse más vértices con un mismo color si fuera menos «ávido» y pasara por alto algún vértice que legalmente se podría colorear. Por ejemplo, considérese el grafo de la figura 1.4, donde, después de colorear con rojo el vértice 1, se puede asignar el mismo color a los vértices 3 y 4, siempre que no se coloree antes el vértice 2. Si se consideran los vértices en orden numérico, con el algoritmo exhaustivo se asignaría el color rojo a los vértices 1 y 2.

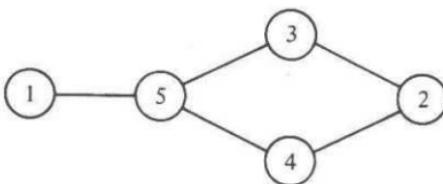


Fig. 1.4. Un grafo.

Como ejemplo del enfoque ávido aplicado a la figura 1.2, supóngase que se empieza por asignar a AB el color azul. Se pueden colorear AC , AD y BA de azul, pues ninguno de estos tres vértices tiene una arista común con AB . No es posible asignar el color azul al vértice BC , porque existe una arista entre AB y BC . Del mismo modo, no se pueden colorear de azul BD , DA ni DB , porque están unidos por una arista a uno o más vértices de los ya coloreados. Sin embargo, sí se puede asignar el azul a DC . Así, EA , EB y EC no se pueden colorear con azul, pero ED sí.

Ahora se empieza con el segundo color, por ejemplo, coloreando BC de rojo. BD puede colorearse de rojo, pero DA no, a causa de la arista entre BD y DA . De igual manera, a DB no se le puede asignar el color rojo y DC ya es azul, pero EA sí se puede colorear de rojo. Los demás vértices sin color tienen ahora una arista con un vértice rojo, de modo que ningún otro vértice puede ser rojo.

Los vértices que aún no tienen color son DA , DB , EB y EC . Si se colorea DA de verde, DB también podrá ser verde, pero EB y EC no. Estos deben colorearse con un cuarto color, por ejemplo, amarillo. La asignación de colores se resume en la figura 1.5. Al aplicar el enfoque ávido se determina que los giros «adicionales» de cierto color son compatibles con aquellos que ya tenían asignado ese mismo color y entre sí. Cuando el semáforo permita giros de un color, también permitirá los giros adicionales sin colisiones.

color	giros	giros adicionales
azul	AB, AC, AD, BA, DC, ED	—
rojo	BC, BD, EA	BA, DC, ED
verde	DA, DB	AD, BA, DC, ED
amarillo	EB, EC	BA, DC, EA, ED

Fig. 1.5. Coloración del grafo de la figura 1.2.

El enfoque ávido no siempre usa el menor número posible de colores. Ahora, es posible utilizar de nuevo la teoría de algoritmos para evaluar la bondad de la solución obtenida. En teoría de grafos, un *grafo completo-k* es un conjunto de k vértices donde cada par de ellos está unido por una arista. Puesto que en un grafo completo no hay dos vértices a los que se pueda asignar el mismo color, es obvio que se necesitan k colores para colorear un grafo completo- k .

En el grafo de la figura 1.2, el conjunto de cuatro vértices AC , DA , BD y EB es un grafo completo-4. Para este grafo no existe, pues, una coloración posible con tres colores o menos y, por tanto, la solución de la figura 1.5 es óptima en el sentido de que usa el menor número posible de colores. Para el problema original, ningún semáforo puede tener menos de cuatro fases para la intersección de la figura 1.1.

Así pues, considérese el control de un semáforo basado en la figura 1.5, donde cada fase del control corresponde a un color. En cada fase, se permiten los giros indicados en la fila de la tabla que corresponde a ese color, y se prohíben los restantes. Este modelo utiliza el menor número posible de fases. □

Seudolenguaje y refinamiento por pasos

Una vez que se tiene un modelo matemático apropiado para un problema, puede formularse un algoritmo basado en ese modelo. Las versiones iniciales del algoritmo a menudo están mezcladas en proposiciones generales que deberán refinarse después en instrucciones más pequeñas y definidas. Por ejemplo, se ha descrito el algoritmo ávido de coloración de grafos con expresiones del tipo «elegir algún vértice no coloreado». Es de esperar que esas instrucciones sean lo bastante claras para com-

prender el razonamiento. Sin embargo, para convertir en programa un algoritmo tan informal, es necesario pasar por varias etapas de formalización (*refinamiento por pasos*) hasta llegar a un programa cuyos pasos tengan un significado formalmente definido en el manual de algún lenguaje de programación.

Ejemplo 1.2. Considérese el algoritmo ávido de coloración de grafos con el fin de convertirlo en un programa en Pascal. A continuación, se supondrá que existe un grafo G con algunos vértices que pueden colorearse. El siguiente programa *ávido* determina un conjunto de vértices llamado *nue_color* a los cuales se puede asignar un nuevo color. Se llama repetidas veces al programa hasta colorear todos los vértices. Sin entrar en detalles, se puede especificar *ávido* en seudolenguaje como se muestra en la figura 1.6.

```

procedure ávido ( var  $G$ : GRAFO; var nue_color: CONJUNTO );
  { ávido asigna a nue_color un conjunto de vértices de  $G$  a los que se
    puede dar el mismo color }
  begin
    (1)   nue_color := Ø; †
    (2)   for cada vértice  $v$  no coloreado de  $G$  do
    (3)     if  $v$  no es adyacente a ningún vértice de nue_color then begin
    (4)       marca  $v$  como coloreado;
    (5)       agrega  $v$  a nue_color
    end
  end; { ávido }

```

Fig. 1.6. Primer refinamiento del algoritmo ávido.

En la figura 1.6 se observan algunas características importantes de este seudolenguaje. La primera de ellas es que se utilizan letras minúsculas negritas para las palabras clave de Pascal que tienen el mismo significado que en Pascal estándar. Las palabras en mayúscula, como GRAFO y CONJUNTO ††, son nombres de «tipos de datos abstractos». Estos se definirán mediante declaraciones tipo Pascal, y las operaciones asociadas con esos tipos de datos abstractos se definirán mediante procedimientos de Pascal al crear el programa final. En las dos secciones siguientes se analizarán con más detalle los tipos de datos abstractos.

Las construcciones de flujo de control de Pascal como if, for y while se pueden utilizar en las proposiciones en seudolenguaje, pero las expresiones condicionales, como la de la línea (3), pueden ser proposiciones informales, en vez de expresiones condicionales de Pascal. Obsérvese que la asignación de la línea (1) utiliza una expresión informal a la derecha, y que el ciclo for de la línea (2) itera sobre un conjunto.

Para su ejecución, el programa en seudolenguaje de la figura 1.6 se debe refinrar hasta convertirlo en un programa convencional en Pascal. Este proceso no se reali-

† El símbolo Ø representa el conjunto vacío.

†† Se debe distinguir entre el tipo de datos abstracto CONJUNTO y el tipo incorporado set de Pascal.

zará en forma completa para el programa de este ejemplo, pero sí se dará un ejemplo de refinamiento, transformando la proposición *if* de la línea (3) de la figura 1.6 en un código más convencional.

Para probar si un vértice *v* es adyacente a otro vértice de *nue_color*, se puede considerar cada miembro *w* de *nue_color* y examinar el grafo *G* para determinar la existencia de una arista entre *v* y *w*. Una manera organizada de realizar esta prueba consiste en utilizar una variable booleana *encontrado* que indique si se ha encontrado una arista. De este modo, las líneas (3) a (5) de la figura 1.6 pueden reemplazarse por el código de la figura 1.7.

```

procedure ávido ( var G: GRAFO; var nue_color: CONJUNTO );
begin
(1)      nue_color := Ø ;
(2)      for cada vértice v no coloreado de G do begin
(3.1)          encontrado := false;
(3.2)          for cada vértice w de nue_color do
(3.3)              if hay una arista entre v y w en G then
(3.4)                  encontrado := true;
(3.5)          if encontrado = false then begin
(4)              { v no es adyacente a ningún vértice de nue_color }
(5)              marca v como coloreado;
(5)              agrega v a nue_color
            end
        end
    end;
end;

```

Fig. 1.7. Refinamiento parcial de la figura 1.6.

Ahora, el algoritmo se ha reducido a una colección de operaciones sobre dos conjuntos de vértices. El ciclo externo, compuesto por las líneas (2) a (5), itera sobre el conjunto de vértices no coloreados de *G*. El ciclo interno, líneas (3.2) a (3.4); itera sobre los vértices que se encuentran en el conjunto *nue_color*. La línea (5) agrega vértices recién coloreados a *nue_color*.

En un lenguaje de programación como Pascal hay muchas formas de representar conjuntos. En los capítulos 4 y 5 se estudiarán varias de esas representaciones. En este ejemplo, los conjuntos de vértices se representarán simplemente por otro tipo de datos abstracto, LISTA, que se puede aplicar mediante una lista de enteros terminada con un valor especial *nulo* (para lo cual se podría aplicar el valor 0). Estos enteros podrían almacenarse, por ejemplo, en un arreglo, aunque hay muchas otras formas de representar una LISTA, como se verá en el capítulo 2.

Se puede ahora reemplazar la proposición *for* de la línea (3.2) por un ciclo en el que *w* se inicializa como primer miembro de *nue_color* y cambia al siguiente miembro en cada repetición del ciclo. También se puede realizar el mismo refinamiento con el ciclo *for* de la línea (2) de la figura 1.6. El procedimiento *ávido* revisado

se muestra en la figura 1.8. Después de esto, aún hay refinamientos por hacer, pero se hará una pausa para repasar lo estudiado. □

```

procedure ávido ( var G: GRAFO; var nue_color: LISTA );
{ ávido asigna a nue_color los vértices a los que se
  puede dar el mismo color }
var
  encontrado: boolean;
  v, w: integer;
begin
  nue_color := Ø;
  v := primer vértice no coloreado de G;
  while v <> nulo do begin
    encontrado := false;
    w := primer vértice de nue_color;
    while w <> nulo do begin
      if hay una arista entre v y w en G then
        encontrado := true;
      w := siguiente vértice de nue_color;
    end;
    if encontrado = false do begin
      marca v como coloreado;
      agrega v a nue_color;
    end;
    v := siguiente vértice no coloreado de G
  end
end; { ávido }

```

Fig. 1.8. Procedimiento *ávido* refinado.

Resumen

En la figura 1.9 se representa el proceso de programación tal como se tratará en este libro. La primera etapa es la modelación, mediante un modelo matemático apropiado, como un grafo. En esta etapa, la solución del problema es un algoritmo expresado de manera muy informal.

En la siguiente etapa, el algoritmo se escribe en seudolenguaje, es decir, en una mezcla de construcciones de Pascal y proposiciones menos formales en español. Esta etapa se alcanza sustituyendo el español informal por secuencias de proposiciones cada vez más detalladas, en un proceso denominado *refinamiento por pasos*. En algún punto de este proceso, el programa en seudolenguaje estará suficientemente detallado para que las operaciones que se deban realizar con los distintos tipos de da-

tos estén bien determinadas. Entonces, se crean los tipos de datos abstractos para cada tipo de datos (con excepción de los datos de tipo elemental como los enteros, los reales o las cadenas de caracteres) dando un nombre de procedimiento a cada operación y sustituyendo los usos de las operaciones por invocaciones a los procedimientos correspondientes.

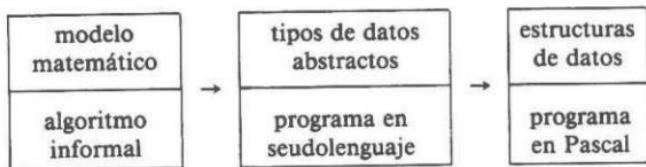


Fig. 1.9. Proceso de solución de problemas.

En la tercera etapa, se elige una aplicación para cada tipo de datos abstracto y se escribe el procedimiento que corresponde a cada operación con ese tipo. También se reemplaza por código Pascal toda proposición informal que quede en el algoritmo en seudolenguaje. El resultado será un programa ejecutable. Despues de depurarlo, será un programa operativo y se espera que por haber aplicado el desarrollo por pasos bosquejado en la figura 1.9, se requiera poca depuración.

1.2 Tipos de datos abstractos

Con seguridad, la mayor parte de los conceptos introducidos en la sección anterior le es familiar al lector desde sus primeros cursos de programación. Quizá la única noción nueva sea la de tipo de datos abstracto; por eso, antes de continuar, se analizará el papel de estos tipos durante el proceso general de diseño de programas. Para comenzar, es útil comparar este concepto con el más familiar, de procedimiento.

Los procedimientos, herramientas esenciales de programación, generalizan el concepto de operador. En vez de limitarse a los operadores incorporados de un lenguaje de programación (suma, resta, etcétera), con el uso de procedimientos, un programador es libre de definir sus propios operadores y aplicarlos a operandos que no tienen por qué ser de tipo fundamental. Un ejemplo de esta aplicación de los procedimientos es una rutina de multiplicación de matrices.

Otra ventaja de los procedimientos es que pueden utilizarse para *encapsular* partes de un algoritmo, localizando en una sección de un programa todas las proposiciones que tienen importancia en relación con cierto aspecto de éste. Un ejemplo de encapsulación es el uso de un procedimiento para leer todas las entradas y verificar su validez. La ventaja de realizar encapsulaciones es que se sabe hacia dónde ir para hacer cambios en el aspecto encapsulado del problema. Por ejemplo, si se decide verificar que todas las entradas sean no negativas, sólo se necesita cambiar unas cuantas líneas de código, y se sabe con exactitud dónde están esas líneas.

Definición de tipo de datos abstracto

Se puede pensar en un *tipo de datos abstracto* (TDA) como en un modelo matemático con una serie de operaciones definidas en ese modelo. Un ejemplo sencillo de TDA son los conjuntos de números enteros con las operaciones de unión, intersección y diferencia. Las operaciones de un TDA pueden tener como operandos no sólo los casos del TDA que se define, sino también otros tipos de operandos, como enteros o casos de otro TDA, y el resultado de una operación puede no ser un caso de ese TDA. Sin embargo, se supone que al menos un operando, o el resultado, de alguna operación pertenece al TDA en cuestión.

Las dos propiedades de los procedimientos mencionadas anteriormente, generalización y encapsulación, son igualmente aplicables a los tipos de datos abstractos. Los TDA son generalizaciones de los tipos de datos primitivos (enteros, reales, etcétera), al igual que los procedimientos son generalizaciones de operaciones primitivas (suma, resta, etcétera). Un TDA encapsula cierto tipo de datos en el sentido de que es posible localizar la definición del tipo y todas las operaciones con ese tipo se pueden localizar en una sección del programa. Así, si se desea cambiar la forma de implantar un TDA, se sabe hacia dónde dirigirse, y revisando una pequeña sección del programa se puede tener la seguridad de que no hay detalles en otras partes que puedan ocasionar errores relacionados con ese tipo de datos. Además, fuera de la sección en la que están definidas las operaciones con el TDA, éste se puede utilizar como si fuese un tipo de datos primitivo, es decir, sin preocuparse por la aplicación. Un inconveniente es que ciertas operaciones pueden implicar más de un TDA, y para que todo marche bien en tales casos, debe hacerse referencia a estas operaciones en las secciones de los dos TDA.

Para ilustrar las ideas básicas considérese el procedimiento *ávido* de la sección anterior (Fig. 1.8), implantado mediante operaciones primitivas en el tipo de datos abstracto LISTA (de enteros). Las operaciones ejecutadas en la LISTA *nue_color* fueron las siguientes:

1. vaciar la lista;
2. obtener el primer miembro de la lista y devolver *nulo* si la lista estaba vacía;
3. obtener el siguiente miembro de la lista y devolver *nulo* si no hay miembro siguiente, y, por último,
4. insertar un entero en la lista.

Existen muchas estructuras de datos que se pueden utilizar para implantar eficientemente a una lista como éstas; este tema se estudiará con profundidad en el capítulo 2. Si en la figura 1.8 se reemplazan esas operaciones por las proposiciones

1. ANULA (*nue_color*);
2. $w := \text{PRIMERO} (\text{nue_color})$;
3. $w := \text{SIGUIENTE} (\text{nue_color})$;
4. INSERTA (*v*, *nue_color*);

entonces se aprecia un aspecto importante de los tipos de datos abstractos. Se puede aplicar un tipo en la forma que se desee sin que los programas, como el de la fi-

gura 1.8, que utilizan objetos de ese tipo sufran cambios; solamente cambian los procedimientos que aplican las operaciones en el tipo.

Volviendo al tipo de datos abstracto GRAFO, se observa la necesidad de las siguientes operaciones:

1. obtener el primer vértice no coloreado;
2. determinar si hay una arista entre dos vértices;
3. marcar un vértice como coloreado, y
4. obtener el siguiente vértice no coloreado.

Es evidente que se necesitan otras operaciones fuera del procedimiento *dvido*, como insertar vértices y aristas en un grafo y marcar todos sus vértices como no coloreados. Hay muchas estructuras de datos que se pueden usar para apoyar grafos con estas operaciones; este tema se estudia en los capítulos 6 y 7.

Es importante resaltar el hecho de que no hay límite para el número de operaciones que se pueden aplicar a diversos casos de un modelo matemático dado. Cada conjunto de operaciones define un TDA distinto. Algunos ejemplos de operaciones que podrían definirse con un tipo de datos abstracto CONJUNTO son los siguientes:

1. ANULA(A). Este procedimiento hace que el valor del conjunto A sea el conjunto vacío.
2. UNION(A, B, C). Este procedimiento toma dos argumentos, A y B , cuyos valores son conjuntos, y hace que el conjunto C tome el valor de la unión de A y B .
3. TAMAÑO(A). Esta función toma un argumento A , cuyo valor es un conjunto, y devuelve un objeto de tipo entero cuyo valor es el número de elementos de A .

La *implantación* de un TDA es la traducción en proposiciones de un lenguaje de programación, de la declaración que define una variable como perteneciente a ese tipo, además de un procedimiento en ese lenguaje por cada operación del TDA. Una implantación elige una *estructura de datos* para representar el TDA; cada estructura de datos se construye a partir de los tipos de datos fundamentales del lenguaje de programación base, utilizando los dispositivos de estructuración de datos disponibles. Los arreglos y las estructuras de registro son dos importantes dispositivos de estructuración de datos de Pascal. Por ejemplo, una implantación posible de la variable S del tipo CONJUNTO sería un arreglo que contuviera los miembros de S .

Una razón importante para afirmar que dos TDA son diferentes si tienen el mismo modelo matemático, pero distintas operaciones, es que lo apropiado de una realización depende en gran medida de las operaciones que se van a realizar. Gran parte de este libro se dedica al examen de algunos modelos matemáticos básicos, como los conjuntos y los grafos, y al desarrollo de las realizaciones preferibles para varios conjuntos de operaciones.

Lo ideal sería escribir programas en lenguajes cuyos tipos de datos y operaciones primitivos estuvieran muy cerca de los modelos y operaciones de los TDA que se utilizarán aquí. En muchos sentidos, Pascal no es el lenguaje más apropiado para implantar varios TDA comunes, pero ninguno de los lenguajes en que un TDA puede declararse más directamente es tan conocido. Consultense las notas bibliográficas del final del capítulo.

1.3 Tipos de datos, estructuras de datos y tipos de datos abstractos

Aunque los términos «tipo de datos» (o simplemente «tipo»), «estructura de datos» y «tipo de datos abstracto» parecen semejantes, su significado es diferente. En un lenguaje de programación, el *tipo de datos* de una variable es el conjunto de valores que ésta puede tomar. Por ejemplo, una variable de tipo booleano puede tomar los valores verdadero o falso, pero ningún otro. Los tipos de datos básicos varían de un lenguaje a otro; en Pascal son entero (*integer*), real (*real*), booleano (*boolean*) y carácter (*char*). Las reglas para construir tipos de datos compuestos a partir de los básicos también varían de un lenguaje a otro; se verá ahora la construcción de esos tipos en Pascal.

Un tipo de datos abstracto es un modelo matemático, junto con varias operaciones definidas sobre ese modelo. Tal como se indicó, en este libro se diseñarán los algoritmos en función de los TDA, pero para implantar un algoritmo en un lenguaje de programación determinado debe hallarse alguna manera de representar los TDA en función de los tipos de datos y los operadores manejados por ese lenguaje. Para representar el modelo matemático básico de un TDA, se emplean *estructuras de datos*, que son conjuntos de variables, quizás de tipos distintos, conectadas entre sí de diversas formas.

El componente básico de una estructura de datos es la *celda*. Se puede representar una celda como una caja capaz de almacenar un valor tomado de algún tipo de datos básico o compuesto. Las estructuras de datos se crean dando nombres a agregados de celdas y (opcionalmente) interpretando los valores de algunas celdas como representantes de conexiones entre celdas (por ejemplo, los apuntadores).

El mecanismo de agregación más sencillo en Pascal y en la mayor parte de los lenguajes de programación es el *arreglo* (unidimensional), que es una sucesión de celdas de un tipo dado al cual se llamará casi siempre «*tipo_celda*». Se puede imaginar un arreglo como una transformación del conjunto índice (como los enteros 1, 2, ..., *n*) al tipo *tipo_celda*. Se puede hacer referencia a una celda de un arreglo dando el nombre del arreglo en unión de un valor tomado del conjunto índice del arreglo. En Pascal, el conjunto índice puede ser un tipo enumerado definido por el programador, como (norte, sur, este, oeste), o un subrango como 1..10. Los valores de las celdas de un arreglo pueden ser de cualquier tipo. Así, la declaración

nombre: array [*tipo_índice*] of *tipo_celda*;

indica que *nombre* es una sucesión de celdas, una por cada valor del tipo *tipo_índice*; el contenido de las celdas puede ser cualquier miembro del tipo *tipo_celda*.

A propósito de esto, Pascal tiene una riqueza considerable de tipos índice. Muchos lenguajes sólo admiten subrangos (conjuntos finitos de enteros consecutivos) como tipos índice. Por ejemplo, en FORTRAN, para indizar un arreglo con letras, debe simularse el efecto utilizando índices enteros, como el índice 1 para representar la 'A', el 2 para la 'B', y así sucesivamente.

Otro mecanismo habitual para agrupar celdas en los lenguajes de programación es la *estructura de registro*. Un *registro* es una celda constituida por un conjunto de celdas llamadas *campos*, que pueden ser de tipos distintos. A menudo, los registros

se agrupan en arreglos; el tipo definido por la agregación de los campos de un registro se convierte en el tipo_celda del arreglo. Por ejemplo, la declaración en Pascal

```
var
  lista_reg: array [1..4] of record
    datos: real;
    siguiente: integer
  end
```

declara que *lista_reg* es un arreglo de cuatro elementos cuyas celdas son registros que tienen dos campos: *datos* y *siguiente*.

Un tercer método de agrupación que se encuentra en Pascal y en otros lenguajes es el *archivo* (*file*). Al igual que un arreglo unidimensional, un archivo es una sucesión de valores de un tipo particular. La diferencia está en que un archivo no tiene tipo índice; los elementos sólo son accesibles en el orden en que aparecen en el archivo. En contraste, tanto el arreglo como el registro son estructuras de «acceso aleatorio», lo cual significa que el tiempo necesario para acceder a un componente de un arreglo o de un registro es independiente del valor del índice del arreglo o del selector de campo. La ventaja de agrupar por archivos en lugar de hacerlo por arreglos es que el número de elementos de un archivo puede ser ilimitado y variable con el tiempo.

Apuntadores y cursores

Además de las características de agrupación-celdas de un lenguaje de programación, es posible representar relaciones entre celdas mediante apuntadores y cursores. Un *apuntador* es una celda cuyo valor indica o señala a otra. Cuando se representan gráficamente estructuras de datos, el hecho de que una celda *A* sea un apuntador a la celda *B* se indica con una flecha de *A* a *B*.

En Pascal, se puede crear una variable *ap* que apunte a celdas de un tipo determinado, como tipo_celda, mediante la declaración

```
var
  ap:  $\dagger$  tipo_celda
```

En Pascal se utiliza una flecha ascendente posfixa como operador de desreferenciación. Así, la expresión *ap* \dagger denota el valor (de tipo tipo_celda) de la celda a la que apunta *ap*.

Un *cursor* es una celda de valor entero que se utiliza como apuntador a un arreglo. Como métodos de conexión, cursores y apuntadores son en esencia lo mismo, con la diferencia de que un cursor se puede utilizar en lenguajes que, como FORTRAN, carecen de los tipos apuntadores explícitos que tiene Pascal. Al tratar una celda de tipo entero como un valor índice para un arreglo, se hace que esa celda apunte a una celda del arreglo. Desafortunadamente, esta técnica sólo sirve para apuntar a celdas contenidas en un arreglo; no hay manera razonable de interpretar un entero como un «apuntador» a una celda que no sea parte de un arreglo.

En este texto, se dibujará una flecha desde una celda cursor hacia la celda a la

que «apunta». En ocasiones, se mostrará también el entero de la celda cursor, para recordar que no se trata de un verdadero apuntador. El lector debe observar que el mecanismo apuntador de Pascal es tal que a las celdas de los arreglos sólo se puede «apuntar» con cursores, y no con verdaderos apuntadores. Ciertos lenguajes, como PL/I o C, permiten que los componentes de un arreglo sean apuntados tanto por cursores como por apuntadores, mientras que en otros, como FORTRAN o ALGOL, no existe el tipo apuntador y sólo es posible usar cursores.

Ejemplo 1.3. En la figura 1.10 se observa una estructura de datos con dos partes, constituida por una cadena de celdas que contienen cursores al arreglo *lista_reg* definido antes. El propósito del campo *siguiente* de *lista_reg* es apuntar a otro registro del arreglo. Por ejemplo, *lista_reg* [4]. *siguiente* es 1, por tanto, el registro 4 va seguido del registro 1. Si se supone que el registro 4 es el primero, el campo *siguiente* de *lista_reg* ordena los registros en la secuencia 4, 1, 3, 2. Obsérvese que el campo *siguiente* del registro 2 es 0, lo cual indica que no existe un registro que lo siga. Una convención útil que se adoptará en este libro consiste en usar 0 como un «apuntador NIL» cuando se trabaje con cursores. Esto sólo da buenos resultados si también se adopta la convención de que los arreglos a los que «apuntan» cursores han de iniciarse a partir de 1, no de 0.

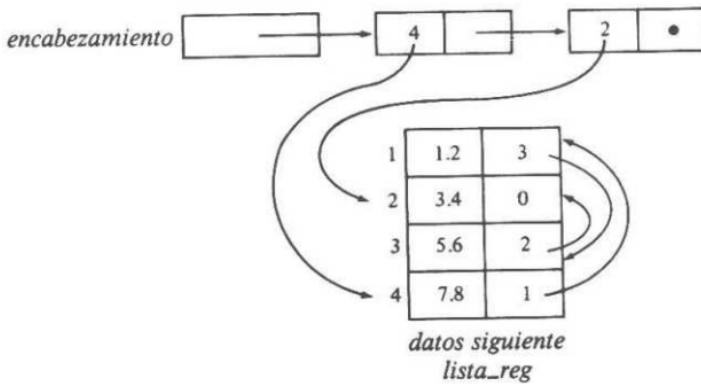


Fig. 1.10. Ejemplo de una estructura de datos.

Las celdas de la cadena de registros de la figura 1.10 son del tipo

```
type
  tipo_reg = record
    cursor: integer;
    ap: ^ tipo_reg
  end
```

A la cadena apunta una variable llamada *encabezamiento*, que es de tipo \uparrow tipo_reg; *encabezamiento* apunta a un registro anónimo de tipo tipo_reg \dagger . Ese registro tiene valor 4 en el campo *cursor*; este 4 se considera como un índice del arreglo *lista_reg*, y en el campo *ap* tiene un verdadero apuntador a otro registro anónimo. Este último tiene un índice en su campo *cursor*, que indica la posición 2 en *lista_reg*, y un apuntador *nil* en su campo *ap*. \square

1.4 Tiempo de ejecución de un programa

Cuando se resuelve un problema, con frecuencia hay necesidad de elegir entre varios algoritmos. ¿Cómo se debe elegir? Hay dos objetivos que suelen contradecirse:

1. Que el algoritmo sea fácil de entender, codificar y depurar.
2. Que el algoritmo use eficientemente los recursos del computador y, en especial, que se ejecute con la mayor rapidez posible.

Cuando se escribe un programa que se va a usar una o pocas veces, el primer objetivo es el más importante. En tal caso, es muy probable que el costo del tiempo de programación exceda en mucho al costo de ejecución del programa, de modo que el costo a optimizar es el de escritura del programa. En cambio, cuando se presenta un problema cuya solución se va a utilizar muchas veces, el costo de ejecución del programa puede superar en mucho al de escritura, en especial si en la mayor parte de las ejecuciones se dan entradas de gran tamaño. Entonces, es más ventajoso, desde el punto de vista económico, realizar un algoritmo complejo siempre que el tiempo de ejecución del programa resultante sea significativamente menor que el de un programa más evidente. Y aun en situaciones como esa, quizás sea conveniente implantar primero un algoritmo simple, con el objeto de determinar el beneficio real que se obtendría escribiendo un programa más complicado. En la construcción de un sistema complejo, a menudo es deseable implantar un prototipo sencillo en el cual se puedan efectuar simulaciones y mediciones antes de dedicarse al diseño definitivo. De esto se concluye que un programador no sólo debe estar al tanto de las formas de lograr que un programa se ejecute con rapidez, sino que también debe saber cuándo aplicar esas técnicas y cuándo ignorarlas.

Medición del tiempo de ejecución de un programa

El tiempo de ejecución de un programa depende de factores como:

1. los datos de entrada al programa,
2. la calidad del código generado por el compilador utilizado para crear el programa objeto,

\dagger El registro no tiene nombre conocido porque se creó con una llamada *new* (*encabezamiento*) cuyo efecto fue que *encabezamiento* apuntara a ese registro recién creado. No obstante, en el interior de la máquina existe una dirección de memoria que puede emplearse para localizar la celda.

3. la naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa, y
4. la complejidad de tiempo del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de la entrada, indica que el tiempo de ejecución de un programa debe definirse como una función de la entrada. Con frecuencia, el tiempo de ejecución no depende de la entrada exacta, sino sólo de su «tamaño». Un buen ejemplo de esto es el proceso conocido como *clasificación (sorting)*, que se analizará en el capítulo 8. En un problema de clasificación, se da como entrada una lista de elementos para ordenar a fin de producir como salida otra lista con los mismos elementos, pero clasificados de menor a mayor o viceversa. Por ejemplo, dada la lista 2, 1, 3, 1, 5, 8 como entrada, se desea producir la lista 1, 1, 2, 3, 5, 8 como salida. Se dice entonces que los elementos de la segunda lista están en *orden de menor a mayor*. La medida natural del tamaño de la entrada a un programa de clasificación es el número de elementos a ordenar o, en otras palabras, la longitud de la lista de entrada. En general, la longitud de la entrada es una medida apropiada de tamaño, y se supondrá que tal es la medida utilizada a menos que se especifique lo contrario.

Se acostumbra, pues, a denominar $T(n)$ al tiempo de ejecución de un programa con una entrada de tamaño n . Por ejemplo, algunos programas pueden tener un tiempo de ejecución $T(n) = cn^2$, donde c es una constante. Las unidades de $T(n)$ se dejan sin especificar, pero se puede considerar a $T(n)$ como el número de instrucciones ejecutadas en un computador idealizado.

Para muchos programas, el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ella. En este caso se define $T(n)$ como el tiempo de ejecución del *peor caso*, es decir, el máximo valor del tiempo de ejecución para entradas de tamaño n . También suele considerarse $T_{\text{prom}}(n)$, el valor medio del tiempo de ejecución de todas las entradas de tamaño n . Aunque $T_{\text{prom}}(n)$ parece una medida más razonable, a menudo es engañoso suponer que todas las entradas son igualmente probables. En la práctica, casi siempre es más difícil determinar el tiempo de ejecución promedio que el del peor caso, pues el análisis se hace intratable en matemáticas, y la noción de entrada «promedio» puede carecer de un significado claro. Así pues, se utilizará el tiempo de ejecución del peor caso como medida principal de la complejidad de tiempo, aunque se mencionará la complejidad del caso promedio cuando pueda hacerse en forma significativa.

Considérense ahora las observaciones 2 y 3 anteriores: a saber, que el tiempo de ejecución depende del compilador y de la máquina utilizados. Este hecho implica que no es posible expresar $T(n)$ en unidades estándares de tiempo, como segundos. Antes bien, sólo se pueden hacer observaciones como «el tiempo de ejecución de tal algoritmo es proporcional a n^2 », sin especificar la constante de proporcionalidad, pues depende en gran medida del compilador, la máquina y otros factores.

Notación asintótica («o mayúscula» y «omega mayúscula»)



Para hacer referencia a la velocidad de crecimiento de los valores de una función se usará la notación conocida como *notación asintótica* («o mayúscula»). Por ejemplo, de-

cir que el tiempo de ejecución $T(n)$ de un programa es $O(n^2)$, que se lee « Θ mayúscula de n al cuadrado» o tan sólo « Θ de n al cuadrado», significa que existen constantes enteras positivas c y n_0 tales que para n mayor o igual que n_0 , se tiene que $T(n) \leq cn^2$.

Ejemplo 1.4. Supóngase que $T(0) = 1$, $T(1) = 4$, y en general $T(n) = (n+1)^2$. Entonces se observa que $T(n)$ es $O(n^2)$ cuando $n_0 = 1$ y $c = 4$; es decir, para $n \geq 1$, se tiene que $(n+1)^2 \leq 4n^2$, que es fácil de demostrar. Obsérvese que no se puede hacer $n_0 = 0$, pues $T(0) = 1$ no es menor que $cn^2 = 0$ para ninguna constante c . \square

A continuación, se supondrá que todas las funciones del tiempo de ejecución están definidas en los enteros no negativos, y que sus valores son siempre no negativos, pero no necesariamente enteros. Se dice que $T(n)$ es $O(f(n))$ si existen constantes positivas c y n_0 tales que $T(n) \leq cf(n)$ cuando $n \geq n_0$. Cuando el tiempo de ejecución de un programa es $O(f(n))$, se dice que tiene *velocidad de crecimiento* $f(n)$.

Ejemplo 1.5. La función $T(n) = 3n^3 + 2n^2$ es $O(n^3)$. Para comprobar esto, sean $n_0 = 0$ y $c = 5$. Entonces, el lector puede mostrar que para $n \geq 0$, $3n^3 + 2n^2 \leq 5n^3$. También se podría decir que $T(n)$ es $O(n^4)$, pero sería una aseveración más débil que decir que $T(n)$ es $O(n^3)$.

A manera de segundo ejemplo, se demostrará que la función 3^n no es $O(2^n)$. Para esto, supóngase que existen constantes n_0 y c tales que para todo $n \geq n_0$, se tiene que $3^n \leq c2^n$. Entonces, $c \geq (3/2)^n$ para cualquier valor de $n \geq n_0$. Pero $(3/2)^n$ se hace arbitrariamente grande conforme n crece y, por tanto, ninguna constante c puede ser mayor que $(3/2)^n$ para toda n . \square

Cuando se dice que $T(n)$ es $O(f(n))$, se sabe que $f(n)$ es una cota superior para la velocidad de crecimiento de $T(n)$. Para especificar una cota inferior para la velocidad de crecimiento de $T(n)$, se usa la notación $T(n)$ es $\Omega(g(n))$, que se lee « $T(n)$ es omega mayúscula de $g(n)$ » o simplemente « $T(n)$ es omega de $g(n)$ », lo cual significa que existe una constante c tal que $T(n) \geq cg(n)$ para un número infinito de valores de n .[†]

Ejemplo 1.6. Para verificar que la función $T(n) = n^3 + 2n^2$ es $\Omega(n^3)$, sea $c = 1$. Entonces, $T(n) \geq cn^3$ para $n = 0, 1, \dots$

En otro ejemplo, sea $T(n) = n$ para $n \geq 1$ impar, y sea $T(n) = n^2/100$ para $n \geq 0$ par. Para verificar que $T(n)$ es $\Omega(n^2)$, sea $c = 1/100$ y considérese el conjunto infinito de valores de n : $n = 0, 2, 4, 6, \dots$ \square

La «tiránica» de la velocidad de crecimiento

Se supondrá que es posible evaluar programas comparando sus funciones de tiempo de ejecución sin considerar las constantes de proporcionalidad. Según este supuesto, un programa con tiempo de ejecución $O(n^2)$ es mejor que uno con tiempo de ejecu-

[†] Obsérvese la asimetría existente entre las notaciones « Θ mayúscula» y «omega mayúscula». La razón de que esta asimetría sea con frecuencia útil, es que muchas veces un algoritmo es rápido con muchas entradas pero no con todas. Por ejemplo, para probar si la entrada es de longitud prima, existen algoritmos que se ejecutan muy rápido si la longitud es par, de modo que para el tiempo de ejecución no es posible obtener una buena cota inferior que sea válida para toda $n \geq n_0$.

ción $O(n^3)$ por ejemplo. Sin embargo, además de los factores constantes debidos al compilador y a la máquina, existe un factor constante debido a la naturaleza del programa mismo. Es posible, por ejemplo, que con una combinación determinada de compilador y máquina, el primer programa tarde $100n^2$ milisegundos, mientras el segundo tarda $5n^3$ milisegundos. En este caso, ¿no es preferible el segundo programa al primero?

La respuesta a esto depende del tamaño de las entradas que se espera que procesen los programas. Para entradas de tamaño $n < 20$, el programa con tiempo de ejecución $5n^3$ será más rápido que el de tiempo de ejecución $100n^2$. Así pues, si el programa se va a ejecutar principalmente con entradas pequeñas, será preferible el programa cuyo tiempo de ejecución es $O(n^3)$. No obstante, conforme n crece, la razón de los tiempos de ejecución, que es $5n^3/100n^2 = n/20$, se hace arbitrariamente grande. Así, a medida que crece el tamaño de la entrada, el programa $O(n^3)$ requiere un tiempo significativamente mayor que el programa $O(n^2)$. Pero si hay algunas entradas grandes en los problemas para cuya solución se están diseñando estos dos programas, será mejor optar por el programa cuyo tiempo de ejecución tiene la menor velocidad de crecimiento.

Otro motivo para al menos considerar programas cuyas velocidades de crecimiento sean lo más pequeñas posibles, es que básicamente es la velocidad del crecimiento quien determina el tamaño de problema que se puede resolver en un computador. Para decirlo de otro modo, conforme los computadores se hacen más veloces, también aumentan los deseos del usuario de resolver con ellos problemas más grandes; sin embargo, a menos que un programa tenga una velocidad de crecimiento baja, como $O(n)$ u $O(n\log n)$, un incremento modesto en la rapidez de un computador no influye gran cosa en el tamaño del problema más grande que es posible resolver en una cantidad fija de tiempo.

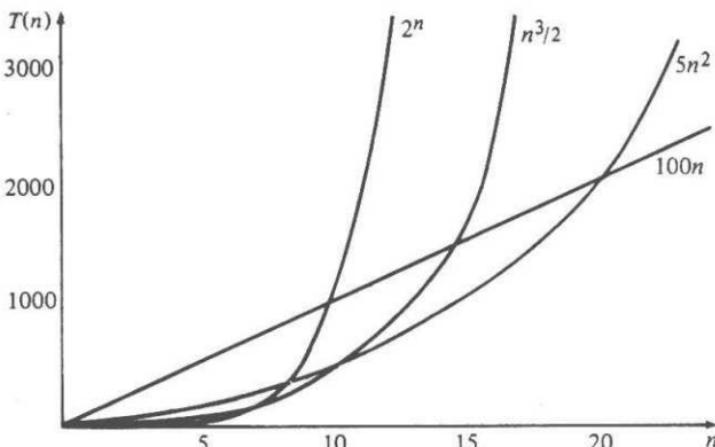


Fig. 1.11. Tiempos de ejecución de cuatro programas.

Ejemplo 1.7. En la figura 1.11 se pueden ver los tiempos de ejecución de cuatro programas de distintas complejidades de tiempo, medidas en segundos, para una combinación determinada de compilador y máquina. Supóngase que se dispone de 1000 segundos, o alrededor de 17 minutos, para resolver un problema determinado. ¿Qué tamaño de problema se puede resolver? Como se puede ver en la segunda columna de la figura 1.12, los cuatro algoritmos pueden resolver problemas de un tamaño similar en 10^3 segundos.

Supóngase ahora que se adquiere una máquina que funciona diez veces más rápido sin costo adicional. Entonces, con el mismo costo, es posible dedicar 10^4 segundos a la solución de problemas que antes requerían 10^3 segundos. El tamaño máximo de problema que es posible resolver ahora con cada uno de los cuatro programas se muestra en la tercera columna de la figura 1.12, y la razón entre los valores de las columnas segunda y tercera se muestra en la cuarta. Se observa que un aumento del 1000% en la velocidad del computador origina apenas un incremento del 30% en el tamaño de problema que se puede resolver con el programa $O(2^n)$. Los aumentos adicionales de un factor de diez en la rapidez del computador a partir de este punto originan aumentos porcentuales aún menores en el tamaño de los problemas. De hecho, el programa $O(2^n)$ sólo puede resolver problemas pequeños, independientemente de la rapidez del computador.

Tiempo de ejecución $T(n)$	Tamaño máximo de problema para 10^3 seg	Tamaño máximo de problema para 10^4 seg	Incremento en el tamaño máximo de problema
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Fig. 1.12. Efecto de multiplicar por diez la velocidad del computador.

En la tercera columna de la figura 1.12 se puede apreciar una superioridad evidente del programa $O(n)$; éste permite un aumento del 1000% en el tamaño de problema para un incremento del 1000% en la rapidez del computador. Se observa que los programas $O(n^3)$ y $O(n^2)$ permiten aumentos de 230% y 320%, respectivamente, en el tamaño de problema, para un incremento del 1000% en la rapidez del computador. Estas razones se mantendrán vigentes para incrementos adicionales en la rapidez del computador. □

Mientras exista la necesidad de resolver problemas cada vez más grandes, se producirá una situación casi paradójica. A medida que los computadores aumenten su rapidez y disminuyan su precio, como con toda seguridad seguirá sucediendo, también el deseo de resolver problemas más grandes y complejos seguirá creciendo. Así, la importancia del descubrimiento y el empleo de algoritmos eficientes (aquellos cuyas velocidades de crecimiento sean pequeñas) irá en aumento, en lugar de disminuir.

Aspectos importantes

Es necesario subrayar de nuevo que la velocidad de crecimiento del tiempo de ejecución del peor caso no es el único criterio, ni necesariamente el más importante, para evaluar un algoritmo o un programa. A continuación se presentan algunas condiciones en las cuales el tiempo de ejecución de un programa se puede ignorar en favor de otros factores.

1. Si un programa se va a utilizar sólo algunas veces, el costo de su escritura y depuración es el dominante, de manera que el tiempo de ejecución raramente influirá en el costo total. En ese caso debe elegirse el algoritmo que sea más fácil de aplicar correctamente.
2. Si un programa se va a ejecutar sólo con entradas «pequeñas», la velocidad de crecimiento del tiempo de ejecución puede ser menos importante que el factor constante de la fórmula del tiempo de ejecución. Determinar qué es una entrada «pequeña», depende de los tiempos de ejecución exactos de los algoritmos implicados. Hay algoritmos (como el de multiplicación de enteros de Schonhage y Strassen [1971]) que son asintóticamente los más eficientes para sus problemas, pero nunca se han llevado a la práctica ni siquiera con los problemas más grandes, debido a que la constante de proporcionalidad es demasiado grande comparada con la de otros algoritmos menos «eficientes».
3. Un algoritmo eficiente pero complicado puede no ser apropiado porque posteriormente puede tener que darle mantenimiento otra persona distinta del escritor. Se espera que al difundir el conocimiento de las principales técnicas de diseño de algoritmos eficientes, se podrán utilizar libremente algoritmos más complejos, pero debe considerarse la posibilidad de que un programa resulte inútil debido a que nadie entiende sus sutiles y eficientes algoritmos.
4. Existen ejemplos de algoritmos eficientes que ocupan demasiado espacio para ser aplicados sin almacenamiento secundario lento, lo cual puede anular la eficiencia.
5. En los algoritmos numéricos, la precisión y la estabilidad son tan importantes como la eficiencia.

1.5 Cálculo del tiempo de ejecución de un programa

Calcular el tiempo de ejecución de un programa arbitrario, aunque sólo sea una aproximación a un factor constante, puede ser un problema matemático complejo. Sin embargo, en la práctica esto suele ser más sencillo; basta con aplicar unos cuantos principios básicos. Antes de presentar estos principios, es importante aprender a sumar y a multiplicar en notación asintótica.

Supóngase que $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos fragmentos de programa P_1 y P_2 , y que $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$. Entonces $T_1(n) + T_2(n)$, el tiempo de ejecución de P_1 seguido de P_2 , es $O(\max(f(n), g(n)))$. Para saber por qué, obsérvese que para algunas constantes, c_1 , c_2 , n_1 y n_2 , si $n \geq n_1$, entonces

$T_1(n) \leq c_1 f(n)$, y si $n \geq n_0$, entonces $T_2(n) \leq c_2 g(n)$. Sea $n_0 = \max(n_1, n_2)$. Si $n \geq n_0$, entonces $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. De aquí se concluye que si $n \geq n_0$, entonces $T_1(n) + T_2(n) \leq (c_1 + c_2) \max(f(n), g(n))$; por tanto, $T_1(n) + T_2(n)$ es $O(\max(f(n), g(n)))$.

Ejemplo 1.8. La regla de la suma anterior puede usarse para calcular el tiempo de ejecución de una secuencia de pasos de programa, donde cada paso puede ser un fragmento de programa arbitrario con ciclos y ramificaciones. Supóngase que se tienen tres pasos cuyos tiempos de ejecución son, respectivamente, $O(n^2)$, $O(n^3)$ y $O(n \log n)$. Entonces, el tiempo de ejecución de los dos primeros pasos ejecutados en secuencia es $O(\max(n^2, n^3))$ que es $O(n^3)$. El tiempo de ejecución de los tres juntos es $O(\max(n^3, n \log n))$, que es $O(n^3)$. □

En general, el tiempo de ejecución de una secuencia fija de pasos, dentro de un factor constante, es igual al tiempo de ejecución del paso con mayor tiempo de ejecución. En raras ocasiones dos pasos pueden tener tiempos de ejecución *incomensurables* (ninguno es mayor que el otro, ni son iguales). Por ejemplo, puede haber pasos con tiempos de ejecución $O(f(n))$ y $O(g(n))$, donde

$$f(n) = \begin{cases} n^4 & \text{si } n \text{ es par} \\ n^2 & \text{si } n \text{ es impar} \end{cases} \quad g(n) = \begin{cases} n^2 & \text{si } n \text{ es par} \\ n^3 & \text{si } n \text{ es impar} \end{cases}$$

En tales casos, la regla de la suma debe aplicarse directamente; en el ejemplo, el tiempo de ejecución es $O(\max(f(n), g(n)))$, esto es, n^4 si n es par y n^3 si n es impar.

Otra observación útil sobre la regla de la suma es que si $g(n) \leq f(n)$ para toda n mayor que una constante n_0 , entonces $O(f(n) + g(n))$ es lo mismo que $O(f(n))$. Por ejemplo, $O(n^2 + n)$ es lo mismo que $O(n^2)$.

La regla del producto es la siguiente: si $T_1(n)$ y $T_2(n)$ son $O(f(n))$ y $O(g(n))$, respectivamente, entonces $T_1(n)T_2(n)$ es $O(f(n)g(n))$. Se aconseja probar este hecho con las mismas ideas que se utilizaron para probar la regla de la suma. Según la regla del producto, $O(c f(n))$ significa lo mismo que $O(f(n))$ si c es una constante positiva cualquiera. Por ejemplo, $O(n^2/2)$ es lo mismo que $O(n^2)$.

Antes de pasar a las reglas generales de análisis de los tiempos de ejecución de los programas, se presentará un ejemplo sencillo para proporcionar una visión general del proceso.

Ejemplo 1.9. Considérese el programa de clasificación *burbuja* de la figura 1.13, que ordena un arreglo de enteros de menor a mayor. El efecto total de cada recorrido del ciclo interno de las proposiciones (3) a (6), es hacer que el menor de los elementos suba hasta el principio del arreglo.

El número n de elementos que se van a clasificar es la medida apropiada del tamaño de la entrada. La primera observación que se hace es que cada proposición de asignación toma cierta cantidad constante de tiempo, independiente del tamaño de la entrada. Esto significa que las proposiciones (4), (5) y (6) toman tiempo $O(1)$ cada una. Obsérvese que $O(1)$ es la notación «o mayúscula» de una «cantidad constante». Por la regla de la suma, el tiempo de ejecución combinado de este grupo de proposiciones es $O(\max(1, 1, 1)) = O(1)$.

Ahora deben tenerse en cuenta las proposiciones condicionales y las de control de ciclos. Las proposiciones **if** y **for** están anidadas unas dentro de otras, de modo

```

procedure burbuja ( var A: array [1..n] of integer );
{ burbuja clasifica el arreglo A de menor a mayor }
var
  i, j, temp: integer;
begin
  for i := 1 to n-1 do
    for j := n downto i + 1 do
      if A[j-1] > A[j] then begin
        { intercambia A[j-1] y A[j] }
        temp := A[j-1];
        A[j-1] := A[j];
        A[j] := temp
      end
    end; { burbuja }
  
```

Fig. 1.13. Clasificación burbuja.

que se debe ir de dentro hacia fuera para obtener el tiempo de ejecución del grupo condicional y de cada ciclo. En cuanto a la proposición **if**, la prueba de la condición requiere tiempo $O(1)$. No se sabe si el cuerpo de la proposición **if** (líneas (4) a (6)) se ejecutará, pero dado que se busca el tiempo de ejecución del peor caso, se supone lo peor, esto es, que se ejecute. Por tanto, el grupo **if** de las proposiciones (3) a (6) requiere tiempo $O(1)$.

Así, siguiendo hacia fuera, se llega al ciclo **for** de las líneas (2) a (6). La regla general para un ciclo es que el tiempo de ejecución total resulta de sumar, en cada iteración, los tiempos empleados en ejecutar el cuerpo del ciclo en esa iteración. Debe acumularse al menos $O(1)$ por cada iteración para justificar el incremento del índice, con el fin de verificar si se alcanzó el límite y para saltar de vuelta al principio del ciclo. Para el ciclo de las líneas (2) a (6), el cuerpo tarda tiempo $O(1)$ en cada iteración. El número de iteraciones es $n-i$, de modo que, por la regla del producto, el total de tiempo invertido en el ciclo de las líneas (2) a (6) es $O((n-i) \times 1)$, o sea, $O(n-i)$.

Se sigue ahora con el ciclo externo, el cual contiene todas las proposiciones ejecutables del programa. La proposición (1) se ejecuta $n-1$ veces, de manera que el tiempo total de ejecución del programa tiene como cota superior una constante multiplicada por

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

que es $O(n^2)$. Por tanto, para ejecutar el programa de la figura 1.13 se necesita un tiempo proporcional al cuadrado del número de elementos que se van a clasificar.

En el capítulo 8 se presentarán programas de clasificación cuyo tiempo de ejecución es $O(n \log n)$, que es considerablemente menor, dado que $\log n$ † es mucho menor que n para valores grandes de n . □

Antes de proseguir con algunas reglas generales de análisis, recuérdese que la determinación de una cota superior precisa para el tiempo de ejecución de un programa, unas veces es sencilla, pero otras puede ser un desafío intelectual profundo. No existe un conjunto completo de reglas para analizar programas; en este libro sólo se proporcionarán ciertas sugerencias e ilustrarán algunos de sus aspectos más sutiles mediante ejemplos.

Se enumerarán ahora algunas reglas generales para el análisis de programas. En general, el tiempo de ejecución de una proposición o de un grupo de ellas puede tener como parámetros el tamaño de la entrada, una o más variables, o ambas cosas. El único parámetro permisible para el tiempo de ejecución del programa completo es n , el tamaño de la entrada.

1. El tiempo de ejecución de cada proposición de asignación (lectura y escritura), por lo común puede tomarse como $O(1)$. Hay unas cuantas excepciones, como en PL/I, donde una asignación puede implicar matrices arbitrariamente grandes, y en cualquier lenguaje donde se permitan llamadas a funciones en las proposiciones de asignación.
2. El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma. Esto es, el tiempo de ejecución de una secuencia es, dentro de un factor constante, el máximo tiempo de ejecución de una proposición de la secuencia.
3. El tiempo de ejecución de una proposición condicional *if* es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición. El tiempo para evaluar la condición, por lo general, es $O(1)$. El tiempo para una construcción *if-then-else* es la suma del tiempo requerido para evaluar la condición más el mayor entre los tiempos necesarios para ejecutar las proposiciones cuando la condición es verdadera y el tiempo de ejecución de las proposiciones cuando la condición es falsa.
4. El tiempo para ejecutar un ciclo es la suma, sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación (este último suele ser $O(1)$). A menudo este tiempo es, despreciando factores constantes, el producto del número de iteraciones del ciclo y el mayor tiempo posible para una ejecución del cuerpo, pero, por seguridad, debe considerarse cada iteración por separado. Por lo común, se conoce con certeza el número de iteraciones, pero en ocasiones no es posible determinarlo con precisión. Incluso puede ocurrir que un programa no sea un algoritmo y que no exista un límite al número de iteraciones de ciertos ciclos.

† A menos que se especifique lo contrario, todos los logaritmos son de base 2. Obsérvese que $O(\log n)$ no depende de la base del logaritmo, puesto que $\log_b n = c \log_a n$, donde $c = \log_a b$.

Llamadas a procedimientos

Si se tiene un programa con procedimientos que no son recursivos, es posible calcular el tiempo de ejecución de los distintos procedimientos, uno a la vez, partiendo de aquellos que no llaman a otros. (Recuérdese que la invocación a una función debe considerarse una «llamada».) Debe haber al menos un procedimiento con esa característica, a menos que como mínimo un procedimiento sea recursivo. Después, puede evaluarse el tiempo de ejecución de los procedimientos que sólo llaman a procedimientos que no hacen llamadas, usando los tiempos de ejecución de los procedimientos llamados evaluados antes. Se continúa el proceso evaluando el tiempo de ejecución de cada procedimiento después de haber evaluado los tiempos correspondientes a los procedimientos que llama.

Si hay procedimientos recursivos, no es posible ordenar las evaluaciones de modo que cada una utilice sólo evaluaciones ya realizadas. Lo que se debe hacer ahora es asociar a cada procedimiento recursivo una función de tiempo desconocida $T(n)$, donde n mide el tamaño de los argumentos del procedimiento. Luego se puede obtener una *recurrencia* para $T(n)$, es decir, una ecuación para $T(n)$ en función de $T(k)$ para varios valores de k .

Se conocen ya técnicas para resolver varias clases de recurrencias; algunas de ellas se presentarán en el capítulo 9. Aquí se mostrará cómo analizar un programa recursivo sencillo.

Ejemplo 1.10. La figura 1.14 muestra un programa recursivo para calcular $n!$, que es el producto de todos los enteros de 1 a n inclusive.

Una medida de tamaño apropiado para esta función es el valor de n . Sea $T(n)$ el tiempo de ejecución para $\text{fact}(n)$. El tiempo de ejecución para las líneas (1) y (2) es $O(1)$, y para la línea (3) es $O(1) + T(n-1)$. Por tanto, para ciertas constantes c y d ,

$$T(n) = \begin{cases} c + T(n-1) & \text{si } n > 1 \\ d & \text{si } n \leq 1 \end{cases} \quad (1.1)$$

```
function fact ( n: integer ) : integer;
  {fact(n) calcula n!}
  begin
    (1)      if n <= 1 then
              fact := 1
            else
              fact := n * fact(n-1)
    end; {fact}
```

Fig. 1.14. Programa recursivo para calcular factoriales.

Suponiendo que $n > 2$, se puede desarrollar $T(n-1)$ en (1.1) para obtener

$$T(n) = 2c + T(n-2) \text{ si } n > 2$$

Esto es, $T(n-1) = c + T(n-2)$, como se puede ver al sustituir n por $n-1$ en (1.1). Así pues, es posible reemplazar $T(n-1)$ con $c + T(n-2)$ en la ecuación $T(n) = c + T(n-1)$. Despues, se puede usar (1.1) para desarrollar $T(n-2)$, con lo que se obtiene

$$T(n) = 3c + T(n-3) \quad \text{si } n > 3$$

y así sucesivamente. En general,

$$T(n) = ic + T(n-i) \quad \text{si } n > i$$

Por último, cuando $i = n-1$, se obtiene

$$T(n) = c(n-1) + T(1) = c(n-1) + d \quad (1.2)$$

Por (1.2) se concluye que $T(n)$ es $O(n)$. Es importante observar que en este análisis se ha supuesto que la multiplicación de dos enteros es una operación de tiempo $O(1)$. En la práctica, no obstante, no se puede emplear el programa de la figura 1.14 para calcular $n!$ cuando los valores de n son grandes, pues el tamaño de los enteros que se calculen excederá del tamaño de palabra de la máquina en cuestión. \square

El método general para resolver una ecuación de recurrencia, tal como se tipifica en el ejemplo 1.10, consiste en reemplazar en forma repetida términos $T(k)$ del lado derecho de la ecuación por el lado derecho completo, donde k se reemplaza por n , hasta obtener una fórmula en la que T no aparezca en el lado derecho como en (1.2). A menudo es necesario calcular la suma de una sucesión o, si no es posible encontrar la suma exacta, obtener una cota superior cercana para la suma a fin de hallar una cota superior para $T(n)$.

Programas con proposiciones GOTO

Hasta ahora, al analizar el tiempo de ejecución de un programa, se supuso de manera tácita que el flujo de control dentro de un procedimiento estaba determinado por construcciones de ciclos y de ramificación. Esto sirvió como base en la determinación del tiempo de ejecución de grupos de proposiciones cada vez más grandes, al suponer que sólo se necesitaba la regla de la suma para agrupar secuencias de proposiciones. Sin embargo, las proposiciones **goto** (proposiciones de transferencia incondicional de control) hacen más complejo el agrupamiento lógico de las proposiciones de un programa. Por esta razón, las proposiciones **goto** deberían evitarse, pero Pascal carece de proposiciones para salir de un ciclo o terminarlo en forma anormal (como *break* y *continue*), por lo que con frecuencia se utiliza **goto** para estos fines.

Para manejar proposiciones **goto** que realicen saltos de un ciclo a código que con seguridad está después del ciclo, se sugiere el siguiente enfoque. (Por lo general,

ésta es la única clase de *goto* que está justificada.) Puesto que es probable que se ejecute el *goto* en forma condicional dentro del ciclo, se puede suponer que nunca se efectúa. Debido a que el *goto* transfiere el control a una proposición que se ejecutará después de terminado el ciclo, esta suposición resulta conservadora; no es posible subestimar el tiempo de ejecución del peor caso si se presume que el ciclo se ejecuta por completo. Sin embargo, es raro el programa en el que ignorar el *goto* es tan conservador que puede llevar a sobreestimar la velocidad de crecimiento del tiempo de ejecución del programa para el peor caso. Obsérvese que si se presentara un *goto* que transfiriera el control a código ejecutado con anterioridad, no sería posible ignorarlo sin correr riesgos, pues ese *goto* podría crear un ciclo que consumiera la mayor parte del tiempo de ejecución.

No por esto debe pensarse que el uso de proposiciones *goto* hacia atrás en sí mismo hace que los tiempos de ejecución sean imposibles de analizar. El enfoque para analizar tiempos de ejecución, que se describió en esta sección, funcionará bien en la medida en que los ciclos de un programa tengan una estructura razonable, es decir, que cualquier par de ciclos sea siempre disjunto o anidado. (No obstante, es responsabilidad del analizador descubrir la estructura de los ciclos.) Así pues, no debe dudarse en aplicar estos métodos de análisis de programas a lenguajes como FORTRAN, donde los *goto* son esenciales, pero los programas tienden a tener una estructura de ciclos razonable.

Análisis de un seudoprograma

Si se conoce la velocidad de crecimiento del tiempo necesario para ejecutar proposiciones informales en español, es posible analizar seudoprogramas como si fueran programas reales. Sin embargo, muchas veces no se conoce el tiempo que demandarán las partes no completamente terminadas de un seudoprograma. Por ejemplo, si se tiene un seudoprograma donde las únicas partes no terminadas son operaciones con TDA, puede elegirse una de las varias implantaciones posibles para un TDA y el tiempo de ejecución total puede depender en gran medida de la realización elegida. Por supuesto, una de las razones en favor de la escritura de programas con TDA es que permite considerar los pros y los contras entre los tiempos de ejecución de varias operaciones que resultan de efectuar distintas realizaciones.

Para analizar seudoprogramas que contengan proposiciones en algún lenguaje de programación y llamadas a procedimientos aún no implantados, tales como operaciones con TDA, se calcula el tiempo de ejecución como una función de los tiempos de ejecución no especificados de esos procedimientos. El tiempo de ejecución de un procedimiento obtiene sus parámetros mediante el «tamaño» de su argumento (o argumentos). Así como sucede con el «tamaño de la entrada», la medida apropiada para el tamaño de un argumento es decisión de quien hace el análisis. Si el procedimiento es una operación con un TDA, el modelo matemático subyacente a menudo indica la noción lógica de tamaño. Por ejemplo, si el TDA está basado en conjuntos, el número de elementos de los conjuntos es con frecuencia la noción correcta de tamaño. En los capítulos siguientes se verán muchos ejemplos de análisis del tiempo de ejecución de seudoprogramas.

1.6 Buenas prácticas de programación

Hay un buen número de ideas que se deberían tener presente al diseñar un algoritmo e implantarlo como programa. Estas ideas a menudo parecen perogrulladas, porque sólo se pueden apreciar en toda su extensión a través de su empleo acertado en situaciones prácticas, más que en el desarrollo de una teoría. A pesar de esto, son lo bastante importantes para repetirlas aquí. El lector debe observar con detenimiento la aplicación de estas ideas en los programas diseñados en este libro, así como buscar oportunidades para ponerlas en práctica en sus propios programas.

1. *Planificar el diseño de un programa.* En la sección 1.1 se mostró cómo diseñar un programa partiendo de un esbozo informal del algoritmo, con el diseño de un seudoprograma y luego con el refinamiento gradual del seudocódigo hasta convertirlo en ejecutable. Esta estrategia de esbozar, y después detallar, tiende a la producción de un programa final más organizado que facilita su depuración y mantenimiento.
2. *Encapsular.* Empléense procedimientos y diversos TDA para que el código asociado a cada operación importante y a cada tipo de datos quede colocado en un solo lugar del listado del programa. Después, si es necesario hacer cambios, la sección de código pertinente se localizará con facilidad.
3. *Usar o modificar un programa existente.* Una de las principales causas de ineficiencia en el proceso de programación es que muchas veces un proyecto se aborda como si fuera el primer programa que se hubiera escrito. Se debería buscar primero un programa ya elaborado que realice la tarea completa o parte de ella. A la inversa, cuando se escribe un programa, se debería pensar en ponerlo a disposición de otras personas para posibles usos no previstos.
4. *Confeccionar herramientas.* En lenguaje de computación, una *herramienta* es un programa que tiene una variedad de usos. Cuando se escriba un programa, debe pensarse en la posibilidad de escribirlo de un modo más general sin mayor esfuerzo adicional. Por ejemplo, supóngase que se pide escribir un programa para la programación de exámenes finales; en lugar de eso, escribase una herramienta que tome un grafo arbitrario y coloréense sus vértices con el mínimo de colores posible, de modo que no haya dos vértices del mismo color unidos por una arista. En el contexto de la programación de exámenes, los vértices son los grupos de estudiantes, los colores son los períodos de exámenes, y una arista entre dos grupos significa que éstos tienen algún estudiante en común. El programa de coloración, junto con rutinas que traduzcan las listas de grupos en grafos y los colores en días y horas específicos, es el programador de exámenes. Sin embargo, el programa de coloración se puede usar para resolver problemas que no tengan relación alguna con la programación de exámenes, como el problema de los somáforos de la sección 1.1.
5. *Programar a nivel de mandatos.* En ocasiones, es imposible encontrar en una biblioteca el programa que se necesita para realizar cierto trabajo, ni adaptar una herramienta para tal efecto. Un sistema operativo bien diseñado permite conectar entre sí una red de programas disponibles sin que sea necesario escribir un programa nuevo, sino sólo una lista de mandatos (*commands*) del sistema ope-

rativo. Para que los mandatos sean pertinentes, es necesario que cada uno se comporte como un *filtro*, o sea como un programa con un archivo de entrada y otro de salida. Obsérvese que es posible componer un número arbitrario de filtros, y si el sistema operativo está diseñado de forma inteligente, una simple lista de mandatos (dispuestos como han de ejecutarse) será suficiente como programa.

Ejemplo 1.11. Como ejemplo de lo anterior, considérese el programa *spell*, tal como fue escrito por S. C. Johnson a partir de mandatos de UNIX †. Este programa toma como entrada un archivo a_1 que contiene un texto en inglés, y produce como salida todas aquellas palabras en a_1 que no se encuentren en un pequeño diccionario ‡‡. *spell* tiende a listar los nombres propios como faltas de ortografía, al igual que las palabras que no figuren en el diccionario, pero dado que la salida típica del programa es pequeña, puede examinarse visualmente, y la inteligencia de cualquier ser humano puede determinar si una palabra de la salida de *spell* tiene faltas de ortografía o no. (La ortografía de la versión original en inglés de este libro se verificó con *spell*.)

El primer filtro utilizado por *spell* es un mandato llamado *translate*; cuando este mandato recibe parámetros apropiados, puede reemplazar las letras mayúsculas por minúsculas y los espacios por saltos de línea, sin alterar los caracteres restantes. La salida de *translate* es un archivo a_2 que contiene las palabras de a_1 , sin mayúsculas y una en cada línea. A continuación, aparece el mandato *sort* que clasifica las líneas de su archivo de entrada en orden lexicográfico (alfabético). La salida de *sort* es un archivo a_3 que tiene todas las palabras de a_2 en orden alfabetico, con repeticiones. Entonces, un mandato *unique* elimina las líneas duplicadas de su archivo de entrada y produce un archivo de salida a_4 que contiene las palabras del archivo original, sin mayúsculas ni duplicaciones, en orden alfabetico. Por último, se aplica a a_4 un mandato *diff*, con un parámetro que indica un segundo archivo a_5 que contiene una lista alfabetizada de las palabras del diccionario, una en cada línea. El resultado tiene todas las palabras de a_4 (y por tanto de a_1) que no están en a_5 , esto es, todas las palabras de la entrada original que no se encuentran en el diccionario. El programa *spell* completo no es más que la siguiente secuencia de mandatos.

```
spell: translate [A-Z] → [a-z], espacio → nueva_línea
      sort
      unique
      diff diccionario
```

La programación a nivel de mandatos requiere disciplina por parte de una comunidad de programadores; éstos deben escribir programas como filtros, y escribir herramientas, en lugar de programas de propósito especial, siempre que les sea posible. La recompensa de esto, en función de la razón global entre trabajo y resultados, es sustancial. □

† UNIX es marca registrada de los Laboratorios Bell.

‡‡ Podría usarse un diccionario no abreviado, pero muchos de los errores ortográficos pueden ser palabras en desuso.

1.7 Súper Pascal

La mayoría de los programas de este libro están escritos en Pascal. No obstante, para aumentar su legibilidad, en ocasiones se usan tres construcciones que no forman parte de Pascal estándar, pero que se pueden traducir mecánicamente a Pascal puro. Una de tales construcciones es la etiqueta no numérica. Las pocas veces que se requieran etiquetas, serán no numéricas, porque facilitan la comprensión de los programas. Por ejemplo, «*goto salida*» siempre es más claro que «*goto 561*». Para convertir a Pascal puro un programa que contenga etiquetas no numéricas, se debe reemplazar cada una de ellas por una etiqueta numérica distinta y después declarar las etiquetas numéricas al principio del programa. Este proceso puede llevarse a cabo mecánicamente.

La segunda construcción no estándar es la proposición de retorno, que se usa porque permite escribir programas más fáciles de entender, sin emplear proposiciones *goto* para interrumpir el flujo de control. La proposición de retorno que aquí se emplea tiene la forma

```
return (expresión)
```

donde (*expresión*) es opcional. Es fácil convertir a Pascal estándar un procedimiento que contenga proposiciones **return**. En primer lugar, se declara una etiqueta nueva, como 999, y con ella se etiqueta la última proposición **end** del procedimiento. Si en una función de nombre *y*, por ejemplo, aparece la proposición **return(x)**, ésta se reemplaza por el bloque

```
begin
  y := x;
  goto 999
end
```

En un procedimiento, la proposición **return**, que puede carecer de argumento, simplemente se reemplaza por **goto 999**.

Ejemplo 1.12. La figura 1.15 muestra el programa para calcular factoriales, escrito con proposiciones **return**. En la figura 1.16 aparece el programa en Pascal resultante de la aplicación sistemática de la transformación indicada a la figura 1.15. □

```
function fact ( n: integer ) : integer;
begin
  if n <= 1 then
    return (1)
  else
    return (n * fact(n-1))
end; {fact}
```

Fig. 1.15. Programa de cálculo de factoriales con proposiciones de retorno.

```
function fact ( n: integer ) : integer;
label
  999;
begin
  if n <= 1 then
    begin
      fact := 1;
      goto 999
    end
  else
    begin
      fact := n * fact(n-1);
      goto 999
    end
  999:
end: {fact }
```

Fig. 1.16. Programa resultante en Pascal.

La tercera extensión a Pascal que se utiliza consiste en usar expresiones como nombres de tipos a lo largo de un programa de manera uniforme. Por ejemplo, una expresión como \uparrow tipo_celda, aunque puede emplearse en cualquier otra parte, no se permite como tipo de un parámetro de un procedimiento o como tipo del valor devuelto por una función. Técnicamente, Pascal requiere que se invente un nombre para la expresión de tipo, como ap_a_celda. En este libro se permitirá este uso de las expresiones de tipo, y se sugiere inventar un nombre para el tipo y que en forma mecánica se reemplacen las expresiones por los nombres. Así, se escribirán proposiciones como

function y (*A*: array [1..10] of integer): \uparrow tipo_celda

en lugar de

function y (*A*: arreglo_de_diez_enteros): ap_a_celda

Para concluir, debe hacerse una observación acerca de las convenciones tipográficas que se utilizan en los programas. Las palabras reservadas de Pascal se escriben en negritas; los tipos, en redondas, y los nombres de variables, en cursivas. Se hará distinción entre letras mayúsculas y minúsculas.

Ejercicios

- 1.1 Una liga de fútbol tiene seis equipos que se representarán con las letras *A*, *B*, *C*, *D*, *E* y *F*. El equipo *A* ya jugó contra los equipos *B* y *C*; el equipo *B*,

además, se enfrentó al *D* y al *F*. El *E* ha jugado contra el *C* y el *F*. Cada equipo tiene un encuentro por semana. Elabórese un calendario tal que cada equipo haya jugado contra todos los demás en el menor número posible de semanas. *Sugerencia:* Diséñese un grafo cuyos vértices representen pares de equipos que aún no se hayan enfrentado. ¿Cuáles deberían ser las aristas para que en una coloración lícita del grafo, cada color pudiera representar los juegos realizados en una semana?

- *1.2 Considérese un brazo robot fijo por un extremo. El brazo tiene dos articulaciones; en cada una de ellas, es posible rotarlo 90 grados hacia arriba o hacia abajo en un plano vertical. ¿Cómo podría modelarse matemáticamente la variedad de movimientos posibles del extremo del brazo? Describase un algoritmo para mover el extremo del brazo robot de una posición permisible a otra.
- *1.3 Supóngase que se desean multiplicar cuatro matrices de números reales: $M_1 \times M_2 \times M_3 \times M_4$, donde M_1 es de 10 por 20, M_2 de 20 por 50, M_3 de 50 por 1 y M_4 de 1 por 100. Supóngase que la multiplicación de una matriz de $p \times q$ por otra de $q \times r$ requiere pqr operaciones escalares, como sucede con el algoritmo usual de multiplicación de matrices. Encuéntrese el orden óptimo de multiplicación de matrices que minimice el número total de operaciones escalares. ¿Cómo se encontraría ese ordenamiento óptimo si se tuviera un número arbitrario de matrices?
- **1.4 Supóngase que se desean dividir las raíces cuadradas de los enteros del 1 al 100 en dos pilas de 50 números cada una, de modo que la suma de los números de la primera pila sea lo más cercana posible a la suma de los números de la segunda. Si sólo se dispusiera de dos minutos de tiempo de computador para resolver el problema, ¿qué cálculo sería aconsejable realizar?
- 1.5 Describase un algoritmo ávido para jugar al ajedrez. ¿Cabe esperar que tenga un buen rendimiento?
- 1.6 En la sección 1.2, se consideró el TDA CONJUNTO con las operaciones ANULA, UNION y TAMAÑO. Supóngase por conveniencia que todos los conjuntos son subconjuntos de $\{0, 1, \dots, 31\}$, y que el TDA CONJUNTO se interpreta como el tipo de datos de Pascal set of 0..31. Escribanse procedimientos en PASCAL para esas operaciones utilizando esta realización de CONJUNTO.
- 1.7 El *máximo común divisor* de dos enteros p y q es el mayor entero d que divide exactamente p y q . Se desea desarrollar un programa para calcular el máximo común divisor de dos enteros p y q con el siguiente algoritmo. Sea r el residuo de dividir p entre q . Si r es igual a cero, entonces q es el máximo común divisor. En caso contrario, hágase que p tome el valor de q , y q el de r , y repítase el proceso.
 - a) Muéstrese que este proceso encuentra el máximo común divisor correcto.

- b) Refíñese el algoritmo en la forma de un programa en seudolenguaje.
 c) Conviértase el programa en seudolenguaje en un programa en Pascal.
- 1.8** Se desea desarrollar un programa para formato de textos que coloque las palabras en las líneas de modo que queden justificadas a la izquierda y a la derecha. El programa tiene dos áreas de almacenamiento transitorio (*buffers*), uno de palabras y otro de líneas; ambas están vacías al inicio. Una palabra se lee al área transitoria, y si hay espacio suficiente en el área de líneas, la palabra se transfiere a ésta. En caso contrario, se insertan espacios adicionales entre las palabras del área transitoria de líneas para completar una línea; al imprimir esta línea, el área se vacía.
- a) Refíñese este algoritmo en la forma de un programa en seudolenguaje.
 b) Conviértase el programa en seudolenguaje en un programa en Pascal.
- 1.9** Considérese un conjunto de n ciudades y una tabla de distancias entre pares de ellas. Escribáse un programa en seudolenguaje que encuentre un camino corto que pase por cada ciudad sólo una vez y vuelva a la ciudad en que empezó. No se conoce un método que no realice una búsqueda exhaustiva para obtener el recorrido más corto. Así pues, inténtese encontrar un algoritmo eficiente para este problema utilizando algún procedimiento heurístico razonable.
- 1.10** Considérense las siguientes funciones de n :

$$f_1(n) = n^2$$

$$f_2(n) = n^2 + 1000n$$

$$f_3(n) = \begin{cases} n & \text{si } n \text{ es impar} \\ n^3 & \text{si } n \text{ es par} \end{cases}$$

$$f_4(n) = \begin{cases} n & \text{si } n \leq 100 \\ n^3 & \text{si } n > 100 \end{cases}$$

Para cada posible par de valores de i y j , indíquese si $f_i(n)$ es $O(f_j(n))$ o no, y si $f_i(n)$ es $\Omega(f_j(n))$ o no.

- 1.11** Considérense las siguientes funciones de n :

$$g_1(n) = \begin{cases} n^2 & \text{para } n \text{ par} \geq 0 \\ n^3 & \text{para } n \text{ impar} \geq 1 \end{cases}$$

$$g_2(n) = \begin{cases} n & \text{para } 0 \leq n \leq 100 \\ n^3 & \text{para } n > 100 \end{cases}$$

$$g_3(n) = n^{2.5}$$

Para cada par de valores de i y j , indíquese si $g_i(n)$ es $O(g_j(n))$ o no, y si $g_i(n)$ es $\Omega(g_j(n))$ o no.

- 1.12 Mediante la notación asintótica, obténganse los tiempos de ejecución del peor caso supuesto para cada uno de los procedimientos siguientes como una función de n .

```

a) procedure prod_mat ( n: integer );
  var
    i, j, k: integer;
  begin
    for i := 1 to n do
      for j := 1 to n do begin
        C[i, j] := 0;
        for k := 1 to n do
          C[i, j] := C[i, j] + A[i, k] * B[k, j]
      end
    end
  end

b) procedure misterio ( n: integer );
  var
    i, j, k: integer;
  begin
    for i := 1 to n - 1 do
      for j := i + 1 to n do
        for k := 1 to j do
          { alguna proposición que requiera tiempo  $O(1)$  }
    end
  end

c) procedure muy_impar ( n: integer );
  var
    i, j, x, y: integer;
  begin
    for i := 1 to n do
      if odd(i) then begin
        for j := i to n do
          x := x + 1;
        for j := 1 to i do
          y := y + 1
      end
    end
  end

*d) function recursiva ( n: integer ) : integer;
  begin
    if n <= 1 then
      return (1)
    else
      return ( recursiva(n - 1) + recursiva(n - 1) )
  end

```

- 1.13** Muéstrese que las siguientes afirmaciones son verdaderas.
- 17 es $O(1)$.
 - $n(n-1)/2$ es $O(n^2)$.
 - $\max(n^3, 10n^2)$ es $O(n^3)$.
 - $\sum_{i=1}^n i^k$ es $O(n^{k+1})$ y $\Omega(n^{k+1})$ para k entero.
 - Si $p(x)$ es cualquier polinomio de grado k donde el coeficiente del término de mayor grado es positivo, entonces $p(n)$ es $O(n^k)$ y $\Omega(n^k)$.
- *1.14** Supóngase que $T_1(n)$ es $\Omega(f(n))$ y $T_2(n)$ es $\Omega(g(n))$. ¿Son verdaderas las siguientes afirmaciones?
- $T_1(n) + T_2(n)$ es $\Omega(\max(f(n), g(n)))$.
 - $T_1(n)T_2(n)$ es $\Omega(f(n)g(n))$.
- *1.15** Algunos autores definen la notación omega mayúscula de la siguiente manera: $f(n)$ es $\Omega(g(n))$ si existen n_0 y $c > 0$ tales que para todo $n \geq n_0$ se tiene que $f(n) \geq cg(n)$.
- ¿Es verdadero para esta definición que $f(n)$ es $\Omega(g(n))$ si, y sólo si, $g(n)$ es $O(f(n))$?
 - ¿Es a) verdadero para la definición de omega mayúscula que se dio en la sección 1.4?
 - ¿Se cumplen con esta definición de omega mayúscula las afirmaciones a) y b) del ejercicio 1.14?
- 1.16** Ordénense las siguientes funciones de acuerdo con su velocidad de crecimiento: a) n , b) \sqrt{n} , c) $\log n$, d) $\log\log n$, e) $\log^2 n$, f) $n/\log n$, g) $\sqrt{n}\log^2 n$, h) $(1/3)^n$, i) $(3/2)^n$, j) 17.
- 1.17** Supóngase que el parámetro n del procedimiento siguiente es una potencia positiva de 2, esto es, $n = 2, 4, 8, 16, \dots$ Proporciónese una fórmula que exprese el valor de la variable *cuenta* en función del valor de n cuando termina el procedimiento.

```

procedure misterio (n: integer);
var
  x, cuenta: integer;
begin
  cuenta := 0;
  x := 2;
  while x < n do begin
    x := 2 * x;
    cuenta := cuenta + 1
  end;
  writeln (cuenta)
end

```

- 1.18 La siguiente es una función *máx* (*i*, *n*) que devuelve el mayor elemento en las posiciones *i* a *i+n-1* de una matriz entera *A*. Por conveniencia, puede suponerse que *n* es una potencia de 2.

```

function máx (i, n: integer) : integer;
var
    m1, m2: integer;
begin
    if n = 1 then
        return (A[i])
    else begin
        m1 := máx(i, n div 2);
        m2 := máx(i+n div 2, n div 2);
        if m1 < m2 then
            return (m2)
        else
            return (m1)
    end
end

```

- a) Sea $T(n)$ el tiempo del peor caso para *máx* con segundo argumento igual a *n*. Esto es, *n* es el número de elementos de entre los cuales se encuentra el mayor. Escribase una ecuación que exprese $T(n)$ en función de $T(j)$ para uno o más valores de *j* menores que *n* y una constante o constantes que representen los tiempos tomados por las proposiciones individuales del programa *máx*.
- b) Obténgase una cota asintótica superior (notación «o mayúscula») lo más cercana posible para $T(n)$. La solución debe ser igual a una cota asintótica inferior (notación «omega mayúscula»), y ser lo más sencilla posible.

Notas bibliográficas

El origen del concepto de tipo de datos abstracto se remonta al tipo *class* en el lenguaje SIMULA 67 (Birtwistle *et al.* [1973]). Desde entonces, se han desarrollado varios lenguajes que manejan tipos de datos abstractos; entre ellos están Alphard (Shaw, Wulf y London [1977]), C con clases (Stroustrup [1982]), CLU (Liskov *et al.* [1977]), MESA (Geschke, Morris y Satterthwaite [1977]) y Russell (Demers y Donahue [1979]). El concepto de TDA se analiza con más profundidad en trabajos tales como Gotlieb y Gotlieb [1978] y Wulf *et al.* [1981].

Knuth [1968] fue el primer trabajo importante en el que se preconizó el estudio sistemático del tiempo de ejecución de programas. Aho, Hopcroft y Ullman [1974] relaciona la complejidad espacial y temporal de los algoritmos con varios modelos computacionales, como las máquinas de Turing y las de acceso aleatorio. Para más referencias sobre el tema del análisis de algoritmos y programas, véanse también las notas bibliográficas del capítulo 9.

Como material adicional sobre programación estructurada véanse Hoare, Dahl y Dijkstra [1972], Wirth [1973], Kernighan y Plauger [1974], y Yourdon y Constantine [1975]. Los problemas organizacionales y psicológicos que se presentan en el desarrollo de proyectos grandes de software se tratan en Brooks [1974] y Weinberg [1971]. En Kernighan y Plauger [1981] se muestra cómo construir herramientas útiles para un ambiente de programación.

2

Tipos de datos abstractos fundamentales

En este capítulo se estudiarán algunos tipos de datos abstractos fundamentales. Se considerarán las listas, que son secuencias de elementos, y dos tipos especiales de listas: las pilas, donde los elementos se insertan y eliminan sólo en un extremo, y las colas, donde los elementos se insertan por un extremo y se eliminan por el otro. Se estudiará después, en forma breve, la correspondencia o memoria asociativa (*mapping*), un TDA que se comporta como una función. Para cada uno de estos TDA, se considerarán varias realizaciones y se analizarán sus méritos relativos.

2.1 El tipo de datos abstracto «lista»

Las listas constituyen una estructura flexible en particular, porque pueden crecer y acortarse según se requiera; los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista. Las listas también pueden concatenarse entre sí o dividirse en sublistas; se presentan de manera rutinaria en aplicaciones como recuperación de información, traducción de lenguajes de programación y simulación. Las técnicas de administración de memoria del tipo de las que se analizan en el capítulo 12 hacen uso extensivo de técnicas de procesamiento de listas. En esta sección se presentarán algunas operaciones básicas con listas, y en el resto del capítulo se presentarán estructuras de datos para representar listas que manejan con eficiencia varios subconjuntos de estas operaciones.

Matemáticamente, una *lista* es una secuencia de cero o más elementos de un tipo determinado (que por lo general se denominará *tipo_elemento*). A menudo se representa una lista como una sucesión de elementos separados por comas

$$a_1, a_2, \dots, a_n$$

donde $n \geq 0$ y cada a_i es del tipo *tipo_elemento*. Al número n de elementos se le llama *longitud* de la lista. Al suponer que $n \geq 1$, se dice que a_1 es el *primer elemento* y a_n el *último elemento*. Si $n = 0$, se tiene una *lista vacía*, es decir, que no tiene elementos.

Una propiedad importante de una lista es que sus elementos pueden estar ordenados en forma lineal de acuerdo con sus posiciones en la misma. Se dice que a_i *precede* a a_{i+1} para $i = 1, 2, \dots, n-1$, y que a_i *sigue* a a_{i-1} para $i = 2, 3, \dots, n$. Se dice que el elemento a_i está en la *posición i*. Es conveniente postular también la existencia de

una posición que sucede a la del último elemento de la lista. La función $\text{FIN}(L)$ devolverá la posición que sigue a la posición n en una lista L de n elementos. Obsérvese que la posición $\text{FIN}(L)$, con respecto al principio de la lista, está a una distancia que varía conforme la lista crece o se reduce, mientras que las demás posiciones guardan una distancia fija con respecto al principio de la lista.

Para formar un tipo de datos abstracto a partir de la noción matemática de lista, se debe definir un conjunto de operaciones con objetos de tipo LISTA †. Como sucede con muchos de los TDA que se analizan en este libro, ningún conjunto de operaciones es adecuado para todas las aplicaciones. Aquí se dará un conjunto representativo de operaciones. En la siguiente sección se mostrarán varias estructuras de datos para representar listas y se escribirán procedimientos para las operaciones características con listas en función de esas estructuras de datos.

Para ilustrar algunas operaciones comunes con listas, considérese una aplicación típica: se tiene una lista de direcciones y se desean eliminar las entradas dobles. En forma conceptual, este problema es bastante fácil de resolver: para cada elemento de la lista, eliminense todos los elementos sucesores equivalentes. Para presentar este algoritmo, sin embargo, es necesario definir operaciones que permitan encontrar el primer elemento de una lista, recorrer los demás elementos sucesivos, y recuperar y eliminar elementos.

Se presentará ahora un conjunto representativo de operaciones con listas. Ahí, L es una lista de objetos de tipo tipo_elemento, x es un objeto de ese tipo y p es de tipo posición. Obsérvese que «posición» es otro tipo de datos cuya implantación cambiará con aquella que se haya elegido para las listas. Aunque de manera informal se piensa en las posiciones como enteros, en la práctica pueden tener otra representación.

1. **INSERTA(x, p, L)**. Esta función inserta x en la posición p de la lista L , pasando los elementos de la posición p y siguientes a la posición inmediata posterior. Esto quiere decir que si L es a_1, a_2, \dots, a_n , se convierte en $a_1, a_2, \dots, a_{p-1}, x, a_p, \dots, a_n$. Si p es $\text{FIN}(L)$, entonces L se convierte en a_1, a_2, \dots, a_n, x . Si la lista L no tiene posición p , el resultado es indefinido.
2. **LOCALIZA(x, L)**. Esta función devuelve la posición de x en la lista L . Si x figura más de una vez en L , la posición de la primera aparición de x es la que se devuelve. Si x no figura en la lista, entonces se devuelve $\text{FIN}(L)$.
3. **RECUPERA(p, L)**. Esta función devuelve el elemento que está en la posición p de la lista L . El resultado no está definido si $p = \text{FIN}(L)$ o si L no tiene posición p . Obsérvese que si se utiliza RECUPERA, los elementos deben ser de un tipo que pueda ser devuelto por una función. No obstante, en la práctica siempre es posible modificar RECUPERA para devolver un apuntador a un objeto de tipo tipo_elemento.
4. **SUPRIME(p, L)**. Esta función elimina el elemento en la posición p de la lista L . Si L es a_1, a_2, \dots, a_n , L se convierte en $a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n$. El resultado no está definido si L no tiene posición p o si $p = \text{FIN}(L)$.

† En sentido estricto, el tipo es «LISTA de tipo_elemento». Sin embargo, las realizaciones de listas que se proponen no dependen del tipo_elemento; precisamente esta independencia es la que justifica la relevancia que se da al concepto de lista. Se usará «LISTA» en lugar de «LISTA de tipo_elemento», y se tratará de manera similar a los demás TDA que dependan de los tipos de elementos.

5. SIGUIENTE(p, L) y ANTERIOR (p, L) devuelven las posiciones siguiente y anterior, respectivamente, a p en la lista L . Si p es la última posición de L , SIGUIENTE(p, L)=FIN(L). SIGUIENTE no está definida si p es FIN(L). ANTERIOR no está definida si p es 1. Ambas funciones están indefinidas cuando L no tiene posición p .
6. ANULA(L). Esta función ocasiona que L se convierta en la lista vacía y devuelve la posición FIN(L).
7. PRIMERO(L). Esta función devuelve la primera posición de la lista L . Si L está vacía, la posición que se devuelve es FIN(L).
8. IMPRIME_LISTA(L). Imprime los elementos de L en su orden de aparición en la lista.

Ejemplo 2.1. Con estos operadores se escribirá un procedimiento PURGA que toma como argumento una lista y elimina sus elementos con doble entrada. Los elementos son de tipo tipo_elemento, y una lista de ellos es de tipo LISTA; esta convención se seguirá en todo el capítulo. Hay una función *mismo(x, y)*, donde x e y son de tipo tipo_elemento, que es verdadera si x e y son «el mismo» elemento, y es falsa en caso contrario. El significado de «ser el mismo» es intencionalmente impreciso. Si tipo_elemento es real, por ejemplo, podría desearse que *mismo(x, y)* fuera verdadera si, y sólo si, $x = y$. Sin embargo, si tipo_elemento es un registro que contiene el número de cuenta, el nombre y la dirección de un suscriptor como en

```
type
  tipo_elemento = record
    num_cta: integer;
    nombre: packed array [1..20] of char;
    dirección: packed array [1..50] of char
  end
```

por consiguiente podría desearse que *mismo(x, y)* fuera verdadera siempre que $x.\text{num_cta} = y.\text{num_cta}$ †.

La figura 2.1 muestra el código de PURGA. Las variables p y q se usan para representar dos posiciones en la lista. Conforme el programa avanza, se eliminan de la lista todas las entradas dobles de cualquier elemento a la izquierda de la posición p . En una iteración del ciclo (2)-(8), q se usa para revisar la lista a partir de la posición p , con el objeto de eliminar cualquier entrada doble del elemento en la posición p . Despues, p avanza a la siguiente posición y el proceso se repite.

En la siguiente sección, se proporcionarán las declaraciones apropiadas para LISTA y posición, y una aplicación de las operaciones, de manera que PURGA se convierta en un programa ejecutable. Como ya se dijo, el programa es independiente de la forma en que se representen las listas, por lo que hay libertad para experimentar con varias realizaciones de listas.

† En este caso, si se eliminan los registros que son «el mismo», podría desearse verificar que los nombres y direcciones fueran iguales; si los números de cuenta fueran iguales, pero los otros datos no, es posible que se haya asignado inadvertidamente el mismo número de cuenta a dos personas. No obstante, es más probable que el mismo suscriptor aparezca más de una vez en la lista, con diferentes números y con los demás datos ligeramente distintos. En tales casos, resulta difícil eliminar las entradas dobles.

```

procedure PURGA ( var L: LISTA );
{PURGA elimina los elementos duplicados de la lista L}
var
  p, q: posición; { p será la posición "actual" en L, y q
    avanzará para encontrar elementos iguales }
begin
  (1)   p := PRIMERO(L);
  (2)   while p <> FIN(L) do begin
  (3)     q := SIGUIENTE(p, L);
  (4)     while q <> FIN(L) do
  (5)       if mismo(RECUPERA(p, L), RECUPERA(q, L)) then
  (6)         SUPRIME(q, L)
  (7)       else
  (8)         q := SIGUIENTE(q, L);
  end
end; { PURGA }

```

Fig. 2.1. Programa para eliminar duplicados.

Un aspecto que es importante observar está relacionado con el cuerpo del ciclo interno, las líneas (4)-(7) de la figura 2.1. Cuando se elimina el elemento de la posición q de la línea (6), los elementos que estaban en las posiciones $q+1$, $q+2$, ..., etcétera, avanzan una posición en la lista. En particular, si q fuera la última posición de L , el valor de q sería entonces $\text{FIN}(L)$. Si después de esto se ejecutara la línea (7), $\text{SIGUIENTE}(\text{FIN}(L), L)$ produciría un resultado indefinido. Por tanto, es esencial que sólo alguna de las líneas, (6) o (7), pero no las dos, se ejecute entre las pruebas para saber si $q = \text{FIN}(L)$ en la línea (4). \square

2.2 Realización de listas

En esta sección se describirán algunas estructuras de datos que pueden utilizarse para representar listas. Se considerarán realizaciones de listas basadas en arreglos, apunadores y cursores. Cada una de ellas permite realizar ciertas operaciones con listas de manera más eficiente que otras.

Realización de listas mediante arreglos

En la realización de una lista mediante arreglos, los elementos de ésta se almacenan en celdas contiguas de un arreglo. Esta representación permite recorrer con facilidad una lista y agregarle elementos nuevos al final. Pero insertar un elemento en la mitad de la lista obliga a desplazarse una posición dentro del arreglo a todos los elementos que siguen al nuevo elemento para concederle espacio. De la misma for-

ma, la eliminación de un elemento, excepto el último, requiere desplazamientos de elementos para llenar de nuevo el vacío formado.

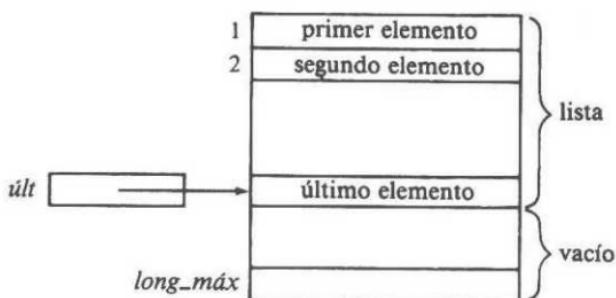


Fig. 2.2. Realización de una lista mediante un arreglo.

En la realización con arreglos se define el tipo LISTA como un registro con dos campos; el primero es un arreglo de elementos que tiene la longitud adecuada para contener la lista de mayor tamaño que se pueda presentar. El segundo campo es un entero *ult* que indica la posición del último elemento de la lista en el arreglo. El *i*-ésimo elemento de la lista está en la *i*-ésima celda del arreglo, para $1 \leq i \leq \text{ult}$, como se muestra en la figura 2.2. Las posiciones en la lista se representan mediante enteros; la *i*-ésima posición, mediante el entero *i*. La función FIN(*L*) sólo tiene que devolver *ult* + 1. Las declaraciones importantes son:

```

const
  long_max = 100 { alguna constante apropiada };
type
  LISTA = record
    elementos: array[1..long_max] of tipo_elemento;
    ult: integer;
  end;
  posición = integer;
function FIN (var L: LISTA): posición†;
begin
  return (L.ult + 1)
end; {FIN}

```

La figura 2.3 muestra cómo se podrían implantar las operaciones INSERTA, SUPRIME y LOCALIZA con esta realización basada en arreglos. INSERTA pasa los elementos de las localidades *p*, *p* + 1, ..., *ult* a las localidades *p* + 1, *p* + 2, ..., *ult* + 1 y después inserta el nuevo elemento en la localidad *p*. Si no hay espacio en el arreglo para insertar un nuevo elemento, se invoca a la rutina *error*, que imprime su argumento.

† Aunque *L* no se modifica, pasa por referencia, pues por lo común es una estructura grande y no se desea perder tiempo copiándola.

```

procedure INSERTA ( x: tipo_elemento; p: posición; var L: LISTA );
{ INSERTA coloca x en la posición p de la lista L }
var
  q: posición;
begin
  if L.últ > = long_máx then
    error ('la lista está llena')
  else if (p > L.últ + 1) or (p < 1) then
    error ('la posición no existe')
  else begin
    for q := L.últ downto p do
      { desplaza los elementos en p, p + 1,... una posición hacia abajo }
      L.elementos [q + 1] := L.elementos[q];
    L.últ := L.últ + 1;
    L.elementos[p] := x
  end
end; { INSERTA }

procedure SUPRIME ( p: posición; var L: LISTA );
{ SUPRIME elimina el elemento en la posición p de la lista L }
var
  q: posición;
begin
  if (p > L.últ) or (p < 1) then
    error ('la posición no existe')
  else begin
    L.últ := L.últ - 1;
    for q := p to L.últ do
      { desplaza los elementos en p + 1, p + 2... una posición hacia
        arriba }
      L.elementos[q] := L.elementos[q + 1]
  end
end; { SUPRIME }

function LOCALIZA ( x: tipo_elemento; var L: LISTA ) : posición;
{ LOCALIZA devuelve la posición de x en la lista L }
var
  q: posición;
begin
  for q := 1 to L.últ do
    if L.elementos[q] = x then
      return (q);
  return (L.últ + 1) { si no se encuentra }
end; { LOCALIZA }

```

Fig. 2.3. Realización basada en arreglos de algunas operaciones con listas.

to y da por terminada la ejecución del programa. SUPRIME elimina el elemento de la posición p pasando los elementos de las posiciones $p + 1, p + 2, \dots, \text{últ}$, a las posiciones $p, p + 1, \dots, \text{últ} - 1$. LOCALIZA revisa el arreglo secuencialmente, en busca de un elemento determinado; si no lo encuentra, LOCALIZA devuelve $\text{últ} + 1$.

Debería estar claro cómo codificar las restantes operaciones con listas con esta forma de implantación. Por ejemplo, PRIMERO siempre devuelve uno; SIGUIENTE devuelve uno más que su argumento, y ANTERIOR, uno menos; cada uno verifica primero si el resultado está dentro del intervalo permisible; ANULA(L) coloca $L.\text{últ}$ en 0. Si el procedimiento PURGA de la figura 2.1 va precedido de

1. las definiciones de tipo_elemento y de la función *mismo*.
2. las definiciones de LISTA, posición y FIN(L) anteriores,
3. la definición de SUPRIME de la figura 2.3, y
4. definiciones adecuadas para los procedimientos triviales PRIMERO, SIGUIENTE y RECUPERA,

entonces se obtiene un procedimiento PURGA ejecutable.

Al principio puede parecer tedioso escribir procedimientos que rijan todos los accesos a las estructuras subyacentes de un programa. Sin embargo, si se logra establecer la disciplina de escribir programas en función de operaciones de manipulación de tipos de datos abstractos en lugar de usar ciertos detalles de implantación particulares, es posible modificar los programas más fácilmente con sólo aplicar de nuevo las operaciones, en lugar de buscar en todos los programas aquellos lugares donde se hacen accesos a las estructuras de datos subyacentes. La flexibilidad que se obtiene con esto, puede ser especialmente importante en proyectos grandes, y no se debe poner en tela de juicio este concepto por los ejemplos, necesariamente pequeños, que se encuentran en este libro.

Realización de listas mediante apuntadores

La segunda forma de realización de listas, celdas enlazadas sencillas, utiliza apuntadores para enlazar elementos consecutivos. Esta implantación permite eludir el empleo de memoria contigua para almacenar una lista y, por tanto, también elude los desplazamientos de elementos para hacer inserciones o llenar vacíos creados por la eliminación de elementos. No obstante, por esto hay que pagar el precio de un espacio adicional para los apuntadores.

En esta representación, una lista está formada por celdas; cada celda contiene un elemento de la lista y un apuntador a la siguiente celda. Si la lista es a_1, a_2, \dots, a_n , la celda que contiene a_i tiene un apuntador a la celda que contiene a a_{i+1} , para $i = 1, 2, \dots, n - 1$. La celda que contiene a_n posee un apuntador **nil**. Existe también una celda de *encabezamiento* que apunta a la celda que contiene a_1 ; esta celda de encabezamiento no tiene ningún elemento†. En el caso de una lista vacía, el apuntador del encabe-

† El empleo de una celda completa para el encabezamiento simplifica la implantación de operaciones con listas en Pascal. Es posible usar apuntadores como encabezamientos si se pretende aplicar operaciones de modo que las inserciones y las eliminaciones al principio de la lista se manejen de manera especial. Véase el análisis de este método en esta misma sección, en el apartado que explica la realización de listas mediante cursores.

zamiento es **nil** y no se tienen más celdas. La figura 2.4 muestra una lista enlazada de esta manera.

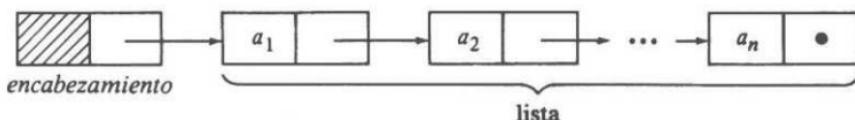


Fig. 2.4. Lista enlazada.

Para las listas enlazadas sencillas es conveniente usar una definición de posición ligeramente distinta de la que se empleó para las listas logradas mediante arreglos. Aquí, la posición i será un apuntador a la celda que contiene el apuntador a a_i para $i = 2, 3, \dots, n$. La posición 1 es un apuntador al encabezamiento, y la posición $\text{FIN}(L)$ es un apuntador a la última celda de L .

El tipo de una lista, en realidad, es el mismo que el de una posición; es un apuntador a una celda particular: el encabezamiento. La siguiente es una definición formal de las partes esenciales de una estructura de datos de lista enlazada.

```
type
    tipo_celda = record
        elemento: tipo_elemento;
        sig: ↑ tipo_celda
    end;
    LISTA = ↑ tipo_celda;
    posición = ↑ tipo_celda;
```

En la figura 2.5 se muestra la función $\text{FIN}(L)$; ésta funciona moviendo el apuntador q por la lista, hasta que alcanza el final, el cual se detecta por el hecho de que q apunta a una celda con un apuntador **nil**. Obsérvese que esta realización de FIN es ineficiente, pues requiere revisar la lista entera cada vez que se desea calcular $\text{FIN}(L)$. Si es necesario hacer esto con frecuencia, como en el programa PURGA de la figura 2.1, puede optarse por cualquiera de las opciones siguientes:

```
function FIN ( L: LISTA ) : posición;
    { FIN devuelve un apuntador a la última celda de  $L$  }
    var
        q: posición;
    begin
        (1)      q := L;
        (2)      while q↑.sig <> nil do
        (3)          q := q↑.sig;
        (4)      return (q)
    end; { FIN }
```

Fig. 2.5. La función FIN.

1. usar una representación de listas que incluya un apuntador de la última celda, o
2. sustituir el uso de $\text{FIN}(L)$ donde sea posible. Por ejemplo, la condición $p <> \text{FIN}(L)$ de la línea (2) de la figura 2.1 podría reemplazarse por $p \uparrow. \text{sig} <> \text{nil}$, al costo de que ese programa dependa de una realización particular de listas.

La figura 2.6 contiene rutinas para las operaciones INSERTA, SUPRIME, LOCALIZA y ANULA usando esta realización de listas con apuntadores. Las operaciones restantes se pueden obtener como rutinas de un solo paso, con la excepción de ANTERIOR, que requiere revisar la lista desde el principio. Estas rutinas se dejan como ejercicio para el lector. Obsérvese que muchos de los mandatos no necesitan el parámetro L , la lista, por lo que se omite.

En la figura 2.7 se muestra la mecánica del manejo de apuntadores en el procedimiento INSERTA de la figura 2.6. La figura 2.7 (a) muestra la situación antes de la ejecución de INSERTA. Se desea insertar un nuevo elemento ante la celda que contiene b , por lo que p es un apuntador a la celda de la lista que contiene el apuntador a b . En la línea (1), temp apunta a la celda que contiene b . En la línea (2), se crea una nueva celda de la lista y se hace que el campo sig de la celda que contiene a apunte hacia esta misma celda. En la línea (3) se almacena x en el campo elemento de la celda recién creada, y en la línea (4), el campo sig toma el valor de temp y, de esta manera, apunta a la celda que contiene b . La figura 2.7 (b) muestra el resultado de la ejecución de INSERTA. Los apuntadores nuevos se representan con líneas punteadas y se marca el paso en el cual fueron creados.

El procedimiento SUPRIME es más sencillo. La figura 2.8 muestra las manipulaciones de apuntadores que realiza el procedimiento SUPRIME de la figura 2.6. Los apuntadores antiguos se representan por medio de líneas de trazo continuo, y los nuevos por medio de líneas punteadas.

Obsérvese que una posición en una implantación de listas enlazadas se comporta de distinta forma que una posición en una realización de arreglos. Supóngase que se tiene una lista con tres elementos a, b, c y una variable p de tipo posición que tiene actualmente el valor 3; es decir, apunta a la celda que contiene b y, por tanto, representa la posición de c . Si se ejecuta un mandato para insertar x en la posición 2, la lista se convierte en a, x, b, c , con lo cual el elemento b pasaría a ocupar la posición 3. Si se utilizara la realización de arreglos descrita anteriormente, b y c se moverían hacia el principio del arreglo, de modo que b ocuparía en realidad la tercera posición.

```

procedure INSERTA (  $x$ : tipo_elemento;  $p$ : posición );
var
    temp: posición;
begin
(1)    temp :=  $p \uparrow. \text{sig}$ ;
(2)    new( $p \uparrow. \text{sig}$ );
(3)     $p \uparrow. \text{sig} \uparrow. \text{elemento} := x$ ;
(4)     $p \uparrow. \text{sig} \uparrow. \text{sig} := \text{temp}$ 
end; { INSERTA }

```

```

procedure SUPRIME ( p: posición );
begin
   $p \uparrow .sig := p \uparrow .sig \uparrow .sig$ 
end; { SUPRIME }

function LOCALIZA ( x: tipo_elemento; L: LISTA ) : posición;
var
  p: posición;
begin
   $p := L$ ;
  while  $p \uparrow .sig <> \text{nil}$  do
    if  $p \uparrow .sig \uparrow .elemento = x$  then
      return (p)
    else
       $p := p \uparrow .sig$ ;
  return (p) { si no es encontrado }
end; { LOCALIZA }

function ANULA ( var L: LISTA ) : posición;
begin
  new (L);
   $L \uparrow .next := \text{nil}$ ;
  return(L)
end; { ANULA }

```

Fig. 2.6. Algunas operaciones con la realización de listas enlazadas.

Pero, si se usara la realización de listas enlazadas, el valor de p , que es un apuntador a la celda que contiene b , no cambiaría como consecuencia de la inserción, de modo que después de ésta el valor de p sería la «posición 4», no la 3. Esta variable de posición debe actualizarse si se va a usar después como posición de $b \uparrow$.

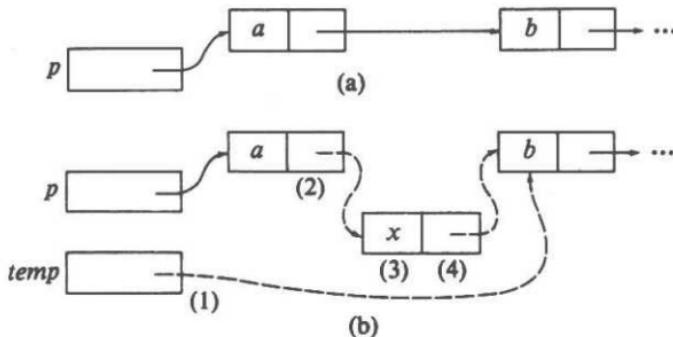


Fig. 2.7. Diagrama de INSERTA.

† Por supuesto, existen muchas situaciones en las que se podría desear que p represente la posición de c .

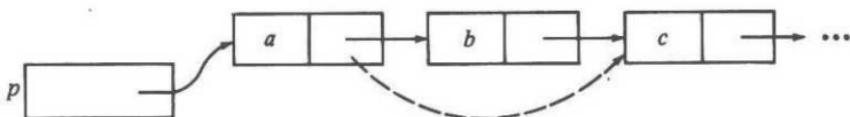


Fig. 2.8. Diagrama de SUPRIME.

Comparación de los métodos

Cabe preguntarse si en determinadas circunstancias es mejor usar una realización de listas basada en apuntadores o una basada en arreglos. A menudo, la respuesta depende de cuáles son las operaciones que se desea realizar, o las que se realizan más frecuentemente. Otras veces, la decisión depende de lo larga que puede llegar a ser la lista. Los aspectos más importantes a tener en cuenta son los siguientes.

1. La realización con arreglos requiere especificar el tamaño máximo de una lista en el momento de la compilación. Si no es posible acotar la probable longitud de la lista, quizás deba optarse por una realización con apuntadores.
2. Ciertas operaciones son más lentas en una realización que en otra. Por ejemplo, INSERTA y SUPRIME tienen un número constante de pasos para una lista enlazada, pero cuando se utiliza la realización con arreglos, requieren un tiempo proporcional al número de elementos que siguen. A la inversa, la ejecución de ANTERIOR y FIN requiere un tiempo constante con la realización con arreglos, pero se usa un tiempo proporcional a la longitud de la lista si se usan apuntadores.
3. Si un programa efectúa inserciones o supresiones que afecten al elemento que ocupa la posición denotada por alguna variable de posición, y el valor de esa variable se va a utilizar posteriormente, entonces la representación con apuntadores no se puede usar de la forma descrita aquí. Como principio general, los apuntadores se deben utilizar con gran cuidado y moderación.
4. La realización con arreglos puede malgastar espacio, puesto que usa la cantidad máxima de espacio, independientemente del número de elementos que en realidad tiene la lista en un momento dado. La realización con apuntadores utiliza sólo el espacio necesario para los elementos que actualmente tiene la lista, pero requiere espacio para el apuntador en cada celda. Así, cualquiera de los dos métodos podría usar más espacio que el otro, dependiendo de las circunstancias.

Realización de listas basadas en cursores

Algunos lenguajes, como FORTRAN y ALGOL, no tienen apuntadores. Si se trabaja con un lenguaje tal, se pueden simular los apuntadores mediante cursores; esto es, con enteros que indican posiciones en arreglos. Se crea un arreglo de registros para

almacenar todas las listas de elementos cuyo tipo es tipo_elemento; cada registro contiene un elemento y un entero que se usa como cursor. Es decir, se define

```

var
  ESPACIO: array [1..long_máx] of record
    elemento: tipo_elemento;
    sig: integer
  end

```

Si L es una lista de elementos, se declara una variable entera, por ejemplo, *Lencab*, como encabezamiento para L . Se puede tratar *Lencab* como un cursor a la celda de encabezamiento en *ESPACIO* con un campo *elemento* vacío. Las operaciones con listas se pueden implantar, entonces, como en la realización basada en apuntadores recién descrita.

Aquí se describirá una realización alternativa que evita el uso de celdas de encabezamiento, al tomar como casos especiales las inserciones y supresiones en la posición 1. Esta técnica se puede usar también con listas enlazadas basadas en apuntadores, para evitar el empleo de celdas de encabezamiento. Para una lista L , el valor de *ESPACIO* [*Lencab*].*elemento* es el valor del primer elemento de L . El valor de *ESPACIO* [*Lencab*].*sig* es el índice de la celda que contiene el segundo elemento, y así sucesivamente. Un valor de 0 ya sea en *Lencab* o en el campo *sig*, representa un «apuntador nil», esto es, no hay un elemento siguiente.

Una lista tendrá tipo entero, ya que el encabezamiento es una variable entera que representa la lista como un todo. Las posiciones serán también de tipo entero. Se adopta aquí la convención de que la posición i de la lista L es el índice de la celda de *ESPACIO* que contiene el elemento $i - 1$ de L , ya que el campo *sig* de esa celda contendrá el cursor al elemento i . Como caso especial, la posición 1 de cualquier lista se representa por 0. Dado que el nombre de la lista es siempre un parámetro de las operaciones que usan posiciones, es posible distinguir entre las primeras posiciones de distintas listas. La posición *FIN(L)* es el índice del último elemento de L .

La figura 2.9 muestra dos listas, $L = a, b, c$ y $M = d, e$, que comparten el arreglo *ESPACIO* con *long_máx* = 10. Obsérvese que las celdas del arreglo no contenidas

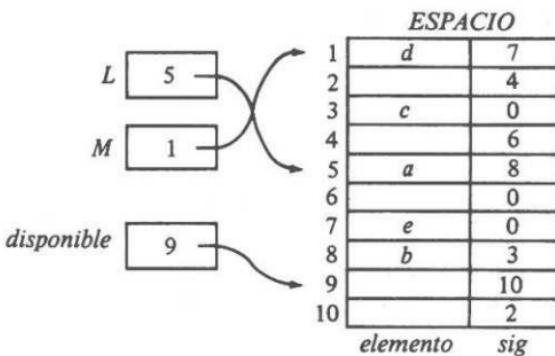


Fig. 2.9. Realización de listas enlazadas mediante cursores.

en L ni en M están enlazadas en otra lista llamada *disponible*. Tal lista es necesaria para obtener una celda vacía cuando se deseé hacer una inserción, y disponer de lugar donde poner las celdas suprimidas para su uso posterior.

Para insertar un elemento x en la lista L , se toma la primera celda de la lista *disponible* y se coloca en la posición correcta en la lista L . El elemento x se pone entonces en el campo *elemento* de esta celda. Para suprimir un elemento x de la lista L , se quita de L la celda que contiene x y se devuelve al principio de la lista *disponible*. Estas dos acciones pueden verse como casos especiales del acto de tomar una celda C apuntada por un cursor p y hacer que otro cursor q apunte a C , a la vez que se hace que p quede apuntando hacia donde C apuntaba y C quede apuntando hacia donde q apuntaba. De esta forma, C se inserta entre q y aquello hacia lo que apuntaba q . Por ejemplo, si se suprime b de la lista L en la figura 2.9, C es la fila 8 de *ESPACIO*, p es *ESPACIO[5].sig* y q es *disponible*. Los cursores antes (en líneas de trazo continuo) y después (en líneas punteadas) de esta acción se muestran en la figura 2.10, y el código está incluido en la función *mueve* de la figura 2.11, que realiza el movimiento si C existe y devuelve falso si C no existe.

La figura 2.12 muestra los procedimientos *INSERTA* y *SUPRIME*, y un procedimiento *val_inicial* que enlaza las celdas del arreglo *ESPACIO* para formar una lista de espacio disponible. Estos procedimientos no incluyen verificaciones para detectar errores; es aconsejable insertarlas como ejercicio para el lector. Se dejan también como ejercicios otras operaciones similares a las de las listas enlazadas basadas en apuntadores.

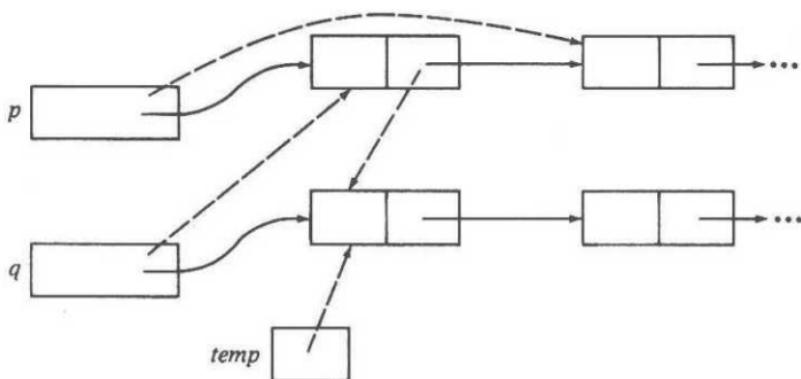


Fig. 2.10. Cambio de una celda C de una lista a otra.

```

function mueve (var  $p, q$ : integer) : boolean;
{mueve coloca la celda apuntada por  $p$  delante de la celda apuntada por  $q$ }
var
    temp: integer;
begin

```

```

if p = 0 then begin { celda inexistente }
    writeln('la celda no existe');
    return (false)
end
else begin
    temp := q;
    q := p;
    p := ESPACIO[q].sig;
    ESPACIO[q].sig := temp;
    return (true)
end
end; { mueve }

```

Fig. 2.11. Código para cambiar una celda.

Listas doblemente enlazadas

En algunas aplicaciones puede ser deseable recorrer eficientemente una lista, tanto hacia adelante como hacia atrás. O, dado un elemento, podría desearse determinar con rapidez el siguiente y el anterior. En tales situaciones, quizás se quisiera poner en cada celda de una lista un apuntador a la siguiente celda y otro a la anterior, como se sugiere en la lista doblemente enlazada de la figura 2.13. En el capítulo 12 se mencionan algunas situaciones específicas en que las listas doblemente enlazadas son esenciales por razones de eficiencia.

```

procedure INSERTA ( x: tipo_elemento; p: posición; var L: LISTA );
begin
    if p = 0 then begin
        { inserta en la primera posición }
        if mueve ( disponible, L ) then
            ESPACIO [L]. elemento := x
    end
    else { inserta en una posición que no es la primera }
        if mueve ( disponible, ESPACIO [p].sig ) then
            { celda que ocupará x ya apuntada por ESPACIO [p].sig }
            ESPACIO [ESPACIO [p].sig]. elemento := x
end; { INSERTA }

procedure SUPRIME ( p: posición; var L: LISTA );
begin
    if p = 0 then
        mueve (L, disponible)
    else
        mueve (ESPACIO [p].sig, disponible)
end; { SUPRIME }

```

```

procedure val_inicial;
{ val_inicial enlaza ESPACIO en una lista de celdas disponibles }
var
  i: integer;
begin
  for i := long_máx-1 downto 1 do
    ESPACIO [i].sig := i + 1;
  disponible := 1;
  ESPACIO [long_máx].sig := 0 { marca el final de la lista de celdas disponibles}
end; { val_inicial }

```

Fig. 2.12. Algunos procedimientos para listas enlazadas basadas en cursores.



Fig. 2.13. Una lista doblemente enlazada.

Otra ventaja importante de las listas doblemente enlazadas es que permiten usar un apuntador a la celda que contiene el i -ésimo elemento de una lista para representar la posición i , en vez de usar el apuntador a la celda anterior, que es menos natural. El único precio que se paga por estas características es la presencia de un apuntador adicional en cada celda, y los procedimientos algo más lentos para algunas de las operaciones básicas con listas. Si se usan apuntadores (en vez de cursores) se puede declarar que las celdas contienen un elemento y dos apuntadores, mediante

```

type
  tipo_celda = record
    elemento: tipo_elemento;
    sig, ant: ↑ tipo_celda
  end;
  posición = ↑ tipo_celda;

```

En la figura 2.14 se da un procedimiento para suprimir un elemento en la posición p en una lista doblemente enlazada. La figura 2.15 muestra los cambios causados en los apuntadores por este procedimiento; los apuntadores anteriores se representan con líneas de trazo continuo, y los nuevos con líneas punteadas, en el supuesto de que la celda suprimida no es la primera ni la última †. Primero se localiza la celda precedente, usando el campo ant . Se hace que el campo sig de esta celda apunte a

† A tal efecto, es práctica común hacer que el encabezamiento de una lista doblemente enlazada sea una celda que efectivamente «cierra el círculo». Esto es, que el campo ant del encabezado apunte a la última celda y que su campo sig apunte a la primera. De esta manera no se necesita hacer verificaciones de campos nil en la figura 2.14.

la celda que sigue a la que ocupa la posición p . Después se hace que el campo ant de esta celda siguiente apunte a la celda que precede a la que ocupa la posición p . La celda apuntada por p queda sin usar y el sistema de tiempo de ejecución de Pascal debe de usarla de nuevo automáticamente.

```

procedure SUPRIME (var p: posición );
begin
  if  $p \uparrow .ant <> \text{nil}$  then
    { la celda a suprimir no es la primera }
     $p \uparrow .ant \uparrow .sig := p \uparrow .sig;$ 
  if  $p \uparrow .sig <> \text{nil}$  then
    { la celda a suprimir no es la última}
     $p \uparrow .sig \uparrow .ant := p \uparrow .ant$ 
end; { SUPRIME }
```

Fig. 2.14. Supresión en una lista doblemente enlazada.

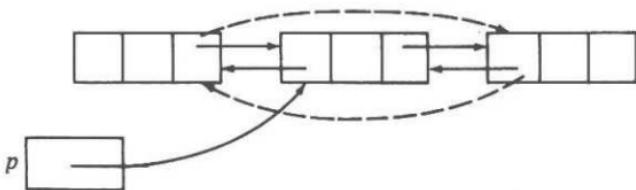


Fig. 2.15. Cambios de los apuntadores para realizar una supresión.

2.3 Pilas

Una *pila* es un tipo especial de lista en la que todas las inserciones y supresiones tienen lugar en un extremo denominado *tope*. A las pilas se les llama también «listas LIFO» (*last in first out*) o listas «último en entrar, primero en salir». El modelo intuitivo de una pila es precisamente una pila de fichas de póquer puesta sobre una mesa, o de libros sobre el piso, o de platos en una estantería, situaciones todas en las que sólo es conveniente quitar o agregar un objeto del extremo superior de la pila, al que se denominará en lo sucesivo «*tope*». Un tipo de datos abstracto de la familia PILA incluye a menudo las cinco operaciones siguientes:

1. ANULA(P) convierte la pila P en una pila vacía. Esta operación es exactamente la misma que para las listas generales.
2. TOPE(P) devuelve el valor del elemento de la parte superior de la pila P . Si se identifica la parte superior de una pila con la posición 1, como suele hacerse, entonces TOPE(P) puede escribirse en función de operaciones con listas como RECUPERA(PRIMERO(P), P).
3. SACA(P), en inglés *POP*, suprime el elemento superior de la pila, es decir, equivale a SUPRIME(PRIMERO(P), P). Algunas veces resulta conveniente implantar

SACA como una función que devuelve el elemento que acaba de suprimir, aunque aquí no se hará eso.

4. METE(x, P), en inglés *PUSH*, inserta el elemento x en la parte superior de la pila P . El anterior tope se convierte en el siguiente elemento, y así sucesivamente. En función de operaciones primitivas con listas, esta operación es INSERTA($x, \text{PRIMERO}(P), P$).
5. VACIA(P) devuelve verdadero si la pila P está vacía, y falso en caso contrario.

Ejemplo 2.2. Los editores de textos siempre permiten usar un carácter (por ejemplo, *back-space*) como *carácter de borrado* que cancela el carácter anterior no cancelado. Por ejemplo, si '#' es el carácter de borrado, la cadena *abc#d##e* es en realidad la cadena *ae*. El primer '#' cancela la *c*, el segundo la *d* y el tercero la *b*.

Los editores de textos también tienen un *carácter de eliminación de línea*, que cancela todos los caracteres anteriores de la línea actual. A efectos de este ejemplo, se usará '@' como carácter de eliminación de línea.

Un editor puede procesar una línea de texto usando una pila. El editor lee un carácter a la vez. Si el carácter leído no es de borrado ni de eliminación de línea, el editor lo mete en la pila. Si el carácter es de borrado, el editor saca un carácter de la pila, y si es de eliminación de línea, vacía la pila. En la figura 2.16 se muestra un programa que ejecuta estas acciones.

```

procedure EDITA;
var
  P: PILA;
  c: char;
begin
  ANULA (P);
  while not eoln do begin
    read(c);
    if c = '#' then
      SACA (P)
    else if c = '@' then
      ANULA (P)
    else { c es un carácter normal }
      METE (c, P)
  end;
  imprime P en orden inverso
end; { EDITA }
```

Fig. 2.16. Programa que logra el efecto de los caracteres de borrado y cancelación de línea.

En este programa, el tipo PILA debe declararse como una lista de caracteres. El proceso de escribir la pila en orden inverso en la última línea del programa tiene un objetivo: sacar de la pila un carácter a la vez da como resultado la inversión de la

línea. Algunas realizaciones de pilas, como la basada en arreglos, que se analizará a continuación, permiten escribir un procedimiento sencillo para imprimir los caracteres de la pila a partir de la base. Sin embargo, en general, para invertir una pila, cada elemento debe sacarse y meterse en otra pila; luego, los elementos pueden sacarse de la segunda pila e imprimirse en el orden en que se sacan. □

Realización de pilas basada en arreglos

Todas las realizaciones de listas descritas sirven para pilas, puesto que una pila con sus operaciones es un caso especial de lista con sus operaciones. La representación de una pila como lista enlazada es sencilla, pues METE y SACA operan sólo con la celda de encabezamiento y con la primera celda de la lista. De hecho, los encabezamientos pueden ser apuntadores o cursores, más que celdas completas, puesto que para las pilas no existe la noción de «posición» para las listas y, por tanto, no es necesario representar la posición 1 en forma análoga a las demás posiciones.

Sin embargo, la realización de listas basada en arreglos que se dio en la sección 2.3 no es particularmente adecuada para las pilas, dado que cada METE o SACA tiene que mover la lista entera hacia arriba o hacia abajo, lo cual lleva un tiempo proporcional al número de elementos de la pila. Un mejor criterio para usar un arreglo es el que tiene en cuenta el hecho de que las inserciones y las supresiones ocurren sólo en la parte superior. Se puede anclar la base de la pila a la base del arreglo (el extremo de índice más alto) y dejar que la pila crezca hacia la parte superior del arreglo (el extremo de índice más bajo). Un cursor llamado *tope* indicará la posición actual del primer elemento de la pila. Esta idea se ilustra en la figura 2.17.

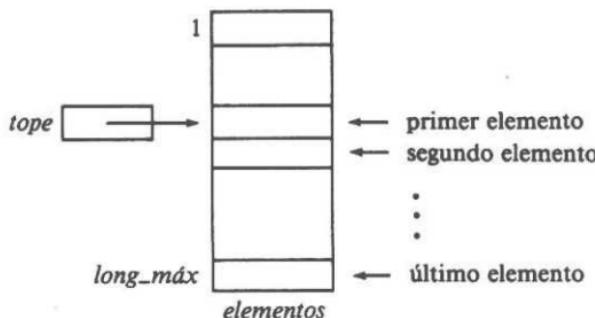


Fig. 2.17. Realización de una pila mediante un arreglo.

Para esta realización de pilas basada en arreglos, el tipo de datos abstracto PILA se define como

```

type
  PILA = record
    tope: integer;
    elementos: array[1..long_máx] of tipo_elemento
  end;

```

Un ejemplo de la pila consiste en la sucesión *elementos[*tope*]*, *elementos[*tope* + 1]*, ..., *elementos[long_máx]*. Obsérvese que si *tope = long_máx + 1*, la pila está vacía.

Las cinco operaciones típicas con pilas están realizadas en la figura 2.18. Obsérvese que para que TOPE devuelva un objeto de tipo tipo_elemento, ese tipo debe ser lícito como resultado de una función. De lo contrario, TOPE debe ser un procedimiento que modifique su segundo argumento asignándole el valor TOPE(*P*) o una función que devuelva un apuntador a tipo_elemento.

```

procedure ANULA ( var P: PILA );
begin
  P.tope := long_máx + 1
end; { ANULA }

function VACIA ( P: PILA ): boolean;
begin
  if P.tope > long_máx then
    return (true)
  else
    return (false)
end; { VACIA }

function TOPE ( var P: PILA ): tipo_elemento;
begin
  if VACIA (P) then
    error ('la pila está vacía')
  else
    return (P.elementos[P.tope])
end; { TOPE }

procedure SACA ( var P: PILA );
begin
  if VACIA(P) then
    error ('la pila está vacía')
  else
    P.tope := P.tope + 1
end; { SACA }

procedure METE ( x: tipo_elemento; P: PILA );
begin
  if P.tope := 1 then
    error ('la pila está llena')
  else begin
    P.tope := P.tope - 1;
    P.elementos[P.tope] := x
  end
end; { METE }

```

Fig. 2.18. Operaciones con pilas.

2.4 Colas .

Una *cola* es otro tipo especial de lista en el cual los elementos se insertan en un extremo (el *posterior*) y se suprimen en el otro (el *anterior* o *frente*). Las colas se conocen también como listas «FIFO» (*first-in first-out*) o listas «primero en entrar, primero en salir». Las operaciones para una cola son análogas a las de las pilas; las diferencias sustanciales consisten en que las inserciones se hacen al final de la lista, y no al principio, y en que la terminología tradicional para colas y listas no es la misma. Se usarán las siguientes operaciones con colas.

1. ANULA(*C*) convierte la cola *C* en una lista vacía.
2. FRENT(*C*) es una función que devuelve el valor del primer elemento de la cola *C*. FRENT(*C*) se puede escribir en función de operaciones con listas, como RECUPERA(PRIMERO(*C*), *C*).
3. PONE_EN_COLA(*x*, *C*) inserta el elemento *x* al final de la cola *C*. En función de operaciones con listas, PONE_EN_COLA(*x*, *C*) es INSERTA(*x*, FIN(*C*), *C*).
4. QUITA_DE_COLA(*C*) suprime el primer elemento de *C*; es decir, QUITA_DE_COLA(*C*) es SUPRIME(PRIMERO(*C*), *C*).
5. VACIA(*C*) devuelve verdadero si, y sólo si, *C* es una cola vacía.

Realización de colas basada en apuntadores

Igual que en el caso de las pilas, cualquier realización de listas es lícita para las colas. No obstante, para aumentar la eficacia de PONE_EN_COLA es posible aprovechar el hecho de que las inserciones se efectúan sólo en el extremo posterior. En lugar de recorrer la lista de principio a fin cada vez que se desea hacer una inserción, se puede mantener un apuntador (o un cursor) al último elemento. Como en las listas de cualquier clase, también se mantiene un apuntador al frente de la lista; en las colas, ese apuntador es útil para ejecutar mandatos del tipo FRENT o QUITA_DE_COLA. En Pascal, se utilizará una celda ficticia como encabezamiento y se tendrá el apuntador frontal dirigido a ella. Esta convención permite manejar convenientemente una cola vacía.

Aquí se desarrollará una implantación de colas basada en apuntadores de Pascal. Se puede desarrollar una realización análoga basada en cursores, pero en el caso de las colas se dispone de una representación orientada a arreglos, mejor que la que se puede obtener por imitación directa de apuntadores mediante cursores. Al final de esta sección se analizará otra implantación, llamada de «arreglos circulares». Para la realización basada en apuntadores, se definirán las celdas como antes:

```

type
  tipo_celda = record
    tipo_elemento: tipo_elemento;
    sig: ^ tipo_celda
  end;

```

Por tanto, es posible definir una cola como una estructura que consiste en apuntadores al extremo anterior de la lista y al extremo posterior. La primera celda de una cola es una celda de encabezamiento cuyo campo elemento se ignora. Como se mencionó, esta convención permite representar de manera simple una cola vacía. Se define:

```

type
  COLA = record
    ant, post: ↑ tipo_celda
  end;

```

La figura 2.19 muestra programas para las cinco operaciones con colas. En ANULA, la primera proposición *new(C.ant)* crea una variable de tipo tipo_celda y asigna su dirección a *C.ant*. La segunda proposición coloca nil en el campo *sig* de esa celda. La tercera proposición hace que el encabezamiento quede como primera y última celda de la cola.

El procedimiento QUITA_DE_COLA(*C*) suprime el primer elemento de *C* desconectando el encabezado antiguo de la cola. El primer elemento de la lista se convierte en la nueva celda ficticia de encabezamiento.

La figura 2.20 muestra los resultados originados por la sucesión de mandatos ANULA(*C*), PONE_EN_COLA(*x, C*), PONE_EN_COLA(*y, C*), QUITA_DE_COLA(*C*). Obsérvese que después de la supresión, el elemento *x* ya no se considera parte de la cola, por estar en el campo elemento de la celda de encabezamiento.

Realización de colas con arreglos circulares

La representación de listas por medio de arreglos analizada en la sección 2.2 puede usarse para las colas, pero no es muy eficiente. Es cierto que con un apuntador al último elemento es posible ejecutar PONE_EN_COLA en un número fijo de pasos, pero QUITA_DE_COLA, que suprime el primer elemento, requiere que la cola complete ascienda una posición en el arreglo. Así pues, QUITA_DE_COLA lleva un tiempo $\Omega(n)$ si la cola tiene longitud *n*.

Para evitar este gasto, se debe adoptar un punto de vista diferente. Imagínese un arreglo como un círculo en el que la primera posición sigue a la última, en la forma sugerida en la figura 2.21. La cola se encuentra en alguna parte de ese círculo, ocupando posiciones consecutivas †, con el extremo posterior en algún lugar a la izquierda del extremo anterior. Para insertar un elemento en la cola, se mueve el apuntador *C.post* una posición en el sentido de las manecillas del reloj, y se escribe el elemento en esa posición. Para suprimir, simplemente se mueve *C.ant* una posición en el sentido de las manecillas del reloj. De esta manera, la cola se mueve en ese mismo sentido conforme se insertan y suprimen elementos. Obsérvese que utilizando

† Obsérvese que la palabra «consecutivas» debe interpretarse en un sentido circular. Esto es, una cola de longitud cuatro puede ocupar, por ejemplo, las dos últimas y las dos primeras posiciones del arreglo.

este modelo los procedimientos PONE_EN_COLA y QUITA_DE_COLA se pueden escribir de tal manera que su ejecución se realice en un número constante de pasos.

Hay una sutileza que surge en la representación de la figura 2.21 y en cualquier variante menor de esta estrategia (es decir, si *C.post* apunta a una posición adelantada con respecto al último elemento, en el sentido de las manecillas del reloj, y no a ese mismo elemento). El problema reside en que no hay manera de distinguir entre una cola vacía y una que ocupa el círculo completo, a menos que se mantenga un bit que sea verdadero si, y sólo si, la cola está vacía. Si no se está dispuesto a mantener ese bit, se debe evitar que la cola llegue a llenar todo el arreglo.

Para comprender la razón de esto, supóngase que la cola de la figura 2.21 tiene *long_máx* elementos. Entonces *C.post* debería apuntar una posición adelante de *C.ant* en el sentido contrario al de las manecillas del reloj. ¿Y qué ocurriría si la cola estuviera vacía? Para ver cómo se representa una cola vacía, considérese primero una que tenga un solo elemento. Entonces *C.ant* y *C.post* apuntan a la misma posición. Si en estas condiciones se suprime ese elemento, *C.ant* avanza una posición en el sentido de las manecillas del reloj, formando una cola vacía. Así, en una cola vacía, *C.post* está a una posición de *C.ant* en sentido contrario al de las manecillas del reloj; es decir, está exactamente en la misma posición relativa que ocuparía si la cola tuviera *long_máx* elementos. Resulta evidente, pues, que aun cuando el arreglo tenga *long_máx* lugares, no se puede permitir que la cola crezca más que *long_máx*-1, a menos que se introduzca un mecanismo para distinguir las colas vacías.

```

procedure ANULA ( var C: COLA );
begin
    new(C.ant); { crea la celda de encabezamiento }
    C.ant^.sig := nil;
    C.post := C.ant { el encabezamiento es la primera y la última celdas }
end; { ANULA }

function VACIA ( C: COLA ): boolean;
begin
    if C.ant = C. post then
        return (true)
    else
        return (false)
end; { VACIA }

function FRENTA ( C: COLA ): tipo_elemento;
begin
    if VACIA(C) then
        error('la cola está vacía')
    else
        return ( C.ant^.sig^.elemento )
end; { FRENTA }

```

```

procedure PONE_EN_COLA ( x: tipo_elemento; var C: COLA );
begin
  new ( C.post↑.sig ); { agrega una nueva celda en el extremo
                        posterior de la cola }
  C.post := C.post↑.sig;
  C.post↑.elemento := x;
  C.post↑.sig := nil
end; { PONE_EN_COLA }

procedure QUITA_DE_COLA ( var C: COLA );
begin
  if VACIA(C) then
    error ('la cola está vacía')
  else
    C.ant := C.ant↑.sig
end; { QUITA_DE_COLA }

```

Fig. 2.19. Implementación de mandatos para colas.

ANULA (C)

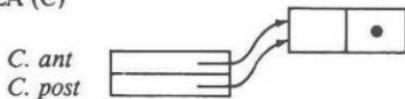
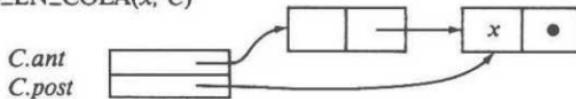
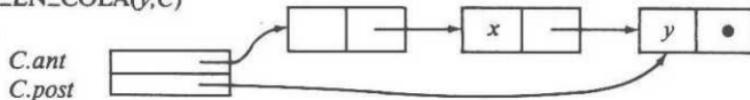
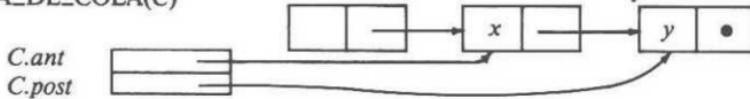
PONE_EN_COLA(*x*, *C*)PONE_EN_COLA(*y*, *C*)QUITA_DE_COLA(*C*)

Fig. 2.20. Sucesión de operaciones con colas.

Se escribirán ahora los cinco mandatos para colas usando esta representación. Formalmente, las colas se definen:

```
type
  COLA = record
    elementos: array[1..long_máx] of tipo_elemento;
    ant, post: integer
  end;
```

Los mandatos aparecen en la figura 2.22. La función *suma_uno(i)* suma uno a la posición *i*, en el sentido circular.

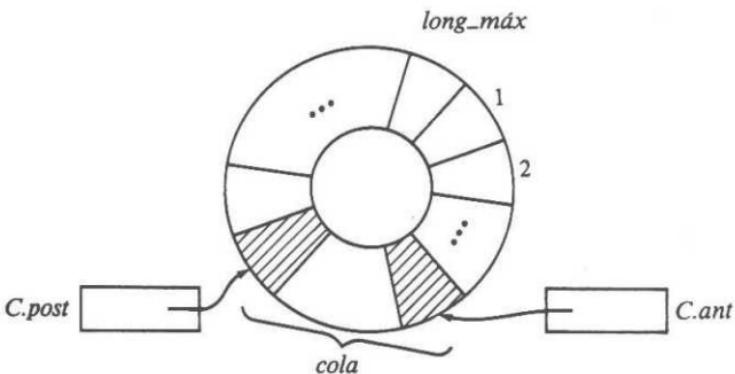


Fig. 2.21. Realización circular para colas.

2.5 Correspondencias

Una *correspondencia* o *memoria asociativa* es una función de elementos de un tipo llamado *tipo_dominio*, a elementos de otro tipo (quizás el mismo) llamado *tipo_contradicomnio*. Se expresa el hecho de que la correspondencia *M* asocia el elemento *r* de tipo *tipo_contradicomnio* con el elemento *d* de tipo *tipo_dominio* por *M(d) = r*.

Ciertas correspondencias, como *cuadrado(i) = i²*, se pueden obtener con facilidad como una función de Pascal por medio de una expresión aritmética u otro medio simple de calcular *M(d)* a partir de *d*. Sin embargo, en el caso de muchas correspondencias, la única manera clara de describir *M(d)* consiste en almacenar para cada *d* el valor de *M(d)*. Por ejemplo, para aplicar una función de nómina que asocie a cada empleado un salario semanal, parece obligatorio almacenar el salario actual de cada empleado. En lo que resta de esta sección se describirá un método para implantar funciones como la función «nómina».

Considérese qué operaciones es deseable realizar en una correspondencia *M*. Dado un elemento *d* de un tipo dominio, se podría desear obtener *M(d)* o saber si *M(d)* está definida (es decir, si *d* pertenece actualmente al dominio de *M*). O quizás

```

function suma_uno ( i : integer ): integer;
begin
    . . .
    return ((i mod long_máx) + 1)
end; { suma_uno }

procedure ANULA ( var C: COLA );
begin
    C. ant := 1;
    C. post := long_máx
end; { ANULA }

function VACIA ( var C: COLA ) : boolean;
begin
    if suma_uno(C. post) = C.ant then
        return (true)
    else
        return (false)
end; { VACIA }

function FRENTA ( var C: COLA ) : tipo_elemento;
begin
    if VACIA (C) then
        error ('la cola está vacía')
    else
        return (C.elementos[C.ant])
end; { FRENTA }

procedure PONE_EN_COLA ( x: tipo_elemento; var C: COLA );
begin
    if suma_uno(suma_uno(C.post)) = C.ant then
        error ('la cola está llena')
    else begin
        C.post := suma_uno(C.post);
        C.elementos[C.post] := x
    end
end; { PONE_EN_COLA }

procedure QUITA_DE_COLA ( var C: COLA );
begin
    if VACIA (C) then
        error ('la cola está vacía')
    else
        C.ant := suma_uno(C.ant)
end; { QUITA_DE_COLA }

```

Fig. 2.22. Realización circular de colas.

se desee introducir nuevos elementos en el dominio actual de M y establecer sus valores asociados de contradominio. Como solución distinta, se podría desechar cambiar el valor de $M(d)$. También se necesita una manera de asignar valor inicial a una correspondencia como la *correspondencia nula* o vacía, cuyo dominio está vacío. Estas operaciones se resumen en los tres mandatos siguientes:

1. ANULA (M). Hace que M sea la correspondencia nula.
2. ASIGNA (M, d, r). Define $M(d)$ como r , tanto si $M(d)$ está definido previamente como si no.
3. CALCULA (M, d, r). Devuelve verdadero y da a r el valor $M(d)$ si este último está definido; en caso contrario, devuelve falso.

Realización de correspondencias mediante arreglos

Muchas veces, el tipo dominio de una correspondencia será un tipo elemental que pueda usarse como tipo índice de un arreglo. En Pascal los tipos índice incluyen todos los subcontradominios finitos de los enteros, como $1..100$ ó $17..23$, el tipo *char* y los subcontradominios de *char*, como ' A '..' Z ', y los tipos enumerados como (norte, sur, este, oeste). Por ejemplo, un programa para descifrar textos podría guardar una correspondencia *cif*, con ' A '..' Z ' como sus tipo_dominio y tipo_contradominio, de modo que *cif*(*letra_texto*) sea la letra que en un momento dado se suponga que representa a la letra *letra_texto*.

Tales correspondencias se pueden realizar sencillamente por medio de arreglos, siempre que haya un valor del tipo_contradominio que signifique «indefinido». Por ejemplo, la correspondencia *cif* anterior podría definirse con *char* como su tipo_contradominio, en lugar de ' A '..' Z ', y '?' podría usarse para denotar «indefinido».

Supóngase que los tipos dominio y contradominio son tipo_dominio y tipo_contradominio, respectivamente, y que tipo_dominio es un tipo de Pascal básico. Entonces, se puede definir el tipo CORRESPONDENCIA (estrictamente hablando, correspondencia de tipo_dominio a tipo_contradominio) por la declaración:

```
type
  CORRESPONDENCIA = array[tipo_dominio] of tipo_contradominio;
```

En el supuesto de que «indefinido» sea una constante de tipo_contradominio y que primer_valor y último_valor sean el primero y el último valores de tipo_dominio †, es posible implantar los tres mandatos en correspondencias como en la figura 2.23.

Realización de correspondencias mediante listas

Existen muchas aplicaciones posibles de las correspondencias con dominios finitos. Por ejemplo, las tablas de dispersión (*hash*) son una excelente alternativa en mu-

† Por ejemplo, primer_valor = 'A' y último_valor = 'Z', si el tipo_dominio es ' A '..' Z '.

chas situaciones, pero su análisis se deja para el capítulo 4. Cualquier correspondencia con un dominio finito se puede representar mediante la lista de pares $(d_1, r_1), (d_2, r_2), \dots, (d_k, r_k)$, donde d_1, d_2, \dots, d_k son todos los miembros actuales del dominio, y r_i es el valor que la correspondencia asocia a d_i , para $i = 1, 2, \dots, k$. Se puede entonces usar cualquier implantación de listas que se elija para representar esa lista de pares.

Para precisar más, el tipo de datos abstracto CORRESPONDENCIA se puede aplicar por medio de listas de tipo_elemento si se define

```
type
  tipo_elemento = record
    dominio: tipo_dominio;
    contradominio: tipo_contradominio
  end;
```

y luego se define CORRESPONDENCIA como se haría con el tipo LISTA (de tipo_elemento) en la realización de listas elegida. En la figura 2.24 se definen los tres mandatos de correspondencias en función de mandatos sobre el tipo LISTA.

```
procedure ANULA ( var M: CORRESPONDENCIA );
  var
    i: tipo_dominio;
  begin
    for i := primer_valor to último_valor do
      M[i] := indefinido
    end; { ANULA }

procedure ASIGNA ( var M: CORRESPONDENCIA;
  d: tipo_dominio; r: tipo_contradominio);
  begin
    M[d] := r
  end; { ASIGNA }

function CALCULA ( var M: CORRESPONDENCIA;
  d: tipo_dominio; var r: tipo_contradominio): boolean;
  begin
    if M[d] = indefinido then
      return (false)
    else begin
      r := M[d];
      return (true)
    end
  end; { CALCULA }
```

Fig. 2.23. Realización de correspondencias mediante arreglos.

2.6 Pilas y procedimientos recursivos

Una aplicación importante de las pilas se da en la aplicación de procedimientos recursivos en los lenguajes de programación. La *organización a tiempo de ejecución* de uno de tales lenguajes es el conjunto de estructuras de datos usadas para representar los valores de las variables de un programa durante su ejecución. Todo lenguaje que, como Pascal, permita procedimientos recursivos, utiliza una pila de *registros de activación* para registrar los valores de todas las variables que pertenecen a cada procedimiento activo de un programa. Cuando se llama a un procedimiento P , se coloca en la pila un nuevo registro de activación para P , con independencia de si ya existe en ella otro registro de activación para ese mismo procedimiento. Cuando P vuelve, su registro de activación debe estar en el tope de la pila, puesto que P no puede volver si no lo han hecho previamente todos los procedimientos a los que P ha llamado. Así, se puede sacar de la pila el registro de activación correspondiente a la llamada actual de P y hacer que el control regrese al punto en el que P fue llamado (este punto, conocido como *dirección de retorno*, se colocó en el registro de activación de P al llamar a este procedimiento).

```

procedure ANULA ( var M: CORRESPONDENCIA );
{ igual que para listas }

procedure ASIGNA ( var M: CORRESPONDENCIA;
d: tipo_dominio; r: tipo_contradominio);
var
  x: tipo_elemento; { el par (d, r) }
  p: posición; { usada para ir de la primera a la última posición de
    la lista M }
begin
  x.dominio := d;
  x.contradominio := r;
  p := PRIMERO(M);
  while p <> FIN(M) do
    if RECUPERA(p, M).dominio = d then
      SUPRIME(p, M) { elimina el elemento con dominio d }
    else
      p := SIGUIENTE(p, M);
    INSERTA (x, PRIMERO(M), M) { coloca (d, r,) al inicio de la lista }
  end; { ASIGNA }

function CALCULA ( var M: CORRESPONDENCIA;
d: tipo_dominio; var r: tipo_contradominio): boolean;
var
  p: posición;
begin
  p: PRIMERO (M);
  while p <> FIN(M) do begin
    if RECUPERA(p, M).dominio = d then begin

```

```

r := RECUPERA(p, M). contradominio;
return (true)
end;
p := SIGUIENTE(p, M)
end;
return (false) { si d no está en el dominio }
end; { CALCULA }

```

Fig. 2.24. Realización de correspondencias en función de listas.

La recursión simplifica la estructura de muchos programas. Sin embargo, en algunos lenguajes las llamadas a procedimientos tienen un costo mucho mayor que el de las proposiciones de asignación, de modo que la ejecución de un programa puede resultar más rápida, en un factor constante considerable, si se eliminan las llamadas recursivas. Al decir esto no se propugna la eliminación habitual de la recursión o de otras llamadas a procedimientos; es frecuente que la sencillez estructural justifique el tiempo de ejecución. Sin embargo, podría ser deseable eliminar la recursión en las porciones de los programas que se ejecutan con mayor frecuencia, y el propósito del análisis que sigue es ilustrar cómo se pueden convertir los procedimientos recursivos en procedimientos no recursivos mediante la introducción de una pila definida por el programador.

Ejemplo 2.3. Considérense dos soluciones, una recursiva y otra no recursiva, a una versión simplificada del clásico *problema de la mochila*, en el cual se da un *objetivo* o y una colección de *pesos* p_1, p_2, \dots, p_n (enteros positivos). Se pide determinar si existe una selección de pesos que totalice exactamente o . Por ejemplo, si $o = 10$ y los pesos son 7, 5, 4, 4 y 1, se pueden elegir el segundo, el tercero y el quinto, ya que $5 + 4 + 1 = 10$.

La justificación del nombre «problema de la mochila» es que se desea llevar en la espalda no más de o kilogramos, y se tiene para elegir un conjunto de objetos de pesos dados. Se supone, además, que la utilidad de los objetos es proporcional a su peso †, y por ello se desea cargar la mochila con un peso lo más cercano posible al objetivo.

En la figura 2.25 se puede ver una función *mochila* que opera en una matriz

pesos: array[1..n] of integer.

La llamada a *mochila(s, i)* determina si existe una colección de los elementos entre *peso[i]* y *peso[n]* que sume exactamente s , e imprime sus pesos de ser tal el caso. Lo primero que *mochila* hace es determinar si puede responder de inmediato. Específicamente, si $s = 0$, entonces el conjunto vacío de pesos es una solución. Si $s < 0$, no puede haber solución, y si $s > 0$ e $i > n$, entonces ya no hay más pesos que considerar y, por tanto, no se puede encontrar una suma de pesos igual a s .

Si no se tiene ninguno de estos casos, entonces sencillamente se hace la llamada a *mochila(s - p_i, i + 1)*, para ver si existe una solución que incluya a p_i . Si esa solu-

† En el «verdadero» problema de la mochila, se dan valores de utilidad y pesos, y se pide maximizar la utilidad de los objetos transportados, con una restricción del peso.

ción existe, todo el problema está resuelto y la solución incluye a p_i , de modo que éste se imprime. Si no hay solución, se efectúa la llamada a $mochila(s, i + 1)$, para ver si existe una solución que no use p_i . \square

Eliminación de la recursión de cola

A menudo, es posible eliminar en forma mecánica la última llamada que un procedimiento se hace a sí mismo. Si un procedimiento $P(x)$ tiene como último paso una llamada a $P(y)$, entonces es posible reemplazar la llamada a $P(y)$ por una asignación $x = y$, seguida de un salto al principio del código de P . Aquí, y puede ser una expresión, pero x debe ser un parámetro pasado por valor, de modo que su valor se almacena en una localidad de memoria privada de esta llamada a P .^f Por supuesto, P podría tener más de un parámetro, en cuyo caso se tratarían exactamente igual que x e y .

```

function mochila ( objetivo: integer; candidato: integer ): boolean;
begin
(1)    if objetivo = 0 then
(2)        return ( true )
(3)    else if (objetivo < 0) or (candidato > n) then
(4)        return(false)
(5)    else { considera soluciones con y sin candidato }
(6)        if mochila(objetivo-pesos[candidato], candidato + 1) then
            begin
(6)            writeln(pesos [candidato] );
(7)            return (true)
            end
(8)        else { la única solución posible es sin candidato }
            return (mochila(objetivo, candidato + 1))
end; { mochila }
```

Fig. 2.25. Solución recursiva al problema de la mochila.

Este cambio funciona, porque volver a ejecutar P con el nuevo valor de x equivale exactamente a llamar a $P(y)$ y luego volver de esa llamada. Obsérvese que el hecho de que algunas de las variables locales de P tengan valores en la segunda llamada no tiene consecuencias. P no podría usar ninguno de esos valores, pues si se hubiera llamado a $P(y)$ como se intentaba en un principio, esos valores no habrían estado definidos.

En la figura 2.25 se ilustra otra variante de la recursión de cola; ahí, el último paso de la función $mochila$ sólo devuelve el resultado de llamarse a sí misma con otros parámetros. En una situación tal, suponiendo que los parámetros se pasan por

^f Alternativamente, x podría pasarse por referencia si y es x .

valor (o por referencia, si el mismo parámetro es el que se pasa a la llamada), se puede reemplazar la llamada por asignaciones a los parámetros y un salto al principio de la función. En el caso de la figura 2.25, se puede reemplazar la línea (8) por

```
candidato := candidato + 1;
goto principio
```

donde *principio* es una etiqueta que se va a asignar a la proposición (1). Obsérvese que no se necesita ningún cambio a *objetivo*, puesto que su valor pasa intacto como primer parámetro. De hecho, se puede advertir que, al no haber cambiado *objetivo*, las pruebas de las proposiciones (1) y (3) que lo incluyen están destinadas a fallar y, por tanto, es posible omitir las líneas (1) y (2), realizar sólo la prueba *candidato > n* en la línea (3) y luego proseguir directamente a la línea (5).

Eliminación completa de la recursión

El procedimiento de eliminación de la recursión de cola suprime la recursión por completo sólo cuando la llamada recursiva se encuentra al final del procedimiento y tiene la forma adecuada. Existe un enfoque más general que convierte cualquier procedimiento (o función) recursivo en no recursivo, pero que introduce una pila definida por el programador. En general, una celda de esa pila contendrá:

1. los valores actuales de los parámetros del procedimiento;
2. los valores actuales de todas las variables locales del procedimiento, y
3. una indicación de la dirección de retorno, esto es, del lugar a donde deberá devolver el control cuando la invocación actual del procedimiento termine.

En el caso de la función *mochila*, se puede hacer algo más sencillo. Primero, se observa que cada vez que se hace una llamada (que implica meter un registro a la pila), *candidato* se incrementa en 1. Así pues, se puede dejar *candidato* como una variable global, incrementando su valor en 1 cada vez que se mete un registro en la pila y disminuyéndolo en 1 cada vez que se saca un registro.

Una segunda simplificación posible consiste en mantener dentro de la pila una «dirección de retorno» modificada. En términos estrictos, la dirección de retorno para esta función es un lugar de otro procedimiento que llama a *mochila*, la llamada de la línea (5) o la llamada de la línea (8). Estas tres posibilidades se representan por una variable «de estado» que tiene uno de los tres valores siguientes:

1. *ninguno*, que indica que la llamada procede de fuera de la función *mochila*.
2. *incluido*, que indica la llamada de la línea (5), la cual incluye *pesos[candidato]* dentro de la solución, o
3. *excluido*, que indica la llamada de la línea (8), la cual excluye a *pesos[candidato]*.

Si se almacena esta variable de estado como la dirección de retorno, se puede manejar *objetivo* como una variable global. Cuando el estado cambia de *ninguno* a *incluido*, se sustrae *pesos[candidato]* de *objetivo*, y se suma de nuevo cuando el estado

cambia de *incluido* a *excluido*. Como ayuda para representar el efecto del retorno de *mochila* cuando indican si se ha hallado la solución, se usa una variable global *bandera-exito*. Una vez que *bandera-exito* toma el valor verdadero, lo conserva y hace que se saquen de la pila los registros, imprimiendo aquellos pesos que tienen asociado un estado *incluido*. Con estas modificaciones, se puede declarar la pila como una pila de estados, mediante.

type

estados = (*ninguno*, *incluido*, *excluido*);

PILA = { declaración adecuada para una pila de estados }

La figura 2.26 muestra el procedimiento resultante no recursivo *mochila*, que opera en un arreglo de *pesos* como antes. Aunque este procedimiento puede ser más rápido que la función *mochila* original, es claramente más largo y más difícil de entender. Por ello sólo se debe eliminar la recursión cuando la velocidad sea muy importante.

```

procedure mochila (objetivo: integer);
var
  candidato: integer;
  bandera_exito: boolean;
  P: PILA;
begin
  candidato := 1;
  bandera_exito := false;
  ANULA(P);
  METE(ninguno, P); { asigna valor inicial a la pila considerando pesos[1] }
repeat
  if bandera_exito then begin
    { saca de la pila e imprime los pesos incluidos en la solución }
    if TOPE(P) = incluido then
      writeln (pesos[candidato]);
    candidato := candidato - 1;
    SACA(P)
  end
  else if objetivo = 0 then begin { se encontró la solución }
    bandera_exito := true;
    candidato := candidato - 1;
    SACA(P)
  end
  else if (((objetivo < 0) and (TOPE(P) = ninguno))
            or (candidato > n)) then begin
    { no hay solución posible con las elecciones hechas }
    candidato := candidato - 1;
    SACA(P)
  end
end

```

```

end
else { aún no hay decisión, se considera el estado del candidato actual }
if TOPE( $P$ ) = ninguno then begin { se intenta primero incluyendo al
candidato }
    objetivo := objetivo-pesos[candidato];
    candidato := candidato +1;
    SACA( $P$ ); METE(incluido,  $P$ ); METE (ninguno,  $P$ )
end
else if TOPE( $P$ ) = excluido then begin { se intenta excluyendo al
candidato }
    objetivo := objetivo + pesos[candidato];
    candidato := candidato +1;
    SACA( $P$ ); METE(excluido,  $P$ ); METE(ninguno,  $P$ )
end
else begin { TOPE( $P$ ) = excluido; la elección actual no da resultado }
    SACA( $P$ );
    candidato := candidato -1
end
until VACIA ( $P$ )
end; { mochila }

```

Fig. 2.26. Procedimiento no recursivo para el problema de la mochila.

Ejercicios

- 2.1 Escribase un programa que imprima los elementos de una lista. En los ejercicios siguientes úsense operaciones con listas para realizar los programas.
- 2.2 Escribanse programas para insertar, suprimir y localizar un elemento en una lista clasificada, usando realizaciones de
 - a) arreglos
 - b) apuntadores y
 - c) cursores.

¿Cuál es el tiempo de ejecución de cada uno de estos programas?
- 2.3 Escribase un programa para intercalar
 - a) dos listas clasificadas,
 - b) n listas clasificadas.
- 2.4 Escribase un programa para concatenar una lista de listas.
- 2.5 Supóngase que se desea manipular polinomios de la forma $p(x) = c_1x^{e_1} + c_2x^{e_2} + \dots + c_nx^{e_n}$, donde $e_1 > e_2 > \dots > e_n \geq 0$. Un polinomio de ese tipo se puede representar mediante una lista enlazada en la que cada celda

tiene tres campos: uno para el coeficiente c_i , otro para el exponente e_i , y otro para el apuntador a la siguiente celda. Escribáse un programa para diferenciar polinomios representados de esta manera.

- 2.6 Escribanse programas para sumar y multiplicar polinomios similares a los del ejercicio 2.5. ¿Cuál es el tiempo de ejecución de los programas en función del número de términos?
- *2.7 Supóngase que se declaran celdas mediante

```
type
  tipo_celda = record
    bit: 0..1;
    sig: ^ tipo_celda
  end;
```

Un número binario $b_1 b_2 \dots b_n$, donde cada b_i es 0 ó 1, tiene el valor numérico $\sum_{i=1}^n b_i 2^{n-i}$. Este número se puede representar por la lista b_1, b_2, \dots, b_n .

Esta lista, a su vez, puede representarse como una lista enlazada de celdas de tipo tipo_celda. Escribase un procedimiento *incrementa(número_bin)* que sume uno al número binario apuntado por *número_bin*. Sugerencia: Hágase *incrementa* recursivo.

- 2.8 Escribase un procedimiento para intercambiar los elementos de las posiciones p y *SIGUIENTE(p)* de una lista enlazada sencilla.
- *2.9 El siguiente procedimiento se hizo con el propósito de suprimir todas las apariciones de un elemento x de una lista L . Explíquese por qué no siempre funciona y sugírerase una manera de arreglarlo de modo que realice la tarea para la que fue propuesto.

```
procedure suprime (x:tipo_elemento; var L: LISTA);
  var
    p: posición;
  begin
    p:= PRIMERO (L);
    while p <> FIN (L) do begin
      if RECUPERA (p, L) = x then
        SUPRIME (p, L);
      p:= SIGUIENTE (p, L)
    end
  end; { suprime }
```

- 2.10 Se desea almacenar una lista en un arreglo A cuyas celdas contienen dos campos: *datos*, que contiene un elemento, y *posición*, que da la posición (entera) del elemento. Un entero *último* indica que la lista ocupa las posiciones $A[1]$ a $A[\text{último}]$. El tipo LISTA se puede definir como:

```

type
  LISTA = record
    ultimo: integer;
    elementos: array[1..long_máx] of record
      datos: tipo_elemento;
      posición: integer
    end
  end;

```

Escríbese un procedimiento SUPRIME(p, L) que elimine el elemento de la posición p . Inclúyanse todas las verificaciones necesarias para detectar errores.

- 2.11 Supóngase que L es una LISTA y que p, q y r son posiciones. En función de la longitud n de la lista L , determínese cuántas veces se ejecutan en el siguiente programa las funciones PRIMERO, FIN y SIGUIENTE.

```

 $p := \text{PRIMERO}(L);$ 
while  $p <> \text{FIN}(L)$  do begin
   $q := p;$ 
  while  $q <> \text{FIN}(L)$  do begin
     $q := \text{SIGUIENTE}(q, L);$ 
     $r := \text{PRIMERO}(L);$ 
    while  $r <> q$  do
       $r := \text{SIGUIENTE}(r, L)$ 
    end;
     $p := \text{SIGUIENTE}(p, L)$ 
  end;

```

- 2.12 Reescríbese el código de las operaciones con listas, suponiendo una representación de listas enlazadas, pero sin celdas de encabezado. Supóngase que se usan apuntadores verdaderos y que la posición 1 se representa por nil.
- 2.13 Agréguese al procedimiento de la figura 2.12 todas las verificaciones necesarias para detectar errores.
- 2.14 Otra representación de listas por medio de arreglos consiste en insertar, como en la sección 2.2, pero suprimir simplemente reemplazando el elemento en cuestión por un valor especial «suprimido», que se supone ya no aparece en las listas. Reescríbanse las operaciones con listas para aplicar esta estrategia. ¿Cuáles son las ventajas y las desventajas de este enfoque en comparación con la representación original de listas mediante arreglos?

- 2.15** Supóngase que se desea utilizar un bit extra en los registros de colas, que indique si una cola está vacía o no. Modifíquense las declaraciones y las operaciones para una cola circular, de manera que se tenga en cuenta esta característica. ¿Cabe esperar que el cambio valga la pena?
- 2.16** Una *cola doble* o de doble extremo es una lista en la cual se pueden insertar o suprimir elementos en cualquiera de los extremos. Desarróllense realizaciones para colas dobles basadas en arreglos, apuntadores y cursosres.
- 2.17** Definase un TDA que maneje las operaciones PONE_EN_COLA, QUIITA_DE_COLA y ESTA_EN_COLA. ESTA_EN_COLA(x) es una función que devuelve verdadero o falso dependiendo de si x está o no en la cola.
- 2.18** ¿Cómo podría obtenerse una cola cuyos elementos son cadenas de longitud arbitraria? ¿Cuánto tiempo lleva poner en cola una cadena?
- 2.19** Otra posible implantación de colas mediante listas enlazadas consiste en no usar una celda de encabezamiento y hacer que *ant* apunte directamente a la primera celda. Si la cola está vacía, se hace que *ant - post = nil*. Obténgase las operaciones con colas para esta representación. ¿Cómo se compara esta aplicación con la que se dio en la sección 2.4, en función de la velocidad, la utilización de espacio y la concisión del código?
- 2.20** Una variante de la cola circular registra la posición del elemento frontal y la longitud de la cola.
- ¿Es necesario en esta realización limitar la longitud de la cola a $long_máx - 1$?
 - Escríbanse las cinco operaciones con colas para esta realización.
 - Compárese esta realización con la de colas circulares de la sección 2.4.
- 2.21** Es posible guardar dos pilas en un solo arreglo, si una de ellas crece desde la posición 1 del arreglo y la otra lo hace desde la última posición. Escríbase un procedimiento METE(x, P) que inserte el elemento x en la pila P , donde P es una u otra de las dos pilas. Inclúyanse en el procedimiento todas las verificaciones de error necesarias.
- 2.22** Se pueden almacenar k pilas en un solo arreglo si se usa la estructura de datos sugerida en la figura 2.27 para el caso $k = 3$. Para meter y sacar de cada pila se opera como se sugiere en la sección 2.3, en relación con la figura 2.17. Sin embargo, si una inserción en la pila i hace que TOPE(i) igualle a BASE($i - 1$), antes de efectuarla hay que desplazar todas las pilas, de modo que quede una separación del tamaño apropiado entre cada par de pilas adyacentes. Por ejemplo, es posible hacer que las separaciones entre pilas sean todas iguales, o que la separación que queda encima de la pila i sea proporcional al tamaño actual de la pila i (en el supuesto de que las pilas más grandes tienen mayor probabilidad de crecer antes, y se desea posergar la siguiente reorganización el mayor tiempo posible).
- Si se dispone de un procedimiento *reorganiza*, al que se llama si las pilas colisionan, escríbase el código de las cinco operaciones con pilas.

- b) Si existe un procedimiento *haz_nuevos_topes* que calcula *nuevo_topo[i]*, la posición «apropiada» para el tope de la pila *i*, con $1 \leq i \leq k$, escribase el procedimiento *reorganiza*. Sugerencia: Obsérvese que la pila *i* puede moverse arriba o abajo y que hay que mover la pila *i* antes que la *j* si la nueva posición de *j* se traslapa con la posición antigua de *i*. Considérense las pilas 1, 2, 3 ... , *k* en orden, pero manténgase una pila de «objetivos» consistentes en desplazar una pila. Si al considerar la pila *i* ésta se puede mover con seguridad, hágase, y reconsiderérese luego la pila cuyo número esté en el tope de la pila de objetivos. Si no hay seguridad, métase *i* en la pila de objetivos.
- c) ¿Qué realización es apropiada para la pila de objetivos de b)? ¿Es realmente necesario mantenerla como una lista de enteros, o sería suficiente con una representación más sucinta?
- d) Obténgase *haz_nuevos_topes* de forma que el espacio que está sobre todas las pilas sea proporcional a los tamaños actuales de éstas.
- e) ¿Qué modificaciones habría que hacer a la figura 2.27 para que esa realización pueda trabajar con colas? ¿Y con listas generales?
- 2.23 Modifíquense las realizaciones de SACA y PONE_EN_COLA de las secciones 2.3 y 2.4, de modo que devuelven el elemento suprimido de la pila o cola. ¿Qué modificaciones habría que hacer si el tipo de los elementos no es alguno que pueda ser devuelto por una función?
- 2.24 Usese una pila para eliminar la recursión de los siguientes procedimientos.
- a) **function** *comb* (*n, m* : integer): integer;
 {calcula $\binom{n}{m}$ suponiendo que $0 \leq m \leq n$ y $n \geq 1$ }
begin
 if (*n* = 1) or (*m* = 0) or (*m* = *n*) then
 return (1)
 else
 return (*comb* (*n* - 1, *m*) + *comb* (*n* - 1, *m* - 1))
end; {*comb*}

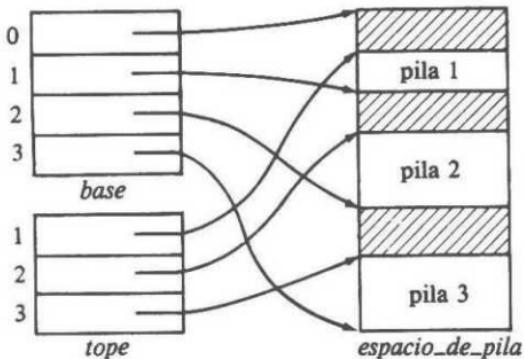


Fig. 2.27. Varias pilas en un arreglo.

```

b) procedure invierte (var L: LISTA);
   { invierte la lista L }
  var
    x: tipo_elemento;
  begin
    if not VACIA(L) then begin
      x := RECUPERA(PRIMERO(L), L);
      SUPRIME(PRIMERO(L), L);
      invierte(L);
      INSERTA(x, FIN(L), L)
    end
  end; { invierte }

```

- *2.25 ¿Es posible eliminar la recursión de cola de los programas del ejercicio 2.24?
De ser así, hágase.

Notas bibliográficas

Knuth [1968] contiene información adicional sobre la realización de listas, pilas y colas. Ciertos lenguajes de programación como LISP y SNOBOL, manejan listas y cadenas de manera apropiada. Véanse Sammet [1969], Nicholls [1975], Pratt [1975] o Wexelblat [1981] sobre una historia y una descripción de muchos de estos lenguajes.

3

Arboles

Un árbol impone una estructura jerárquica sobre una colección de objetos. Los árboles genealógicos y los organigramas son ejemplos comunes de árboles. Entre otras cosas, los árboles son útiles para analizar circuitos eléctricos y para representar la estructura de fórmulas matemáticas. También se presentan naturalmente en diversas áreas de computación. Por ejemplo, se usan para organizar información en sistemas de bases de datos y para representar la estructura sintáctica de un programa fuente en los compiladores. En el capítulo 5 se describe el uso de árboles en la representación de datos. En este libro se usan muchas variantes de árboles. En este capítulo se presentan las definiciones básicas y algunas de las operaciones más comunes con árboles. Despues se describen algunas de las estructuras de datos más frecuentemente usadas para representar árboles que permiten manejar esas operaciones con eficiencia.

3.1 Terminología fundamental

Un árbol es una colección de elementos llamados *nodos*, uno de los cuales se distingue como *raíz*, junto con una relación (de «paternidad») que impone una estructura jerárquica sobre los nodos. Un nodo, como un elemento de una lista, puede ser del tipo que se desee. A menudo se representa un nodo por medio de una letra, una cadena de caracteres o un círculo con un número en su interior. Formalmente, un *árbol* se puede definir de manera recursiva como sigue:

1. Un solo nodo es, por sí mismo, un árbol. Ese nodo es también la raíz de dicho árbol.
2. Supóngase que n es un nodo y que A_1, A_2, \dots, A_k son árboles con raíces n_1, n_2, \dots, n_k , respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos n_1, n_2, \dots, n_k . En dicho árbol, n es la raíz y A_1, A_2, \dots, A_k son los *subárboles* de la raíz. Los nodos n_1, n_2, \dots, n_k reciben el nombre de *hijos* del nodo n .

A veces, conviene incluir entre los árboles el *árbol nulo*, un «árbol» sin nodos que se representa mediante Λ .

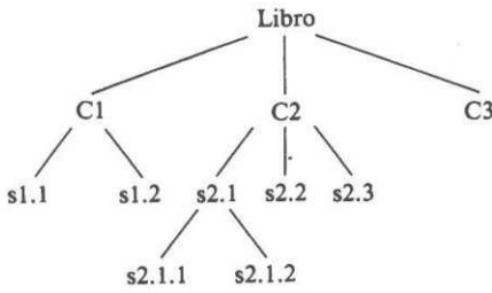
Ejemplo 3.1. Considérese el índice general de un libro, que se representa en la figura 3.1(a). Tal índice es un árbol. Se puede redibujar en la forma mostrada en la

figura 3.1(b). La relación padre-hijo entre dos nodos se representa por una línea que los une. Los árboles normalmente se dibujan de arriba hacia abajo, como en la figura 3.1(b), con el padre encima de los hijos.

La raíz, el nodo llamado «Libro», tiene tres subárboles que corresponden a los capítulos C1, C2 y C3. Esta relación se representa por medio de las líneas descendentes que unen a Libro con C1, C2 y C3. Libro es el padre de C1, C2 y C3, y estos tres nodos son los hijos de Libro.

Libro
C1
s1.1
s1.2
C2
s2.1
s2.1.1
s2.1.2
s2.2
s2.3
C3

(a)



(b)

Fig. 3.1. Un índice general y su representación como árbol.

El tercer subárbol de Libro, que tiene raíz C3, es un árbol de un solo nodo, en tanto que los otros dos subárboles tienen una estructura no trivial. Por ejemplo, el subárbol con raíz C2 tiene tres subárboles que corresponden a las secciones s2.1, s2.2 y s2.3; estas dos últimas son árboles de un solo nodo, mientras que la primera tiene dos subárboles que corresponden a las subsecciones s2.1.1 y s2.1.2. □

El ejemplo 3.1 es típico de cierta clase de datos cuya mejor representación se logra mediante un árbol. En el ejemplo, la relación de paternidad representa inclusión: un nodo padre está constituido por sus hijos, como Libro está constituido por C1, C2 y C3. En esta obra se encontrarán otras relaciones que se pueden representar por la relación de paternidad en un árbol.

Si n_1, n_2, \dots, n_k , es una sucesión de nodos de un árbol tal que n_i es el padre de n_{i+1} para $1 \leq i < k$, entonces la secuencia se denomina *camino* del nodo n_1 al nodo n_k . La *longitud* de un camino es el número de nodos del camino menos 1. Por tanto, hay un camino de longitud cero de cualquier nodo a sí mismo. Por ejemplo, en la figura 3.1 hay un camino de longitud 2, a saber, (C2, s2.1, s2.1.2), de C2 a s2.1.2.

Si existe un camino de un nodo a a otro b , entonces a es un *antecesor* de b , y b es un *descendiente* de a . Por ejemplo, en la figura 3.1, los antecesores de s2.1 son él mismo, C2 y Libro. Obsérvese que cada nodo es a la vez un antecesor y un descendiente de sí mismo.

Un antecesor o un descendiente de un nodo que no sea él mismo recibe el nombre de *antecesor propio* o *descendiente propio*, respectivamente. En un árbol, la raíz es el único nodo que no tiene antecesores propios. Un nodo sin descendientes pro-

pios se denomina *hoja*. Un subárbol de un árbol es un nodo junto con todos sus descendientes.

La *altura* de un nodo en un árbol es la longitud del camino más largo de ese nodo a una hoja. En la figura 3.1 el nodo C1 tiene altura 1, C2 altura 2 y el nodo C3 altura 0. La *altura del árbol* es la altura de la raíz. La *profundidad* de un nodo es la longitud del camino único desde la raíz a ese nodo.

Orden de los nodos

A menudo los hijos de un nodo se ordenan de izquierda a derecha. Así, los dos árboles de la figura 3.2 son diferentes porque los dos hijos del nodo *a* aparecen en distintos órdenes en los dos árboles. Si se desea ignorar explícitamente el orden de los hijos, se habla entonces de un árbol *no ordenado*.

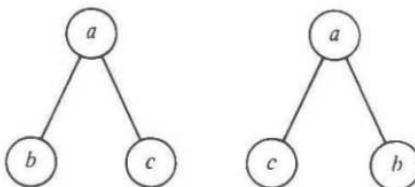


Fig. 3.2. Dos árboles (ordenados) distintos.

El orden de izquierda a derecha de los *hermanos* (hijos del mismo nodo) se puede extender para comparar dos nodos cualesquiera entre los cuales no exista la relación antecesor-descendiente. La regla que se aplica es que si *a* y *b* son hermanos y *a* está a la izquierda de *b*, entonces todos los descendientes de *a* están a la izquierda de todos los descendientes de *b*.

Ejemplo 3.2. Considérese el árbol de la figura 3.3. El nodo 8 está a la derecha del nodo 2, a la izquierda de los nodos 9, 6, 10, 4 y 7, y no está a la izquierda ni a la derecha de sus antecesores 1, 3 y 5.

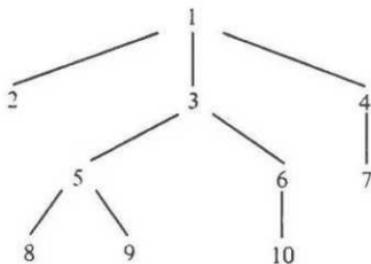


Fig. 3.3. Un árbol.

Dado un nodo n , una regla sencilla para determinar qué nodos están a su izquierda y cuáles a su derecha, consiste en dibujar el camino de la raíz a n . Todos los nodos que salen a la izquierda de este camino, y todos sus descendientes, están a la izquierda de n . Los nodos, y sus descendientes, que salen a la derecha, están a la derecha de n . \square

Orden previo, orden posterior y orden simétrico

Existen varias maneras útiles de ordenar sistemáticamente todos los nodos de un árbol. Los tres ordenamientos más importantes se llaman orden previo (*preorder*), orden simétrico (*inorder*) y orden posterior (*postorder*); tales ordenamientos se definen recursivamente como sigue:

- Si un árbol A es nulo, entonces la lista vacía es el listado de los nodos de A en los órdenes previo, simétrico y posterior.
- Si A contiene un solo nodo, entonces ese nodo constituye el listado de los nodos de A en los órdenes previo, simétrico y posterior.

Si ninguno de los anteriores es el caso, sea A un árbol con raíz n y subárboles A_1, A_2, \dots, A_k , como se representa en la figura 3.4.

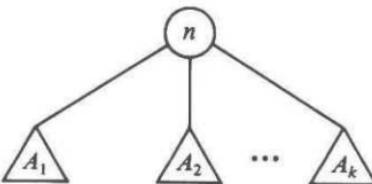


Fig. 3.4. Árbol A .

1. El *listado en orden previo* (o *recorrido en orden previo*) de los nodos de A está formado por la raíz de A , seguida de los nodos de A_1 en orden previo, luego por los nodos de A_2 en orden previo y así sucesivamente hasta los nodos de A_k en orden previo.
2. El *listado en orden simétrico* de los nodos de A está constituido por los nodos de A_1 en orden simétrico, seguidos de n y luego por los nodos de A_2, \dots, A_k , con cada grupo de nodos en orden simétrico.
3. El *listado en orden posterior* de los nodos de A tiene los nodos de A_1 en orden posterior, luego los nodos de A_2 en orden posterior y así sucesivamente hasta los nodos de A_k en orden posterior y por último la raíz n .

La figura 3.5(a) muestra el esbozo de un procedimiento para listar los nodos de un árbol en orden previo. Para convertirlo en un procedimiento de recorrido en orden posterior, basta invertir el orden de los pasos (1) y (2). La figura 3.5(b) es un esbozo de un procedimiento de recorrido en orden simétrico. En los tres casos, el or-

denamiento deseado de un árbol se produce llamando al procedimiento apropiado en la raíz del árbol.

Ejemplo 3.3. Listense en orden previo los nodos del árbol de la figura 3.3. Primero se lista 1, y luego se llama recursivamente a ORD_PRE en el primer subárbol de 1, o sea el subárbol con raíz 2. Este subárbol tiene un solo nodo, de manera que simplemente se lista. Luego se sigue con el segundo subárbol de 1, el que tiene raíz 3. Se lista 3 y luego se vuelve a llamar a ORD_PRE en el primer subárbol de 3. Esta llamada origina que se listen 5, 8 y 9, en ese orden. Continuando de esta forma, se obtiene el recorrido completo en orden previo de la figura 3.3: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7.

```
procedure ORD_PRE ( n: nodo );
begin
```

- (1) lista n ;
- (2) for cada hijo c de n , si los hay, en orden desde la izquierda do
 ORD_PRE(c)
 end; { ORD_PRE }

(a) procedimiento ORD_PRE.

```
procedure ORD_SIM ( n: nodo );
```

```
begin
```

```
if  $n$  es una hoja then
```

```
    lista  $n$ 
```

```
else begin
```

```
    ORD_SIM(hijo de  $n$  de más a la izquierda);
```

```
    lista  $n$ ;
```

```
    for cada hijo  $c$  de  $n$ , excepto el de más a la izquierda, en orden  
        desde la izquierda do
```

```
        ORD_SIM( $c$ )
```

```
    end
```

```
end; { ORD_SIM }
```

(b) procedimiento ORD_SIM.

Fig. 3.5. Procedimientos recursivos de ordenamiento.

De manera similar, simulando la figura 3.5(a) con el orden de los pasos invertido, se puede descubrir que el recorrido en orden posterior de la figura 3.3 es 2, 8, 9, 5, 10, 6, 3, 7, 4, 1. Simulando la figura 3.5(b), se encuentra que el listado en orden simétrico de la figura 3.3 es 2, 1, 8, 5, 9, 3, 10, 6, 7, 4. \square

Un truco útil para producir los tres ordenamientos de nodos es el siguiente. Imáginese que se camina por la periferia del árbol, partiendo de la raíz, y que se avanza en sentido contrario al de las manecillas del reloj, manteniéndose tan cerca del árbol

como sea posible; el camino previsto para la figura 3.3 está representado en la figura 3.6.

Para el orden previo se lista un nodo la primera vez que se pasa por él. En el caso del orden posterior, se lista un nodo la última vez que se pasa por él, conforme se sube hacia su padre. Tratándose del orden simétrico, se lista una hoja la primera vez que se pasa por ella, y un nodo interior, la segunda vez que se pasa por él. A manera de ejemplo, en la figura 3.6 se pasa por el nodo 1 al empezar, y otra vez al pasar por la «bahía» entre los nodos 2 y 3. Obsérvese que el orden de las hojas en los tres ordenamientos corresponde al mismo ordenamiento de izquierda a derecha de las hojas. Sólo el orden de los nodos interiores y su relación con las hojas varía entre los tres ordenamientos.

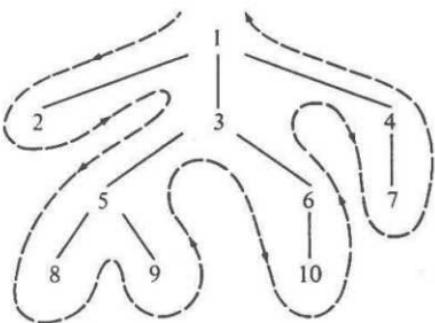


Fig. 3.6. Recorrido de un árbol.

Arboles etiquetados y árboles de expresiones

A menudo es útil asociar una *etiqueta*, o valor, a cada nodo de un árbol, siguiendo la misma idea con que se asoció un valor a un elemento de una lista en el capítulo anterior. Esto es, la etiqueta de un nodo no será el nombre del nodo, sino un valor «almacenado» en él. En algunas aplicaciones, incluso se cambiará la etiqueta de un nodo sin modificar su nombre. Una analogía útil a este respecto es: árbol es a lista como etiqueta es a elemento y nodo es a posición.

Ejemplo 3.4. La figura 3.7 muestra un árbol etiquetado que representa la expresión aritmética $(a + b) * (a + c)$, donde n_1, n_2, \dots, n_7 son los nombres de los nodos, cuyas etiquetas se muestran, por convención, en las proximidades de los nodos correspondientes. Las reglas para representar una expresión mediante un árbol etiquetado son éstas:

1. Cada hoja está etiquetada con un operando y sólo consta de ese operando. Por ejemplo, el nodo n_4 representa la expresión a .
2. Cada nodo interior n está etiquetado con un operador. Supóngase que n está etiquetado con el operador binario θ , como $+ o *$, y que el hijo izquierdo de n re-

presenta la expresión E_1 y el hijo derecho la expresión E_2 . Entonces, n representa la expresión $(E_1) \theta (E_2)$. Los paréntesis se pueden quitar si no son necesarios.

Por ejemplo, el nodo n_2 tiene al operador $+$ como etiqueta, y sus hijos izquierdo y derecho representan las expresiones a y b , respectivamente. Por tanto, n_2 representa $(a) + (b)$, o, más simple, $a + b$. El nodo n_1 representa $(a + b) * (a + c)$, puesto que $*$ es la etiqueta de n_1 , y $a + b$ y $a + c$ son las correspondientes expresiones representadas por n_2 y n_3 . \square

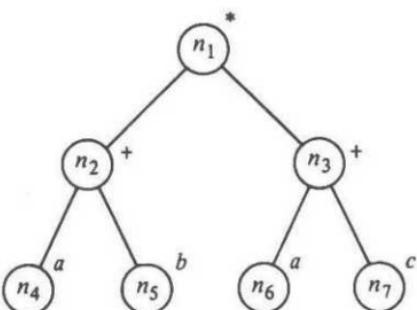


Fig. 3.7. Árbol de expresión con etiquetas.

A menudo, cuando se recorre un árbol en orden previo, simétrico o posterior, se prefiere listar las etiquetas de los nodos, en vez de sus nombres. En el caso de un árbol que representa una expresión, el listado en orden previo de las etiquetas da lo que se conoce como forma *prefija* de la expresión, en la cual un operador precede a sus operandos izquierdo y derecho. Para precisar, la expresión prefija correspondiente a un solo operando a es a mismo. La expresión prefija correspondiente a $(E_1) \theta (E_2)$, donde θ es un operador binario, es $\theta P_1 P_2$, donde P_1 y P_2 son las expresiones prefijas correspondientes a E_1 y E_2 . Obsérvese que en la expresión prefija no se necesitan paréntesis, dado que es posible revisar la expresión prefija $\theta P_1 P_2$ e identificar únicamente a P_1 como el prefijo más corto (y único) de $P_1 P_2$, que es además una expresión prefija válida.

Por ejemplo, el listado en orden previo de las etiquetas de la figura 3.7 es $*+ab+ac$. La expresión prefija correspondiente a n_2 , que es $+ab$, es el prefijo válido más corto de $+ab+ac$.

De manera similar, el listado en orden posterior de las etiquetas de un árbol de expresión da lo que se conoce como representación *postfija* (o *polaca*). La expresión $(E_1) \theta (E_2)$ se representa con la expresión postfija $P_1 P_2 \theta$, donde P_1 y P_2 son las representaciones postfijas de E_1 y E_2 , respectivamente. De nuevo, los paréntesis son innecesarios, porque se puede identificar a P_2 buscando el sufijo más corto de $P_1 P_2$ que sea una expresión postfija válida. Por ejemplo, la expresión postfija correspondiente a la figura 3.7 es $ab+ac+*$. Si se escribe una expresión en la forma $P_1 P_2 *$, entonces P_1 es $ac+$, el sufijo más corto de $ab+ac+$ que es una expresión postfija válida.

El recorrido en orden simétrico de un árbol de expresión da la expresión infija misma, pero sin paréntesis. Por ejemplo, el listado en orden simétrico de las etiquetas de la figura 3.7 es $a+b * a+c$. Se invita a elaborar un algoritmo que recorra un árbol de expresión, y que produzca una expresión infija con todos los pares de paréntesis necesarios.

Determinación de los antecesores

Los recorridos de un árbol en los órdenes previo y posterior permiten determinar los antecesores de un nodo. Supóngase que $ord_post(n)$ es la posición del nodo n en el listado en orden posterior de los nodos de un árbol, y que $desc(n)$ es el número de descendientes propios del nodo n . Por ejemplo, en el árbol de la figura 3.7 se verá cómo las posiciones en orden posterior de los nodos n_2 , n_4 y n_5 son 3, 1 y 2, respectivamente.

Las posiciones en orden posterior de los nodos tienen la útil propiedad de que los nodos de un subárbol con raíz n ocupan posiciones consecutivas de $ord_post(n)-desc(n)$ a $ord_post(n)$. Para determinar si un vértice x es descendiente de un vértice y , basta verificar que se cumple

$$ord_post(y)-desc(y) \leq ord_post(x) \leq ord_post(y)$$

Una propiedad similar se cumple para el recorrido en orden previo.

3.2 EL TDA ARBOL

En el capítulo 2, se vieron las listas, pilas, colas y correspondencias como tipos de datos abstractos (TDA). En este capítulo, los árboles se considerarán como TDA y también como estructuras de datos. Uno de los usos más importantes de los árboles se presenta en el diseño de implantaciones de varios TDA que se estudian en este libro. Por ejemplo, en la sección 5.1 se verá cómo un «árbol binario de búsqueda» puede usarse para obtener tipos de datos abstractos basados en el modelo matemático de un conjunto junto con operaciones como INSERTA, SUPRIME y MIEMBRO (esta última para probar si un elemento es o no elemento de un conjunto). Los dos capítulos siguientes presentan otras realizaciones de varios TDA mediante árboles.

En esta sección se describirán varias operaciones útiles con árboles y se verá cómo diseñar algoritmos para árboles en función de esas operaciones. Igual que con las listas, hay una gran variedad de operaciones que se pueden efectuar con árboles. Aquí se considerarán las siguientes:

1. **PADRE(n , A)**. Esta función devuelve el padre del nodo n en el árbol A . Si n es la raíz, que no tiene padre, se devuelve Λ . En este contexto, Λ es un «nodo nulo», que se usa como señal de que se ha salido del árbol.
2. **HIJO_MAS_IZQ(n , A)** devuelve el hijo más a la izquierda del nodo n en el árbol A , y devuelve Λ si n es una hoja y, por tanto, no tiene hijos.

3. HERMANO_DER(n, A) devuelve el hermano a la derecha del nodo n en el árbol A , el cual se define como el nodo m que tiene el mismo padre p que n , de forma que m está inmediatamente a la derecha de n en el ordenamiento de los hijos de p . Por ejemplo, para el árbol de la figura 3.7, HIJO_MAS_IZQ(n_2) = n_4 , HERMANO_DER(n_4) = n_5 y HERMANO_DER(n_5) = Λ .
4. ETIQUETA(n, A) devuelve la etiqueta del nodo n en el árbol A . Sin embargo, no se requiere que haya etiquetas definidas para cada árbol.
5. CREA*i*(v, A_1, A_2, \dots, A_i) es un miembro de una familia infinita de funciones, una para cada valor de $i = 0, 1, 2, \dots$. CREA*i* crea un nuevo nodo r con etiqueta v y le asigna i hijos que son las raíces de los árboles A_1, A_2, \dots, A_i , en ese orden desde la izquierda. Se devuelve el árbol con raíz r . Obsérvese que si $i = 0$, entonces r es a la vez una hoja y la raíz.
6. RAIZ(A) devuelve el nodo raíz del árbol A , o Λ si A es el árbol nulo.
7. ANULA(A) convierte A en el árbol nulo.

Ejemplo 3.5. Escribábase un procedimiento recursivo y otro que no lo sea, para listar las etiquetas de los nodos de un árbol en orden previo. Se supone que los tipos de datos nodo y ARBOL ya están definidos, y que el tipo de datos ARBOL es para árboles con etiquetas del tipo tipo_etiqueta. La figura 3.8 muestra un procedimiento recursivo que, dado un nodo n , lista las etiquetas del subárbol con raíz n en orden previo. Para obtener un listado en orden previo correspondiente al árbol A se hace la llamada ORD_PRE(RAIZ(A)).

```

procedure ORD_PRE (  $n$ : nodo );
  { lista las etiquetas de los descendientes de  $n$  en orden previo }
  var
    c: nodo;
  begin
    imprime (ETIQUETA( $n, A$ ));
     $c :=$  HIJO_MAS_IZQ( $n, A$ );
    while  $c <> \Lambda$  do begin
      ORD_PRE( $c$ );
       $c :=$  HERMANO_DER( $c, A$ )
    end
  end;
end; { ORD_PRE }

```

Fig. 3.8. Procedimiento recursivo de listado en orden previo.

También se desarrollará un procedimiento no recursivo para imprimir las etiquetas en orden previo. Para ubicarse en el árbol se usará una pila P , cuyo tipo PILA es, en realidad, «pila de nodos». La idea básica del algoritmo es que cuando se esté en un nodo n , la pila contendrá el camino de la raíz a n , con la raíz en la base de la pila y el nodo n en la parte superior †.

† Recuérdese el análisis sobre recursividad de la sección 2.6, donde se mostró que la implantación de un procedimiento recursivo requiere una pila de registros de activación. Si analiza la figura 3.8, se observa

Una manera de efectuar un recorrido de un árbol en orden previo no recursivo se da en el programa ORD_PRE_NO_REC de la figura 3.9. Este programa tiene dos modos de operación. En el primero, desciende por el camino más a la izquierda aún no explorado del árbol, imprimiendo y apilando los nodos a lo largo del camino, hasta que alcanza una hoja.

Luego, el programa entra en el segundo modo de operación, en el cual regresa por el camino apilado, sacando de la pila los nodos del camino hasta que encuentra un nodo que tiene un hermano derecho. El programa vuelve entonces al primer modo de operación, iniciando el descenso a partir de ese hermano derecho aún no explorado.

El programa empieza en el primer modo de operación en la raíz, y termina cuando la pila queda vacía. En la figura 3.9 se muestra el programa completo.

3.3 Realizaciones de árboles

En esta sección se presentarán varias realizaciones básicas de árboles y se analizarán sus funciones para manejar las operaciones con árboles presentadas en la sección 3.2.

Representación de árboles mediante arreglos

Sea A un árbol cuyos nodos tienen nombres $1, 2, \dots, n$. Tal vez la representación más sencilla de A que es capaz de manejar la operación PADRE sea un arreglo lineal L en la cual cada entrada $L[i]$ constituya un apuntador o un cursor al padre del nodo i . La raíz de A se puede distinguir dándole como padre un apuntador nulo o que apunte a sí misma. En Pascal no se pueden usar apuntadores a elementos de arreglos, por tanto, habrá que emplear un esquema de cursos donde $L[i] = j$ si el nodo j es el padre del nodo i , y $L[i] = 0$ si el nodo i es la raíz.

Esta representación se basa en la propiedad de que en los árboles cada nodo tiene un parente único, y permite hallar el parente de un nodo en un tiempo constante. Un camino ascendente en un árbol, es decir de un nodo a su parente, de éste a su parente y así sucesivamente, se puede recorrer en un tiempo proporcional al número de nodos del camino. También es posible manejar la operación ETIQUETA agregando otro arreglo E , tal que $E[i]$ sea la etiqueta del nodo i , o haciendo que los elementos del arreglo L sean registros que contengan un entero (cursor) y una etiqueta.

Ejemplo 3.6. El árbol de la figura 3.10(a) tiene la representación por apuntadores al parente que se muestra en el arreglo A de la figura 3.10(b). □

que cuando se llama a ORD_PRE(n), las llamadas a procedimientos activas, y por tanto la pila de registros de activación, corresponden a las llamadas a ORD_PRE para todos los antecesores de n . Así, el procedimiento no recursivo de orden previo, como el ejemplo de la sección 2.6, modela con exactitud la forma de implantar el procedimiento recursivo.

La representación por apuntadores al padre no facilita las operaciones que requieren información de los hijos. Dado un nodo n , resulta caro determinar sus hijos o su altura. Además, esta representación no especifica el orden de los hijos de un nodo. Por tanto, las operaciones como HIJO_MAS_IZQ y HERMANO_DER no están bien definidas. Se puede imponer un orden artificial, por ejemplo, numerando los hijos de cada nodo después de numerar al padre, y numerando los hijos en orden

```

procedure ORD_PRE_NO_REC ( A: ARBOL );
  { recorrido en orden previo no recursivo del árbol A }
  var
    m: nodo; { temporal }
    P: PILA; { pila de nodos que contiene el camino de la raíz al padre TO-
              PE(P) del nodo "actual" m }
  begin
    { asigna valor inicial }
    ANULA(P);
    m := RAIZ(A);
    while true do
      if m <> Λ then begin
        print(ETIQUETA(m, A));
        METE(m, P);
        { explora el hijo más a la izquierda de m }
        m := HIJO_MAS_IZQ(m, A)
      end
      else begin
        { la exploración del camino contenido en la pila está completa }
        if VACIA(P) then
          return;
        { explora el hermano derecho del nodo al tope de la pila }
        m := HERMANO_DER(TOPE(P), A);
        SACAR(P)
      end
    end; { ORD_PRE_NO_REC }
  
```

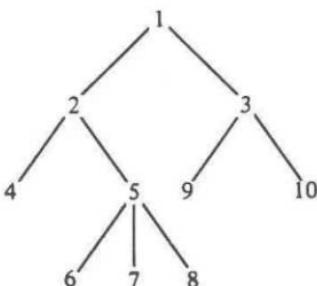
Fig. 3.9. Procedimiento no recursivo de listado en orden previo.

ascendente, de izquierda a derecha. Con esa suposición se ha escrito la función HERMANO_DER de la figura 3.11, para los tipos nodo y ARBOL que se definen como sigue:

```

type
  nodo = integer;
  ARBOL = array [1..nodos_máx] of nodo;
  
```

Para esta representación se supone que el nodo nulo Λ se representa por 0.



(a) un árbol

A	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	2	5	5	5	3	3

(b) representación por apuntadores al padre.

Fig. 3.10. Un árbol y su representación mediante apuntadores al padre.

```

function HERMANO_DER ( n: nodo; A: ARBOL ) : nodo;
  { devuelve el hermano derecho del nodo n del árbol A }
  var
    i, padre: nodo;
  begin
    padre := A[n];
    for i := n + 1 to nodos_máx do
      { busca un nodo después de n que tenga el mismo padre }
      if A[i] = padre then
        return (i);
    return (0) { devuelve el nodo nulo si no se encontró hermano
                derecho }
  end; { HERMANO_DER }
  
```

Fig. 3.11. Operación HERMANO_DER usando la representación por arreglos.

Representación de árboles mediante listas de hijos

Una manera importante y útil de representar árboles consiste en formar una lista de los hijos de cada nodo. Las listas se pueden representar con cualquiera de los métodos sugeridos en el capítulo 2, pero como el número de hijos que cada nodo puede tener es variable, la representación por medio de listas enlazadas es a menudo más apropiada.

La figura 3.12 sugiere cómo se podría representar el árbol de la figura 3.10(a). Hay un arreglo de celdas de encabezamiento indizadas por nodos, los cuales se supone

que están numerados del 1 al 10. Cada encabezamiento apunta a una lista enlazada de «elementos» que son nodos del árbol. Los elementos de la lista encabezada por *encabezamiento[i]* son los hijos del nodo *i*; por ejemplo, 9 y 10 son los hijos de 3.

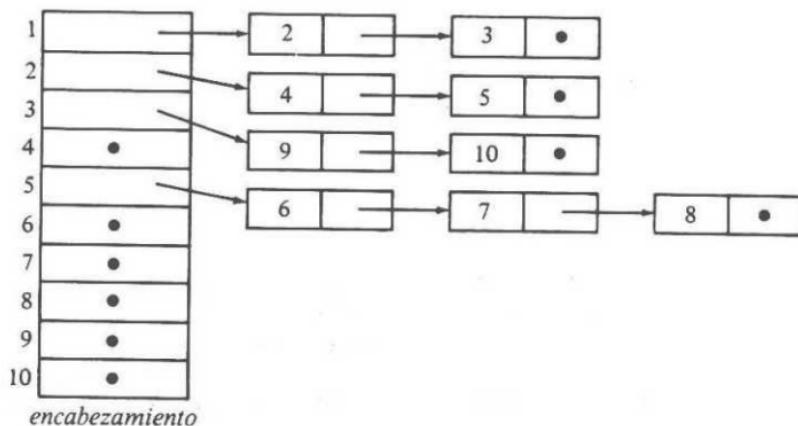


Fig. 3.12. Representación de un árbol por listas enlazadas.

Primero se desarrollarán las estructuras de datos necesarias en función de un tipo de datos abstracto LISTA (de nodos), y luego se dará una realización particular para las listas y se verá cómo las abstracciones compaginan entre sí. Despues se mostrarán algunas simplificaciones posibles. Se empieza con las siguientes declaraciones de tipo:

```

type
    nodo = integer;
    LISTA = { definición apropiada para listas de nodos };
    posición = { definición apropiada para posiciones en listas };
    ARBOL = record
        encabezamiento: array [1..nodos_máx] of LISTA;
        etiquetas: array [1..nodos_máx] of tipo_etiqueta;
        raíz: nodo
    end;

```

Se supone que la raíz de cada árbol está almacenada explícitamente en el campo *raíz*, y que se usa 0 para representar el nodo nulo.

La figura 3.13 muestra el código para la operación HIJO_MAS_IZQ. Como ejercicio, sería útil escribir el código para las restantes operaciones.

```

function HIJO_MAS_IZQ ( n: nodo; A: ARBOL ) : nodo;
    { devuelve el hijo más a la izquierda del nodo n del árbol A }
    var
        L: LISTA; { abreviatura para la lista de hijos de n }

```

```

begin
  L := A.encabezamiento[n];
  if VACIA(L) then { n es una hoja }
    return (0)
  else
    return (RECUPERA(PRIMERO(L), L))
end; { HIJO_MAS_IZQ }

```

Fig. 3.13. Función para hallar el hijo más a la izquierda.

Ahora se elegirá una implantación particular para las listas, en la cual tanto LISTA como posición sean enteros, empleados como cursores a un arreglo *espacio_celdas* de registros:

```

var
  espacio_celdas: array [1..nodos_máx] of record
    nodo: integer;
    sig: integer
  end;

```

Para simplificar, no se insistirá en que las listas de hijos tienen celdas de encabezamiento. Más bien, se hará que *A.encabezamiento[n]* apunte directamente a la primera celda de la lista, como se sugiere en la figura 3.12. La figura 3.14(a) muestra la función HIJO_MAS_IZQ de la figura 3.13, escrita otra vez para esta realización en particular. La figura 3.14(b) muestra el operador PADRE, que es más difícil de escribir usando esta representación de listas, puesto que se requiere una búsqueda en todas ellas para determinar en cuál aparece un nodo dado.

```

function HIJO_MAS_IZQ ( n: nodo; A: ARBOL ) : nodo;
  { devuelve el hijo más a la izquierda del nodo n del árbol A }
  var
    L: integer; { un cursor al principio de la lista de hijos de n }
  begin
    L := A.encabezado[n];
    if L = 0 then { n es una hoja }
      return (0)
    else
      return (espacio_celdas[L].nodo)
  end; { HIJO_MAS_IZQ }

```

(a) La función HIJO_MAS_IZQ.

```

function PADRE ( n: nodo; A: ARBOL ) : nodo;
  { devuelve el padre del nodo n del árbol A }

```

```

var
  p: nodo; { recorre los posibles padres de n }
  i: posición; { recorre la lista de hijos de p }
begin
  for p := 1 to nodos_máx do begin
    i := A.encabezado[p];
    while i <> 0 do { verifica si n está entre los hijos de p }
      if espacio_celdas[i].nodo = n then
        return (p)
      else
        i := espacio_celdas[i].sig
  end;
  return (0) { devuelve el nodo nulo si no se encontró padre }
end; { PADRE }

```

(b) La función PADRE.

Fig. 3.14. Dos funciones que emplean representación por listas enlazadas de árboles.

Representación «hijo más a la izquierda — hermano derecho»

La estructura de datos descrita antes adolece, entre otras cosas, de no ser adecuada para crear árboles grandes a partir de árboles más chicos por medio de los operadores CREA*i*. La razón es que, a pesar de que todos los árboles comparten *espacio_celdas* para las listas ligadas de hijos, cada uno tiene su arreglo propio de encabezamientos correspondiente a sus nodos. Por ejemplo, si se deseara obtener CREA2(*v*, *A*₁, *A*₂), se tendría que copiar *A*₁ y *A*₂ en un tercer árbol y agregar un nuevo nodo con etiqueta *v* y dos hijos: las raíces de *A*₁ y *A*₂.

Si se desea construir árboles a partir de otros menores, es mejor que las representaciones de los nodos de todos los árboles compartan un área. La extensión lógica de la figura 3.12 para lograr esto consiste en reemplazar el arreglo de encabezamientos por un arreglo *espacio_nodos* que contenga registros con dos campos *etiqueta* y *encabezamiento*. Este arreglo contendrá los encabezamientos de la totalidad de nodos de todos los árboles. Así pues, se declara:

```

var
  espacio_nodos: array [1..nodos_máx] of record
    etiqueta: tipo_etiqueta;
    encabezamiento: integer { cursor a espacio_celdas }
  end;

```

Entonces, puesto que ya no hay nodos con nombres 1, 2, ..., *n*, sino que los nodos están representados por índices arbitrarios de *espacio_nodos*, no es posible que el campo *nodo* de *espacio_celdas* represente el «número» de un nodo; en cambio, *nodo* es ahora un cursor a *espacio_nodos*, que indica la posición de ese nodo. El tipo ARBOL es sencillamente un cursor a *espacio_nodos* que indica la posición de la raíz.

Ejemplo 3.7. La figura 3.15(a) muestra un árbol, y la figura 3.15(b), la estructura de datos correspondiente, en la cual los nodos con etiquetas *A*, *B*, *C*, y *D* se han colocado de manera arbitraria en las posiciones 10, 5, 11 y 2 de *espacio_nodos*. También se ha hecho una elección arbitraria de las celdas de *espacio_celdas* utilizadas para las listas de hijos. □

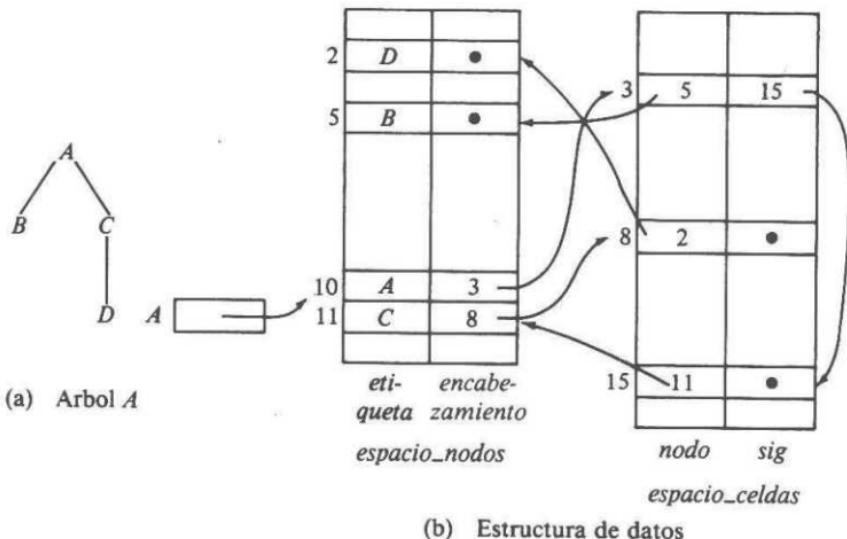


Fig. 3.15. Otra estructura de listas enlazadas para árboles.

La estructura de la figura 3.15(b) es adecuada para combinar árboles por medio de operaciones CREAr. Sin embargo, esta estructura se puede simplificar de modo significativo. Primero, se observa que las cadenas de apuntadores *sig* de *espacio_celdas* son en realidad apuntadores a hermanos derechos.

Usando estos apuntadores, se pueden obtener los hijos más a la izquierda como sigue. Supóngase que *espacio_celdas[i].nodo = n*. (Recuérdese que el «nombre» de un nodo, en oposición a su etiqueta, es en realidad su índice en *espacio_nodos*, que es el valor de *espacio_celdas[i].nodo*.) Entonces, *espacio_nodos[n].encabezamiento* indica la celda ocupada por el hijo más a la izquierda de *n* en *espacio_celdas*, en el sentido de que el campo *nodo* de esa celda es el nombre de tal nodo en *espacio_nodos*.

Se pueden simplificar las cosas si se identifica un nodo no por su índice en *espacio_nodos*, sino por el índice de la celda de *espacio_celdas* que lo representa como hijo. Entonces los apuntadores *sig* de *espacio_celdas* apuntarán en realidad a hermanos derechos, y la información contenida en el arreglo *espacio_nodos* se podrá guardar introduciendo un campo *hijo_más_izq* en *espacio_celdas*. El tipo de datos ARBOL se convertirá en un entero empleado como cursor a *espacio_celdas* para indicar la raíz del árbol. Se declara *espacio_celdas* para tener la siguiente estructura:

```

var
  espacio_celdas: array [1..nodos_máx] of record
    etiqueta: tipo_etiqueta;
    hijo_más_izq: integer;
    hermano_der: integer
  end;

```

Ejemplo 3.8. El árbol de la figura 3.15(a) se representa con la nueva estructura de datos en la figura 3.16. Para los nodos se han utilizado los mismos índices arbitrarios de la figura 3.15(b). □

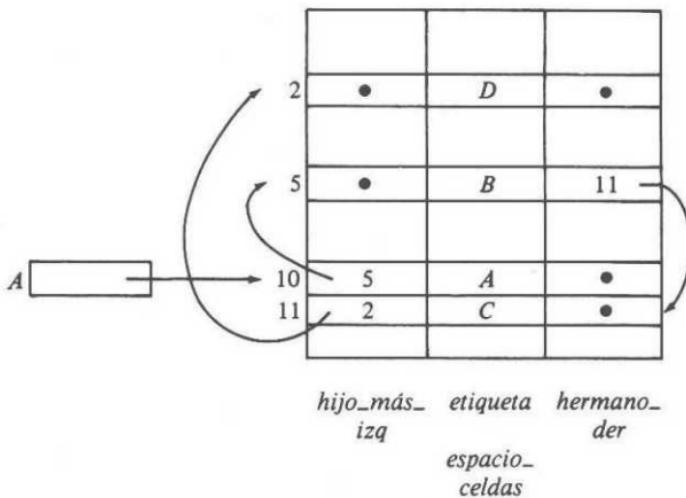


Fig. 3.16. Representación "hijo más a la izquierda-hermano derecho" de un árbol.

En la representación hijo más a la izquierda-hermano derecho, todas las operaciones son fáciles de realizar, excepto PADRE. Esta requiere de una búsqueda en todo *espacio_celdas*. Si es necesario realizar la operación PADRE eficientemente, se puede agregar a *espacio_celdas* un cuarto campo que indique al padre de un nodo directamente.

Como ejemplo de una operación con árboles escrita para utilizar una estructura hijo más a la izquierda-hermano derecho, como la de la figura 3.16, se da la función CREA2 en la figura 3.17. Se supone que las celdas no utilizadas están enlazadas en una lista de espacio disponible, encabezada por *disp*, y que este enlace utiliza los campos hermano derecho. La figura 3.18 muestra los apuntadores antiguos (con líneas de trazo continuo) y los nuevos (con líneas punteadas).

```

function CREA2 ( v: tipo_etiqueta; A1, A2: integer ) : integer;
  { devuelve un nuevo árbol con raíz v, que tiene A1 y A2 como subárboles }
  var
    temp: integer; { guarda el índice de la primera celda disponible
      para la raíz del nuevo árbol }
  begin
    temp := disp;
    disp := espacio_celdas[disp].hermano_der;
    espacio_celdas[temp].hijo_más_izq := A1;
    espacio_celdas[temp].etiqueta := v;
    espacio_celdas[temp].hermano_der := 0;
    espacio_celdas[A1].hermano_der := A2;
    espacio_celdas[A2].hermano_der := 0; { no es necesario; ese
      campo debería valer 0, pues la celda era una raíz }
    return (temp)
  end; { CREA2 }

```

Fig. 3.17. La función CREA2.

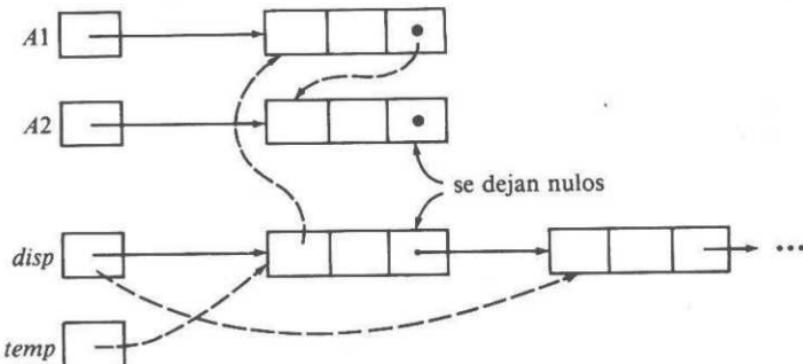


Fig. 3.18. Cambios de apuntadores producidos por CREA2.

Como alternativa, se puede usar menos espacio, pero más tiempo, si se coloca en el campo hermano derecho del hijo más a la derecha un apuntador al padre, en lugar del apuntador nulo que, de lo contrario, estaría ahí. Para evitar confusiones, se necesitaría en cada celda un bit que indicase si el campo hermano derecho tiene un apuntador al hermano derecho o al padre.

Así, dado un nodo, se encontraría su padre siguiendo apuntadores hermano derecho hasta encontrar uno que fuera apuntador al padre. Como todos los hermanos tienen el mismo parente, se encontraría el camino al parente del nodo del que se partió. El tiempo necesario para encontrar el parente de un nodo en esta representación depende del número de hermanos que tenga el nodo.

3.4 Arboles binarios

El árbol que se definió en la sección 3.1 recibe a veces el nombre de árbol *ordenado orientado*, porque los hijos de cada nodo están ordenados de izquierda a derecha, y porque existe un camino orientado (un camino en una dirección particular) de cada nodo a sus descendientes. Otro concepto útil y muy distinto de «árbol» es el de *árbol binario*, que define un árbol vacío o un árbol en el que cada nodo no tiene hijos, tiene un *hijo izquierdo*, un *hijo derecho* o un hijo izquierdo y un hijo derecho. El hecho de que cada hijo se designe en este caso como hijo izquierdo o como hijo derecho hace que un árbol binario sea distinto de un árbol ordenado orientado, tal como se definió en la sección 3.1.

Ejemplo 3.9. Si se adopta la convención de que los hijos izquierdos se dibujan hacia la izquierda de su padre y los hijos derechos hacia la derecha, las figuras 3.19(a) y (b) representan dos árboles binarios distintos, aunque ambos «se parecen» al árbol ordinario (ordenado orientado) de la figura 3.20. Sin embargo, se hace hincapié en que los de la figura 3.19 no son iguales entre sí ni son en ningún sentido iguales al de la figura 3.20, por la sencilla razón de que los árboles binarios no son directamente comparables con los árboles ordinarios. Por ejemplo, en la figura 3.19(a), 2 es el hijo izquierdo de 1, que no tiene hijo derecho; en cambio, en la figura 3.19(b), 1 no tiene hijo izquierdo, pero tiene a 2 como hijo derecho. En ambos árboles binarios, 3 es el hijo izquierdo de 2 y su hijo derecho es 4. □

Los listados en órdenes previo y posterior de un árbol binario son similares a los de un árbol ordinario dados en la página 79. El listado en orden simétrico de los nodos de un árbol binario con raíz n , subárbol izquierdo A_1 y subárbol derecho A_2 , es el listado en orden simétrico de A_1 seguido de n y del listado en orden simétrico de A_2 . Por ejemplo, 35241 es el listado en orden simétrico de los nodos del árbol de la figura 3.19(a).

Representación de árboles binarios

Una forma conveniente de estructurar datos para representar un árbol binario consiste en dar a sus nodos los nombres 1, 2, ..., n , y utilizar un arreglo de registros declarado como

```

var
    espacio_celdas: array [1..nodos_máx] of record
        hijo_izq: integer;
        hijo_der: integer
    end;

```

La idea es que $espacio_celdas[i].hijo_izq$ sea el hijo izquierdo del nodo i , y que suceda lo mismo con $hijo_der$. Un valor 0 en cualquiera de esos campos indicará la ausencia de un hijo.

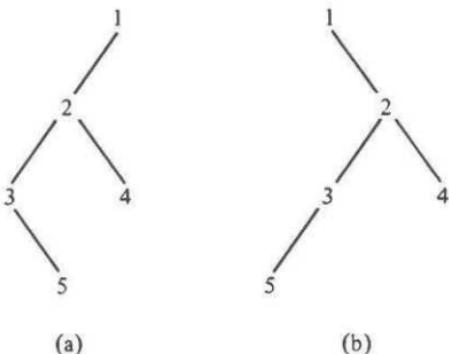


Fig. 3.19. Dos árboles binarios.

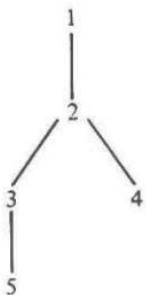


Fig. 3.20. Un árbol «ordinario».

Ejemplo 3.10. El árbol binario de la figura 3.19(a) puede representarse como se muestra en la figura 3.21. □

Un ejemplo: códigos de Huffman

Para exemplificar la forma en que los árboles binarios pueden usarse como estructuras de datos, se verá un problema particular que consiste en la construcción de «códigos de Huffman». Supóngase que se tienen mensajes compuestos por secuencias de caracteres. En cada mensaje, los caracteres son independientes entre sí, y aparecen con una probabilidad conocida en cualquier posición dada del mensaje; las probabilidades son las mismas para todas las posiciones. Como ejemplo, supóngase que se tiene un mensaje constituido por los caracteres *a*, *b*, *c*, *d*, *e*, los cuales aparecen con probabilidades 0.12, 0.4, 0.15, 0.08 y 0.25, respectivamente.

Se desea codificar cada carácter por medio de una sucesión de ceros y unos, de manera que ningún código de un carácter sea prefijo del código de otro carácter. Esta *propiedad de prefijos* permitiría decodificar una cadena de ceros y unos, eliminando repetidas veces de la cadena los prefijos que sean códigos de caracteres.

	<i>hijo_izq</i>	<i>hijo_der</i>
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

Fig. 3.21. Representación de un árbol binario.

Ejemplo 3.11. La figura 3.22 muestra dos códigos posibles para el alfabeto de cinco símbolos que se está utilizando. Es evidente que el código 1 tiene la propiedad de prefijos, puesto que ninguna sucesión de tres bits puede ser prefijo de otra sucesión con igual número de bits. El algoritmo de decodificación para el código 1 es sencillo. Basta tomar tres bits de cada vez y traducir cada grupo de éstos en un carácter. Por supuesto, las sucesiones 101, 110 y 111 no pueden presentarse si la cadena de bits codifica realmente caracteres de acuerdo con el código 1. Por ejemplo, si se recibe el mensaje codificado 001010011, se puede saber que el mensaje original era *bcd*.

Símbolo	Probabilidad	Código 1	Código 2
<i>a</i>	0.12	000	000
<i>b</i>	0.40	001	11
<i>c</i>	0.15	010	01
<i>d</i>	0.08	011	001
<i>e</i>	0.25	100	10

Fig. 3.22. Dos códigos binarios.

También es fácil verificar que el código 2 tiene la propiedad de prefijos. Se puede decodificar una cadena de bits tomando repetidas veces los prefijos que sean códigos de caracteres y quitándolos, tal como se hizo con el código 1. La única diferencia consiste en que no se puede «rebanar» la sucesión completa de bits de una sola vez, porque tomar dos o tres bits para un carácter depende de los bits. Por ejemplo, si una cadena comienza con 1101001, se puede asegurar de nuevo que los caracteres codificados fueron *bcd*. Los dos primeros bits, 11, deben proceder de *b*, por tanto, es posible quitarlos y preocuparse sólo de 01001. Se deduce, entonces, que los bits 01 proceden de *c*, y así sucesivamente. □

El problema que se enfrenta es: dado un conjunto de caracteres y sus probabilidades, encontrar un código con la propiedad de prefijos, tal que la longitud media del código de un carácter sea mínima. La razón por la que se desea minimizar la longitud media de un código es que ello permite comprimir la longitud de un mensaje tipo. Cuanto más corta sea la longitud media de un carácter, tanto más corta será la longitud de un mensaje codificado. Por ejemplo, el código 1 tiene una longitud

media de código de 3. Esta longitud se obtiene multiplicando las longitudes de los códigos de todos los símbolos por sus respectivas probabilidades de aparición. El código 2 tiene una longitud media de 2.2, puesto que los símbolos *a* y *d*, que en conjunto aparecen el 20% de las veces, tienen códigos de longitud 3, y los otros símbolos, códigos de longitud 2.

¿Se puede obtener un código mejor que el 2? Una respuesta completa a esta pregunta consistiría en exhibir un código con la propiedad de prefijos que tuviera una longitud media de 2.15. Ese sería el mejor código posible para las probabilidades dadas de aparición de los símbolos. Una técnica que permite hallar códigos de prefijos óptimos recibe el nombre de *algoritmo de Huffman*. Funciona seleccionando dos caracteres, como *a* y *b*, que tengan las probabilidades más pequeñas, y reemplazándolos con un único carácter (imaginario), por ejemplo *x*, cuya probabilidad de aparición sea la suma de las probabilidades de *a* y *b*. Despues se encuentra un código de prefijos óptimo para este conjunto más pequeño de caracteres, usando recursivamente el mismo procedimiento. Los códigos para el conjunto original de caracteres se obtienen usando el código para *x* seguido de un cero, como código para *a*, y seguido de un uno, como código para *b*.

Se pueden considerar los códigos de prefijos como caminos en árboles binarios. Piénsese que seguir el camino de un nodo a su hijo izquierdo es como agregarle un cero a un código, y que tomar el camino hacia el hijo derecho es como agregarle un uno. Si se etiquetan las hojas de un árbol binario con los caracteres representados, se puede simbolizar cualquier código de prefijos por medio de un árbol binario. La propiedad de prefijos garantiza que ningún carácter puede tener un código que corresponda a un nodo interior y, reciprocamente, etiquetar las hojas de cualquier árbol binario con caracteres da un código con la propiedad de prefijos de esos caracteres.

Ejemplo 3.12. Los árboles binarios correspondientes a los códigos 1 y 2 de la figura 3.22 se muestran en la figura 3.23(a) y (b), respectivamente. □

Se obtendrá el algoritmo de Huffman mediante un *bosque* o conjunto de árboles, cada uno de los cuales tendrá sus hojas etiquetadas con los caracteres cuyos códigos se desea seleccionar, y sus raíces etiquetadas con la suma de las probabilidades de todas las etiquetas de las hojas. Estas sumas se denominan *pesos* de los árboles. Inicialmente, cada carácter estará en un árbol de un solo nodo, y cuando el algoritmo termine se tendrá un solo árbol con todos los caracteres como hojas. En este árbol, el camino de la raíz a cualquier hoja representará el código para la etiqueta de esa hoja, de acuerdo con el esquema izquierda = 0, derecha = 1 de la figura 3.23.

El paso fundamental del algoritmo consiste en seleccionar los dos árboles del bosque que tengan los pesos más bajos (decidiendo arbitrariamente en caso de empate), y en combinar esos dos árboles en uno solo, cuyo peso sea la suma de los pesos de ambos. Para combinar los árboles se crea un nuevo nodo que queda como raíz que tiene las raíces de los dos árboles dados como sus hijos izquierdo y derecho (sin importar cuál sea uno y otro). Este proceso continúa hasta que quede un solo árbol representativo del código que, para las probabilidades dadas, tenga la menor longitud media de código posible.

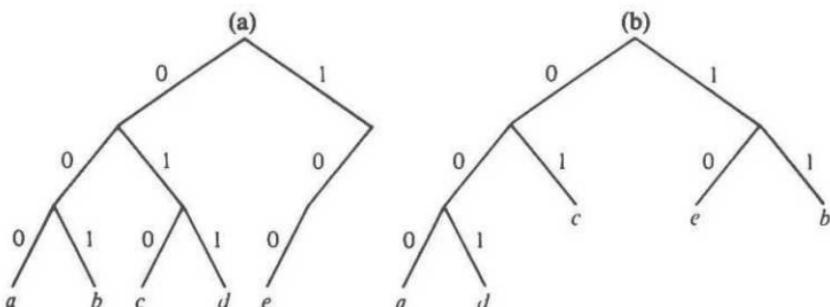


Fig. 3.23. Arboles binarios que representan códigos con la propiedad de prefijos.

Ejemplo 3.13. La secuencia de pasos seguidos para los caracteres y probabilidades del ejemplo que se está desarrollando se muestra en la figura 3.24. En la parte (e), se puede ver que las palabras de código para *a*, *b*, *c*, *d* y *e* son 1111, 0, 110, 1110 y 10. En este ejemplo, hay un solo árbol no trivial, pero, en general, puede haber varios. Por ejemplo, si las probabilidades de *b* y *e* fueran 0.33 y 0.32, entonces, después del paso (c) de la figura habría que combinar *b* y *e*, en vez de añadir *e* al árbol más grande, como se hizo en el paso (d). □

Se describen ahora las estructuras de datos necesarias. Primero, se usa un arreglo ARBOL de registros del tipo

```

record
    hijo_izq: integer;
    hijo_der: integer;
    padre: integer
end

```

para representar los árboles binarios. Los apunadores al padre facilitan el hallazgo de caminos de las hojas a la raíz, que permiten descubrir los códigos de los caracteres. En segundo término, se usa un arreglo ALFABETO de registros del tipo

```

record
    símbolo: char;
    probabilidad: real;
    hoja: integer { cursor a un árbol }
end

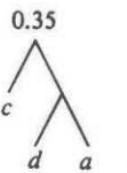
```

0.12	0.40	0.15	0.08	0.25
•	•	•	•	•
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>

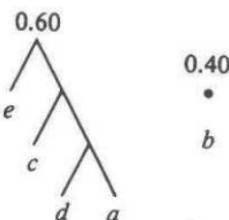
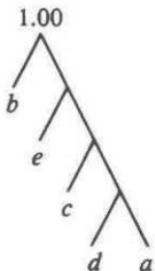
(a) Inicial

0.20	0.40	0.15	0.25
•	•	•	•
<i>d</i>	<i>a</i>	<i>b</i>	<i>c</i>

(b) Combina *a* y *d*

(c) Combina a, d con c

0.40 0.25
• •
b e

(d) Combina a, c, d con e 

(e) Arbol final

Fig. 3.24. Pasos para la construcción de un árbol de Huffman.

para asociar a cada símbolo del alfabeto a codificar la hoja que le corresponda. Este arreglo también registra la probabilidad de cada carácter. En tercer lugar, se necesita un arreglo *BOSQUE* de registros que representen los árboles mismos. El tipo de esos registros es

record

peso: real;

raíz: integer { cursor a un árbol }

end

Los valores iniciales de todos los arreglos correspondientes a los datos de la figura 3.24(a) se muestran en la figura 3.25. En la figura 3.26 se muestra un esbozo del programa para construir el árbol de Huffman.

1	0.12	1
2	0.40	2
3	0.15	3
4	0.08	4
5	0.25	5

peso raíz

1	a	0.12	1
2	b	0.40	2
3	c	0.15	3
4	d	0.08	4
5	e	0.25	5

símbolo- proba- hoja
lo bilitad

ALFABETO

1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

hijo_ hijo_ padre
izq der

ARBOL

Fig. 3.25. Contenido inicial de los arreglos.

```

(1) while haya más de un árbol en el bosque do
begin
(2)     i := índice del árbol de menor peso en BOSQUE;
(3)     j := índice del segundo árbol de menor peso en BOSQUE;
(4)     crea un nuevo nodo con hijo izquierdo BOSQUE[i].raíz
           e hijo derecho BOSQUE[j].raíz;
(5)     reemplaza el árbol i de BOSQUE por un árbol cuya
           raíz es el nuevo nodo y cuyo peso es
           BOSQUE[i].peso + BOSQUE[j].peso;
(6)     elimina el árbol j de BOSQUE
end;

```

Fig. 3.26. Esbozo de la construcción de árboles de Huffman.

Para aplicar la línea (4) de la figura 3.26, la cual incrementa el número de celdas empleadas del arreglo *ARBOL*, y las líneas (5) y (6), que disminuyen el número de celdas utilizadas de *BOSQUE*, se introducen los cursores *últ_arbol* y *últ_nodo*, que apuntan hacia *BOSQUE* y *ARBOL*, respectivamente. Se supone que las celdas 1 a *últ_arbol* de *BOSQUE* y 1 a *últ_nodo* de *ARBOL* están ocupadas †. Se supone también que los arreglos de la figura 3.25 tienen declaradas sus longitudes, pero en lo que sigue se omitirán las comparaciones entre esos límites y los valores de los cursores.

```

procedure más_ligeros ( var menor, segundo: integer );
{ asigna menor y segundo a los índices en BOSQUE de los dos árboles
  de menor peso; se supone que últ_arbol ≥ 2. }
var
  i: integer;
begin { asigna valor inicial a menor y a segundo teniendo en cuenta
        los dos primeros árboles }
  if BOSQUE[1].peso <= BOSQUE[2].peso then
    begin menor := 1; segundo := 2 end
  else
    begin menor := 2; segundo := 1 end;
{ Ahora recorre i desde 3 a últ_arbol. En cada iteración, menor es
  el árbol de menor peso entre los i primeros de BOSQUE; y
  segundo, el siguiente en peso entre los mismos }
  for i := 3 to últ_arbol do
    if BOSQUE[i].peso < BOSQUE[menor].peso then
      begin segundo := menor; menor := i end
    else if BOSQUE[i].peso < BOSQUE[segundo].peso then
      segundo := i
  end; { más_ligeros }

```

† En la fase de lectura, aquí omitida, también se requiere un cursor para el arreglo *ALFABETO*, mientras se llena con los símbolos y sus probabilidades.

```

function crea ( árbol_izq, árbol_der; integer ) : integer;
  { devuelve un nuevo nodo cuyos hijos son BOSQUE[árbol_izq].raíz
    y BOSQUE[árbol_der].raíz }
begin
  últ_nodo := últ_nodo + 1;
  { la celda para el nuevo nodo es ARBOL[últ_nodo] }
  ARBOL[últ_nodo].hijo_izq := BOSQUE[árbol_izq].raíz;
  ARBOL[últ_nodo].hijo_der := BOSQUE[árbol_der].raíz;
  { ahora asigna apuntadores padre al nuevo nodo y a sus hijos }
  ARBOL[últ_nodo].padre := 0;
  ARBOL[BOSQUE[árbol_izq].raíz].padre := últ_nodo;
  ARBOL[BOSQUE[árbol_der].raíz].padre := últ_nodo;
  return (últ_nodo)
end; { crea }

```

Fig. 3.27. Dos procedimientos.

La figura 3.27 muestra dos procedimientos útiles; el primero es una realización de las líneas (2) y (3) de la figura 3.26, en las que se seleccionan los índices de los dos árboles de menor peso. El segundo es la orden *crea(n_1, n_2)*, que crea un nuevo nodo y hace que n_1 y n_2 se conviertan en sus hijos izquierdo y derecho.

Ahora es posible describir con más detalles los pasos de la figura 3.26. En la figura 3.28 se puede ver un procedimiento *Huffman*, que no tiene parámetros sino que trabaja con las estructuras globales de la figura 3.25.

```

procedure Huffman;
var
  i, j: integer; { los dos árboles de menor peso en BOSQUE }
  nueva_raíz: integer;
begin
  while últ_arbol > 1 do begin
    más_ligeros(i, j);
    nueva_raíz := crea(i, j);
    { Ahora reemplaza el árbol i por el árbol cuya raíz es nueva_raíz }
    BOSQUE[i].peso := BOSQUE[i].peso + BOSQUE[j].peso;
    BOSQUE[i].raíz := nueva_raíz;
    { luego reemplaza el árbol j, que ya no se necesita,
      por últ_arbol, y acorta BOSQUE en uno }
    BOSQUE[j] := BOSQUE[últ_arbol];
    últ_arbol := últ_arbol - 1
  end
end; { Huffman }

```

Fig. 3.28. Algoritmo de Huffman.

La figura 3.29 muestra la estructura de datos de la figura 3.25 después de que *últ_árbol* ha quedado reducido a 3; es decir, cuando el bosque tiene el aspecto de la figura 3.24(c).

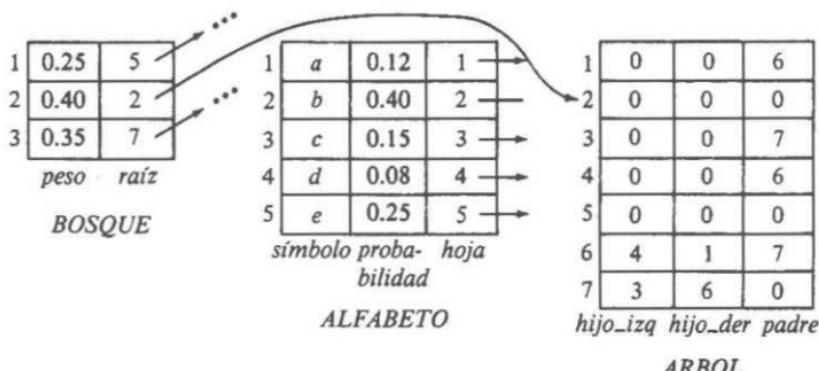


Fig. 3.29. Estructura de datos de un árbol después de dos iteraciones.

Después de terminada la ejecución del algoritmo, el código de cada símbolo se puede determinar como sigue. Encuéntrese el símbolo en el campo *símbolo* del arreglo *ALFABETO*. Sigase el campo *hoja* del mismo registro para obtener un registro del arreglo *ARBOL*; este registro corresponde a la hoja asociada al símbolo en cuestión. Sígase repetidamente el apuntador *padre* desde el registro «actual», por ejemplo, el del nodo *n*, al registro en el arreglo *ARBOL* de su padre *p*. Hágase esto sin olvidar el nodo *n*, de manera que sea posible examinar los apuntadores *hijo_izq* e *hijo_der* de *p* y determinar cuál de ellos apunta a *n*. En el primer caso, imprímase 0; en el segundo, 1. La secuencia de bits impresa de esta forma será el código del símbolo con el orden de los bits invertido. Si se desea imprimir los bits en el orden correcto, se pueden meter en una pila conforme se sube por el árbol, tras lo cual se sacan en forma reiterada bits de la pila, imprimiéndolos al mismo tiempo.

Realización de árboles binarios basada en apuntadores

En vez de utilizar cursores para apuntar a los hijos izquierdos y derechos (y a los padres, si se desea), es posible usar apuntadores genuinos de Pascal. Por ejemplo, se puede declarar

```

type
  nodo = record
    hijo_izq: ^nodo;
    hijo_der: ^nodo;
    padre: ^nodo
  end

```

Por otro lado, si se usara este tipo de nodos para un árbol binario, la función *crea* de la figura 3.27 se escribiría como se muestra en la figura 3.30.

```
function crea (árbol_izq, árbol_der: ↑ nodo) : ↑ nodo;
var
    raíz: ↑ nodo;
begin
    new(raíz);
    raíz↑.hijo_izq := árbol_izq;
    raíz↑.hijo_der := árbol_der;
    raíz↑.padre := 0;
    árbol_izq↑.padre := raíz;
    árbol_der↑.padre := raíz;
    return (raíz)
end; { crea }
```

Fig. 3.30. Realización de árboles binarios basada en apuntadores.

Ejercicios

3.1 Respóndase a las siguientes preguntas acerca del árbol de la figura 3.31.

- ¿Qué nodos son hojas?
- ¿Qué nodo es la raíz?
- ¿Cuál es el parente del nodo C?
- ¿Qué nodos son los hijos de C?
- ¿Qué nodos son los antecesores de C?
- ¿Qué nodos son los descendientes de E?
- ¿Cuáles son los hermanos derechos de D y E?
- ¿Qué nodos están a la izquierda y a la derecha de G?
- ¿Cuál es la profundidad del nodo C?
- ¿Cuál es la altura del nodo C?

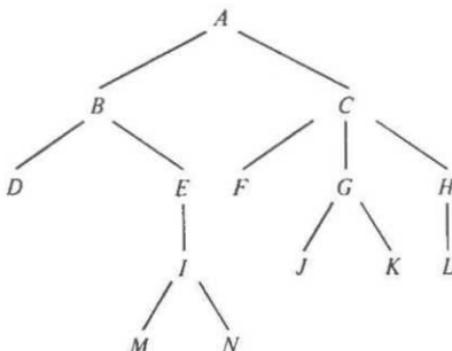


Fig. 3.31. Un árbol.

- 3.2 ¿Cuántos caminos distintos de longitud 3 hay en el árbol representado en la figura 3.31?
- 3.3 Escríbanse programas para calcular la altura de un árbol, usando cada una de las tres representaciones de árboles de la sección 3.3.
- 3.4 Lístense los nodos de la figura 3.31 en
- orden previo,
 - orden posterior, y
 - orden simétrico.
- 3.5 Muéstrese que si m y n son dos nodos distintos del mismo árbol, exactamente una de las siguientes afirmaciones es verdadera:
- m está a la izquierda de n
 - m está a la derecha de n
 - m es un antecesor propio de n
 - m es un descendiente propio de n .
- 3.6 Márquese la intersección de la fila i con la columna j si las situaciones representadas por esa fila y esa columna pueden ocurrir simultáneamente.

	$ord_pre(n)$ $< ord_pre(m)$	$ord_sim(n)$ $< ord_sim(m)$	$ord_post(n)$ $< ord_post(m)$
n está a la izquierda de m			
n está a la derecha de m			
n es un antecesor propio de m			
n es un descendiente propio de m			

Por ejemplo, póngase una marca en la intersección de la fila 3 y la columna 2 si se cree que n puede ser a la vez un antecesor propio de m y preceder a m en orden simétrico.

- 3.7 Supóngase que se tienen los arreglos $ORD_PRE[n]$, $ORD_SIM[n]$ y $ORD_POST[n]$ que dan las posiciones en los órdenes previo, simétrico y posterior, respectivamente, de cada nodo n de un árbol. Describáse un algoritmo que diga si un nodo i es un antecesor o no de un nodo j , para cualquier par de nodos i, j . Explíquese cómo trabaja el algoritmo.
- *3.8 Es posible probar si un nodo m es un descendiente propio o no de un nodo n , determinando si m precede a n en orden X, pero sucede a n en orden Y, donde X e Y se eligen {pre, post, sim}. Determínense todos los pares X, Y para los cuales se cumple esta afirmación.

- 3.9 Escribase un programa para recorrer un árbol binario en
- orden previo,
 - orden posterior,
 - orden simétrico.
- 3.10 El listado en *orden de nivel* de los nodos de un árbol lista primero la raíz, luego todos los nodos de profundidad 1, después todos los de profundidad 2, y así sucesivamente. Los nodos que estén a la misma profundidad se listan en orden de izquierda a derecha. Escribase un programa para listar los nodos de un árbol en orden de nivel.
- 3.11 Conviértase la expresión $((a + b) * c * (d + e) + f) * (g + h)$ en
- expresión prefija
 - expresión postfixia.
- 3.12 Dibújense representaciones de árbol para las expresiones
- $* a + b * c + d e$
 - $* a + * b + c d e$.
- 3.13 Sea A un árbol en el que cada nodo que no es hoja tiene dos hijos. Escribase un programa para convertir
- un listado en orden previo de A en un listado en orden posterior;
 - un listado en orden posterior de A en un listado en orden previo;
 - un listado en orden previo de A en un listado en orden simétrico.
- 3.14 Escribase un programa para evaluar expresiones aritméticas en
- orden previo
 - orden posterior.
- 3.15 Como modelo matemático, es posible definir un árbol binario como un TDA con estructura de árbol binario y con operaciones como HIJO-IZQ(n), HIJO-DER(n), PADRE(n) y NULO(n). Las tres primeras operaciones devuelven el hijo izquierdo, el hijo derecho y el padre del nodo n (Λ , si no existe), y la última devuelve verdadero si, y sólo si, n es Λ . Implántense estos procedimientos con la representación de árboles binarios de la figura 3.21.
- 3.16 Obténganse las siete operaciones con árboles de la sección 3.2, usando las siguientes realizaciones de árboles:
- apuntadores al padre,
 - listas de hijos,
 - apuntadores al hijo más a la izquierda, hermano derecho.
- 3.17 El *grado* de un nodo es el número de hijos que tiene. Muéstrese que en cualquier árbol binario el número de hojas es uno más que el número de nodos de grado dos.

- 3.18** Muéstrese que el máximo número de nodos de un árbol binario de altura h es $2^{h+1} - 1$. Un árbol binario de altura h con ese máximo número de nodos se denomina árbol binario *lleno*.
- *3.19** Supóngase que los árboles están realizados por medio de apuntadores al hijo más a la izquierda, al hermano derecho y al padre. Elabórense algoritmos no recursivos de recorrido en los órdenes previo, posterior y simétrico, que no utilicen «estados» ni una pila, como se hace en la figura 3.9.
- 3.20** Supóngase que los caracteres a, b, c, d, e , y f tienen probabilidades 0.07, 0.09, 0.12, 0.22, 0.23 y 0.27, respectivamente. Encuéntrese un código de Huffman óptimo y dibújese el árbol de Huffman correspondiente. ¿Cuál es la longitud media del código?
- *3.21** Supóngase que A es un árbol de Huffman y que la hoja del símbolo a tiene una profundidad mayor que la del símbolo b . Demuéstrese que la probabilidad del símbolo b no es menor que la del símbolo a .
- *3.22** Demuéstrese que el algoritmo de Huffman funciona, es decir, que produce un código óptimo para las probabilidades dadas. *Sugerencia.* Utilícese el ejercicio 3.21.

Notas bibliográficas

Berge [1958] y Harary [1969] analizan las propiedades matemáticas de los árboles. Knuth [1973] y Nievergelt [1974] contienen información adicional sobre los árboles de búsqueda binarios. Muchos de los trabajos sobre grafos y sus aplicaciones, a las que se hace referencia en el capítulo 6, también contienen material sobre árboles.

El algoritmo dado en la sección 3.4 para encontrar un árbol con mínima longitud ponderada de camino es de Huffman [1952]. Parker [1980] contiene algunas exploraciones más recientes en ese algoritmo.

4

Operaciones básicas con conjuntos

Los conjuntos constituyen la estructura básica que fundamenta las matemáticas. En el diseño de algoritmos, los conjuntos se utilizan como la base de una gran cantidad de tipos de datos abstractos y se han desarrollado muchas técnicas para la implantación de tipos de datos abstractos basadas en conjuntos. En este capítulo se hace una revisión de las operaciones básicas con conjuntos y se presentan algunas de las formas más sencillas de realización de conjuntos; se estudian los «diccionarios» y las «colecciones de prioridad», dos tipos de datos abstractos basados en el modelo de conjunto. Las realizaciones de estos tipos de datos abstractos se cubren en este capítulo y el siguiente.

4.1 Introducción a los conjuntos

Un conjunto es una colección de *miembros* (o *elementos*); cada miembro de un conjunto puede ser un conjunto, o un elemento primitivo que recibe el nombre de *átomo*. Todos los miembros de un conjunto son distintos, lo cual significa que ningún conjunto puede contener dos copias de un mismo elemento.

Por lo general, cuando los átomos se usan como herramientas para el diseño de algoritmos y estructuras de datos, son enteros, caracteres o cadenas, y todos los elementos de cualquier conjunto suelen ser del mismo tipo. A menudo se hará la suposición de que los átomos están ordenados linealmente por una relación que se designa mediante el símbolo «<», y se lee «menor que» o «precede a». Un *orden lineal* < sobre un conjunto S satisface dos propiedades:

1. Para cualesquiera a y b en S , sólo una de las afirmaciones $a < b$, $a = b$ o $b < a$ es verdadera.
2. Para todo a , b y c en S , si $a < b$ y $b < c$, entonces $a < c$ (transitividad).

Los enteros, reales, caracteres y cadenas tienen un orden lineal natural que en Pascal se designa con el símbolo <. Se puede definir un orden lineal sobre los objetos que consisten en conjuntos de objetos ordenados. Como ejercicio, determinese una forma de desarrollar un orden de este tipo. Por ejemplo, una pregunta que se debe contestar al desarrollar un orden lineal para un conjunto de enteros es si el conjunto que contiene los enteros 1 y 4 debe considerarse menor o mayor que el conjunto que contiene los enteros 2 y 3.

Notación de conjuntos

Un conjunto de átomos por lo general se representa encerrando sus miembros entre llaves; así, $\{1, 4\}$ denota el conjunto cuyos únicos miembros son 1 y 4. Debe recordarse que un conjunto no es una lista, a pesar de que se representen los conjuntos como si fueran listas; en el caso de los conjuntos, el orden en que se listan los elementos no es importante, y se pudo haber escrito $\{4, 1\}$ en lugar de $\{1, 4\}$. Se debe observar también que cada elemento aparece en un conjunto sólo una vez, de modo que $\{1, 4, 1\}$ no es un conjunto †.

Algunas veces se representan los conjuntos mediante una expresión de la forma

$$\{x \mid \text{proposición sobre } x\}$$

donde la proposición sobre x es un predicado que dice con exactitud qué se necesita para que un objeto arbitrario x pertenezca al conjunto. Por ejemplo, $\{x \mid x \text{ es un entero positivo y } x \leq 1000\}$ es otra manera de representar el conjunto $\{1, 2, \dots, 1000\}$, y $\{x \mid \text{para algún entero } y, x = y^2\}$ designa el conjunto de los cuadrados perfectos. Obsérvese que el conjunto de los cuadrados perfectos es infinito y no se puede representar listando sus miembros.

La relación fundamental en teoría de conjuntos es la de pertenencia, que se indica por el símbolo \in . Esto es, $x \in A$ significa que x pertenece a A o que el elemento x es un miembro del conjunto A , y puede ser un átomo u otro conjunto, pero A no puede ser un átomo. Se utiliza $x \notin A$ para señalar que « x no es un miembro de A ». Existe un conjunto especial que no tiene miembros, y se simboliza con \emptyset , que se denomina *conjunto nulo* o *conjunto vacío*. Obsérvese que \emptyset es un conjunto y no un átomo, a pesar de no tener miembros. La diferencia radica en que $x \in \emptyset$ es falso para toda x , mientras que si y es un átomo, entonces $x \in y$ no tiene sentido, porque más que una afirmación falsa, es un error de sintaxis.

Se dice que un conjunto A está *incluido* (*o contenido*) en un conjunto B , y se escribe $A \subseteq B$ o $B \supseteq A$, si todo miembro de A también es miembro de B . También se dice en este caso que A es un *subconjunto* de B o que B es un *supraconjunto* de A . Por ejemplo, $\{1, 2\} \subseteq \{1, 2, 3\}$, pero $\{1, 2, 3\}$ no es un subconjunto de $\{1, 2\}$ porque 3 pertenece al primero de estos conjuntos, pero no al segundo. Todo conjunto está incluido en sí mismo, y el conjunto vacío está incluido en todo conjunto. Dos conjuntos son iguales si cada uno de ellos está incluido en el otro, esto es, si sus miembros son los mismos. Un conjunto A es un *subconjunto propio* o un *supraconjunto propio* de otro B , si $A \neq B$ y $A \subset B$ o $A \supset B$, respectivamente.

Las operaciones elementales con conjuntos son unión, intersección y diferencia. Si A y B son conjuntos, entonces $A \cup B$, la *unión* de A y B , es el conjunto de los elementos que son miembros de A , de B o de ambos. La *intersección* de A y B , que se escribe $A \cap B$, es el conjunto de los elementos que pertenecen tanto a A como a B , y la *diferencia*, $A - B$, es el conjunto de los elementos de A que no pertenecen a B .

† Algunas veces el término *conjunto múltiple* se aplica a un «conjunto con repeticiones», es decir, el conjunto múltiple $\{1, 4, 1\}$ tiene (dos veces) 1 y (una vez) 4 como miembros. Un conjunto múltiple tampoco es una lista, por lo que el anterior se pudo haber escrito 4, 1, 1 ó 1, 1, 4.

Por ejemplo, si $A = \{a, b, c\}$ y $B = \{b, d\}$, entonces $A \cup B = \{a, b, c, d\}$, $A \cap B = \{b\}$ y $A - B = \{a, c\}$.

Tipos de datos abstractos basados en conjuntos

Se considerarán ahora los TDA que incorporan diversas operaciones con conjuntos. Algunas colecciones de estas operaciones reciben nombres especiales y tienen realizaciones muy eficientes. Algunas de las operaciones con conjuntos más comunes son las siguientes.

- 1.-3. Los procedimientos UNION(A, B, C), INTERSECCION(A, B, C) y DIFERENCIA(A, B, C) toman los argumentos A y B cuyos valores son conjuntos y asignan el resultado, $A \cup B$, $A \cap B$ o $A - B$, respectivamente, a la variable C de conjuntos.
4. Algunas veces se emplea una operación llamada *combinación* o *unión de conjuntos disjuntos*, que no difiere de la unión, pero supone que sus operandos son disjuntos (no tienen miembros en común). El procedimiento COMBINA(A, B, C) asigna a la variable C de conjuntos el valor $A \cup B$, pero no está definido cuando $A \cap B \neq \emptyset$, es decir, si los conjuntos A y B no son disjuntos.
5. La función MIEMBRO(x, A) toma el conjunto A y el objeto x , cuyo tipo es el de los elementos de A , y devuelve un valor booleano: verdadero si $x \in A$ y falso si $x \notin A$.
6. El procedimiento ANULA(A), hace que el conjunto nulo sea el valor de la variable conjunto A .
7. El procedimiento INSERTA(x, A), donde A es una variable cuyos valores son conjuntos y x es un elemento del tipo de los miembros de A , hace de x un miembro de A . Esto es, el nuevo valor de A es $A \cup \{x\}$. Obsérvese que si x ya es miembro de A , entonces INSERTA(x, A) no modifica el conjunto A .
8. SUPRIME(x, A) elimina x de A , es decir, A se reemplaza por $A - \{x\}$. Si x no está originalmente en A , SUPRIME(x, A) deja sin cambios el conjunto A .
9. ASIGNA(A, B) hace que el valor de la variable A de conjuntos sea igual al valor de la variable B , también de conjuntos.
10. La función MIN(A) devuelve el elemento menor del conjunto A . Esta operación sólo es aplicable cuando los miembros del conjunto parámetro están ordenados linealmente. Por ejemplo, $\text{MIN}(\{2, 3, 1\}) = 1$ y $\text{MIN}(\{'a', 'b', 'c'\}) = 'a'$. También se utiliza la función MAX, cuyo significado es obvio.
11. IGUAL(A, B) es una función cuyo valor es verdadero si, y sólo si, los conjuntos A y B contienen los mismos elementos.
12. La función ENCUENTRA(x) opera en un ambiente donde hay una colección de conjuntos disjuntos. ENCUENTRA(x) devuelve el nombre del (único) conjunto del cual x es miembro.

4.2 Un TDA con UNION, INTERSECCION y DIFERENCIA

Para empezar, se definirá un TDA para el modelo matemático «conjunto» con las tres operaciones básicas de la teoría de conjuntos, unión, intersección y diferencia. Primero se dará un ejemplo en el que un TDA es útil y después se analizarán varias realizaciones sencillas del mismo.

Ejemplo 4.1. Se desea escribir un programa que efectúe una forma sencilla de «análisis de flujo de datos» sobre diagramas de flujo que representen procedimientos. El programa usará variables de un tipo de datos abstracto CONJUNTO, cuyas operaciones son UNION, INTERSECCION, DIFERENCIA, IGUAL, ASIGNA Y ANULA, según se definieron en la sección anterior.

En la figura 4.1 se observa un diagrama de flujo cuyas casillas tienen los nombres B_1, \dots, B_8 y contienen *definiciones de datos* (proposiciones de lectura y asignación) numeradas del 1 al 9. Este diagrama de flujo es una aplicación del algoritmo de Euclides para calcular el máximo común divisor de sus entradas p y q , pero los detalles del algoritmo no tienen importancia para el ejemplo.

En general, un *análisis de flujo de datos* se refiere a la parte de un compilador que examina la representación de un programa fuente en forma de un diagrama de flujo, como la figura 4.1, y reúne información acerca de lo que puede ser cierto conforme el control llega a cada casilla del diagrama. A menudo, las casillas reciben el nombre de *bloques* o *bloques básicos* y representan grupos de proposiciones a través de los cuales el flujo de control procede en forma secuencial. La información que se colecta en el análisis de flujo de datos sirve para mejorar el código generado por el compilador. Por ejemplo, si el análisis permitió determinar que cada vez que el control llegó al bloque B , la variable x tuvo el valor 27, entonces se podría sustituir x por 27 en el bloque B , a menos que se hubiera asignado un nuevo valor a x dentro de ese bloque. Si el acceso a las constantes fuera más rápido que el acceso a las variables, este cambio haría que el código generado por el compilador se ejecutara a mayor velocidad.

En el ejemplo se desea determinar el lugar donde una variable pudo haber recibido, por última vez, un valor nuevo. Esto es, se desea calcular para cada bloque B_i el conjunto $DEF_ENT[i]$ de definiciones de datos d , de modo que exista un camino de B_1 a B_i en el cual aparezca d , pero que no esté seguida de otra definición de la misma variable que define d . El conjunto $DEF_ENT[i]$ recibe el nombre de *definiciones de alcance* de B_i .

Para ver cómo podría ser útil esa información, considérese la figura 4.1. El primer bloque B_1 es un bloque «ficticio» con tres definiciones de datos que hacen que las variables t , p y q tomen valores «indefinidos». Si, por ejemplo, se descubre que $DEF_ENT[7]$ incluye la definición 3, la cual da a q un valor indefinido, entonces el programa podría tener un error, puesto que aparentemente podría imprimir q sin antes haberle asignado un valor válido. Por fortuna, se descubrirá que es imposible alcanzar el bloque B_7 sin haber hecho una asignación a q , es decir, 3 no pertenece a $DEF_ENT[7]$.

Varias reglas ayudan a calcular los $DEF_ENT[i]$. Primero, se hace un cálculo pre-

vio, para cada bloque i , de los conjuntos $GEN[i]$ y $ELIM[i]$. $GEN[i]$ es el conjunto de las definiciones de datos del bloque i , con la excepción de que si B_i contiene dos o más definiciones de una variable x , entonces sólo la última de ellas pertenece a $GEN[i]$. Así, $GEN[i]$ es el conjunto de las definiciones de B_i que son «generadas» por B_i ; alcanzan el final de B_i sin que sus variables sean redefinidas.

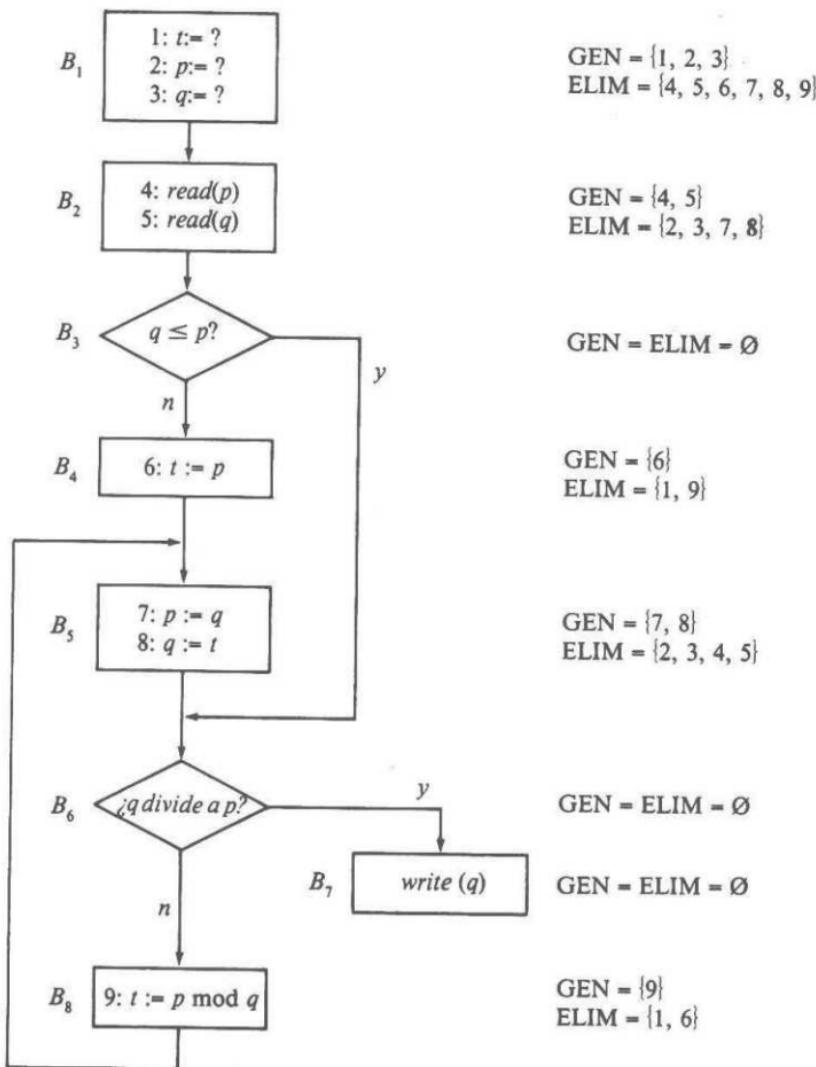


Fig. 4.1. Diagrama de flujo para el algoritmo de Euclides.

El conjunto $ELIM[i]$ es el conjunto de las definiciones d que no están en B_i y tales que B_i contiene una definición de la misma variable que d . Por ejemplo, en la figura 4.1, $GEN[4] = \{6\}$, puesto que la definición 6 (de la variable t) está en B_4 y no hay ninguna definición posterior de t en B_4 . $ELIM[4] = \{1, 9\}$, puesto que 1 y 9 son las definiciones de la variable t que no se encuentran en B_4 .

Además de los $DEF_ENT[i]$, se calcula el conjunto $DEF_SAL[i]$ para cada uno de los bloques B_i . De la misma manera que $DEF_ENT[i]$ es el conjunto de definiciones que alcanzan el principio de B_i , $DEF_SAL[i]$ es el conjunto de definiciones que alcanzan el final de B_i . Existe una fórmula muy sencilla que relaciona DEF_ENT y DEF_SAL :

$$DEF_SAL[i] = (DEF_ENT[i] - ELIM[i]) \cup GEN[i] \quad (4.1)$$

Esto es, la definición d alcanza el final de B_i si, y sólo si, d alcanza el principio de B_i y no es eliminada por B_i o es generada en B_i . La segunda regla que relaciona DEF_ENT y DEF_SAL es que $DEF_ENT[i]$ es la unión, sobre todos los predecesores p de B_i , de $DEF_SAL[p]$, es decir:

$$DEF_ENT[i] = \bigcup_{B_p \text{ precede a } B_i} DEF_SAL[p] \quad (4.2)$$

La regla 4.2 establece que una definición de datos entra en B_i si, y sólo si, alcanza el final de uno de los predecesores de B_i . Como caso especial, si B_i no tiene predecesores, como ocurre con B_1 en la figura 4.1, entonces $DEF_ENT[i] = \emptyset$.

Como ya se han presentado varios conceptos nuevos en este ejemplo, no se profundizará al escribir un algoritmo general para calcular las definiciones de entrada de un diagrama de flujo arbitrario. En vez de ello, se escribirá una parte de un programa que supone que $GEN[i]$ y $ELIM[i]$ se conocen para $i = 1, \dots, 8$, que calcula $DEF_ENT[i]$ y $DEF_SAL[i]$ para $i = 1, \dots, 8$, en el caso particular del diagrama de flujo de la figura 4.1. Este fragmento de programa supone la existencia de un TDA CONJUNTO con las operaciones UNION, INTERSECCION, DIFERENCIA, IGUAL, ASIGNA y ANULA. Más adelante se darán varias implantaciones alternativas de este TDA.

El procedimiento *propaga(GEN, ELIM, DEF_ENT, DEF_SAL)* aplica la regla 4.1. al cálculo de DEF_SAL para un bloque, dado DEF_ENT . En el caso de un programa sin ciclos, el cálculo de DEF_SAL sería directo. La presencia de un ciclo en este caso exige un procedimiento iterativo. La aproximación a $DEF_ENT[i]$ da comienzo con $DEF_ENT[i] = \emptyset$ y $DEF_SAL[i] = GEN[i]$ para toda i , y después aplica repetidas veces (4.1) y (4.2) hasta que ya no ocurran más cambios en los DEF_ENT y DEF_SAL . Como puede mostrarse que cada nuevo valor asignado a los $DEF_ENT[i]$ y $DEF_SAL[i]$ es un supraconjunto (no necesariamente propio) de su valor anterior, y dado que en cualquier programa hay solamente un número finito de definiciones de datos, el proceso debe converger finalmente a una solución de (4.1) y (4.2).

En la figura 4.3 se muestran los valores sucesivos de $DEF_ENT[i]$ después de cada iteración del ciclo `repeat` de la figura 4.2. Obsérvese que ninguna de las asignaciones ficticias 1, 2 y 3 alcanza un bloque donde se emplea su variable, por tanto, en el programa de la figura 4.1 no hay usos de variables indefinidas. Obsérvese también que al diferir la aplicación de (4.2) para B_i hasta antes de aplicar (4.1) para B_i , haría que en general el proceso de la figura 4.2 convergiera en menos iteraciones. \square

4.3 Realización de conjuntos mediante vectores de bits

Obtener la mejor realización para un TDA CONJUNTO depende de las operaciones que se vayan a efectuar y del tamaño de los conjuntos. Cuando todos los conjuntos del dominio en cuestión son subconjuntos de un pequeño «conjunto universal» cuyos elementos son los enteros 1, 2, .., N , para algún N fijo, se puede usar una realización basada en un *vector de bits* (arreglo booleano). Un conjunto se representa mediante un vector de bits en el que el i -ésimo bit es verdadero si i es un elemento del conjunto. La principal ventaja de esta representación radica en que las operaciones MIEMBRO, INSERTA y SUPRIME se pueden realizar en un tiempo constante mediante una referencia directa al bit apropiado. UNION, INTERSECCION y DIFERENCIA se pueden realizar en un tiempo proporcional al tamaño del conjunto universal.

```

var
  GEN, ELIM, DEF_ENT, DEF_SAL: array[1..8] of CONJUNTO;
  { se supone que GEN y ELIM se calculan por fuera }
  i: integer;
  hay_cambios: boolean;

procedure propaga ( G,K,I: CONJUNTO; var O: CONJUNTO );
  { aplica (4.1) y asigna verdadero a hay_cambios si se detecta algún cambio
    en DEF_SAL }
var
  TEMP: CONJUNTO;
begin
  DIFERENCIA(I, K, TEMP);
  UNION(TEMP, G, TEMP);
  if not IGUAL(TEMP, O) do begin
    ASIGNA(O, TEMP);
    hay_cambios := true
  end
end; { propaga }

begin
  for i := 1 to 8 do
    ASIGNA(DEF_SAL[i], GEN[i]);

```

```

repeat
    hay_cambios := false;
    { las ocho proposiciones siguientes aplican (4.2) al diagrama de la figura
        4.1 solamente }
    ANULA(DEF_ENT[1]);
    ASIGNA(DEF_ENT[2], DEF_SAL[1]);
    ASIGNA(DEF_ENT[3], DEF_SAL[2]);
    ASIGNA(DEF_ENT[4], DEF_SAL[3]);
    UNION(DEF_SAL[4], DEF_SAL[8], DEF_ENT[5]);
    UNION(DEF_SAL[3], DEF_SAL[5], DEF_ENT[6]);
    ASIGNA(DEF_ENT[7], DEF_SAL[6]);
    ASIGNA(DEF_ENT[8], DEF_SAL[6]);
    for i := 1 to 8 do
        propaga(GEN[i], ELIM[i], DEF_ENT[i], DEF_SAL[i]);
    until
        not hay_cambios
end.

```

Fig. 4.2. Programa para calcular definiciones de alcance.

<i>i</i>	paso 1	paso 2	paso 3	paso 4
1	\emptyset	\emptyset	\emptyset	\emptyset
2	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}
3	{4,5}	{1,4,5}	{1,4,5}	{1,4,5}
4	\emptyset	{4,5}	{1,4,5}	{1,4,5}
5	{6,9}	{6,9}	{4,5,6,7,8,9}	{4,5,6,7,8,9}
6	{7,8}	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}
7	\emptyset	{7,8}	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}
8	\emptyset	{7,8}	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}

Fig. 4.3. *DEF_ENT [i]* después de cada iteración.

Si el conjunto universal es suficientemente pequeño para que un vector de bits quepa en una palabra de computador, entonces UNION, INTERSECCION y DIFERENCIA se pueden realizar mediante una sola operación lógica del lenguaje de la máquina en cuestión. Algunos conjuntos pequeños se pueden representar directamente en Pascal por medio de la construcción set. El tamaño máximo de tales conjuntos depende del compilador particular que se esté usando y, por desgracia, a menudo es demasiado pequeño para problemas típicos con conjuntos. Sin embargo, al escribir los propios programas no es necesario restringirse a un límite para el tamaño de los conjuntos, siempre que sea posible tratar cualquier conjunto como un subconjunto de algún conjunto universal $\{1, \dots, N\}$. Se pretende que si *A* es un conjunto representado por un arreglo booleano, entonces *A*[*i*] será verdadero si, y sólo si, el elemento

i es miembro de *A*. Así, se puede definir un TDA CONJUNTO mediante la declaración en Pascal

```
const
  N = {cualquier valor adecuado};
type
  CONJUNTO = packed array[1..N] of boolean;
```

Se puede implantar entonces el procedimiento UNION tal como se muestra en la figura 4.4. Para implantar INTERSECCION y DIFERENCIA basta reemplazar or en la figura 4.4, por and y and not, respectivamente. Es posible poner en práctica las otras operaciones mencionadas en la sección 4.1 (exceptuando COMBINA y ENCUENTRA, que tienen poco sentido en este contexto) a manera de ejercicios sencillos.

Se puede utilizar la implantación de conjuntos mediante vectores de bits, cuando el conjunto universal es un conjunto finito diferente de un conjunto de enteros consecutivos. De ordinario, se necesitaría una manera de «traducir» entre los miembros del conjunto universal y los enteros 1, ..., *N*. Así, en el ejemplo 4.1, se supuso que las definiciones de datos tenían asignados números del 1 al 9. En general, las traducciones en ambas direcciones se pueden efectuar mediante el TDA CORRESPONDENCIA descrito en el capítulo 2. Sin embargo, la traducción en sentido inverso de enteros a elementos del conjunto universal puede lograrse mejor con un arreglo *A* en el cual *A*[*i*] sea el elemento correspondiente al entero *i*.

```
procedure UNION ( A, B: CONJUNTO; var C: CONJUNTO );
  var
    i: integer;
  begin
    for i := 1 to N do
      C[i] := A[i] or B[i]
  end
```

Fig. 4.4. Realización de UNION.

4.4 Realización de conjuntos mediante listas enlazadas

Debe ser evidente que también los conjuntos se pueden representar por medio de listas enlazadas, cuyos elementos son los miembros del conjunto. A diferencia de la representación mediante vectores de bits, la representación mediante listas utiliza un espacio proporcional al tamaño del conjunto representado, y no al del conjunto universal. Más aún, la representación mediante listas es más general, puesto que puede manejar conjuntos que no necesitan ser subconjuntos de algún conjunto universal finito.

Cuando hay operaciones como INTERSECCION con conjuntos representados por listas enlazadas, se presentan varias opciones. Si el conjunto universal tiene un

orden lineal, puede representarse un conjunto por medio de una lista clasificada. Esto es, se supone que todos los miembros de un conjunto son comparables por medio de una relación « $<$ » y los miembros de un conjunto aparecen en una lista en el orden e_1, e_2, \dots, e_n , donde $e_1 < e_2 < \dots < e_n$. La ventaja de una lista clasificada radica en que no es necesario revisarla toda para determinar si un elemento se encuentra en la lista.

Un elemento está en la intersección de las listas L_1 y L_2 si, y sólo si, aparece en ambas listas. Para determinar la intersección con listas no clasificadas, es necesario parear cada elemento de L_1 con cada elemento de L_2 en un proceso de $O(n^2)$ pasos con listas de longitud n . La razón por la cual la clasificación de listas facilita la intersección y algunas otras operaciones, consiste en que si se desea parear un elemento e de una lista L_1 con los elementos de otra lista L_2 , sólo es necesario observar los elementos de L_2 hasta que se encuentre e o un elemento mayor que e ; en el primer caso, se habrá encontrado el par, mientras que en el segundo, se sabe que no existe. Es más, si d es el elemento de L_1 que precede inmediatamente a e , y ya se ha encontrado en L_2 el primer elemento, por ejemplo f , tal que $d \leq f$, para buscar en L_2 una aparición de e , se puede empezar con f . La conclusión de este razonamiento es que se pueden encontrar pares para todos los elementos de L_1 , si existen, buscando en L_1 y L_2 una sola vez, siempre que se hagan avanzar los marcadores de posición de las dos listas en el orden apropiado, es decir, avanzando siempre el que apunta al elemento más pequeño. La rutina para realizar INTERSECCION se muestra en la figura 4.5. Ahí, los conjuntos se representan mediante listas enlazadas de «celdas» cuyo tipo se define como

```
type
  tipo_celda = record
    elemento: tipo_elemento;
    sig: ^ tipo_celda
  end
```

En la figura 4.5, se supone que tipo_elemento es un tipo, por ejemplo un entero, que se puede comparar mediante $<$. En caso contrario, es necesario escribir una función que, dados dos elementos, determine cuál de ellos precede al otro.

Las listas enlazadas de la figura 4.5 están encabezadas por celdas vacías que sirven como puntos de entrada a las listas. Como ejercicio, escríbese este programa en una forma abstracta más general, usando primitivas de listas. Sin embargo, el programa de la figura 4.5 puede ser más eficiente que el programa más abstracto. Por ejemplo, en la figura 4.5 se usan apuntadores a celdas particulares, en vez de variables «de posición» que apunten a celdas anteriores. Se puede hacer esto porque sólo se agrega al final de la lista C , y las listas A y B sólo se revisan sin hacer en ellas inserciones ni supresiones.

Las operaciones UNION y DIFERENCIA se pueden escribir de modo que se asemejen en forma sorprendente al procedimiento INTERSECCION de la figura 4.5. Para UNION, se deben agregar todos los elementos de la lista A o de la B , a la lista C en el orden apropiado de clasificación, de modo que cuando los elementos no sean iguales (líneas 12-14) se agregue el menor a la lista C , como se hace cuando los

```

procedure INTERSECCION ( encab_a, encab_b: ↑ tipo_celda;
    var apc: ↑ tipo_celda );
    { calcula la intersección de las listas clasificadas A y B con celdas de en-
    cabezamiento encab_a y encab_b, dejando el resultado como una
    lista clasificada a cuyo encabezamiento apunta apc }
var
    actual_a, actual_b, actual_c: ↑ tipo_celda;
    { las celdas actuales de las listas A y B y la última celda agregada a la
    lista C }
begin
    (1) new(apc); { crea el encabezamiento de la lista C }
    (2) actual_a := encab_a↑.sig;
    (3) actual_b := encab_b↑.sig;
    (4) actual_c := apc;
    (5) while (actual_a <> nil) and (actual_b <> nil) do begin
        { compara los elementos actuales de las listas A y B }
    (6) if actual_a↑.elemento = actual_b↑.elemento then begin
            { agrega a la intersección }
            (7) new(actual_c↑.sig);
            (8) actual_c := actual_c↑.sig;
            (9) actual_c↑.elemento := actual_a↑.elemento;
            (10) actual_a := actual_a↑.sig;
            (11) actual_b := actual_b↑.sig
        end
        else { elementos distintos }
            (12) if actual_a↑.elemento < actual_b↑.elemento then
                (13) actual_a := actual_a↑.sig
            else
                (14) actual_b := actual_b↑.sig
        end;
        (15) actual_c↑.sig := nil
    end; { INTERSECCION }

```

Fig. 4.5. Procedimiento de intersección que utiliza listas clasificadas.

elementos son iguales. También se deben agregar a la lista C todos los elementos de la lista que no se haya agotado cuando la prueba de la línea (5) falle. Para DIFERENCIA, no se agrega un elemento a la lista C cuando se encuentren elementos iguales; sólo se agrega el elemento actual de la lista A a la lista C cuando sea menor que el elemento actual de la lista B, porque entonces se sabe que el primero no puede encontrarse en la lista B. Además, se añaden a C los elementos de A cuando, y si la prueba de la línea (5) falla porque B se agotó.

El operador ASIGNA(*A, B*) copia la lista *A* en la lista *B*. Obsérvese que este operador no se puede implantar haciendo simplemente que la celda de encabezamiento de *A* apunte al mismo lugar que la celda de encabezamiento de *B*, porque en tal caso cualquier cambio posterior en *B* causaría cambios no esperados en *A*. El operador MIN

es sencillo; sólo devuelve el primer elemento de la lista. SUPRIME y ENCUENTRA se pueden realizar encontrando el elemento objetivo como se expuso para las listas generales y en el caso de SUPRIME, eliminando su celda.

Por último, la inserción no es difícil de implantar, pero debe hacerse de modo que se inserte el nuevo elemento en la posición apropiada. La figura 4.6 muestra un procedimiento INSERTA que toma como parámetros un elemento y un apuntador a la celda de encabezamiento de una lista, e inserta el elemento en la lista. La figura 4.7 muestra las celdas y apuntadores cruciales antes (líneas de trazo continuo) y después (líneas punteadas) de la inserción.

```

procedure INSERTA ( x: tipo_elemento; a: ↑ tipo_celda );
{ inserta x en la lista a cuyo encabezamiento apunta p }
var
    actual, cel_nue: ↑ tipo_celda;
begin
    actual := a;
    while actual↑.sig <> nil do begin
        if actual↑.sig↑.elemento = x then
            return; { si x ya está en la lista, regresa }
        if actual↑.sig↑.elemento > x then
            goto agrega; { salida del ciclo }
        actual := actual↑.sig
    end;
agrega: { actual es ahora la celda después de la cual se debe insertar x }
    new(cel_nue);
    cel_nue↑.elemento := x;
    cel_nue↑.sig := actual↑.sig;
    actual↑.sig := cel_nue
end; { INSERTA }
```

Fig. 4.6. Procedimiento de inserción.

4.5 El diccionario

Cuando se usa un conjunto en el diseño de un algoritmo, quizás no sean necesarias algunas operaciones poderosas como unión e intersección. A menudo, lo único que se necesita es mantener un conjunto de objetos «actuales», con inserciones y supresiones periódicas en el conjunto. Con cierta frecuencia, también puede ser necesario saber si un elemento particular está en el conjunto. Un TDA CONJUNTO con las operaciones INSERTA, SUPRIME y MIEMBRO recibe el nombre de *diccionario*. Se incluirá también ANULA como una operación de diccionario para asignar un valor inicial a cualquier estructura utilizada en la aplicación. Se considerará un ejemplo de realización del diccionario para después analizar algunas realizaciones apropiadas para representar diccionarios.

Ejemplo 4.2. La Sociedad para la Prevención de Injusticias con el Atún (SPIA) mantiene una base de datos que registra los votos más recientes de los legisladores en asuntos de importancia para los amantes del atún. Desde el punto de vista conceptual, esta base de datos consta de dos conjuntos de nombres de legisladores llamados *chicos_buenos* y *chicos_malos*. La sociedad olvida con facilidad los errores del pasado y, de igual modo, tiende a olvidar a quienes fueron amigos. Por ejemplo, si se efectuara una votación para decidir acerca de la prohibición de la pesca de atún en el lago Erie, todos los legisladores que votaran a favor se incluirían en *chicos_buenos* y se excluirían de *chicos_malos*, y lo contrario sucedería con los que votaran en contra. Los legisladores que se abstuvieran, permanecerían en el conjunto donde se encontraran, si estuvieran en alguno.

Cuando está en operación, el sistema de bases de datos acepta tres mandatos, cada uno representado por un solo carácter, seguido de una cadena de diez caracteres que denota el nombre de un legislador; cada mandato está en una línea aparte. Los mandatos son:

1. F (sigue un legislador con voto favorable)
2. D (sigue un legislador con voto desfavorable)
3. ? (determina el estado del legislador que sigue).

También se permite el carácter 'E' en la línea de entrada para señalar el fin del proceso. La figura 4.8 muestra un esbozo del programa, escrito en función del TDA DICCIONARIO aún no definido, que en este caso se pretende que sea un conjunto de cadenas de longitud 10. □

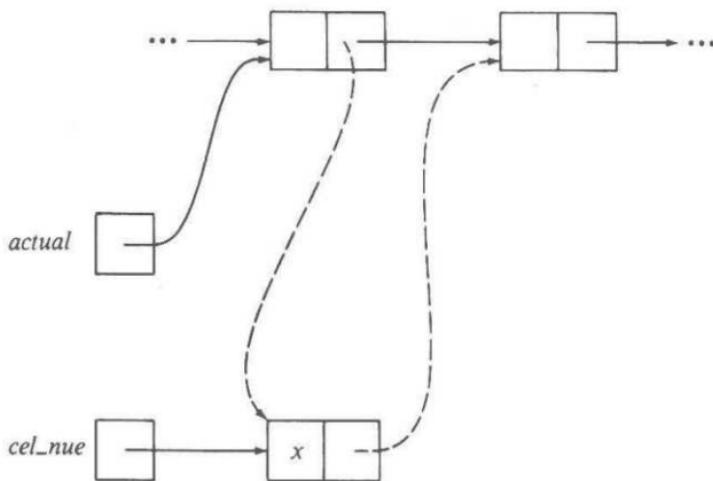


Fig. 4.7. Representación de la inserción.

4.6 Realizaciones sencillas de diccionarios

Un diccionario puede implantarse mediante una lista enlazada, clasificada o no. Otra posible implantación de un diccionario es mediante un vector de bits, siempre que los elementos del conjunto en cuestión estén restringidos a los enteros 1, 2, ..., N para algún N , o a un conjunto que se pueda hacer corresponder con ese conjunto de enteros.

Una posible tercera implantación de diccionarios consiste en usar un arreglo de longitud fija con un apuntador a la última entrada del arreglo en uso. Esta realización sólo es factible si se puede suponer que los conjuntos nunca serán más grandes que la longitud del arreglo. Tiene la ventaja de la sencillez sobre la representación con listas enlazadas, mientras sus desventajas son que 1) los conjuntos no pueden crecer arbitrariamente, 2) la supresión es más lenta, y 3) el espacio no se utiliza de forma eficiente si los conjuntos son de tamaño variable.

A causa de la última razón, no se estudió la realización mediante arreglos en relación con conjuntos cuyas uniones e intersecciones se realizan con frecuencia. No obstante, dado que los arreglos, al igual que las listas, se pueden clasificar, podría considerarse la realización con arreglos que ahora se describe para los diccionarios como una posible realización para conjuntos en general. La figura 4.9 muestra las declaraciones y procedimientos necesarios para complementar la figura 4.8, haciendo de ella un programa ejecutable.

```

program atún ( input, output );
  { base de datos de legisladores }
  type
    tipo_nombre = array[1..10] of char;
  var
    mandato: char;
    legislador: tipo_nombre;
    chicos_buenos, chicos_malos: DICCCIONARIO;
  procedure favorable ( amigo: tipo_nombre );
  begin
    INSERTA(amigo, chicos_buenos);
    SUPRIME(amigo, chicos_malos)
  end; { favorable }

  procedure desfavorable ( enemigo: tipo_nombre );
  begin
    INSERTA(enemigo, chicos_malos);
    SUPRIME(enemigo, chicos_buenos)
  end; { desfavorable }

  procedure consulta ( sujeto: tipo_nombre );
  begin
    if MIEMBRO(sujeto, chicos_buenos) then
      writeln(sujeto, 'es un amigo')
    else
      writeln(sujeto, 'no es un amigo')
  end; { consulta }
end.

```

```

else if MIEMBRO(sujeto, chicos_malos) then
    writeln(sujeto, 'es un enemigo')
else
    writeln('no se tiene información sobre', sujeto)
end; { consulta }

begin { programa principal }
    ANULA(chicos_buenos);
    ANULA(chicos_malos);
    read(mandato);
    while mandato <> 'E' do begin
        readln(legislador);
        if mandato = 'F' then
            favorable(legislador)
        else if mandato = 'D' then
            desfavorable(legislador)
        else if mandato = "?" then
            consulta(legislador)
        else
            error('mandato desconocido');
        read(mandato)
    end
end. { atún }

```

Fig. 4.8. Esbozo del programa de base de datos SPIA.

```

const
    tamaño_máx = { algún número adecuado };
type
    DICCCIONARIO = record
        últ: integer;
        datos: array[1..tamaño_máx] of tipo_nombre
    end;
procedure ANULA ( var A: DICCCIONARIO );
begin
    A.últ := 0
end; { ANULA }

function MIEMBRO ( x: tipo_nombre; var A: DICCCIONARIO ) : boolean;
var
    i: integer;
begin
    for i := 1 to A.últ do
        if A.datos[i] = x then return (true);
    return (false) { si no se encuentra x }
end; { MIEMBRO }

```

```

procedure INSERTA ( x: tipo_nombre; var A: DICCIONARIO );
begin
  if not MIEMBRO(x, A) then
    if A.últ < tamaño_máx then begin
      A.últ := A.últ + 1;
      A.datos[A.últ] := x
    end
    else error('la base de datos está llena')
  end; { INSERTA }

procedure SUPRIME ( x: tipo_nombre; var A: DICCIONARIO );
var
  i: integer;
begin
  if A.últ > 0 then begin
    i := 1;
    while (A.datos[i] <> x) and (i < A.últ) do
      i := i + 1;
    { cuando se llega aquí, se ha encontrado x o el elemento
      actual es el último del conjunto A, o ambas cosas }
    if A.datos[i] = x then begin
      A.datos[i] := A.datos[A.últ];
      { pasa el último elemento al lugar ocupado por x; ob-
        sérvase que si i = A.últ, este paso no hace nada,
        pero el siguiente elimina x }
      A.últ := A.últ - 1
    end
  end
end;
end; { SUPRIME }

```

Fig. 4.9. Declaraciones de tipos y procedimientos para un diccionario basado en arreglos.

4.7 La estructura de datos tabla de dispersión

En promedio, la realización de diccionarios mediante arreglos requiere $O(N)$ pasos para la ejecución de una sola instrucción INSERTA, SUPRIME o MIEMBRO en un diccionario de N elementos; si se usa una realización mediante listas, se obtiene una velocidad similar. En la realización mediante vectores de bits, cualquiera de estas tres operaciones utiliza un tiempo constante, pero existe la limitación de trabajar con conjuntos de enteros en un intervalo pequeño para la realización que sea factible.

Existe otra técnica importante y muy útil para implantar diccionarios, y se denomina «dispersión» (*hashing*). La dispersión utiliza tiempo constante por operación, en promedio, y no existe la exigencia de que los conjuntos sean subconjuntos de algún conjunto universal finito. En el peor caso, este método requiere, para

cada operación, un tiempo proporcional al tamaño del conjunto, como sucede con las realizaciones mediante arreglos y listas. Sin embargo, con un diseño cuidadoso es posible hacer que sea arbitrariamente pequeña la probabilidad de que la dispersión demande más de un tiempo constante para cada operación.

Se considerarán dos formas de dispersión algo diferentes. La primera, llamada dispersión *abierta* o *externa*, permite que el conjunto se almacene en un espacio potencialmente ilimitado, por lo que no impone un límite al tamaño del conjunto. La segunda, llamada dispersión *cerrada* o *interna*, usa un espacio fijo para el almacenamiento, por lo que limita el tamaño de los conjuntos †.

Dispersión abierta

En la figura 4.10 se puede ver la estructura de datos básica para la dispersión abierta. La idea fundamental es que el conjunto (posiblemente infinito) de miembros potenciales se divide en un número finito de clases. Si se desea tener B clases, numeradas de 0 a $B-1$, se usa una función de dispersión h tal que para cada objeto x del tipo de datos de los miembros del conjunto que se va a representar, $h(x)$ sea uno de los enteros de 0 a $B-1$. Lógicamente, el valor de $h(x)$ es la clase a la cual x pertenece. A menudo se da a x el nombre de *clave* y a $h(x)$ el de *valor de dispersión* de x . A las «clases» se les da el nombre de *cubetas* y se dice que x pertenece a la cubeta $h(x)$.

En un arreglo llamado *tabla de cubetas*, indizado por los *números de cubeta* 0, 1, ..., $B-1$, se tienen los encabezamientos de B listas. Los elementos de la i -ésima lista son los miembros del conjunto que se está representando y que pertenecen a la clase i , esto es, aquellos elementos x del conjunto tales que $h(x) = i$.

Se espera que las cubetas tengan casi el mismo tamaño, de modo que la lista para

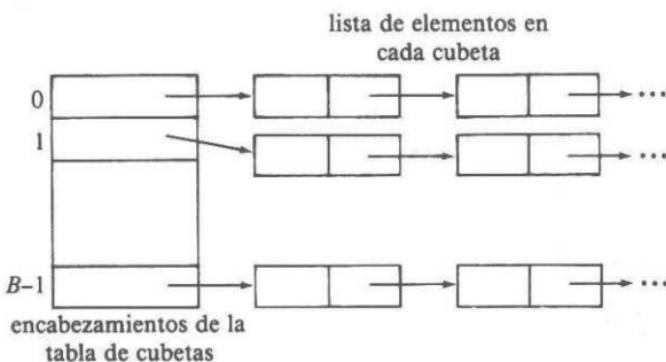


Fig. 4.10. Organización de datos en la dispersión abierta.

† Es posible hacer cambios en la estructura de datos para acelerar la dispersión abierta y permitir que la cerrada maneje conjuntos más grandes. Estas técnicas se describirán después de estudiar los métodos básicos.

cada cubeta sea corta. Entonces, si hay N elementos en el conjunto, en promedio, cada cubeta tendrá N/B miembros. Si se puede estimar N y elegir B aproximadamente igual de grande, entonces una cubeta tendrá en promedio sólo uno o dos miembros, y las operaciones con el diccionario tendrán, en promedio, un número pequeño y constante de pasos, independiente del valor que tenga N (o, en forma equivalente, B).

No siempre está clara la posibilidad de elegir h de modo que un conjunto típico tenga sus miembros distribuidos de forma relativamente uniforme entre las cubetas. Más adelante se estudiará otra vez la forma de elegir h de modo que, en verdad, «disperse» su argumento, es decir, para que $h(x)$ sea un valor «aleatorio» que no dependa de x de ninguna forma trivial.

```
function h ( x: tipo_nombre ) : 0..B-1;
var
    i, suma: integer;
begin
    suma := 0;
    for i := 1 to 10 do
        suma := suma + ord(x[i]);
    h := suma mod B
end; { h }
```

Fig. 4.11. Una función de dispersión sencilla h .

Con el fin de especificar, se presenta ahora una función de dispersión para cadenas de caracteres que es muy buena, aunque no perfecta. La idea es considerar a los caracteres como enteros, utilizando el código de caracteres de la máquina para definir la correspondencia. Pascal proporciona la función incorporada ord , donde $ord(c)$ es el código entero del carácter c . Así, si x es una clave y el tipo de las claves es $\text{array}[1..10] \text{ of char}$ (lo que se llamó `tipo_nombre` en el Ejemplo 4.2), se podría declarar la función de dispersión h como en la figura 4.11. En esa función, se suman los enteros para cada carácter y se divide el resultado entre B , tomando el residuo, que es un entero entre 0 y $B-1$.

En la figura 4.12 se observan las declaraciones de las estructuras de datos para una tabla de dispersión abierta y los procedimientos que realizan las operaciones para un diccionario. El tipo de los miembros del diccionario se supone que es `tipo_nombre` (un arreglo de caracteres), de modo que estas declaraciones se pueden usar en forma directa en el ejemplo 4.2. Debe observarse que en la figura 4.12 se ha hecho que los encabezamientos de las listas de cubetas sean apuntadores a celdas, en lugar de celdas completas. Esto se hizo así, porque la tabla de cubetas, donde se encuentran los encabezamientos, podría ocupar tanto espacio como las propias listas si fuera un arreglo de celdas, en lugar de uno de apuntadores. Obsérvese, sin embargo, que se paga un precio por este ahorro de espacio. Ahora, el código del procedimiento `SUPRIME` debe hacer distinción entre la primera celda y las restantes.

```

const
     $B = \{ \text{constante adecuada} \}$ 
type
    tipo_celda = record
        elemento: tipo_nombre;
        sig:  $\uparrow$  tipo_celda
    end;
    DICCCIONARIO = array[0..B-1] of  $\uparrow$  tipo_celda;

procedure ANULA ( var A: DICCCIONARIO );
var
    i: integer;
begin
    for i := 0 to B-1 do
        A[i] := nil
end; { ANULA }

function MIEMBRO ( x: tipo_nombre; var A: DICCCIONARIO ) : boolean;
var
    actual:  $\uparrow$  tipo_celda;
begin
    actual := A[h(x)];
    { actual es inicialmente el encabezado de la cubeta para x }
    while actual <> nil do
        if actual^.elemento = x then
            return (true)
        else
            actual := actual^.sig;
    return (false) { si no se encuentra x }
end; { MIEMBRO }

procedure INSERTA ( x: tipo_nombre; var A: DICCCIONARIO );
var
    cubeta: integer;
    encab_ant:  $\uparrow$  tipo_celda;
begin
    if not MIEMBRO(x, A) then begin
        cubeta := h(x);
        encab_ant := A[cubeta];
        new(A[cubeta]);
        A[cubeta]^.elemento := x;
        A[cubeta]^.sig := encab_ant
    end
end; { INSERTA }

procedure SUPRIME ( x: tipo_nombre; var A: DICCCIONARIO );
var

```

```

actual: ↑ tipo_celda;
cubeta: integer;
begin
  cubeta: = h(x);
  if A[cubeta] <> nil then begin
    if A[cubeta]↑.elemento = x then { x está en la primera celda }
      A[cubeta] := A[cubeta]↑.sig { suprime x de la lista }
    else begin { x, si está, no ocupa la primera celda de la cubeta }
      actual := A[cubeta]; { actual apunta a la celda anterior }
      while actual↑.sig <> nil do
        if actual↑.sig↑.elemento = x then begin
          actual↑.sig := actual↑.sig↑.sig
          { suprime x de la lista }
        return { salida del ciclo }
      end
      else { aún no se encuentra x }
        actual := actual↑.sig
    end
  end
end; { SUPRIME }

```

Fig. 4.12. Realización de un diccionario mediante una tabla de dispersión abierta.

Dispersión cerrada

Una tabla de dispersión cerrada guarda los miembros del diccionario en la tabla de cubetas, en vez de usar esa tabla para almacenar encabezamientos de listas. En consecuencia, parece que sólo es posible colocar un elemento en una cubeta; sin embargo, la dispersión cerrada tiene asociada una *estrategia de redispersión*. Si se intenta colocar x en la cubeta $h(x)$ y esta ya tiene un elemento —situación denominada *colisión*—, la estrategia de redispersión elige una sucesión de localidades alternas $h_1(x)$, $h_2(x)$, ... dentro de la tabla de cubetas, en la cual es posible colocar x . Se prueba en todas esas localidades, en orden, hasta encontrar una vacía. Si ninguna está vacía, la tabla está llena y no es posible insertar x .

Ejemplo 4.3. Supóngase que $B = 8$ y que las claves a, b, c y d tienen valores de dispersión $h(a) = 3, h(b) = 0, h(c) = 4, h(d) = 3$. Se usará la estrategia de redispersión más sencilla, denominada *dispersión lineal*, en la que $h_i(x) = (h(x) + i) \bmod B$. Así, por ejemplo, si se deseara insertar a y se encontrara que la cubeta 3 ya está llena, se probaría con las cubetas 4, 5, 6, 7, 0, 1 y 2, en ese orden.

En principio, se supone que la tabla está vacía, esto es, que cada cubeta guarda un valor especial *vacio*, que no es igual a ningún valor que podría intentarse insertar \dagger . Si se insertan a, b, c y d , en ese orden, en una tabla inicialmente vacía, resulta

\dagger Si el tipo de los miembros del diccionario no sugiere un valor adecuado para *vacio*, se puede hacer que cada cubeta tenga un campo adicional de un bit que informe si está vacía o no.

que a va a la cubeta 3, b a la 0 y c a la 4. Al insertar d , primero se hace la prueba con $h(d) = 3$ para encontrar que está llena. Luego se intenta con $h_1(d) = 4$ y ocurre lo mismo. Por último, se prueba con $h_2(d) = 5$, se encuentra un espacio vacío y d se coloca allí. Las posiciones resultantes están en la figura 4.13. \square

La prueba de pertenencia de un elemento x al conjunto requiere examinar $h(x)$, $h_1(x)$, $h_2(x)$, ..., hasta encontrar x o una cubeta vacía. Para ver por qué es posible detenerse al alcanzar una cubeta vacía, supóngase primero que las supresiones no están permitidas. Si $h_3(x)$ es la primera cubeta vacía encontrada en la serie, no es posible que x esté en las cubetas $h_4(x)$, $h_5(x)$ o más adelante en la sucesión, porque x no pudo haber sido colocada allí a menos que $h_3(x)$ hubiera estado llena en el momento de insertarla.

0	<i>b</i>
1	
2	
3	<i>a</i>
4	<i>c</i>
5	<i>d</i>
6	
7	

Fig. 4.13. Tabla de dispersión parcialmente llena.

Sin embargo, obsérvese que si se permiten las supresiones, nunca puede existir la seguridad, si se alcanza una cubeta vacía sin encontrar x , de que x no se encuentra en alguna otra parte, y la cubeta ahora vacía estaba ocupada cuando se insertó x . Cuando deban hacerse supresiones, el enfoque más efectivo para resolver este problema es colocar una constante llamada *suprimido* en una cubeta que contenga un elemento que se desee suprimir. Es importante que haya una diferencia entre *suprimido* y *vacío*, la constante que se encuentra en todas las cubetas que nunca se han llenado. De esta manera, es posible permitir supresiones sin tener que buscar en la tabla completa durante la prueba MIEMBRO. En el momento de la inserción, es posible tratar *suprimido* como un espacio disponible, de modo que con suerte el espacio de un elemento suprimido puede volver a utilizarse. Sin embargo, como el espacio de un elemento suprimido no es reclamable de inmediato, cosa que sí ocurre en la dispersión abierta, puede preferirse el esquema abierto al cerrado.

Ejemplo 4.4. Supóngase que se desea probar si e está en el conjunto representado en la figura 4.13. Si $h(e) = 4$, se prueba con las cubetas 4, 5 y luego 6. La cubeta 6 está vacía y como e no se ha encontrado, la conclusión es que no está en el conjunto.

Si se suprime c , es necesario colocar la constante *suprimido* en la cubeta 4. Así,

al buscar d y comenzar en $h(d) = 3$, se pueden examinar 4 y 5 para encontrar d , y no detenerse en 4 como se hubiera hecho de haber puesto *vacio* en esa cubeta. \square

En la figura 4.14 se observan las declaraciones de tipos y las operaciones para el TDA DICCIONARIO con miembros de conjunto del tipo tipo_nombre y la tabla de dispersión cerrada como estructura fundamental. Se utiliza una función de dispersión arbitraria h , de la cual la figura 4.11 es una posibilidad, y se usa la estrategia de dispersión lineal para redispersar en caso de colisiones. Por conveniencia, se identifica *vacio* con una cadena de diez espacios y *suprimido* con una de diez asteriscos, con la suposición de que ninguna de esas cadenas puede ser una clave real. (En la base de datos SPIA esas cadenas tienen poca probabilidad de ser nombres de legisladores.) El procedimiento INSERTA(x, A) primero usa *localiza* para determinar si x ya está en A , y si no, utiliza una rutina especial *localizal* para encontrar una localidad en la cual se pueda insertar x . Obsérvese que *localizal* busca localidades marcadas tanto con *vacio* como con *suprimido*.

```

const
    vacio = '          '; { 10 espacios }
    suprimido = '*****'; { 10 asteriscos }
type
    DICCIONARIO = array[0..B-1] of tipo_nombre;
procedure ANULA ( var A: DICCIONARIO );
var
    i: integer;
begin
    for i := 0 to B-1 do
        A[i] := vacio
end; { ANULA }

function localiza ( x: tipo_nombre ) : integer;
{ localiza examina el DICCIONARIO desde la cubeta para h(x) hasta que
  se encuentre x, o una cubeta vacía, o se haya revisado toda la tabla
  y determinado que no contiene a x; localiza devuelve el índice
  de la cubeta en la que se detiene por cualquiera de esas razones }

var
    inicial, i: integer;
    { inicial guarda h(x); i cuenta el número de cubetas examinadas
      hasta el momento cuando se busca x }
begin
    inicial := h(x);
    i := 0;
    while (i < B) and (A[(inicial + i) mod B] <> x) and
        (A[(inicial + i) mod B] <> vacio) do
        i := i + 1;
    return ((inicial + i) mod B)
end; { localiza }

```

```

function localiza ( x: tipo_nombre ): integer;
  { como localiza, pero también se detiene en una entrada con suprimido
    y devuelve ese valor }

function MIEMBRO ( x: tipo_nombre; var A: DICCIONARIO ) : boolean;
begin
  if A[localiza(x)] = x then
    return (true)
  else
    return (false)
end; { MIEMBRO }

procedure INSERTA ( x: tipo_nombre; var A: DICCIONARIO );
var
  cubeta: integer;
begin
  if A[localiza(x)] = x then
    return; { x ya está en A }
  cubeta := localiza(x);
  if (A[cubeta] = vacío) or (A[cubeta] = suprimido) then
    A[cubeta] := x
  else
    error ('INSERTA falló: la tabla está llena')
end; { INSERTA }

procedure SUPRIME ( x: tipo_nombre; var A: DICCIONARIO );
var
  cubeta: integer;
begin
  cubeta := localiza(x);
  if A[cubeta] = x then
    A[cubeta] := suprimido
end; { SUPRIME }

```

Fig. 4.14. Realización de un diccionario mediante una tabla de dispersión cerrada.

4.8 Estimación de la eficiencia de las funciones de dispersión

Como ya se mencionó, la dispersión es una manera eficiente de representar diccionarios y otros tipos de datos abstractos basados en conjuntos. En esta sección se examinará el tiempo promedio por operación de diccionario en una tabla de dispersión abierta. Si hay B cubetas y N elementos almacenados en la tabla de dispersión, las cubetas tienen, en promedio, N/B miembros y se puede esperar que una operación normal INSERTA, SUPRIME o MIEMBRO lleve un tiempo $O(1 + N/B)$. La constante 1 representa el tiempo necesario para hallar la cubeta, y N/B , el tiempo para buscar en ella. Si puede elegirse B casi igual a N , este tiempo se convierte

en una constante para cada operación. Por tanto, el tiempo promedio para insertar, suprimir y probar la pertenencia de un elemento al conjunto, suponiendo que un elemento cualquiera tiene la misma probabilidad de ser dispersado en cualquier cubeta, es una constante que no depende de N .

Supóngase que se tiene un programa escrito en algún lenguaje como Pascal y se desea insertar todos los identificadores que aparecen en el programa en una tabla de dispersión. Cada vez que se encuentra la declaración de un nuevo identificador, éste se inserta en la tabla de dispersión después de verificar que ya no se encuentra ahí. Durante esta fase, es razonable suponer que un identificador tiene la misma probabilidad de ser dispersado en cualquier cubeta. Por tanto, se puede construir una tabla de dispersión de N elementos en tiempo $O(N(1 + N/B))$. Al elegir B igual a N , este tiempo es $O(N)$.

En la siguiente fase, los identificadores se hallan en el cuerpo del programa; es necesario localizarlos en la tabla de dispersión para recuperar información asociada con ellos. Pero, ¿cuál es el tiempo estimado para localizar un identificador? Si el elemento que se busca tiene la misma probabilidad que cualquiera de los elementos de la tabla, entonces el tiempo estimado para buscarlo es simplemente el tiempo promedio que demanda la inserción de un elemento. Para comprender esto, obsérvese que el tiempo que demanda buscar una vez cada elemento en la tabla es el empleado en insertarlo, suponiendo que los elementos siempre se agregan al final de la lista de la cubeta apropiada. Entonces, el tiempo estimado para una búsqueda es también $O(1 + N/B)$.

El análisis anterior supone que una función de dispersión distribuye los elementos de manera uniforme en las cubetas. ¿Existen funciones de esta clase? Una función como la de la figura 4.11 (que convierte caracteres en enteros, suma y toma el residuo módulo B) puede considerarse una función típica de dispersión. El siguiente ejemplo examina su funcionamiento.

Ejemplo 4.5. Supóngase que se emplea la función de la figura 4.11 para dispersar 100 claves que consisten en las cadenas de caracteres A0, A1, ..., A99, en una tabla de 100 cubetas. Si se establece que $ord(0)$, $ord(1)$, ..., $ord(9)$ forman una progresión aritmética, como ocurre con los códigos de caracteres más comunes ASCII y EBCDIC, es fácil verificar que las claves se dispersan hasta en 29 de las 100 cubetas † y que la cubeta más grande contiene A18, A27, A36, ..., A90, es decir, 9 de los 100 elementos. Si se calcula el número promedio de pasos para una inserción, partiendo del hecho de que la inserción del i -ésimo elemento en una cubeta requiere $i+1$ pasos, se obtienen 395 pasos para las 100 claves. En comparación, el estimado $N(1+N/B)$ sugiere 200 pasos. □

La función de dispersión sencilla de la figura 4.11 puede tratar ciertos conjuntos de entradas, como las cadenas consecutivas del ejemplo 4.5, de manera no aleatoria. Hay funciones de dispersión «más aleatorias»; como ejemplo, puede usarse la idea de elevar al cuadrado y tomar los dígitos medios. Así, si se tiene como

† Obsérvese que A2 y A20 no necesariamente se dispersan en la misma cubeta, pero, por ejemplo, A23 y A41 deben hacerlo.

clave un número n de 5 dígitos y se eleva al cuadrado, se obtiene un número de 9 ó 10 dígitos. Los «dígitos medios», como los que están entre las posiciones cuarta y séptima a la derecha, tienen valores que dependen de casi todos los dígitos de n ; por ejemplo, el cuarto dígito depende de todos ellos excepto del que se encuentre más a la izquierda de n , y el quinto depende de todos los dígitos de n . De modo que si $B = 100$, se podrían tomar los dígitos sexto y quinto para formar el número de cubeta.

Esta idea se puede generalizar para situaciones en las que B no sea una potencia de 10. Supóngase que las claves son enteros en el intervalo 0, 1, ..., K . Si se escoge un entero C tal que BC^2 sea casi igual a K^2 , la función

$$h(n) = \lfloor n^2/C \rfloor \bmod B$$

efectivamente extrae un dígito de base B del centro de n^2 .

Ejemplo 4.6. Si $k = 1000$ y $B = 8$, se puede elegir $C = 354$. Entonces,

$$h(456) = \lfloor \frac{207936}{354} \rfloor \bmod 8 = 587 \bmod 8 = 3. \quad \square$$

Para usar la estrategia de «elevar al cuadrado y tomar el medio» cuando las claves son cadenas de caracteres, primero se agrupan los caracteres en la cadena de derecha a izquierda, en bloques de un número fijo de caracteres, como 4, rellenando a la izquierda con espacios si es necesario. Se trata cada bloque como un solo entero formado al concatenar los códigos binarios de los caracteres. Por ejemplo, ASCII maneja un código de caracteres de 7 bits, de modo que los caracteres pueden considerarse como «dígitos» de base 2^7 ó 128. De este modo, se puede considerar la cadena de caracteres $abcd$ como el entero $(128)^3 a + (128)^2 b + (128)c + d$. Después de convertir todos los bloques a enteros, se suma † y se procede como se sugirió con anterioridad para los enteros.

Análisis de dispersión cerrada

En un esquema de dispersión cerrada, la velocidad de inserción y de otras operaciones no sólo depende de lo aleatoriamente que la función de dispersión distribuye los elementos en las cubetas, sino también de lo bien que la estrategia de redispersión evita colisiones adicionales cuando una cubeta ya esté llena. Por ejemplo, la estrategia lineal para resolver colisiones no es la mejor posible. Aunque el análisis de este hecho no se efectúa en este libro, es posible observar lo siguiente. Tan pronto como se llenen unas cuantas cubetas consecutivas, cualquier clave que se dispersa a una de ellas será enviada al final del grupo por la estrategia de redispersión, con lo cual el tamaño de ese grupo de cubetas consecutivas se incrementará. De este modo, es probable encontrar sucesiones más largas de cubetas consecutivas

† Si las cadenas pueden ser muy largas, esta suma tendría que hacerse para alguna constante c . Por ejemplo, c valdría uno más que el mayor entero que se pudiera obtener de un solo bloque.

llenas que si los elementos llenaran las cubetas al azar. Más aún, las sucesiones de bloques llenos causan largas secuencias de intentos antes de que un elemento encuentre una cubeta vacía, de modo que tener grupos de cubetas llenas extraordinariamente grandes hace más lentas la inserción y otras operaciones.

Podría desearse saber cuántos intentos (o *sondeos*) se necesitan en promedio para insertar un elemento cuando N de las B cubetas están llenas, al suponer que todas las posibles combinaciones de N de las B cubetas tienen la misma probabilidad de estar llenas. Por lo general se presume, aunque no se demuestre, que ninguna estrategia de dispersión cerrada puede tener un rendimiento temporal medio mejor que ésa para operaciones con diccionarios. Se derivará entonces una fórmula para el costo de inserción cuando las localidades alternas empleadas por la estrategia de redispersión se elijan al azar. Por último, se considerarán algunas estrategias de redispersión, que se aproximan a ese comportamiento aleatorio.

La probabilidad de una colisión en el sondeo inicial es N/B . Suponiendo una colisión, la primera redispersión intentará con una de $B - 1$ cubetas, de las cuales $N - 1$ estarán llenas, de manera que la probabilidad de al menos dos colisiones es $\frac{N(N-1)}{B(B-1)}$. De modo similar, la probabilidad de al menos i colisiones es

$$\frac{N(N-1) \cdots (N-i+1)}{B(B-1) \cdots (B-i+1)}. \quad (4.3)$$

Si B y N son grandes, esta probabilidad se approxima a $(N/B)^i$. El número promedio de sondeos es uno (para inserción exitosa) más la suma con todos los $i \geq 1$ de la probabilidad de al menos i colisiones, esto es, alrededor de $1 + \sum_{i=1}^{\infty} (N/B)^i$, o $\frac{B}{B-N}$. Se puede demostrar que el valor exacto de la sumatoria, cuando la fórmula (4.3) se usa para $(N/B)^i$, es $\frac{B+1}{B+1-N}$, de manera que la aproximación obtenida es buena, excepto cuando N se acerca bastante a B .

Obsérvese que $\frac{B+1}{B+1-N}$ crece en una forma muy lenta conforme N comienza a crecer de 0 a $B-1$, que es el mayor valor de N para el cual es posible otra inserción. Por ejemplo, si N es la mitad de B , se necesitan alrededor de dos sondeos para la siguiente inserción. El costo de inserción promedio por cubeta para llenar M de las B cubetas es $\frac{1}{M} \sum_{N=0}^{M-1} \frac{B+1}{B+1-N}$, que se approxima a $\frac{1}{M} \int_0^{M-1} \frac{B}{B-x} dx$, o $\frac{B}{M} \log_e \left(\frac{B}{B-M+1} \right)$. Llenar por completo la tabla ($M = B$) requiere un promedio de $\log_e B$ sondeos por cubeta o $B \log_e B$ sondeos en total. Sin embargo, llenar la tabla al 90% de su capacidad ($M = 0.9 B$) sólo requiere $B((10/9)\log_{10} 10)$ o unos 2.56 B sondeos. El costo medio de la prueba de pertenencia para un elemento no existente es idén-

tico al de insertar el siguiente elemento, pero el costo de la prueba de pertenencia para un elemento que está en el conjunto, es el costo promedio de todas las inserciones realizadas hasta el momento, el cual es menor en gran medida si la tabla está bastante llena. Las supresiones tienen el mismo costo medio que las pruebas de pertenencia, pero, a diferencia de la dispersión abierta, las supresiones de una tabla de dispersión cerrada no ayudan a acelerar las inserciones o pruebas de pertenencia posteriores. Se debe subrayar que si no se permite que las tablas de dispersión cerradas se llenen en más de una fracción fija, menor que uno, de su capacidad total, el costo medio de las operaciones es una constante; esa constante crece conforme lo hace la fracción de la capacidad que se permite usar. En la figura 4.15 se puede ver una gráfica del costo de las inserciones, supresiones y pruebas de pertenencia como una función del porcentaje de la tabla que está lleno cuando se realiza la operación.

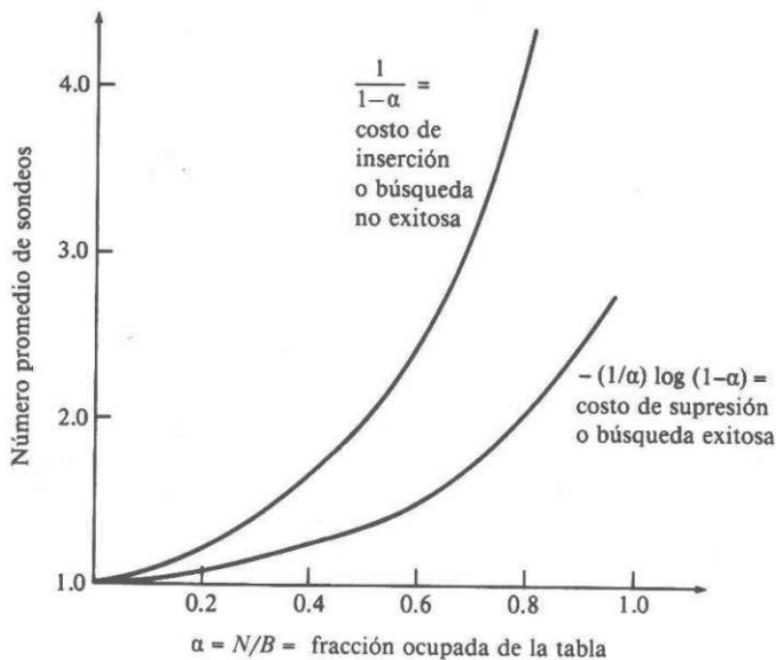


Fig. 4.15. Costo medio de operación.

Estrategias «aleatorias» de resolución de colisiones

Se ha observado que la estrategia de redispersión lineal tiende a agrupar las cubetas llenas en bloques grandes consecutivos. Tal vez se pueda lograr un comportamiento más «aleatorio» si se hacen los sondeos a intervalos constantes mayores que

uno. Esto es, sea $h_i(x) = (h(x) + ci) \bmod B$ para algún $c > 1$. Por ejemplo, si $B = 8$, $c = 3$ y $h(x) = 4$, se sondearían las cubetas 4, 7, 2, 5, 0, 3, 6 y 1, en ese orden. Por supuesto, si c y B tienen un factor común mayor que uno, esta estrategia no permite siquiera buscar en todas las cubetas; como ejemplo, pruébese con $B = 8$ y $c = 2$. Pero todavía más significativo es el hecho de que, aun si c y B son *primos relativos* (no tienen factores comunes), se tiene el mismo problema de «amontonamiento» que con la dispersión lineal, aunque aquí tienden a aparecer las secuencias de cubetas llenas, separadas por una distancia c . Este fenómeno hace más lentas las operaciones, como en la dispersión lineal, puesto que un intento de inserción en una cubeta llena originará un desplazamiento en una cadena de cubetas llenas separadas por la distancia c , y la longitud de esa cadena se incrementará en uno.

En realidad, cualquier estrategia de redispersión en la que el blanco de un sondeo sólo dependa del blanco del sondeo anterior (en lugar de depender del número de sondeos no exitosos ya realizados, la cubeta original $h(n)$, o el valor x de la clave misma) exhibirá la propiedad de amontonamiento de la dispersión lineal. Quizá la estrategia más simple en la que no se presenta este problema sea hacer que $h_i(x) = (h(x) + d_i) \bmod B$, donde d_1, d_2, \dots, d_{B-1} , es una permutación «aleatoria» de los enteros 1, 2, ..., $B - 1$. Desde luego, en todas las inserciones, supresiones y pruebas de pertenencia se usa la misma sucesión d_1, \dots, d_{B-1} ; la disposición «aleatoria» de los enteros se decide una sola vez, al diseñar el algoritmo de redispersión.

La generación de números «aleatorios» es un asunto complejo, pero por fortuna muchos métodos comunes producen una sucesión de ellos que van de 1 a cierto límite. Estos generadores de números aleatorios, cuando se reajustan sus valores iniciales para cada operación con la tabla de dispersión, sirven para generar la sucesión deseada d_1, \dots, d_{B-1} .

Un enfoque efectivo utiliza «sucesiones de registros de desplazamiento». Sea B una potencia de 2, y k una constante entre 1 y $B - 1$. Empiezan con algún número d_1 en el intervalo de 1 a $B - 1$, generan los números siguientes de la sucesión tomando el valor anterior, lo doblan y, si el resultado es mayor que B , le sustraen B y toman la suma módulo 2 bit a bit del resultado y la constante seleccionada k . La suma módulo 2 bit a bit de x e y , que se escribe $x \oplus y$, se calcula escribiendo x e y en binario, con ceros a la izquierda si hace falta, para que ambos sean de la misma longitud, y formando los números cuya representación binaria tenga un 1 en las posiciones en que x o y , pero no ambos, también lo tengan.

Ejemplo 4.7. $25 \oplus 13$ se calcula tomando

$$\begin{array}{rcl} 25 & = & 11001 \\ 13 & = & 01101 \\ \hline 25 \oplus 13 & = & 10100 \end{array}$$

Obsérvese que esta «suma» se puede considerar como una suma binaria ordinaria en la que se ignora el residuo. □

No todo valor de k produce una permutación de 1, 2, ..., $B - 1$; algunas veces, un número se repite antes de generarlos todos. Sin embargo, para una B dada, existe

una probabilidad pequeña, pero finita, de que algún valor particular de k funcione, y sólo hace falta encontrar una k para cada B .

Ejemplo 4.8. Sea $B = 8$. Si se toma $k = 3$, será fácil generar todos los enteros del 1 al 7. Por ejemplo, si se comienza con $d_1 = 5$, se calcula luego d_2 , doblando primero d_1 para obtener 10. Como $10 > 8$, se resta 8 para obtener 2 y luego se calcula $d_2 = 2 \oplus 3 = 1$. Obsérvese que $x \oplus 3$ se puede calcular al tomar el complemento de los dos últimos bits de x .

Es bueno observar las representaciones binarias en 3 bits de d_1, d_2, \dots, d_7 , que se muestran en la figura 4.16, junto con su método de cálculo. Obsérvese que la multiplicación por 2, en binario corresponde a un desplazamiento hacia la izquierda. De ahí se obtiene una indicación acerca del origen del término «sucesión de registros de desplazamiento».

Se deberá verificar que también se genera una permutación de 1, 2, ..., 7 si se elige $k = 5$, pero que otros valores de k fracasan. \square

	$d_1 = 101 = 5$
desplazamiento	1010
supresión del primer 1	010
$\oplus 3$	$d_2 = 001 = 1$
desplazamiento	$d_3 = 010 = 2$
desplazamiento	$d_4 = 100 = 4$
desplazamiento	1000
supresión del primer 1	000
$\oplus 3$	$d_5 = 011 = 3$
desplazamiento	$d_6 = 110 = 6$
desplazamiento	1100
supresión del primer 1	100
$\oplus 3$	$d_7 = 111 = 7$

Fig. 4.16. Cálculo de una sucesión de registros de desplazamiento.

Reestructuración de tablas de dispersión

Si se usa una tabla de dispersión abierta, el tiempo medio de las operaciones crece con N/B , cantidad que crece con rapidez conforme el número de elementos excede el número de cubetas. De manera similar, para una tabla de dispersión cerrada, como se vio en la figura 4.15, la eficiencia disminuye a medida que N se acerca a B , y no es posible que N exceda a B .

Para mantener el tiempo constante por operación, que en teoría es posible con las tablas de dispersión, se sugiere que si N crece demasiado, por ejemplo $N \geq 0.9B$ para una tabla cerrada o $N \geq 2B$ para una abierta, sencillamente se produzca una nueva tabla de dispersión con el doble de cubetas. La inserción de los elementos actuales del conjunto en la nueva tabla llevará, en promedio, menos tiempo que el requerido para insertarlas en la tabla más pequeña, y ese costo se compensa cuando se realizan operaciones de diccionario posteriores.

4.9 Realización del TDA CORRESPONDENCIA

Recuérdese el análisis del TDA CORRESPONDENCIA del capítulo 2, en el cual se definió una correspondencia como una función de los elementos de un dominio a los elementos de un contradominio. Las operaciones de este TDA son:

1. ANULA(A) inicializa la correspondencia A , dejando cada elemento del dominio sin valor asignado en el contradominio.
2. ASIGNA(A, d, r) define $A(d)$ como r .
3. CALCULA(A, d, r) devuelve verdadero y hace que r tome el valor $A(d)$ si $A(d)$ está definido; en caso contrario, devuelve falso.

Las tablas de dispersión son una forma efectiva de obtener correspondencias. Las operaciones ASIGNA y CALCULA se realizan de la misma manera que las operaciones INSERTA y MIEMBRO para un diccionario. Considerese primero una tabla de dispersión abierta. Se supone que la función de dispersión $h(d)$ asocia elementos del dominio con números de cubeta. Mientras que para el diccionario las cubetas consistían en listas enlazadas de elementos, para la correspondencia se necesita una lista de elementos del dominio pareados con sus correspondientes valores del contradominio. Para esto, la definición de celda de la figura 4.12 se reemplaza por

```
type
  tipo_celda = record
    elemento_dominio: tipo_dominio;
    contradominio: tipo_contradominio;
    sig: ^tipo_celda
  end
```

donde tipo_dominio y tipo_contradominio son cualesquiera tipos que tengan los elementos del dominio y contradominio de la correspondencia. La declaración de CORRESPONDENCIA queda como

```
type
  CORRESPONDENCIA = array[0..B - 1] of ^tipo_celda
```

Este arreglo es el arreglo de cubetas para una tabla de dispersión. El procedimiento ASIGNA está escrito en la figura 4.17. Los códigos de ANULA y CALCULA se dejan como ejercicio.

De manera parecida, es posible usar una tabla de dispersión cerrada como una correspondencia. Se definen las celdas como campos de elementos del dominio y de elementos del contradominio, y se declara una CORRESPONDENCIA como un arreglo de celdas. Como ocurre con las tablas de dispersión abiertas, se hace que la función de dispersión se aplique a los elementos del dominio, no a los del contradominio. Se deja como ejercicio la realización de las operaciones de correspondencia usando una tabla de dispersión cerrada.

4.10 Colas de prioridad

La cola de prioridad es un TDA basado en el modelo de conjunto con las operaciones INSERTA y SUPRIME-MIN, así como con ANULA para asignar valores iniciales a la estructura de datos. Con el fin de definir la nueva operación SUPRIME-MIN, primero se supone que los elementos del conjunto tienen una función «prioridad» definida sobre ellos; para cada elemento a , $p(a)$, la *prioridad* de a es un número real o, en términos más generales, un miembro de algún conjunto linealmente ordenado. La operación INSERTA tiene el significado habitual, mientras que SUPRIME-MIN es una función que devuelve un elemento de más baja prioridad y, como efecto colateral, lo suprime del conjunto. Así, como su nombre indica, SUPRIME-MIN es una combinación de las operaciones SUPRIME y MIN estudiadas antes en este capítulo.

```

procedure ASIGNA ( var A: CORRESPONDENCIA; a: tipo_dominio;
    r: tipo_contradominio );
var
    cubeta: integer;
    actual: ↑ tipo_celda;
begin
    cubeta := h(d);
    actual := A[cubeta];
    while actual < > nil do
        if actual ↑. elemento_dominio = d then begin
            actual ↑. contradominio := r; { reemplaza el valor anterior
                de d }
            return
        end
        else
            actual := actual ↑. sig;
    { en este punto, d no se encontró en la lista }
    actual := A[cubeta]; { usa actual para almacenar la primera celda }
    new(A[cubeta]);
    A[cubeta] ↑. elemento_dominio := d;
    A[cubeta] ↑. contradominio := r;
    A[cubeta] ↑. sig := actual
end; { ASIGNA }
```

Fig. 4.17. El procedimiento ASIGNA para una tabla de dispersión abierta.

Ejemplo 4.9. El término «cola de prioridad» proviene de la siguiente forma de usar este TDA. La palabra «cola» sugiere la espera de cierto servicio por ciertas personas u objetos, y la palabra «prioridad» sugiere que ese servicio no se proporciona por medio de la disciplina «primero en llegar, primero en ser atendido», que es la base del TDA COLA, sino que cada persona tiene una prioridad basada en la urgencia de su necesidad. Un ejemplo es la sala de espera de un hospital, donde los pacientes

con problemas potencialmente mortales serán atendidos antes que los demás, sin importar el tiempo de espera.

Como ejemplo más corriente del uso de las colas de prioridad, un sistema de cómputo de tiempo compartido necesita mantener un conjunto de procesos que esperan servicio. En general, los diseñadores de estos sistemas desean hacer que los procesos cortos parezcan instantáneos (en la práctica, una respuesta en uno o dos segundos parece instantánea), de modo que a estos procesos se les da prioridad sobre los que ya tienen un consumo sustancial de tiempo. Un proceso que requiera varios segundos de tiempo de cómputo no puede ser instantáneo, por lo que es razonable la estrategia de diferir estos procesos hasta que todos los que tienen posibilidades de parecer instantáneos hayan sido atendidos. Sin embargo, si no se tiene cuidado, los procesos que ya hayan tomado un tiempo mucho mayor que el promedio, quizás no vuelvan a obtener una porción de tiempo y queden esperando para siempre.

Una posible manera de favorecer los procesos cortos sin bloquear los largos, consiste en dar a un proceso P la prioridad $100t_{\text{cons}}(P) - t_{\text{inic}}(P)$. El parámetro t_{cons} es el tiempo consumido hasta el momento por el proceso, y t_{inic} es el momento en que se inició el proceso, medido a partir de cierto «instante cero». Obsérvese que, en general, las prioridades serán enteros negativos grandes, a menos que se elija medir t_{inic} a partir de un instante en el futuro. Obsérvese también que 100 en la fórmula anterior es un «número mágico»; se selecciona de forma que sea algo mayor que el mayor número de procesos que puedan estar activos a la vez. Así, si siempre se elige el proceso con el número de prioridad más bajo y no hay demasiados procesos cortos, a la larga, un proceso que no termine rápido recibirá el 1% del tiempo de procesador. Si esto es mucho o muy poco, otra constante puede reemplazar el 100 de la fórmula de la prioridad.

Los procesos se representan mediante registros con un identificador de proceso y un número de prioridad. Esto es, se define

```
type
  tipo_proceso = record
    id: integer;
    prioridad: integer
  end;
```

La prioridad de un proceso es el valor del campo de prioridad, que se ha definido como un entero. La función de prioridad se puede definir como sigue:

```
function p (a: tipo_proceso): integer;
begin
  return (a.prioridad)
end;
```

Para seleccionar el proceso que reciba una porción de tiempo, el sistema mantiene una cola de prioridad ESPERA con elementos de tipo tipo_proceso, y emplea dos procedimientos, *inicial* y *selecciona*, para manejar la cola de prioridad mediante las operaciones *INSERTA* y *SUPRIME_MIN*. Siempre que un proceso comienza,

se llama al procedimiento *inicial*, el cual coloca un registro para ese proceso en ESPERA. El procedimiento *selecciona* es llamado cuando el sistema dispone de una porción de tiempo assignable a algún proceso. El registro del proceso que resulte seleccionado se elimina de ESPERA, pero *selecciona* lo retiene para reingresarlo en la cola con una nueva prioridad, que es la anterior incrementada en 100 veces la cantidad de tiempo utilizada.

Se emplea la función *tiempo_actual*, que devuelve el tiempo actual en cualesquier unidades que el sistema use, como microsegundos, y el procedimiento *ejecuta* (*P*) que hace que el proceso con identificador *P* se ejecute durante una porción de tiempo. La figura 4.18 muestra los procedimientos *inicial* y *selecciona*. □

```

procedure inicial ( P: integer );
{ inicial coloca un proceso con identificador P en la cola }
var
    proceso: tipo_proceso;
begin
    proceso.id := P,
    proceso.prioridad := -tiempo_actual;
    INSERTA (proceso, ESPERA)
end; { inicial }

procedure selecciona;
{ selecciona asigna una porción de tiempo al proceso de mayor prioridad }
var
    tiempo_inicio, tiempo_fin: integer;
    proceso: tipo_proceso;
begin
    proceso := ↑ SUPRIME_MIN(ESPERA)
    { SUPRIME_MIN devuelve un apuntador al elemento eliminado }
    tiempo_inicio := tiempo_actual;
    ejecuta(proceso.id);
    tiempo_fin := tiempo_actual;
    proceso.prioridad := proceso.prioridad + 100* (tiempo_fin -
        tiempo_inicio);
    { ajusta la prioridad para incorporar la cantidad de tiempo usado }
    INSERTA (proceso, ESPERA)
    { devuelve el proceso seleccionado a la cola con la nueva prioridad }
end; { selecciona }

```

Fig. 4.18. Asignación de tiempo a procesos.

4.11 Realizaciones de colas de prioridad

Con excepción de la tabla de dispersión, todas las realizaciones estudiadas hasta el momento para los conjuntos son también apropiadas para las colas de prioridad. La razón de que la tabla de dispersión no sea adecuada, es que no proporciona

un medio conveniente de encontrar el menor elemento, por lo que la dispersión sólo agrega complicaciones y no mejora el rendimiento respecto a una lista enlazada, por ejemplo.

En caso de manejarse una lista enlazada, se tienen las opciones de clasificarla o dejarla sin clasificar. Si la lista se clasifica, es fácil encontrar un mínimo; basta tomar su primer elemento. En cambio, la inserción requiere revisar en promedio la mitad de la lista para mantenerla clasificada. Por otro lado, es posible dejar la lista sin clasificar, lo que facilita la inserción y dificulta la selección del mínimo.

Ejemplo 4.10. Se obtendrá SUPRIME_MIN para una lista no clasificada de elementos del tipo tipo_proceso, que se definió en el ejemplo 4.9. El encabezado de la lista es una celda vacía; las aplicaciones de INSERTA y ANULA son directas, y la aplicación por medio de listas clasificadas se deja como ejercicio. La figura 4.19 proporciona la declaración de las celdas, para el tipo COLA_PRIORIDAD, y para el procedimiento SUPRIME_MIN. □

Realización de colas de prioridad mediante árboles parcialmente ordenados

Tanto si se desea emplear listas clasificadas como no clasificadas para representar colas de prioridad, se debe gastar un tiempo proporcional a n para realizar INSERTA o SUPRIME_MIN en conjuntos de tamaño n . Existe otra realización en la que ambas operaciones requieren $O(\log n)$ pasos, una mejora sustancial para valores grandes de n (como $n \geq 100$). La idea básica consiste en organizar los elementos de la cola de prioridad en un árbol binario que esté lo más balanceado posible; hay un ejemplo en la figura 4.20. En el nivel más bajo, en el cual pueden faltar algunas hojas, se exige que las hojas que falten estén a la derecha de las que se encuentran en el nivel más bajo.

Más importante aún, el árbol es *parcialmente ordenado*, es decir, la prioridad del nodo v no es mayor que la de los hijos de v , donde la prioridad de un nodo es el número de prioridad del elemento almacenado en ese nodo. Obsérvese en la figura 4.20 que los nodos con número de prioridad pequeño no necesitan estar a niveles más altos que los de número de prioridad más grande. Por ejemplo, el nivel tres tiene 6 y 8, que son números de prioridad menores que el 9, que aparece en el nivel dos. En cambio, el padre de 6 y 8 tiene prioridad 5, la cual es, y debe ser, al menos tan pequeña como las prioridades de sus hijos.

Para ejecutar SUPRIME_MIN, se devuelve el elemento de menor prioridad, que, como se puede ver con facilidad, debe estar en la raíz. Sin embargo, si lo único que se hace es eliminar la raíz, lo que queda ya no es un árbol. Para que se mantenga la propiedad de árbol parcialmente ordenado, lo más balanceado y con hojas tan a la izquierda posible, se toma la hoja de más a la derecha del nivel más bajo y se coloca de modo provisional en la raíz. La figura 4.21(a) muestra este cambio a partir de la figura 4.20. Despues, este elemento se coloca en el árbol lo más abajo posible, intercambiándolo con el hijo que tenga la prioridad más baja, hasta que el elemento

esté en una hoja o en una posición cuya prioridad no sea mayor que la de cualquiera de sus hijos.

En la figura 4.21(a) se debe intercambiar la raíz con su hijo de menor prioridad, el cual tiene prioridad 5. El resultado de este intercambio se muestra en la figura 4.21(b). El elemento que se está desplazando hacia abajo es todavía mayor que sus hijos, los cuales tienen prioridades 6 y 8. Al intercambiarlo con el menor de los dos, 6, se logra el árbol de la figura 4.21(c). Este árbol tiene ya la propiedad de ser parcialmente ordenado.

En esta filtración, si un nodo v tiene un elemento con prioridad a y sus hijos tienen prioridades b y c , y si al menos b o c son menores que a , el intercambio de a con el menor de b y c logra que v tenga un elemento cuya prioridad sea menor que la de cualquiera de sus hijos. Para demostrar esto, supóngase que $b \leq c$. Después del intercambio, el nodo v adquiere la prioridad b , y sus hijos, las prioridades a y c . Se supuso que $b \leq c$ y se dijo que a es mayor que b o c . Por tanto, es seguro que se cumplirá que $b \leq a$. Así, el proceso de inserción esbozado con anterioridad filtrará un elemento hacia abajo en el árbol hasta que no haya más violaciones de la propiedad de orden parcial.

```

type
  tipo_celda = record
    elemento: tipo_proceso;
    siguiente: ↑ tipo_celda
  end;

  COLA_CON_PRIORIDAD = ↑ tipo_celda;
  { la celda apuntada es un encabezado de lista }

function SUPRIME_MIN ( var A: COLA_CON_PRIORIDAD ) : ↑ tipo_celda;
  var
    actual: ↑ tipo_celda; { celda anterior a la "examinada" }
    menor_prioridad: integer; { prioridad más baja encontrada }
    preganador: ↑ tipo_celda; { celda anterior a la del elemento con menor
      prioridad }

  begin
    if A ↑.siguiente = nil then
      error ('no puede encontrar el menor de una lista vacía')
    else begin
      menor_prioridad := p(A ↑.siguiente^.elemento);
      { p devuelve la prioridad del primer elemento. Obsérvese que
        A apunta a una celda de encabezado que no contiene un
        elemento }

      preganador := A;
      actual := A ↑.siguiente;
      while actual↑.siguiente <> nil do begin
        { compara las prioridades del ganador actual con las del elemento
          siguiente }
    
```

```

if  $p(actual^{\dagger}.siguiente^{\dagger}.elemento) < menor\_prioridad$  then begin
    peganador := actual;
    menor_prioridad :=  $p(actual^{\dagger}.siguiente^{\dagger}.elemento)$ 
end;
actual :=  $actual^{\dagger}.siguiente$ 
end;
SUPRIME_MIN := peganador^{\dagger}.siguiente;
{ devuelve el apuntador al ganador }
peganador^{\dagger}.siguiente := peganador^{\dagger}.siguiente^{\dagger}.siguiente
{ elimina el ganador de la lista }
end
end; { SUPRIME_MIN }

```

Fig. 4.19. Realización de una cola de prioridad mediante listas enlazadas.

Obsérvese también que SUPRIME_MIN consume un tiempo $O(\log n)$, aplicado a un conjunto de n elementos. Esto ocurre porque ningún camino del árbol tiene más de $1 + \log n$ nodos y el proceso de forzar el descenso de un elemento en el árbol consume un tiempo constante por nodo. Obsérvese que para cualquier constante c , la cantidad $c(1 + \log n)$ es, a lo sumo, igual a $2c \log n$ para $n \geq 2$. Por consiguiente, $c(1 + \log n)$ es $O(\log n)$.

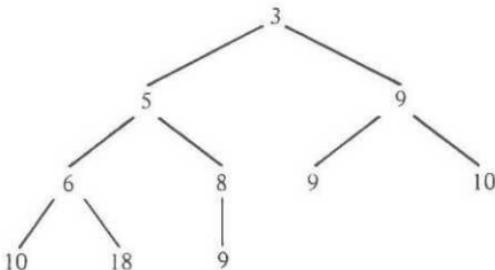


Fig. 4.20. Un árbol parcialmente ordenado.

Ahora se considerará cómo debería trabajar INSERTA. En primer lugar, se coloca el nuevo elemento lo más a la izquierda posible en el nivel más bajo, iniciando un nuevo nivel si el más bajo actual se encuentra lleno en su totalidad. La figura 4.22(a) muestra el resultado de colocar un elemento con prioridad 4 en el árbol de la figura 4.20. Si el nuevo elemento tiene una prioridad más baja que la de su padre, se intercambia con él. El nuevo elemento queda entonces en una posición de menor prioridad que cualquiera de sus hijos, pero puede ser también que tenga menor prioridad que su padre, en cuyo caso se debe intercambiar con él y repetir el proceso hasta que el nuevo elemento llegue hasta la raíz o alcance una posición en la que tenga mayor prioridad que su padre. La figura 4.22(b) y (c) muestra el ascenso de 4 en este proceso.

Se podría demostrar que los pasos anteriores dan como resultado un árbol parcialmente ordenado. No se intentará una demostración rigurosa de este hecho, pero sí se observará que un elemento con prioridad a puede llegar a ser el padre de un elemento con prioridad b de tres maneras. (En el razonamiento siguiente se identificará un elemento con su prioridad.)

1. a es el nuevo elemento y asciende en el árbol reemplazando al padre anterior de b . Sea c la prioridad del padre anterior de b . Entonces, $a < c$, de otro modo, el intercambio no habría tenido lugar. Pero $c \leq b$, puesto que el árbol original estaba parcialmente ordenado. Por tanto, $a < b$. Tómese como ejemplo la figura 4.22(c), donde 4 se convierte en el padre de 6 después de reemplazar a un padre de mayor prioridad, 5.
2. a fue desplazado hacia abajo en el árbol debido a un intercambio con el nuevo elemento. En este caso, a tiene que haber sido un antecesor de b en el árbol parcialmente ordenado original. Por tanto, $a \leq b$. Por ejemplo, en la figura 4.22(c), 5 se convierte en el padre de los elementos con prioridad 8 y 9. Originalmente, 5 era el padre del primero y el «abuelo» del segundo.
3. b sería el nuevo elemento y asciende hasta ser un hijo de a . Si ocurriera que $a > b$, entonces a y b se intercambiarían en el siguiente paso, con lo que se eliminaría la violación de la propiedad parcialmente ordenada.

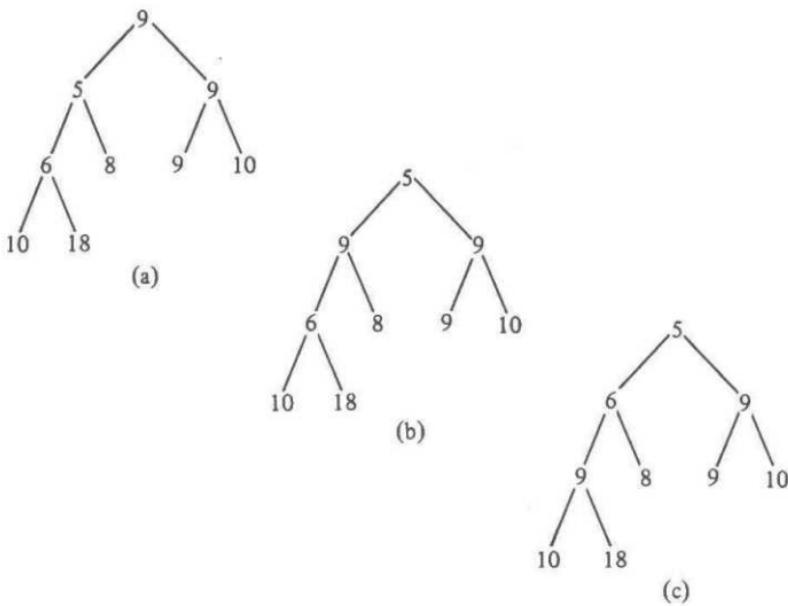


Fig. 4.21. Descenso de un elemento en un árbol.

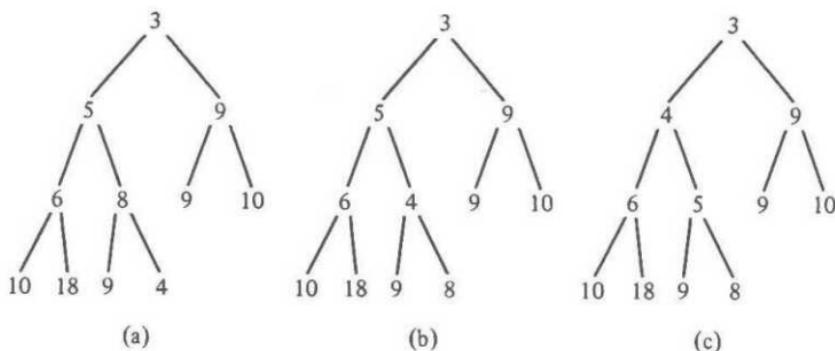


Fig. 4.22. Inserción de un elemento.

El tiempo necesario para efectuar una inserción es proporcional a la distancia que el elemento asciende en el árbol. Como en el caso de SUPRIME-MIN, se observa que esta distancia tal vez no sea mayor que $1 + \log n$, de modo que tanto INSERTA como SUPRIME-MIN dan $O(\log n)$ pasos.

Realización por arreglos de los árboles parcialmente ordenados

El hecho de que los árboles que se han estado considerando sean binarios, lo más balanceados posible, y tengan sus hojas del nivel más bajo lo más a la izquierda posible, permitirá utilizar para ellos una representación poco usual, conocida como *montículo (heap)*. De haber n nodos, se usan las n primeras posiciones de un arreglo A . $A[1]$ contiene la raíz. El hijo izquierdo del nodo $A[i]$, si existe, está en $A[2i]$, y el hijo derecho, si existe, en $A[2i + 1]$. Una manera equivalente de expresar esto es decir que el padre de $A[i]$ es $A[i \text{ div } 2]$, para $i > 1$. Otra observación, también equivalente, es que los nodos del árbol llenan $A[1], A[2], \dots, A[n]$ nivel a nivel, desde arriba, y de izquierda a derecha dentro de cada nivel. Por ejemplo, la figura 4.20 corresponde a un arreglo que contiene los elementos 3, 5, 9, 6, 8, 9, 10, 10, 18, 9.

Se puede declarar una cola de prioridad con elementos de cierto tipo, como tipo_proceso del ejemplo 4.9, como un arreglo de tipos tipo_proceso y un entero *últ* que indica cuál es el último elemento de un arreglo que en un momento dado está en uso. Si se supone que *tam_máx* es el tamaño deseado para los arreglos de colas de prioridad, se puede declarar:

```

type
  COLA_DE_PRIORIDAD = record
    contenido: array[1..tam_máx] of tipo_proceso;
    últ: integer
  end;

```

La realización de las operaciones con colas de prioridad se puede ver en la figura 4.23.

```

procedure ANULA ( var A: COLA_DE_PRIORIDAD );
begin
  A.últ := 0
end; { ANULA }

procedure INSERTA ( x: tipo_proceso; var A: COLA_DE_PRIORIDAD );
var
  i: integer;
  temp: tipo_proceso;
begin
  if A.últ >= long_máx then
    error('la cola de prioridad está llena')
  else begin
    A.últ := A.últ + 1;
    A.contenido[A.últ] := x;
    i := A.últ; { i es el índice de la posición actual de x }
    while (i > 1) and (p(A.contenido[i]) < p(A.contenido[i div 2])) do
      begin { sube x en el árbol y lo intercambia con su padre
              de mayor prioridad. Recuérdese que p calcula
              la prioridad de un elemento de tipo_proceso }
      temp := A.contenido[i];
      A.contenido[i] := A.contenido[i div 2];
      A.contenido[i div 2] := temp;
      i := i div 2
    end
  end
end;
end; { INSERTA }

function SUPRIME_MIN ( var A: COLA_DE_PRIORIDAD ) : ↑ tipo_proceso;
var
  i, j: integer;
  temp: tipo_proceso;
  mínimo: ↑ tipo_proceso;
begin
  if A.últ = 0 then
    error('la cola de prioridad está vacía')
  else begin
    new(mínimo);
    mínimo↑ := A.contenido[1];
    { debe devolverse un apuntador a una copia de la raíz de A }
    A.contenido[1] := A.contenido[A.últ];
    A.últ := A.últ - 1;
    { mueve el último elemento hacia el principio }
    i := 1; { i es la posición actual del que antes era
              el último elemento }
  end
end;

```

```

while i < = A.últ div 2 do begin
    { lleva hacia abajo el anterior último elemento
      en el árbol }
    if (p(A.contenido[2*i]) < p(A.contenido[2*i + 1]))
        or (2*i = A.últ) then
            j := 2*i
        else
            j := 2*i + 1;
    { j será el hijo de i que tiene la menor prioridad o bien,
      si 2*i = A.últ, j será el único hijo de i }
    if p(A.contenido[i]) > p(A.contenido[j]) then begin
        { intercambia el anterior último elemento con el hijo
          de menor prioridad }
        temp := A.contenido[i];
        A.contenido[i] := A.contenido[j];
        A.contenido[j] := temp;
        i := j
    end
    else
        return (mínimo) { no puede meterlo más }
end;
return (mínimo) { se metió hasta llegar a una hoja }
end
end; { SUPRIME_MIN }

```

Fig. 4.23. Realización de colas de prioridad mediante arreglos.

4.12 Algunas estructuras complejas de conjuntos

En esta sección se considerarán dos usos más complejos de los conjuntos para representar datos. El primer problema será el de representar asociaciones de muchos a muchos, como las que pueden presentarse en un sistema de bases de datos. Un segundo caso de estudio mostrará cómo un par de estructuras de datos que representan el mismo objeto (una correspondencia en el ejemplo que se analizará), pueden dar una representación más eficiente que cualquiera de ellas por separado.

Asociaciones de muchos a muchos y la estructura de listas múltiples

Un ejemplo de asociación de muchos a muchos entre estudiantes y cursos está representada en la figura 4.24. Esta asociación se llama «de muchos a muchos» porque puede haber muchos estudiantes que tomen un curso y cada estudiante puede tomar muchos cursos.

	CS101	CS202	CS303
Alan	X	X	X
Alejandro			
Alicia			X
Amanda	X		
Andrés		X	X
Angélica	X		X

*Inscripción***Fig. 4.24.** Ejemplo de relación entre estudiantes y cursos.

De vez en cuando, el responsable de los cursos puede desear insertar o eliminar estudiantes de los cursos, determinar cuáles estudiantes están tomando un curso dado, o saber qué cursos está tomando un estudiante dado. La estructura de datos más simple con la que es posible responder estas preguntas es obvia; basta usar el arreglo bidimensional sugerido por la figura 4.24, donde el valor 1 (o verdadero) reemplaza las X y el valor 0 (o falso) reemplaza los espacios.

Por ejemplo, para insertar un estudiante en un curso se necesita una correspondencia, *CE*, tal vez aplicada mediante una tabla de dispersión, que traduzca nombres de estudiante a índices del arreglo y otra, *CC*, que traduzca nombres de curso a índices del arreglo. Entonces, para insertar al estudiante *e* en el curso *c*, simplemente se asigna

$$\text{Inscripción}[\text{CE}(e), \text{CC}(c)] := 1.$$

Las supresiones se realizan haciendo que este elemento tome el valor 0. Para hallar los cursos que toma un estudiante de nombre *e*, se recorre la fila *CE(e)* y, de manera similar, se recorre la columna *CC(c)* para hallar los estudiantes que están en el curso *c*.

¿Por qué sería deseable buscar una estructura de datos más apropiada? Considérese una universidad grande con, tal vez, 1000 cursos y 20 000 estudiantes, que toman en promedio tres cursos cada uno. El arreglo sugerido por la figura 4.24 tendría 20 000 000 de elementos, de los cuales 60 000, o el 0.3 %, tendrían el valor 1 †. Un arreglo así, que recibe el nombre de *dispersa*, para indicar que casi todos sus elementos valen cero, se puede representar en mucho menos espacio con sólo listar las entradas que no son nulas. Más aún, se puede gastar mucho tiempo revisando una columna de 20 000 entradas en busca de, en promedio, 60 que no son nulos y las revisiones de filas pueden llevar también mucho tiempo.

Una forma de mejorar esto, consiste en plantear el problema como si se tratara del mantenimiento de una colección de conjuntos. Dos de esos conjuntos son *E* y *C*, los conjuntos de todos los estudiantes y todos los cursos. Cada elemento de *E* es en realidad un registro de un tipo como

† Si ésta fuera una base de datos real, el arreglo se guardaría en almacenamiento secundario. Sin embargo, esta estructura de datos malgastaría mucho espacio.

```

type
  tipo_estudiante = record
    ident: integer;
    nombre: array[1..30] of char;
  end

```

y habrá que inventar un tipo de registro parecido para los cursos. Para obtener la estructura pensada, se necesita un tercer conjunto, I , cuyos elementos representen las inscripciones. Cada uno de los elementos de I representará una de las celdas del arreglo de la figura 4.24 que contenga una X. Los elementos de I serían registros de un tipo fijo. Hasta este punto no se sabe qué campos van en esos registros †, pero pronto se sabrá de ellos. Por el momento, basta postular que hay un registro de inscripción por cada entrada marcada con X en la matriz y que los registros de inscripción son distinguibles unos de otros de alguna manera.

También se requieren conjuntos que representen respuestas a las preguntas cruciales: dado un estudiante o un curso, ¿cuáles son los cursos o estudiantes, según el caso, asociados? Sería interesante tener, para cada estudiante e , un conjunto C_e de todos los cursos que e estuviera tomando y, por otro lado, un conjunto E_c de todos los estudiantes que siguen el curso c . Tales conjuntos serían difíciles de obtener, porque no habría límite para el número de conjuntos en los que podría estar cualquier elemento, lo cual daría lugar a complicados registros de estudiantes y cursos. En cambio, si se podría hacer que E_c y C_e fueran conjuntos de apuntadores, en vez de registros, pero existe un método que permite un ahorro significativo de espacio y ofrece respuestas igual de rápidas a las preguntas sobre estudiantes y cursos.

Sea cada conjunto C_e el conjunto de registros de inscripción correspondientes al estudiante e y algún curso c . Es decir, si se considera una inscripción como un par (e, c) , entonces

$$C_e = \{(e, c) \mid e \text{ está tomando el curso } c\}.$$

De la misma manera, se puede definir

$$E_c = \{(e, c) \mid e \text{ está tomando el curso } c\}.$$

Obsérvese que la única diferencia en el significado de las propiedades de estos dos conjuntos es que, en el primer caso, e es constante, y en el segundo lo es c . Por ejemplo, con base en la figura 4.24, $C_{\text{Alejandro}} = \{(\text{Alejandro}, \text{CS101}), (\text{Alejandro}, \text{CS202})\}$ y $E_{\text{CS101}} = \{(\text{Alejandro}, \text{CS101}), (\text{Amanda}, \text{CS101}), (\text{Angélica}, \text{CS101})\}$.

Estructuras de listas múltiples

En general, una estructura de listas múltiples es cualquier colección de celdas, donde algunas contienen más de un apuntador y pueden, por tanto, pertenecer a más de una lista a la vez. Para cada tipo de celda de una estructura de listas múltiples, es importante distinguir entre los campos apuntadores, de modo que se pueda se-

† En la práctica, sería útil colocar algunos campos, como calificaciones y otros, en los registros de inscripción, pero el planteamiento original del problema no los requiere.

guir una lista en particular sin que haya confusión con respecto a cuál de los diferentes apuntadores de una celda en particular se debe seguir.

Para el caso en cuestión, es posible colocar un campo apuntador en cada registro de estudiante y curso que apunte al primer registro de inscripción en C_e o E_c , respectivamente. Cada registro de inscripción necesita dos campos apuntadores: uno que se llamará c_sig , para apuntar a la siguiente inscripción en la lista que representa al conjunto C_e , al cual pertenece el registro, y otro, e_sig , para apuntar al siguiente elemento del conjunto E_c al que pertenece.

Resulta que un registro de inscripción no indica en forma explícita el estudiante ni el curso que representa. Esta información está implícita en las listas en las cuales aparece el registro de inscripción. Llámese *propietarios* del registro de inscripción a los registros de estudiante y de curso que encabezan estas listas. Entonces, para poder decir qué cursos toma el estudiante e , es preciso examinar los registros de inscripción de C_e y hallar para cada uno su registro de curso propietario. Esto podría hacerse colocando un apuntador en cada registro de inscripción para el registro de curso propietario, y podría necesitarse también un apuntador para el registro de estudiante propietario.

Si bien se pueden usar esos apuntadores con el objeto de responder a las preguntas en el menor tiempo posible, existe la alternativa de lograr un ahorro considerable de espacio †, al costo de hacer más lentos algunos cálculos, si se eliminan los apuntadores y se coloca al final de cada lista E_c un apuntador al registro de curso propietario, y al final de cada lista C_e , un apuntador al registro de estudiante propietario. En esta forma, cada registro de estudiante y de curso será parte de un anillo que incluya todos los registros de inscripción de los cuales es propietario. Estos anillos están representados en la figura 4.25, para los datos de la figura 4.24. Obsérvese que los registros de inscripción tienen como primer apuntador c_sig , y como segundo, e_sig .

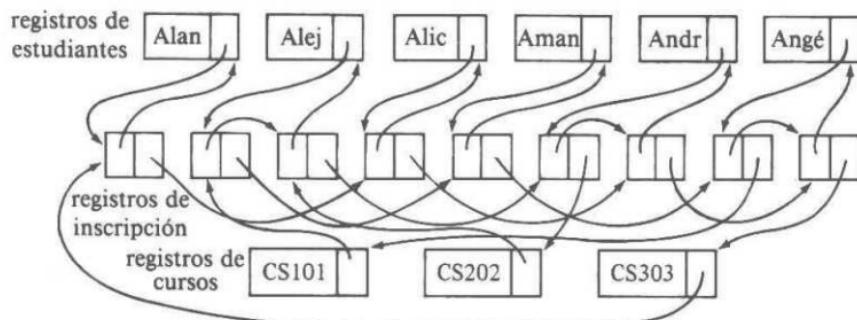


Fig. 4.25. Representación de la figura 4.24 con listas múltiples.

† Obsérvese que es probable que haya muchos más registros de inscripción que registros de estudiantes o cursos, de modo que reducir de tamaño los registros de inscripción permite disminuir la necesidad total de espacio casi en la misma proporción.

Ejemplo 4.11. Para responder a una pregunta como «¿qué estudiantes toman el curso CS101?», se busca el registro de curso correspondiente a CS101. Cómo encontrar este registro depende de la forma en que se mantiene el conjunto de cursos. Por ejemplo, podría haber una tabla de dispersión que tuviera todos esos registros, y para obtener el registro deseado se aplicaría cierta función de dispersión a «CS101».

Luego se sigue el apuntador del registro de CS101 al primer registro de inscripción del anillo de CS101. Este es, en la figura, el segundo registro de inscripción desde la izquierda. Después, hay que buscar al estudiante propietario de este registro de inscripción, lo cual se logra siguiendo los apuntadores *c-sig* (el primero en los registros de inscripción), hasta llegar a un registro de estudiantes \dagger . En este caso, después del tercer registro de inscripción se llega al registro de estudiante para Alejandro; ahora se sabe que Alejandro está tomando el curso CS101.

A continuación, se debe encontrar el siguiente estudiante CS101, y se hace siguiendo el apuntador *e-sig* (el segundo apuntador) del segundo registro de inscripción, el cual conduce al quinto registro de inscripción. El apuntador *c-sig* de ese registro conduce directamente a su propietario, Amanda, de modo que ella está en el curso CS101. Por último, se sigue el apuntador *e-sig* del quinto registro de inscripción hasta el octavo. El anillo de apuntadores *c-sig* de ese registro conduce al noveno registro de inscripción, y de ahí, al registro de estudiante de Angélica, de modo que ella está inscrita en CS101. El apuntador *e-sig* del octavo registro de inscripción conduce de vuelta a CS101, por tanto no hay más estudiantes inscritos en CS101. \square

La operación del ejemplo 4.11 se puede expresar en términos abstractos como sigue:

```
for cada registro de inscripción en el conjunto para CS101 do begin
    e := el estudiante propietario del registro de inscripción;
    imprime (e)
end
```

Esta asignación a *e* se puede expresar como

```
f := e;
repeat
    f := f. c-sig
until
    f es un apuntador a un registro de estudiante;
    e := campo nombre-estudiante del registro apuntado por f;
```

donde *e* es un apuntador al primer registro de inscripción en el conjunto para CS101.

Para aplicar una estructura como la de la figura 4.25 en Pascal, se necesita un solo tipo de registro, con variantes para cubrir los casos de registros de estudiantes,

\dagger Se debe tener alguna forma de identificar los tipos de registro y se dará momentáneamente una manera de hacerlo.

cursos e inscripciones. En Pascal esto tiene que ser así, ya que los campos *c-sig* y *e-sig* tienen la capacidad de apuntar a tipos de registro distintos. Sin embargo, esta estructura resuelve uno de los problemas que quedan, pues ahora es fácil decir a qué tipo de registro se llega conforme se recorre un anillo. La figura 4.26 muestra una posible declaración de los registros y un procedimiento que imprime los nombres de los estudiantes inscritos en un curso en particular.

Estructuras de datos duales para mayor eficiencia

Con frecuencia, un problema aparentemente simple de representación de un conjunto o correspondencia, conlleva un difícil problema de elección de estructuras de datos. La elección de una estructura de datos para el conjunto simplifica ciertas operaciones, pero hace que otras lleven demasiado tiempo, y, al parecer, no existe una estructura de datos que haga más sencillas todas las operaciones. En tales casos, la solución suele ser el uso simultáneo de dos o más estructuras diferentes para el mismo conjunto o correspondencia.

Supóngase que se desea mantener una «escala de tenistas» en la que cada jugador esté situado en un solo «peldaño». Los jugadores nuevos se agregan en la base de la escala, es decir, en el peldaño con la numeración más alta. Un jugador puede retar a otro que esté en el peldaño inmediato superior, y si le gana, cambia de peldaño con él. Se puede representar esta situación mediante un tipo de datos abstracto cuyo modelo fundamental sea una correspondencia de nombres (cadenas de caracteres) con peldaños (los enteros 1, 2, ...).

Las tres operaciones a realizar son:

1. AGREGA(*nombre*) agrega a la persona nombrada al peldaño de numeración más alta.
2. RETA(*nombre*) es una función que devuelve el nombre de la persona del peldaño *i* - 1 si el jugador nombrado está en el peldaño *i*, *i* > 1.
3. CAMBIA(*i*) intercambia los nombres de los jugadores que estén en los peldaños *i* e *i* - 1, *i* > 1.

type

```

tipo_e = array[1..20] of char;
tipo_c = array[1..5] of char;
clase_registro = (estudiante, curso, inscripción);
tipo_registro = record
    case clase : clase_registro of
        estudiante : (nombre_estudiante: tipo_e;
                      primer_curso: ^tipo_registro);
        curso: (nombre_curso: tipo_c;
                  primer_estudiante: ^tipo_registro);
        inscripción: (c_sig, e_sig: ^tipo_registro)
end;
```

```

procedure imprime_estudiantes ( nombre_c: tipo_c );
var
  c, e, f: tipo_registro;
begin
  c := apuntador al registro de curso con c^.nombre_curso = nom-
  bre_c;
  { depende de la aplicación del conjunto curso }
  e := c^.primer_estudiante;
  { e recorre el anillo de inscripciones apuntadas por c }
  while e^.clase = inscripción do begin
    f := e;
    repeat
      f := f^.c.sig
    until
      f^.clase = estudiante;
    { ahora f apunta al estudiante a quien pertenece la inscripción e^. }
    writeln(f^.nombre_estudiante);
    e := e^.e.sig
  end
end

```

Fig. 4.26. Realización de una búsqueda en una lista múltiple.

Obsérvese que se ha elegido pasar a CAMBIA sólo el número de peldaño más alto, mientras que las otras dos operaciones tienen un nombre como argumento.

Como alternativa, se podría considerar el uso de un arreglo *ESCALA*, en la que *ESCALA[i]* sea el nombre de la persona del peldaño *i*. Si además se lleva la cuenta del número de jugadores, la adición de un jugador al primer peldaño desocupado se puede hacer en un pequeño número constante de pasos.

CAMBIA también es fácil, puesto que sólo hay que intercambiar dos elementos del arreglo, pero RETA(*nombre*) requiere examinar todo el arreglo en busca del nombre, lo cual lleva un tiempo $O(n)$, si *n* es el número de jugadores en la escala.

Por otra parte, se podría considerar el uso de una tabla de dispersión para representar la correspondencia de nombres a peldaños. En el supuesto de que es posible mantener el número de cubetas proporcional al número de jugadores, AGRE-GA llevaría en promedio un tiempo $O(1)$. A un reto le llevaría un tiempo promedio $O(1)$ buscar el nombre dado, un tiempo $O(n)$ encontrar el nombre que ocupa el peldaño con el siguiente número más bajo, dado que para eso se necesitaría buscar en toda la tabla de dispersión. El intercambio de jugadores requeriría un tiempo $O(n)$ para hallar los jugadores en los peldaños *i* e *i* - 1.

Supóngase, sin embargo, que se combinan las dos estructuras. Las celdas de la tabla de dispersión contendrán pares compuestos de un nombre y un peldaño, mientras que el arreglo tendrá en *ESCALA[i]* un apuntador a la celda correspondiente al jugador que ocupe el peldaño *i*, como se sugiere en la figura 4.27.

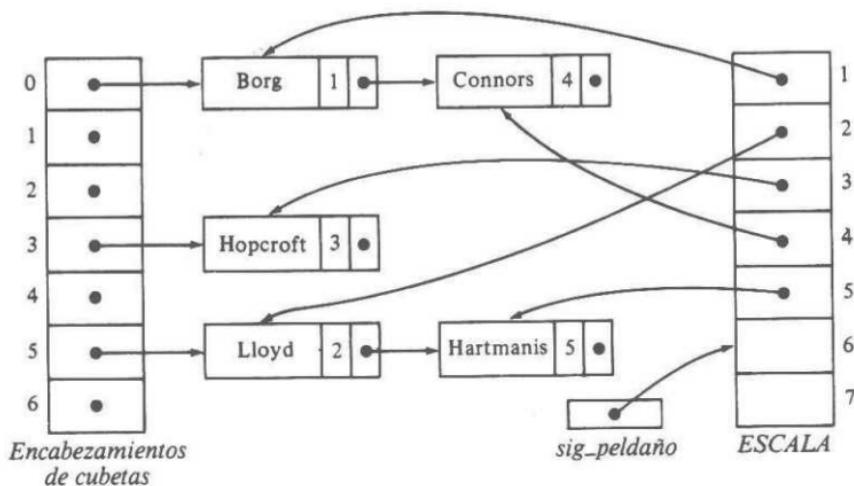


Fig. 4.27. Estructura combinada para alto rendimiento.

Ahora se puede agregar un nombre mediante una inserción en la tabla de dispersión en un tiempo $O(1)$ en promedio, colocando también un apuntador a la celda recién creada dentro del arreglo *ESCALA* en la posición marcada por el cursor *sig-peldaño* de la figura 4.27. Para los retos, se busca el nombre en la tabla de dispersión en un tiempo $O(1)$ en promedio, obteniéndose el peldaño i correspondiente al jugador dado y luego se sigue el apuntador en *ESCALA*[$i - 1$] a la celda del jugador que se va a retar. La consulta de *ESCALA*[$i - 1$] lleva un tiempo constante en el peor caso y la búsqueda en la tabla de dispersión demanda un tiempo $O(1)$ en promedio, de modo que RETA es $O(1)$ en el caso promedio.

A CAMBIA(i) le lleva un tiempo $O(1)$ hallar las celdas de los jugadores en los peldaños i e $i - 1$, intercambiar los números de peldaño de esas celdas, e intercambiar en *ESCALA* los apuntadores a las dos celdas. Así, CAMBIA requiere un tiempo constante incluso en el peor caso.

Ejercicios

4.1 Si $A = \{1, 2, 3\}$ y $B = \{3, 4, 5\}$, ¿cuáles son los resultados de

- a) UNION(A, B, C)
- b) INTERSECCION(A, B, C)
- c) DIFERENCIA(A, B, C)
- d) MIEMBRO(1, A)
- e) INSERTA(1, A)
- f) SUPRIME(1, A)
- g) MIN(A)?

- *4.2 Escribase un procedimiento en función de las operaciones básicas con conjuntos que imprima todos los elementos de un conjunto (finito). Puede suponerse que se dispone de un procedimiento para imprimir un objeto del tipo de los elementos. El conjunto a imprimir no debe quedar destruido. ¿Qué estructuras de datos serían las más apropiadas para implantar conjuntos en este caso?
- 4.3 La realización de conjuntos mediante vectores de bits se puede usar siempre que el «conjunto universal» se pueda traducir a los enteros de 1 a N . Describase cómo se haría esa traducción si el conjunto universal fuera
- los enteros 0, 1, ..., 99
 - los enteros de n a m para cualquier $n \leq m$
 - los enteros $n, n+2, n+4, \dots, n+2k$, para cualesquiera n y k
 - los caracteres 'a', 'b', ..., 'z'
 - arreglos de dos caracteres, cada uno de ellos elegidos entre 'a' y 'z'.
- 4.4 Escribanse procedimientos ANULA, UNION, INTERSECCION, MIEMBRO, MIN, INSERTA y SUPRIME para conjuntos representados mediante listas enlazadas, por medio de las operaciones abstractas del TDA lista clasificada. Obsérvese que la figura 4.5 es un procedimiento para INTERSECCION que maneja una realización específica del TDA lista.
- 4.5 Repítase el ejercicio 4.4 para las realizaciones de conjuntos siguientes:
- tabla de dispersión abierta (úsense operaciones abstractas con listas dentro de las cubetas).
 - tabla de dispersión cerrada con resolución lineal de colisiones.
 - lista no clasificada (empléense operaciones abstractas con listas).
 - un arreglo de longitud fija con un apuntador a la última posición usada.
- 4.6 Para cada una de las operaciones y aplicaciones de los ejercicios 4.4 y 4.5, proporcionese el orden de magnitud del tiempo de ejecución con conjuntos de tamaño n .
- 4.7 Supóngase que se están dispersando enteros en una tabla de dispersión de siete cubetas sirviéndose de la función de dispersión $h(i) = i \bmod 7$.
- Muéstrese la tabla de dispersión abierta si se insertan los cubos perfectos 1, 8, 27, 125, 216, 343.
 - Repítase el apartado a) usando una tabla de dispersión cerrada con resolución lineal de colisiones.
- 4.8 Supóngase que se está usando una tabla de dispersión cerrada con cinco cubetas y la función de dispersión $h(i) = i \bmod 5$. Muéstrese la tabla de dispersión cerrada con resolución lineal de colisiones que resulta de insertar la sucesión 23, 48, 35, 4, 10, en una tabla inicialmente vacía.
- 4.9 Obténganse las operaciones del TDA correspondencia, con tablas de dispersión abiertas y cerradas.

- 4.10** Para mejorar la velocidad de las operaciones podría desearse reemplazar una tabla de dispersión abierta con B_1 cubetas con más de B_1 elementos por otra tabla de dispersión con B_2 cubetas. Escribase un procedimiento para construir la nueva tabla a partir de la anterior, usando las operaciones del TDA lista para procesar cada cubeta.
- 4.11** En la sección 4.8 se habló de las funciones de dispersión «aleatorias» para las cuales $h_i(X)$, la cubeta en la que se va a probar después de i colisiones, es $(h(x) + d_i) \bmod B$ para cierta sucesión d_1, d_2, \dots, d_{B-1} . También se sugirió que una manera de calcular una sucesión semejante apropiada era elegir una constante k , y un $d_1 > 0$ arbitrario, y hacer

$$d_i = \begin{cases} 2d_{i-1} & \text{si } 2d_{i-1} < B \\ (2d_{i-1} - B) \oplus k & \text{si } 2d_{i-1} \geq B \end{cases}$$

donde $i > 1$, B es una potencia de 2, y \oplus representa la suma módulo 2 bit a bit. Si $B = 16$, encuéntrense los valores de k para los cuales la sucesión d_1, d_2, \dots, d_{15} incluye todos los enteros entre 1 y 15.

- 4.12** a) Muéstrese el árbol parcialmente ordenado que resulta cuando los enteros 5, 6, 4, 9, 3, 1, 7 se insertan en un árbol vacío.
 b) ¿Cuál es el resultado de tres operaciones SUPRIME-MIN sucesivas en el árbol de a)?
- 4.13** Supóngase que se representa el conjunto de cursos mediante
 a) una lista enlazada
 b) una tabla de dispersión
 c) un árbol binario de búsqueda.

Modifíquense las declaraciones de la figura 4.26 para cada una de estas estructuras.

- 4.14** Modifíquese la estructura de datos de la figura 4.26 de modo que cada registro de inscripción tenga un apuntador directo al estudiante y curso propietarios. Escribase otra vez el procedimiento *imprime_estudiantes* de la figura 4.26, aprovechando esta estructura.
- 4.15** Supóngase que hay 20 000 estudiantes, 1000 cursos y cada estudiante en un promedio de tres cursos; compárese la estructura de datos de la figura 4.26 con la modificación sugerida en el ejercicio 4.14 en lo referente a
 a) la cantidad de espacio requerida
 b) el tiempo promedio de ejecución de *imprime_estudiantes*
 c) el tiempo promedio de ejecución del procedimiento análogo que imprima los cursos que toma un estudiante dado.
- 4.16** Considérese la estructura de datos de la figura 4.26, dado un registro de curso c y un registro de estudiante e y escribanse procedimientos para insertar y suprimir el hecho de que e toma c .

- 4.17 Si existe, ¿cuál es la diferencia entre la estructura de datos del ejercicio 4.14 y la estructura en la que los conjuntos C_e y E_e se representan mediante listas de apuntadores a los registros de cursos y estudiantes, respectivamente?
- 4.18 Los empleados de cierta compañía se representan en la base de datos de la compañía por su nombre (que se supone único), número de empleado y número de seguridad social. Sugírase una estructura de datos que permita, dada una representación de un empleado, encontrar las otras dos representaciones del mismo individuo. ¿Qué rápida, en promedio, puede lograrse que sea cada una de esas operaciones?

Notas bibliográficas

Knuth [1973] es una buena fuente de información adicional sobre dispersión. La dispersión se desarrolló en la segunda mitad de la década de 1950, y Peterson [1957] es de los primeros artículos fundamentales sobre el tema. Morris [1968] y Maurer y Lewis [1975] dan buena información sobre la dispersión.

La lista múltiple es la estructura de datos central de los sistemas de bases de datos basados en redes propuestos en DBTG [1971]. Ullman [1982] proporciona información adicional sobre las aplicaciones, en bases de datos, de las estructuras de ese tipo.

La aplicación mediante montículos de los árboles parcialmente ordenados se basa en una idea de Williams [1964]. Las colas de prioridad se estudian más a fondo en Knuth [1973].

Reingold [1972] analiza la complejidad computacional de las operaciones básicas con conjuntos. Las técnicas de análisis de flujo de datos basadas en conjuntos se tratan con detalle en Cocke y Allen [1976] y en Aho y Ullman [1977].

5 Métodos avanzados de representación de conjuntos

Este capítulo presenta estructuras de datos para conjuntos que permiten obtener colecciones comunes de operaciones sobre conjuntos más eficientes que las presentadas en el capítulo anterior. Sin embargo, estas estructuras son más complejas y con frecuencia sólo son apropiadas para conjuntos muy grandes. Todas están basadas en varias clases de árboles, como árboles binarios de búsqueda, *tries* (árboles de recuperación de información) y árboles balanceados.

5.1 Árboles binarios de búsqueda

Se comenzará con los árboles binarios de búsqueda, una estructura de datos básica para la representación de conjuntos cuyos elementos están clasificados de acuerdo con algún orden lineal. Como de costumbre, se designará ese orden por medio del signo $<$. Esta estructura de datos es útil cuando se tiene un conjunto de elementos de un universo tan grande, que no es práctico emplear los propios elementos del conjunto como índices de arreglos. Un ejemplo de tal universo puede ser el conjunto de identificadores en un programa en Pascal. Un árbol binario de búsqueda puede manejar las operaciones de conjuntos INSERTA, SUPRIME, MIEMBRO y MIN, tomando en promedio $O(\log n)$ pasos por operación para un conjunto de n elementos.

Un *árbol binario de búsqueda* es un árbol binario en el cual los nodos están etiquetados con elementos de un conjunto. La propiedad importante de este tipo de árboles es que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo x son menores que el elemento almacenado en x , y todos los elementos almacenados en el subárbol derecho de x son mayores que el elemento almacenado en ese sitio. Esta condición, conocida como *propiedad del árbol binario de búsqueda*, se cumple para todo nodo de un árbol binario de búsqueda, incluyendo la raíz.

La figura 5.1 muestra dos árboles binarios de búsqueda que representan el mismo conjunto de enteros. Obsérvese la interesante propiedad de que si se listan los nodos del árbol en orden simétrico, los elementos almacenados en dichos nodos quedan clasificados.

Supóngase que se usa un árbol binario de búsqueda para representar un conjunto. La propiedad del árbol binario de búsqueda hace que sea simple la prueba de pertenencia al conjunto. Para determinar si x es un miembro del conjunto, primero se compara x con el elemento r que se encuentre en la raíz. Si $x = r$, no hay problema, la respuesta a la pregunta de pertenencia es «cierto». Si $x < r$, entonces x , si existe,

sólo puede ser un descendiente del hijo izquierdo de la raíz, a causa de la propiedad del árbol binario de búsqueda \dagger . De igual modo, si $x > r$, x sólo puede ser un descendiente del hijo derecho de la raíz.

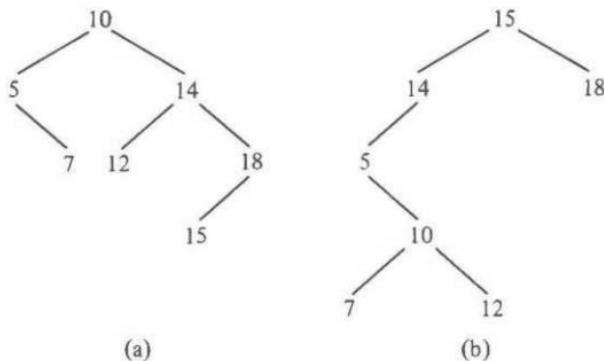


Fig. 5.1. Dos árboles binarios de búsqueda.

Se escribirá una función recursiva simple $MIEMBRO(x, A)$ para realizar esta prueba de pertenencia. Se supondrá que los elementos del conjunto son de un tipo no especificado que se denominará tipo_elemento. Por conveniencia, se supone que tipo_elemento es un tipo para el cual están definidos $<$ e $=$. Si no es así, se deben definir las funciones $MQ(a, b)$ e $IG(a, b)$, donde a y b son del tipo tipo_elemento, tal que $MQ(a, b)$ es cierto si, y sólo si, a es «menor que» b , e $IG(a, b)$ es cierto si, y sólo si, a y b son iguales.

El tipo de los nodos consta de un elemento y dos apuntadores a otros nodos:

```

type
  tipo_nodo = record
    elemento: tipo_elemento;
    hijo_izq, hijo_der: ^ tipo_nodo
  end;

```

Entonces es posible definir el tipo CONJUNTO como un apuntador a un nodo, que aquí será la raíz del árbol binario de búsqueda que representa el conjunto. Esto es:

type CONJUNTO = † tipo-nodo;

Ahora se puede especificar en su totalidad la función MIEMBRO de la figura 5.2. Obsérvese que debido a que CONJUNTO y «apuntador a tipo_nodo» son sinóni-.

[†] Recuérdese que el hijo izquierdo de la raíz es un descendiente de sí mismo, así que no debe eliminarse la posibilidad de que x sea hijo izquierdo de la raíz.

mos, MIEMBRO puede llamarse a sí mismo en subárboles, como si éstos representaran conjuntos. De hecho, el conjunto puede dividirse en el subconjunto de los miembros menores que x y en el subconjunto de los miembros mayores que x .

```
function MIEMBRO (  $x$ : tipo_elemento;  $A$ : CONJUNTO ) : boolean;
{ devuelve verdadero si  $x$  está en  $A$ , y falso en caso contrario }
begin
    if  $A = \text{nil}$  then
        return (false) {  $x$  nunca está en  $\emptyset$  }
    else if  $x = A \uparrow.\text{elemento}$  then
        return (true)
    else if  $x < A \uparrow.\text{elemento}$  then
        return (MIEMBRO( $x$ ,  $A \uparrow.\text{hijo\_izq}$ ))
    else {  $x > A \uparrow.\text{elemento}$  }
        return (MIEMBRO( $x$ ,  $A \uparrow.\text{hijo\_der}$ ))
end; { MIEMBRO }
```

Fig. 5.2. Prueba de pertenencia en un árbol binario de búsqueda.

El procedimiento INSERTA(x, A), que agrega un elemento x al conjunto A , también es fácil de escribir. La primera acción que INSERTA debe efectuar es probar si $A = \text{nil}$, esto es, si el conjunto está vacío. De ser así, se crea un nodo nuevo para colocar x y hacer que A le apunte. Si el conjunto no está vacío, se busca x más o menos como lo hace MIEMBRO, pero al encontrar un apuntador nil durante la búsqueda, se reemplaza por un apuntador a un nodo nuevo que contenga x . Entonces x estará en el lugar correcto, esto es, donde la función MIEMBRO lo encuentre. El código de INSERTA se muestra en la figura 5.3.

```
procedure INSERTA (  $x$ : tipo_elemento; var  $A$ : CONJUNTO );
{ agrega  $x$  al conjunto  $A$  }
begin
    if  $A = \text{nil}$  then begin
        new( $A$ );
         $A \uparrow.\text{elemento} := x$ 
         $A \uparrow.\text{hijo\_izq} := \text{nil}$ ;
         $A \uparrow.\text{hijo\_der} := \text{nil}$ 
    end
    else if  $x < A \uparrow.\text{elemento}$  then
        INSERTA( $x$ ,  $A \uparrow.\text{hijo\_izq}$ )
    else if  $x > A \uparrow.\text{elemento}$  then
        INSERTA( $x$ ,  $A \uparrow.\text{hijo\_der}$ )
    { if  $x = A \uparrow.\text{elemento}$ , no se hace nada;  $x$  ya está en el conjunto }
end; { INSERTA }
```

Fig. 5.3. Inserción de un elemento en un árbol binario de búsqueda.

La eliminación presenta algunos problemas. Primero, se debe localizar el elemento x que se elimine del árbol. Si x está en una hoja, tal vez baste eliminar esa hoja. Sin embargo, x puede estar en un nodo interior $nodo_i$, y eliminar $nodo_i$ podría desconectar el árbol.

Si $nodo_i$ tiene sólo un hijo, como el nodo 14 de la figura 5.1(b), es posible sustituir $nodo_i$ por ese hijo, y el árbol binario de búsqueda quedará bien construido. Si $nodo_i$ tiene dos hijos, como el nodo 10 de la figura 5.1(a), es necesario encontrar el menor elemento de los descendientes del hijo derecho \dagger . Por ejemplo, en el caso de que el elemento 10 sea borrado de la figura 5.1(a), se debe reemplazar por 12, el descendiente del hijo derecho de 10 con valor más bajo.

Para escribir SUPRIME, es útil tener una función SUPRIME_MIN(A) que elimine el elemento más pequeño de un árbol no vacío y devuelva el valor del elemento eliminado. El código de SUPRIME_MIN se muestra en la figura 5.4. El código de SUPRIME usa SUPRIME_MIN y se muestra en la figura 5.5.

```
function SUPRIME_MIN ( var A: CONJUNTO ) : tipo_elemento;
  { devuelve y elimina el elemento más pequeño del conjunto A }
begin
  if A^.hijo_izq = nil then begin
    { A apunta al elemento más pequeño }
    SUPRIME_MIN := A^.elemento;
    A := A^.hijo_der;
    { reemplaza el nodo apuntado por A por su hijo derecho }
  end
  else { el nodo apuntado por A tiene un hijo izquierdo }
    SUPRIME_MIN := SUPRIME_MIN(A^.hijo_izq)
end; { SUPRIME_MIN }
```

Fig. 5.4. Eliminación del elemento más pequeño.

```
procedure SUPRIME ( x: tipo_elemento; var A: CONJUNTO );
  { elimina x del conjunto A }
begin
  if A <> nil then
    if x < A^.elemento then
      SUPRIME(x, A^.hijo_izq)
    else if x > A^.elemento then
      SUPRIME(x, A^.hijo_der)
    { si se llega aquí, x es el nodo apuntado por A }
    else if (A^.hijo_izq = nil) and (A^.hijo_der = nil) then
      A := nil { suprime la hoja que contiene a x }
    else if A^.hijo_izq = nil then
      A := A^.hijo_der
```

\dagger Puede ser también el nodo con valor más alto entre los descendientes del hijo izquierdo.

```

else if A↑.hijo_der = nil then
    A := A↑.hijo_izq
else { ambos hijos están presentes }
    A↑.elemento := SUPRIME_MIN(A↑.hijo_der)
end; { SUPRIME }

```

Fig. 5.5. Eliminación en un árbol binario de búsqueda.

Ejemplo 5.1. Supóngase que se intenta eliminar 10 de la figura 5.1(a). Entonces, en la última proposición de SUPRIME se llama a SUPRIME_MIN con un argumento apuntador al nodo 14. Ese apuntador es el campo *hijo_der* de la raíz. Esta llamada produce otra llamada a SUPRIME_MIN. El argumento es ahora un apuntador al nodo 12; este apuntador se encuentra en el campo *hijo_izq* del nodo 14. Se encuentra que 12 no tiene hijo izquierdo, así que se devuelve el elemento 12 y se hace que el hijo izquierdo de 14 sea el hijo derecho de 12, el cual resulta ser nil. Después, SUPRIME toma el valor 12 devuelto por SUPRIME_MIN, que reemplaza a 10. El árbol resultante se muestra en la figura 5.6. □

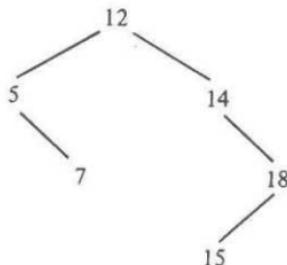


Fig. 5.6. Árbol de la figura 5.1(a) después de suprimir 10.

5.2 Análisis en tiempo de las operaciones para árboles binarios de búsqueda

En esta sección se analiza el comportamiento promedio de distintas operaciones para árboles binarios de búsqueda. Se demuestra que al insertar n elementos aleatorios en un árbol binario de búsqueda inicialmente vacío, la longitud de camino promedio de la raíz a una hoja es $O(\log n)$. La prueba de pertenencia, por tanto, lleva un tiempo $O(\log n)$.

Es fácil ver que si un árbol binario de n nodos está completo (todos los nodos, excepto los que están en el nivel más bajo, tienen dos hijos), ningún camino tendrá más de $1 + \log n$ nodos †. Así, los procedimientos MIEMBRO, INSERTA, SUPRIME y SUPRIME_MIN llevan un tiempo $O(\log n)$. Para comprenderlo, obsérvese que todos toman una cantidad de tiempo constante en un nodo, por lo que pueden llamarse a sí mismos en forma recursiva a lo sumo en un hijo. Por tanto, la secuencia

† Recuérdese que todos los logaritmos son de base 2, a menos que se indique lo contrario.

de nodos en la cual se realizan las llamadas forma un camino desde la raíz. Como el camino es de longitud $O(\log n)$, el tiempo total consumido para seguir el camino es $O(\log n)$.

Sin embargo, al insertar n elementos en un orden «aleatorio», no es forzoso que se acomoden en forma de árbol binario completo. Por ejemplo, si sucede que el primer elemento insertado en orden clasificado es el más pequeño, el árbol resultante será una cadena de n nodos, donde cada nodo, excepto el más bajo en el árbol, tendrá un hijo derecho, pero no un hijo izquierdo. En este caso, es fácil mostrar que como lleva $O(i)$ pasos insertar el i -ésimo elemento y $\sum_{i=1}^n i = n(n+1)/2$, dicho proceso de n inserciones necesita $O(n^2)$ pasos, u $O(n)$ pasos por operación.

Es preciso determinar si el árbol binario de búsqueda «promedio» con n nodos se acerca en estructura al árbol completo y no a la cadena, esto es, si el tiempo promedio por operación en un árbol «aleatorio» necesita $O(\log n)$ pasos, $O(n)$ pasos, o un tiempo intermedio. Como es difícil saber la verdadera frecuencia de inserciones y supresiones, o si los elementos eliminados poseen alguna propiedad especial (por ejemplo, si siempre se elimina el mínimo), sólo se puede analizar la longitud del camino promedio de árboles «aleatorios» adoptando algunas suposiciones: los árboles se forman sólo a partir de inserciones, y todas las magnitudes de los n elementos insertados tienen igual probabilidad.

Con esas suposiciones naturales, se puede calcular $P(n)$, el número promedio de nodos del camino que va de la raíz hacia algún nodo (no necesariamente una hoja). Se supone que el árbol se formó con la inserción de n nodos aleatorios en un árbol que se encontraba vacío en un inicio. Es evidente que $P(0) = 0$ y $P(1) = 1$. Supóngase que se tiene una lista de $n \geq 2$ elementos para insertar en un árbol vacío. El primer elemento en la lista, llamado a , es probable que sea el primero, el segundo o el n -ésimo en el orden de clasificación. Considérese que i elementos en la lista son menores que a , de modo que $n - i - 1$ son mayores que a . Al construir el árbol, a aparecerá en la raíz, los i elementos más pequeños serán descendientes izquierdos de la raíz, y los restantes $n - i - 1$ serán descendientes derechos. (Véase Fig. 5.7.)

Como todos los órdenes de los i elementos pequeños y de los $n - i - 1$ elementos más grandes tienen igual probabilidad, se espera que los subárboles izquierdo y derecho de la raíz tengan longitudes de camino promedio $P(i)$ y $P(n - i - 1)$, respectivamente. Como es posible acceder a esos elementos desde la raíz del árbol comple-

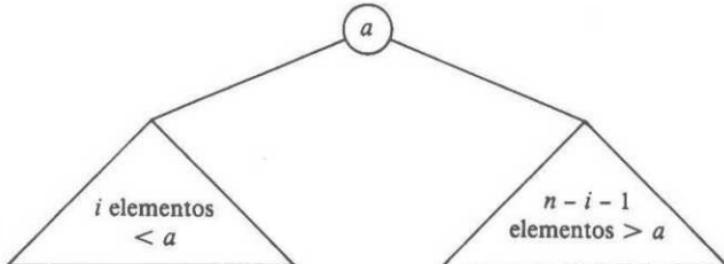


Fig. 5.7. Árbol binario de búsqueda.

to, es necesario agregar 1 al número de nodos de cada camino. Así, para todo i entre 0 y $n - 1$, $P(n)$ puede calcularse obteniendo el promedio de la suma

$$\frac{i}{n} (P(i) + 1) + \frac{(n-i-1)}{n} (P(n-i-1) + 1) + \frac{1}{n}$$

El primer término es la longitud de camino promedio en el subárbol izquierdo, ponderando su tamaño. El segundo término es la cantidad análoga del subárbol derecho y el término $1/n$ representa la contribución de la raíz. Al promediar la suma anterior para toda i entre 1 y n , se obtiene la recurrencia

$$P(n) = 1 + \frac{1}{n^2} \sum_{i=0}^{n-1} (iP(i) + (n-i-1)P(n-i-1)) \quad (5.1)$$

La primera parte de la sumatoria (5.1), $\sum_{i=0}^{n-1} iP(i)$, se puede hacer idéntica a la segunda parte $\sum_{i=0}^{n-1} (n-i-1)P(n-i-1)$ si se sustituye i por $n-i-1$ en la segunda parte. Además, el término para $i=0$ del sumatorio $\sum_{i=0}^{n-1} iP(i)$ es cero, por lo que es posible empezar el sumatorio en 1. Así, (5.1) puede escribirse

$$P(n) = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} iP(i) \quad \text{para } n \geq 2 \quad (5.2)$$

Se demuestra, por inducción sobre n , comenzando en $n=1$, que $P(n) \leq 1 + 4\log n$. Con seguridad esta proposición es verdadera para $n=1$, ya que $P(1)=1$. Supóngase que es verdadera para toda $i < n$. Entonces, por (5.2)

$$\begin{aligned} P(n) &\leq 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} (4ilogi + i) \\ &\leq 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} 4ilogi + \frac{2}{n^2} \sum_{i=1}^{n-1} i \\ &\leq 2 + \frac{8}{n^2} \sum_{i=1}^{n-1} ilogi \end{aligned} \quad (5.3)$$

El último paso está justificado, ya que $\sum_{i=1}^{n-1} i \leq n^2/2$ y, por tanto el último término de la segunda línea es a lo sumo 1. Al dividir los términos del sumatorio de (5.3) en dos partes, aquellos para los que $i \leq [n/2] - 1$, lo cual no excede de $i\log(n/2)$, y aquellos para los que $i > [n/2] - 1$, que no excede de $i\log n$. Así, (5.3) puede escribirse otra vez

$$P(n) \leq 2 + \frac{8}{n^2} \left[\sum_{i=1}^{[n/2]-1} ilog(n/2) + \sum_{i=[n/2]}^{n-1} ilogn \right] \quad (5.4)$$

Sea n par o impar, es posible demostrar que la primera suma de (5.4) no excede de $(n^2/8)\log(n/2)$, lo cual es $(n^2/8)\log n - (n^2/8)$, y la segunda suma no excede de $(3n^2/8)\log n$. Así, (5.4) se puede escribir

$$\begin{aligned} P(n) &\leq 2 + \frac{8}{n^2} \left[\frac{n^2}{2} \log n - \frac{n^2}{8} \right] \\ &\leq 1 + 4\log n \end{aligned}$$

como se deseaba probar. Este paso completa la inducción y demuestra que el tiempo promedio para seguir un camino de la raíz a un nodo aleatorio de un árbol binario de búsqueda construido mediante inserciones aleatorias es $O(\log n)$, lo cual es, en un factor constante, tan bueno como si el árbol fuera completo. Un análisis más cuidadoso demuestra que la constante 4 es en realidad cercana a 1.4.

Se concluye de lo anterior que el tiempo de la prueba de pertenencia de un miembro aleatorio del conjunto lleva un tiempo $O(\log n)$. Un análisis similar muestra que si se incluyen en la longitud de camino promedio sólo aquellos nodos que carecen de ambos hijos o sólo aquellos que no tienen hijo izquierdo, la longitud del camino promedio aún se ajustará a una ecuación similar a (5.1) y es, por tanto, $O(\log n)$. Es posible aseverar entonces que la prueba de pertenencia de un elemento aleatorio que no está en el conjunto, la inserción de un nuevo elemento aleatorio y la eliminación de un elemento aleatorio también llevan un tiempo $O(\log n)$ en promedio.

Evaluación del rendimiento de los árboles binarios de búsqueda

Las realizaciones de diccionarios por medio de tablas de dispersión requieren un tiempo constante por operación en promedio. Aunque este rendimiento es mejor que el de un árbol binario de búsqueda, una tabla de dispersión requiere $O(n)$ pasos para la operación MIN; así, si MIN se usa con frecuencia, el árbol binario de búsqueda será la mejor opción; si MIN no se usa, tal vez sería preferible la tabla de dispersión.

El árbol binario de búsqueda debe compararse también con el árbol parcialmente ordenado empleado para las colas de prioridad del capítulo 4. Un árbol parcialmente ordenado con n elementos requiere sólo $O(\log n)$ pasos para cada operación INSERTA y SUPRIME-MIN no sólo en el promedio, sino también en el peor caso. Más aún, la constante real de proporcionalidad que acompaña al factor $\log n$ será más pequeña para un árbol parcialmente ordenado que para un árbol binario de búsqueda. Sin embargo, este último permite las operaciones generales SUPRIME y MIN, así como la combinación SUPRIME-MIN, mientras que el árbol parcialmente ordenado sólo permite la última. Además, MIEMBRO requiere $O(n)$ pasos en un árbol parcialmente ordenado, pero sólo $O(\log n)$ pasos en un árbol binario de búsqueda. Así, mientras que el árbol parcialmente ordenado es adecuado para realizar colas de prioridad, no puede efectuar de forma tan eficiente ninguna de las operaciones adicionales que el árbol binario de búsqueda puede hacer.

5.3 Tries

En esta sección se presenta una estructura especial para representar conjuntos de cadenas de caracteres. El mismo método funciona para la representación de tipos de datos que son cadenas de objetos de cualquier tipo, como las cadenas de enteros. Esta estructura se conoce como *trie*, derivada de las letras centrales de la palabra *retrieval* (recuperación) †. A manera de introducción, considérese el siguiente uso de un conjunto de cadenas de caracteres.

Ejemplo 5.2. Como se indicó en el capítulo 1, una forma de implantar un revisor de ortografía es leer un archivo de texto, separarlo en palabras (cadenas de caracteres separados por espacios y caracteres de nueva línea) y encontrar las palabras que no estén en un diccionario estándar de palabras de uso común. Las palabras que estén en el texto, pero no en el diccionario, se imprimen como posibles faltas de ortografía. La figura 5.8 muestra el esbozo de un posible programa *orto*. Este utiliza un procedimiento *toma_palabra(x, t)* que asigna a *x* la siguiente palabra en el archivo de texto *t*; la variable *x* es del tipo llamado tipo_palabra, que se define más adelante. La variable *A* es de tipo CONJUNTO; las operaciones necesarias sobre CONJUNTO son INSERTA, SUPRIME, ANULA e IMPRIME. El operador IMPRIME imprime los miembros del conjunto. □

```

program orto ( input, output, diccionario );
type
  tipo_palabra = { a definir }
  CONJUNTO = { a definir mediante la estructura de trie };
var
  A: CONJUNTO; { retiene las palabras de entrada no encontradas
    en el diccionario }
  siguiente_palabra: tipo_palabra;
  diccionario: file of char;

procedure toma_palabra ( var x: tipo_palabra; f: file of char );
{ procedimiento a definir que hace que x
  sea la siguiente palabra en el archivo f }

procedure INSERTA ( x: tipo_palabra; var A: CONJUNTO );
{ a definir }

procedure SUPRIME ( x: tipo_palabra; var A: CONJUNTO );
{ a definir }

procedure ANULA ( var A: CONJUNTO );
{ a definir }

```

† *Trie* se pensó originalmente como un homónimo de *tree* (que en inglés se pronuncia «tri»), pero para distinguir estos términos, mucha gente prefiere pronunciarlo igual que *pie* (que en inglés se pronuncia «pay»).

```

procedure IMPRIME ( var A: CONJUNTO );
{ a definir }

begin
  ANULA(A);
  while not eof(input) do begin
    toma_palabra(siguiente_palabra, input);
    INSERTA(siguiente_palabra, A)
  end
  while not eof(diccionario) do begin
    toma_palabra(siguiente_palabra, diccionario);
    SUPRIME(siguiente_palabra, A)
  end;
  IMPRIME(A);
end; { orto }

```

Fig. 5.8. Esbozo de revisor de ortografía.

La estructura trie maneja esas operaciones cuando los elementos del conjunto son palabras, esto es, cadenas de caracteres. Es apropiada cuando muchas palabras comienzan con la misma secuencia de letras, es decir, cuando el número de prefijos distintos entre todas las palabras del conjunto es mucho menor que la longitud total de todas las palabras.

En un trie, cada camino de la raíz a una hoja corresponde a una palabra del conjunto representado. De esta forma, los nodos del trie corresponden a los prefijos de las palabras del conjunto. Para evitar confusión entre palabras como ELLO y ELLOS, se añade un símbolo especial *marca-fin*, \$, al final de todas las palabras, y así ningún prefijo de una palabra puede ser una palabra por sí mismo.

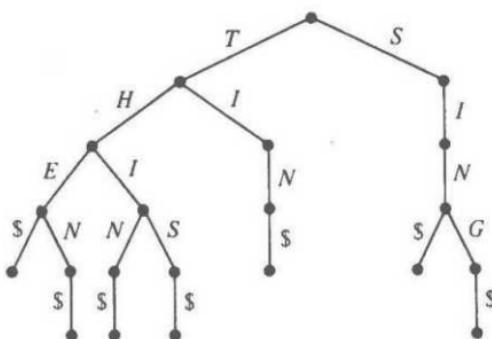


Fig. 5.9. Un trie.

Ejemplo 5.3. En la figura 5.9 hay un trie que representa el conjunto de las palabras {THE, THEN, THIN, THIS, TIN, SIN, SING}. Esto es, la raíz corresponde a la ca-

dena vacía y sus dos hijos corresponden a los prefijos T y S. La hoja que está más a la izquierda representa la palabra THE, la siguiente, la palabra THEN, y así sucesivamente. □

Considérense las siguientes observaciones sobre la figura 5.9.

1. Cada nodo tiene hasta 27 hijos, uno para cada letra y \$.
2. La mayor parte de los nodos tiene mucho menos de 27 hijos.
3. Una hoja que sigue a una arista etiquetada con \$ no puede tener hijos e incluso podría no existir.

Nodos de un trie como TDA

Un nodo de trie puede considerarse como una correspondencia cuyo dominio es {A, B, ..., Z, \$} (o cualquier alfabeto que se escoga) y cuyo conjunto de valores es de tipo «apuntador a nodo de trie». Más aún, el trie mismo puede identificarse con su propia raíz, por lo que los TDA TRIE y NODO-TRIE tienen el mismo tipo de datos, aunque sus operaciones son sustancialmente diferentes. En un NODO-TRIE, se necesitan las operaciones siguientes:

1. procedimiento ASIGNA(*nodo, c, p*) que asigna el valor *p* (un apuntador a un nodo) al carácter *c* de *nodo*.
2. función VALOR-DE(*nodo, c*) que produce el valor asociado con el carácter *c* de *nodo* †, y
3. procedimiento TOMA-NUEVO(*nodo, c*) para hacer que el valor de *nodo* para el carácter *c* apunte a un nodo nuevo.

Técnicamente, también se requiere un procedimiento ANULA(*nodo*) para hacer que *nodo* sea la correspondencia nula. Una realización simple de los nodos de un trie es mediante un arreglo *nodo* de apuntadores a nodos, siendo el conjunto de índices {A, B, ..., Z, \$}. Esto es, se define

```
type
  cars = ('A', 'B', ..., 'Z', '$');
  NODO-TRIE = array [cars] of ^NODO-TRIE;
```

Si *nodo* es un nodo de trie, *nodo[c]* es VALOR-DE(*nodo, c*) para cualquier *c* del conjunto de caracteres. Para evitar la creación de muchas hojas que sean hijos correspondientes a '\$', se adoptará la convención de que *nodo['\$']* es nil o un apuntador al propio nodo. En el primer caso, *nodo* no tiene hijos que correspondan a '\$', y en el segundo caso, se determina que tiene tal hijo, aunque nunca se haya creado. Entonces, se pueden escribir los procedimientos para nodos de trie como en la figura 5.10.

† VALOR-DE es una versión de la función CALCULA de la sección 2.5.

```

procedure ANULA ( var nodo: NODO_TRIE );
{ hace de nodo una hoja, es decir, una correspondencia nula }
var
  c: char;
begin
  for c := 'A' to '$' do
    nodo[c] := nil
end; { ANULA }

procedure ASIGNA ( var nodo: NODO_TRIE; c: char; p: ↑NODO_TRIE );
begin
  nodo[c] := p
end; { ASIGNA }

function VALOR_DE ( var nodo: NODO_TRIE; c: char ) : ↑ NODO_TRIE;
begin
  return (nodo[c])
end; { VALOR_DE }

procedure TOMA_NUEVO ( var nodo: NODO_TRIE; c: char );
begin
  new(nodo[c]);
  ANULA(nodo[c])
end; { TOMA_NUEVO }

```

Fig. 5.10. Operaciones en nodos de un trie.

Ahora, se define

```

type
  TRIE = ↑ NODO_TRIE;

```

Se supondrá que tipo_palabra es un arreglo de caracteres de cierta longitud fija. Siempre se partirá del supuesto de que el valor de tal arreglo contiene al menos '\$'; se considerará como fin de la palabra representada el primer '\$', sin importar lo que siga (quizá más '\$'). Con esta suposición, se escribe el procedimiento INSERTA(*x*, *palabras*) para insertar *x* en el conjunto *palabras* representado por un trie, como se muestra en la figura 5.11. Se deja como ejercicio la escritura de ANULA, SUPRIME e IMPRIME para tries representados como arreglos.

```

procedure INSERTA ( x: tipo_palabra; var palabras: TRIE );
var
  i: integer; { cuenta las posiciones en la palabra x }
  t: TRIE; { empleado para apuntar a nodos del trie que corresponden
            a los prefijos de x }

```

```

begin
    i := 1;
    t := palabras;
    while x[i] <> '$' do begin
        if VALOR_DE(t↑, x[i]) = nil then
            { si el nodo actual no tiene hijo para el carácter x[i], crea uno }
            TOMA_NUEVO(t↑, x[i]);
        t := VALOR_DE(t↑, x[i]);
        { prosigue al hijo de t para el carácter x[i], sin importar si
          ese hijo fue creado o no }
        i := i+1 { se mueve en la palabra x }
    end;
    { ahora se ha alcanzado el primer '$' en x }
    ASIGNA(t↑, '$', t)
    { hace un ciclo para '$' para representar una hoja }
end; { INSERTA }

```

Fig. 5.11. Procedimiento INSERTA.

Representación de nodos de un trie por medio de listas

La representación de nodos de un trie mediante arreglos toma una colección de palabras, teniendo en ellas p diferentes prefijos, y las representa con $27p$ bytes de almacenamiento. Esta cantidad de espacio puede exceder con facilidad la longitud total de las palabras del conjunto. Sin embargo, existe otra realización de tries que puede ahorrar espacio. Recuérdese que cada nodo del trie es una correspondencia, como se expuso en la sección 2.6. En principio, cualquier implantación de correspondencias puede funcionar, pero en la práctica se desea una representación adecuada para correspondencias con un dominio pequeño y para las definidas por relativamente pocos miembros del dominio. La representación con listas enlazadas satisface en buena medida esos requisitos. Se puede representar una correspondencia, que es un nodo de un trie, por medio de una lista enlazada de caracteres para la cual el valor asociado no es el apuntador `nil`. O sea, un nodo de un trie es una lista enlazada de celdas de tipo

```

type
    tipo_celda = record
        dominio: char;
        valor: † tipo_celda;
        {apuntador a la primera celda de la lista para el nodo hijo}
        siguiente: † tipo_celda;
        {apuntador a la siguiente celda de la lista}
    end;

```

Se dejan como ejercicios los procedimientos ASIGNA, VALOR_DE, ANULA y TOMA_NUEVO para esta realización de nodos de trie. Después de escribir esos pro-

cedimientos, las operaciones sobre tries como INSERTA, de la figura 5.11, y otras que quedaron como ejercicios, deben funcionar correctamente.

Evaluación de la estructura de datos trie

Compárense el tiempo y el espacio necesarios para representar n palabras con un total de p prefijos diferentes y una longitud total l usando una tabla de dispersión y un trie. En lo que sigue, se supondrá que los apuntadores requieren cuatro bytes. Quizás el medio más eficaz en cuanto al espacio para almacenar palabras y al manejo de las operaciones INSERTA y SUPRIME sea una tabla de dispersión. Si las palabras son de longitud variable, las celdas de las cubetas no deben contener las palabras mismas, sino que deben constar de dos apuntadores, uno para enlazar entre sí las celdas de la cubeta y otro para apuntar al principio de la palabra que pertenece a la cubeta.

Las palabras mismas se almacenan en un gran arreglo de caracteres, y el fin de cada palabra se indica por medio de un carácter de fin como '\$'. Por ejemplo, las palabras THE, THEN, y THIN pueden almacenarse como

THE\$THEN\$THIN\$...

Los apuntadores de las tres palabras son cursores a las posiciones 1, 5 y 10 del arreglo. La cantidad de espacio utilizado en las cubetas y el arreglo de caracteres es

1. $8n$ bytes para las celdas de las cubetas, siendo una celda para cada una de las n palabras; una celda tiene dos apuntadores u 8 bytes.
2. $l + n$ bytes para que el arreglo de caracteres almacene las n palabras de longitud total l y sus marcas de final.

Así, el espacio total es $9n + l$ bytes más la cantidad empleada para los encabezamientos de las cubetas.

En comparación, un trie con nodos aplicados por medio de listas enlazadas requiere $p + n$ celdas, una celda para cada prefijo y otra para el fin de cada palabra. Cada celda del trie tiene un carácter y dos apuntadores, y necesita nueve bytes, para un espacio total de $9n + 9p$. Si l más el espacio para los encabezados de las cubetas excede de $9p$, el trie usa menos espacio. Sin embargo, para aplicaciones como el almacenamiento de un diccionario en donde l/p es menor que 3, la tabla de dispersión puede ocupar menor espacio.

A favor del trie, sin embargo, obsérvese que se puede recorrer y realizar operaciones tales como INSERTA, SUPRIME y MIEMBRO en un tiempo proporcional a la longitud de la palabra en cuestión. Una función de dispersión que sea realmente «aleatoria» debe comprender cada carácter de la palabra que se esté dispersando. Por tanto, es justo establecer que el cálculo de la función de dispersión lleva tanto tiempo como realizar una operación como MIEMBRO sobre el trie. Por supuesto, el tiempo utilizado en el cálculo de la función de dispersión no incluye el tiempo empleado en resolver las colisiones o la inserción, la eliminación, o la prueba de pertenencia en la tabla de dispersión, así que se puede esperar que los tries sean bas-

tante más rápidos que las tablas de dispersión para diccionarios cuyos elementos son cadenas de caracteres.

Otra ventaja del trie es que permite la realización eficiente de la operación MIN, mientras que las tablas de dispersión, no. Más aún, en la organización con las tablas de dispersión ya descrita, no es posible volver a usar fácilmente el espacio del arreglo de caracteres cuando se borra una palabra (véase en el Cap. 12 los métodos para resolver este problema).

5.4 Realización de conjuntos con árboles balanceados

En las secciones 5.1 y 5.2 se vio cómo realizar conjuntos mediante árboles binarios de búsqueda, y que las operaciones como INSERTA pueden ejecutarse en un tiempo proporcional a la profundidad promedio de los nodos del árbol. Más aún, se hizo patente que esta profundidad promedio es $O(\log n)$ para un árbol «aleatorio» de n nodos. Sin embargo, algunas secuencias de inserciones y eliminaciones pueden producir árboles binarios de búsqueda cuya profundidad promedio sea proporcional a n . Esto sugiere que se puede hacer el intento de reordenar el árbol después de cada inserción y eliminación para que siempre esté completo; entonces el tiempo para las operaciones como INSERTA y similares puede ser siempre $O(\log n)$.

En la figura 5.12(a) se muestra un árbol de seis nodos que se convierte en el árbol completo de 7 nodos mostrado en la figura 5.12(b) cuando se inserta el elemento 1. Todos los elementos de la figura 5.12(a), sin embargo, tienen un parente diferente en la figura 5.12(b), así que deben tomarse n pasos para insertar el 1 en un árbol como el de la figura 5.12(a), si se desea conservar el árbol lo más balanceado posible. Así, es improbable que la sola insistencia en que el árbol binario de búsqueda sea completo lleve a la implantación de un diccionario, cola de prioridad u otro TDA que incluya INSERTA entre sus operaciones, en un tiempo $O(\log n)$.

Existen otros enfoques que dan en el peor caso un tiempo $O(\log n)$ por operación para diccionarios y colas de prioridad, como el llamado «árbol 2-3». Un árbol 2-3 tiene las siguientes propiedades.

1. Cada nodo interior tiene dos o tres hijos.
2. Todos los caminos que van de la raíz a una hoja tienen idéntica longitud.

También se considerará un árbol con uno o con cero nodos, como casos especiales de un árbol 2-3.

Los conjuntos de elementos que están clasificados de acuerdo con algún orden lineal $<$, se representan como sigue. Los elementos están colocados en las hojas; si el elemento a está a la izquierda del elemento b , entonces debe cumplirse que $a < b$. Se supondrá que el ordenamiento « $<$ » de elementos está basado en un campo de un registro que forma el tipo del elemento; este campo se conoce como *clave*. Por ejemplo, los elementos pueden representar personas y cierta información acerca de ellas; en ese caso, el campo clave puede ser el «número de seguridad social».

En cada nodo interior se coloca la clave del elemento más pequeño que sea descendiente del segundo hijo, y si existe un tercer hijo, se coloca también la clave del

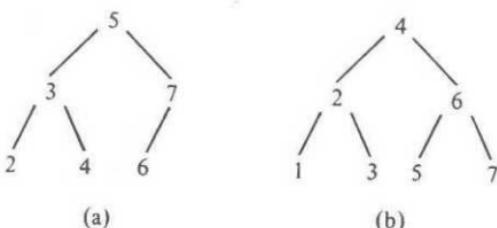


Fig. 5.12. Árboles completos.

elemento más pequeño que descienda de ese hijo \dagger . La figura 5.13 es un ejemplo de árbol 2-3. En ese ejemplo y los siguientes, se identificará un elemento con su campo clave, para que el orden de los elementos sea evidente.

Obsérvese que un árbol 2-3 de k niveles tiene entre 2^{k-1} y 3^{k-1} hojas. Dicho de otra manera, un árbol 2-3 que represente un conjunto de n elementos requiere al menos $1 + \log_3 n$ y no más de $1 + \log_2 n$ niveles. Así, las longitudes de los caminos en el árbol son $O(\log n)$.

Es posible probar la pertenencia de un registro con clave x en un conjunto representado por un árbol 2-3 en un tiempo $O(\log n)$, simplemente descendiendo en el árbol, usando los valores de los elementos registrados en los nodos internos para guiar el camino. En un nodo *nodo*, se compara x con el valor y que representa al menor elemento descendiente del segundo hijo de *nodo*. (Recuérdese que los elementos se están tratando como si consistiesen sólo en un campo clave.) Si $x < y$, hay que ir al primer hijo de *nodo*. Si $x \geq y$ y *nodo* tiene sólo dos hijos, será preciso ir al segundo hijo de *nodo*. Si *nodo* tiene tres hijos y $x \geq y$, debe compararse x con z , el segundo valor registrado en el nodo, el valor que indica el descendiente más pequeño del tercer hijo de *nodo*. Si $x < z$, habrá que ir al segundo hijo, y si $x \geq z$, al tercero. De esta forma, se llega finalmente a una hoja, y x está en el conjunto representado si, y sólo si, x está en la hoja. Es evidente que si durante este proceso $x = y$ o $x = z$,

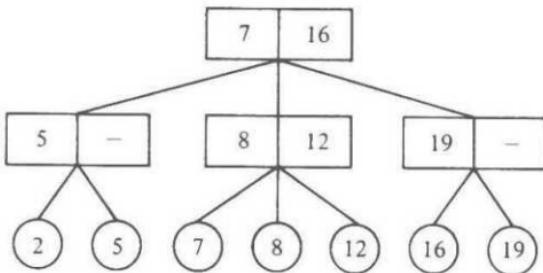


Fig. 5.13. Un árbol 2-3.

\dagger Existe otra versión de árboles 2-3 que coloca registros completos en nodos internos, tal como se hace en los árboles binarios de búsqueda.

se puede parar de inmediato. Sin embargo, el algoritmo quedó así porque en algunos casos es deseable encontrar la hoja con x , además de verificar su existencia.

Inserción en un árbol 2-3

Para insertar un nuevo elemento x en un árbol 2-3, se procede al principio como si se probara la pertenencia de x al conjunto. Sin embargo, justo en el nivel superior al de las hojas, se estará en un nodo *nodo* cuyos hijos no incluyen x . Si *nodo* tiene sólo dos hijos, se hace que x sea el tercero, colocándolo en el orden adecuado. Después se ajustan los dos números de *nodo* para reflejar la nueva situación.

Por ejemplo, al insertar el 18 en la figura 5.13, se termina con *nodo* igual al nodo del extremo derecho en el nivel medio. Se coloca el 18 entre los hijos de *nodo*, cuyo orden correcto es 16, 18, 19. Los dos valores registrados en *nodo* se convierten en 18 y 19, los elementos del segundo y tercer hijos. El resultado se muestra en la figura 5.14.

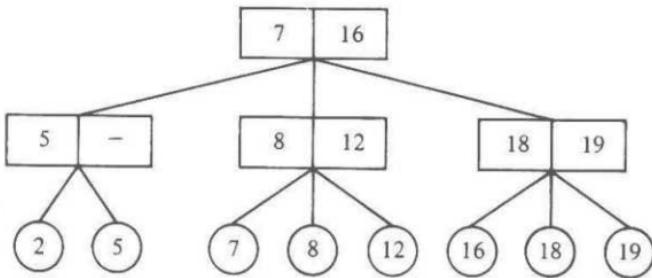


Fig. 5.14. Árbol 2-3 con el 18 insertado.

Sin embargo, supóngase que x es el cuarto hijo de *nodo*, y no el tercero, no es posible tener un nodo con cuatro hijos en un árbol 2-3, por lo que es necesario partir *nodo* en dos: *nodo* y *nodo'*. Los dos elementos más pequeños entre los cuatro hijos de *nodo* permanecen ahí, mientras que los dos más grandes pasan a ser hijos de *nodo'*. Ahora, se debe insertar *nodo'* entre los hijos de *p*, el padre de *nodo*. Esta parte de la inserción es análoga a la inserción de una hoja como hija de *nodo*. Esto es, si *p* tiene dos hijos, se hace que *nodo'* sea el tercero y se coloca inmediatamente a la derecha de *nodo*. Si *p* tiene tres hijos antes de crear *nodo'*, *p* se divide en *p* y *p'*, dejando *p* para los dos hijos de la izquierda y *p'* para los dos restantes, y después se inserta *p'* entre los hijos del padre de *p*, en forma recursiva.

Un caso especial ocurre cuando se llega a dividir la raíz. En este caso se crea una raíz nueva, cuyos hijos son los dos nodos en los cuales se dividió la raíz primaria. De esta manera es como se incrementa el número de niveles en un árbol 2-3.

Ejemplo 5.4. Supóngase que se inserta 10 en el árbol de la figura 5.14. El supuesto padre de 10 ya tiene los hijos 7, 8 y 12, así que se divide en dos nodos. El primero

tiene como hijos a 7 y 8, y el segundo, a 10 y 12. El resultado se muestra en la figura 5.15(a). Ahora es necesario insertar en el lugar apropiado un nodo nuevo cuyos hijos sean 10 y 12, como hijo de la raíz de la figura 5.15(a). Al hacerlo, resulta que la raíz tiene cuatro hijos, por lo que se divide, y se crea una raíz nueva, como se muestra en la figura 5.15(b). Los detalles de cómo se lleva hacia arriba la información relacionada con los elementos más pequeños de los subárboles, se darán al desarrollar el programa del mandato INSERTA. □

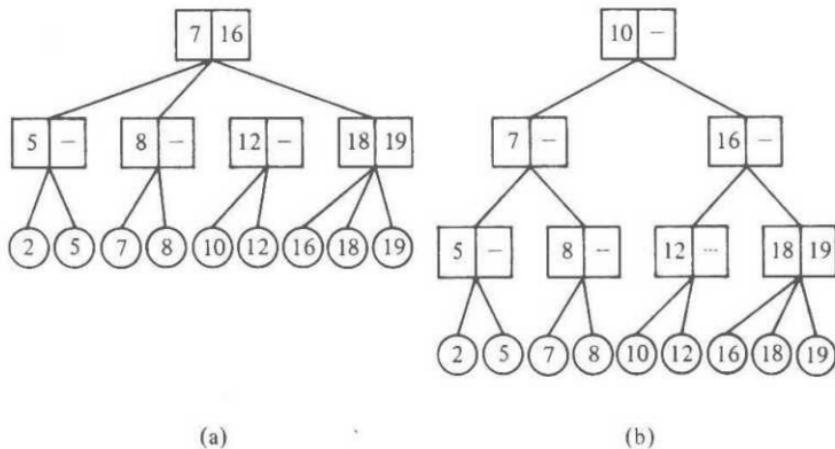


Fig. 5.15. Inserción de 10 en el árbol de la figura 5.14.

Supresión en un árbol 2-3

Al suprimir una hoja, es posible dejar a su *nodo* padre con sólo un hijo. Si *nodo* es la raíz, se suprime *nodo* y se deja el hijo único como nueva raíz. De otra forma, *p* será el padre de *nodo*. Si *p* tiene otro hijo, adyacente a *nodo* por la derecha o por la izquierda, y ese hijo de *p* tiene tres hijos, se puede transferir el más adecuado de esos tres a *nodo*. Entonces *nodo* tendrá dos hijos y se habrá terminado.

Si los hijos de *p* adyacentes a *nodo* tienen sólo dos hijos, se transfiere el único hijo de *nodo* al hermano adyacente de *nodo* y se elimina *nodo*. Si ahora *p* queda sólo con un hijo, se repite todo lo anterior recursivamente, con *p* en lugar de *nodo*.

Ejemplo 5.5. Considérese el árbol de la figura 5.15(b). Si se elimina el 10, su padre tiene sólo un hijo, pero el abuelo tiene otro hijo con tres hijos, 16, 18 y 19. Este nodo está a la derecha del nodo deficitario, por lo que se pasa a ese nodo el elemento más pequeño, 16, quedando el árbol 2-3 de la figura 5.16(a).

A continuación, supóngase que se elimina el 7 del árbol de la figura 5.16(a). Ahora su padre sólo tiene un hijo, 8, y el abuelo no tiene hijos con tres hijos. Por tanto, se hace al 8 hermano de 2 y 5, quedando el árbol de la figura 5.16(b). Ahora el nodo marcado con un asterisco en la figura 5.16(b) tiene sólo un hijo, y su padre no tiene

otros hijos con tres hijos. Se borra entonces el nodo marcado, haciendo que su hijo pase a ser hijo del hermano de ese nodo. Ahora, la raíz tiene sólo un hijo, que se elimina, dejando el árbol de la figura 5.16(c).

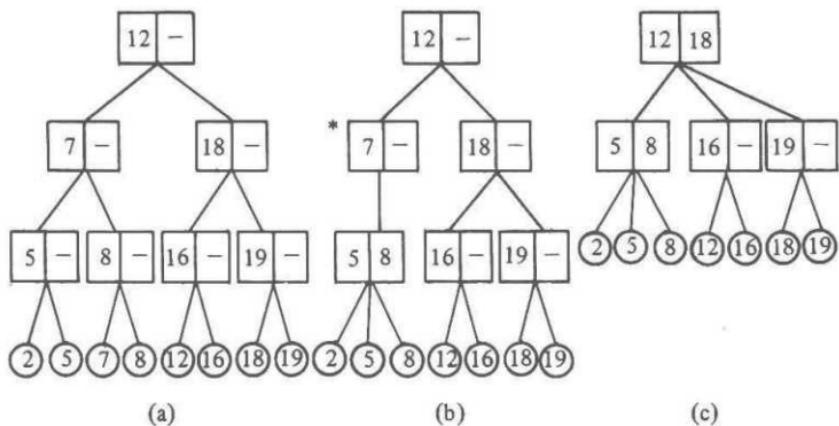


Fig. 5.16. Supresión en un árbol 2-3.

Obsérvese en los ejemplos anteriores la frecuente manipulación de los valores de los nodos interiores. Aunque siempre es posible calcular esos valores recorriendo el árbol, puede hacerse también manipulando el árbol mismo, siempre que se recuerde el valor más pequeño entre los descendientes de cada nodo en el camino que va de la raíz a la hoja eliminada. Esta información puede calcularse con un algoritmo recursivo de eliminación, y con cada llamada en un nodo se pasa, desde arriba, la cantidad correcta (o el valor «menos infinito» si se está en el camino del extremo izquierdo). Los detalles requieren un análisis cuidadoso del caso, y se expondrán al considerar el programa para la operación SUPRIME. □

Tipos de datos para árboles 2-3

Aquí sólo se representarán con árboles 2-3 los conjuntos de elementos cuyas claves sean números reales. La naturaleza de otros campos que van con la clave, para formar un registro de tipo tipo_elemento, se dejará sin especificar, ya que no tiene relación con lo que sigue.

En Pascal, los padres de las hojas deben ser registros que consten de dos números reales (las claves de los elementos más pequeños en el segundo y tercer subárboles) y de tres apuntadores a elementos. Los padres de esos nodos son registros que comprenden dos números reales y tres apuntadores a padres de hojas. Esta progresión continúa indefinidamente: cada nivel de un árbol 2-3 es de un tipo diferente al de los otros niveles. Esta situación haría imposible programar en Pascal las operaciones de árboles 2-3, pero por fortuna, Pascal ofrece un mecanismo, la estructura

de registro variante, que permite considerar a todos los nodos del árbol 2-3 como del mismo tipo, aun cuando algunos sean elementos y otros sean registros con apuntadores y números reales †. Se pueden definir los nodos como en la figura 5.17. Entonces se declararía un conjunto, representado mediante un árbol 2-3, como un apuntador a la raíz, como se muestra en la figura 5.17.

```

type
  tipo_elemento = record
    clave: real;
    { los demás campos requeridos }

  end;
  tipos_nodo = (hoja, interior);
  nodo_dos_tres = record
    case clase: tipos_nodo of
      hoja : (elemento: tipo_elemento);
      interior: (primer_hijo, segundo_hijo, tercer_hijo: ↑ nodo_dos_tres;
                  menor_de_segundo, menor_de_tercero: real)

    end;
  CONJUNTO = ↑ nodo_dos_tres;

```

Fig. 5.17. Definición de un nodo en un árbol 2-3.

Realización de INSERTA

Los detalles de las operaciones en árboles 2-3 son muy complicados, aunque los principios son simples. Así pues, se describe con detalle sólo una operación, la inserción; las otras, supresión y prueba de pertenencia, son similares en esencia, y encontrar el mínimo requiere una búsqueda trivial en el camino del extremo izquierdo. Se escribirá la rutina de inserción como procedimiento principal, INSERTA, que se llama en la raíz, y un procedimiento *inserta1*, al cual se llama recursivamente en el árbol. Por conveniencia, se supondrá que un árbol 2-3 no es un árbol vacío o con un solo nodo. Esos dos casos requieren una secuencia directa de pasos que se recomienda obtener como ejercicio.

Se desea que *inserta1* devuelva un apuntador a un nuevo nodo, si ha de crearlo, y la clave del elemento más pequeño que desciende del nuevo nodo. Como el mecanismo de Pascal para crear tal función es complejo, se declarará *inserta1* como un procedimiento que asigna valores a los parámetros *ap_nuevo* y *menor* en caso que deba «devolver» un nodo nuevo. En la figura 5.18 se muestra un esbozo de *inserta1*. El procedimiento completo se muestra en la figura 5.19; para ahorrar espacio, en la 5.19 se han omitido algunos comentarios de la figura 5.18.

† Sin embargo, todos los nodos toman la mayor cantidad de espacio necesaria para cualquiera de los tipos variantes, así que Pascal no es en realidad el mejor lenguaje para la realización práctica de árboles 2-3.

```

procedure insertal ( nodo: ↑ nodo_dos_tres;
x: tipo_elemento; { x se insertará en el subárbol de nodo }
var ap_nuevo: ↑ nodo_dos_tres; { apuntador al nodo recién creado
a la derecha de nodo }
var menor: real ); { elemento más pequeño del subárbol al que
apunta ap_nuevo}

begin
  ap_nuevo := nil;
  if nodo es una hoja then begin
    if x no es el elemento que está en nodo then begin
      crea un nodo nuevo apuntado por ap_nuevo;
      pone x en el nodo nuevo;
      menor := x.llave
    end
  end
  else begin { nodo es un nodo interno }
    sea w el hijo de nodo a cuyo subárbol pertenece x;
    insertal(w, x, ap_atrás, menor_atrás);
    if ap_atrás <> nil then begin
      inserta el apuntador ap_atrás entre los hijos de
      nodo justo a la derecha de w;
      if nodo tiene cuatro hijos then begin
        crea un nodo nuevo apuntado por ap_nuevo;
        da al nuevo nodo los hijos tercero y cuarto de nodo;
        ajusta menor_de_segundo y menor_de_tercero en nodo
        y el nodo nuevo;
        coloca menor como la menor clave entre los hijos
        del nodo nuevo
      end
    end
  end
end;
{ insertal }

```

Fig. 5.18. Esbozo del programa para inserción en árboles 2-3.

```

procedure insertal ( nodo: ↑ nodo_dos_tres; x: tipo_elemento;
var ap_nuevo: ↑ nodo_dos_tres; var menor: real );

var
  ap_atrás: ↑ nodo_dos_tres;
  menor_atrás: real;
  hijo: 1..3 ; { indica qué hijo de nodo se sigue en la llamada
  recursiva (véase w en la Fig. 5.18)}
  w: ↑ nodo_dos_tres; { apuntador al hijo }

```

```

begin
    ap_nuevo := nil;
    if nodo↑.clase = hoja then begin
        if nodo↑.elemento.clave <> x.clave then begin
            { crea una hoja nueva que contiene x.clave y "devuelve"
              este nodo }
            new(ap_nuevo, hoja);
            if (nodo↑.elemento.clave < x.clave) then
                { coloca x en el nuevo nodo a la derecha del nodo actual }
                begin ap_nuevo↑.elemento := x; menor := x.clave end
            else begin { x está a la izquierda del elemento en el
                         nodo actual }
                ap_nuevo↑.elemento := nodo↑.elemento;
                nodo↑.elemento := x;
                menor := ap_nuevo↑.elemento.clave
            end
        end
    end
    else begin { nodo es un nodo interno }
        { selecciona el hijo de nodo que se debe seguir }
        if x.clave < nodo↑.menor_de_segundo then
            begin hijo := 1; w := nodo↑.primer_hijo end
        else if (nodo↑.tercer_hijo = nil) or (x.clave < nodo↑.menor_de_tercero)
            then begin
                { x está en el segundo subárbol }
                hijo := 2;
                w := nodo↑.segundo_hijo
            end
        else begin { x está en el tercer subárbol }
            hijo := 3;
            w := nodo↑.tercer_hijo
        end;
        insertar(w, x, ap_atrás, menor_atrás);
        if ap_atrás <> nil then
            { debe insertarse un nuevo hijo de nodo }
            if nodo↑.tercer_hijo = nil then
                { nodo tiene sólo dos hijos, así que se inserta el nuevo en el
                  lugar adecuado }
                if hijo = 2 then begin
                    nodo↑.tercer_hijo := ap_atrás;
                    nodo↑.menor_de_tercero := menor_atrás
                end
            else begin { hijo = 1 }
                nodo↑.tercer_hijo := nodo↑.segundo_hijo;
                nodo↑.menor_de_tercero := nodo↑.menor_de_segundo;
                nodo↑.segundo_hijo := ap_atrás;
            end
        end
    end
end;

```

```

nodo†.menor_de_segundo := menor_atrás
end
else begin { nodo ya tiene tres hijos }
  new(ap_nuevo, interior);
  if hijo = 3 then begin
    { ap_atrás y el tercer hijo se convierten en hijos del nuevo
      nodo }
    ap_nuevo†.primer_hijo := nodo†.tercer_hijo;
    ap_nuevo†.segundo_hijo := ap_atrás;
    ap_nuevo†.tercer_hijo := nil;
    ap_nuevo†.menor_de_segundo := menor_atrás;
    { menor_de_tercero está indefinido para ap_nuevo }
    menor := nodo†.menor_de_tercero;
    nodo†.tercer_hijo := nil
  end
  else begin { hijo ≤ 2; pasa el tercer hijo de nodo a ap_nuevo }
    ap_nuevo†.segundo_hijo := nodo†.tercer_hijo;
    ap_nuevo†.menor_de_segundo := nodo†.menor_de_tercero;
    ap_nuevo†.tercer_hijo := nil;
    nodo†.tercer_hijo := nil
  end;
  if hijo = 2 then begin
    { ap_atrás se convierte en el primer hijo de ap_nuevo }
    ap_nuevo†.primer_hijo := ap_atrás;
    menor := menor_atrás
  end;
  if hijo = 1 then begin
    { el segundo hijo de nodo pasa a ap_nuevo; ap_atrás
      se convierte en el segundo hijo de nodo }
    ap_nuevo†.primer_hijo := nodo†.segundo_hijo;
    menor := nodo†.menor_de_segundo;
    nodo†.segundo_hijo := ap_atrás;
    nodo†.menor_de_segundo := menor_atrás
  end
end
end
end; { inserta1 }

```

Fig.5.19. Procedimiento *inserta1*.

Ahora es posible escribir el procedimiento INSERTA, que llama a *inserta1*. Si *inserta1* «devuelve» un nodo nuevo, entonces INSERTA debe crear una raíz nueva. El código se muestra en la figura 5.20 con la suposición de que el tipo CONJUNTO es \dagger nodo_dos_tres, esto es, un apuntador a la raíz de un árbol 2-3 cuyas hojas contienen los miembros del conjunto.

```

procedure INSERTA ( x: tipo_elemento; var S: CONJUNTO );
var
  ap_atrás: ↑nodo_dos_tres; { apuntador al nuevo nodo devuelto por insert1 }
  menor_atrás: real; { menor valor en el subárbol de ap_atrás }
  guardaS: CONJUNTO; { lugar para almacenar una copia temporal del
    apuntador S }
begin
  { prueba si S está vacío o si hay un solo nodo, y debe incluirse un
    procedimiento de inserción apropiado }
  insert1(S, x, ap_atrás, menor_atrás);
  if ap_atrás <> nil then begin
    { crea la raíz nueva; sus hijos están ahora apuntados por S y ap_atrás }
    guardaS := S;
    nuevo(S);
    S↑.primer_hijo := guardaS;
    S↑.segundo_hijo := ap_atrás;
    S↑.menor_de_segundo := menor_atrás;
    S↑.tercer_hijo := nil
  end
end; { INSERTA }

```

Fig. 5.20. INSERTA para conjuntos representados por árboles 2-3.

Realización de SUPRIME

Ahora se bosqueja una función *suprime1* que toma un apuntador al nodo *nodo* y un elemento *x*, y elimina una hoja que desciende de *nodo* que tiene el valor *x*, si es que existe ↑. La función *suprime1* devuelve *true* (verdadero) si después de la eliminación *nodo* tiene sólo un hijo, y devuelve *false* (falso) si *nodo* permanece con dos o tres hijos. Un esbozo del código para *suprime1* se muestra en la figura 5.21.

```

function suprime1 (nodo: ↑nodo_dos_tres; x: tipo_elemento) : boolean;
var
  sólo_uno: boolean; { para guardar el valor devuelto por una llamada a
    suprime1 }
begin
  suprime1 := false;
  if los hijos de nodo son hojas then begin
    if x está entre esas hojas then begin
      elimina x;
      desplaza los hijos de nodo que están a la derecha de x una posición
      hacia la izquierda;

```

† Una variante útil tomaría sólo una clave y eliminaría todo elemento con esa clave.

```

if ahora nodo tiene un hijo then
    suprime1 := true
end
else begin { nodo está en el nivel dos o en un nivel mayor }
    determinar cuál de los hijos de nodo podría tener a x como descendiente;
    sólo_uno:=suprime1(w, x); { w significa nodo.primer_hijo,
        nodo.segundo_hijo o bien nodo.tercer_hijo, según sea lo apropiado }
    if sólo_uno then begin { arregla los hijos de nodo }
        if w es el primer hijo de nodo then
            if y, el segundo hijo de nodo, tiene tres hijos then
                hace que el primer hijo de y sea el segundo hijo de w
            else begin { y tiene dos hijos }
                hace que el hijo de w sea el primer hijo de y;
                elimina w de entre los hijos de nodo;
            if ahora nodo tiene un hijo then
                suprime1 := true
            end;
        if w es el segundo hijo de nodo then
            if y, el primer hijo de nodo, tiene tres hijos then
                hace que el tercer hijo de y sea el primer hijo de w
            else { y tiene dos hijos }
                if z, el tercer hijo de nodo, existe y tiene tres hijos then
                    hace que el primer hijo de z sea el segundo hijo de w
                else begin { ningún otro hijo de nodo tiene tres hijos }
                    hace que el hijo de w sea el tercer hijo de y;
                    elimina w de entre los hijos de nodo;
                if ahora nodo tiene un hijo then
                    suprime1 := true
                end;
            if w es el tercer hijo de nodo then
                if y, el segundo hijo de nodo, tiene tres hijos then
                    hace que el tercer hijo de y sea el segundo hijo de w
                else begin { y tiene dos hijos }
                    hace que el hijo de w sea el tercer hijo de y;
                    elimina w de entre los hijos de nodo
                end { obsérvese que con seguridad nodo tiene aún dos hijos en
                    este caso }
            end
        end
    end; { suprime1 }

```

Fig. 5.21. Procedimiento recursivo de supresión.

El código detallado de la función *suprime1* se deja como ejercicio. Otro ejercicio es escribir un procedimiento SUPRIME (*S*, *x*) que pruebe los casos especiales en

que el conjunto S consta de una sola hoja o está vacío, y en otro caso llame a *suprime1* (S, x); si *suprime1* devuelve *true*, el procedimiento elimina la raíz (el nodo al que apunta S) y hace que S apunte al hijo que quedó solo.

5.5 Conjuntos con las operaciones COMBINA y ENCUENTRA

En ciertos problemas, se empieza con una colección de objetos, cada uno de ellos contenido en un conjunto; después se combinan los conjuntos en algún orden dado, y de vez en cuando se pregunta en qué conjunto se encuentra algún elemento en particular. Estos problemas pueden resolverse por medio de las operaciones COMBINAR y ENCUENTRA. La operación $\text{COMBINA}(A, B, C)$ hace C igual a la unión de los conjuntos A y B , bajo el supuesto de que A y B son disjuntos (no tienen miembros en común); COMBINA está indefinida si A y B no son disjuntos. $\text{ENCUENTRA}(x)$ es una función que devuelve el conjunto del cual x es un miembro; en caso de que x esté en dos o más conjuntos, o en ninguno, ENCUENTRA no está definida.

Ejemplo 5.6. Una *relación de equivalencia* es una relación reflexiva, simétrica y transitiva. Esto es, si \equiv es una relación de equivalencia en el conjunto S , para cualesquiera miembros a, b y c de S (no necesariamente distintos), se cumplen las siguientes propiedades:

1. $a \equiv a$ (*reflexividad*).
2. Si $a \equiv b$, entonces $b \equiv a$ (*simetría*).
3. Si $a \equiv b$ y $b \equiv c$, entonces $a \equiv c$ (*transitividad*).

La relación «igual a» ($=$) es la relación de equivalencia ejemplar en cualquier conjunto S . Para a, b y c de S , se tiene 1) $a = a$, 2) si $a = b$, entonces $b = a$, y 3) si $a = b$ y $b = c$, entonces $a = c$. Existen muchas otras relaciones de equivalencia, sin embargo, y pronto se verán varios ejemplos adicionales.

En general, siempre que se divide una colección de objetos en grupos disjuntos, la relación $a = b$ es de equivalencia si, y sólo si, a y b están en el mismo grupo. «Igual a» es el caso especial donde todo elemento está en grupo por sí solo.

Más formalmente, si un conjunto S tiene una relación de equivalencia definida en él, el conjunto S puede dividirse en subconjuntos disjuntos S_1, S_2, \dots , llamados *clases de equivalencia*, cuya unión es S . Cada subconjunto S_i consta de miembros equivalentes de S . Esto es, $a = b$ para todo a y b en S_i , y $a \neq b$ si a y b están en subconjuntos diferentes. Por ejemplo, la relación congruencia módulo n ^f es una relación de equivalencia en el conjunto de los enteros. Para comprobar que esto es así, obsérvese que $a - a = 0$, que es un múltiplo de n (reflexividad); si $a - b = dn$, entonces $b - a = (-d)n$ (simetría), y si $a - b = dn$ y $b - c = en$, entonces $a - c = (d + e)n$ (transitividad). En el caso de la congruencia módulo n existen n clases de equiva-

^f Se dice que a es congruente con b módulo n si a y b tienen los mismos residuos cuando se dividen entre n , o dicho de otra forma, $a - b$ es múltiplo de n .

lencia, las cuales son el conjunto de los enteros congruentes con 0, el conjunto de los enteros congruentes con 1, ..., el conjunto de los enteros congruentes con $n - 1$.

El problema de equivalencia puede formularse de la siguiente manera. Se dan un conjunto S y una secuencia de proposiciones de la forma « a es equivalente a b ». Hay que procesar las proposiciones en orden, de manera que en cualquier momento pueda determinarse a qué clase de equivalencia pertenece un elemento dado. Por ejemplo, supóngase que $S = \{1, 2, \dots, 7\}$ y se tiene la secuencia de proposiciones

$$1=2 \quad 5=6 \quad 3=4 \quad 1=4$$

para procesar. Es necesario construir la siguiente secuencia de clases de equivalencia, suponiendo que inicialmente cada elemento de S está en una clase de equivalencia propia.

$$1=2 \quad \{1,2\} \quad \{3\} \quad \{4\} \quad \{5\} \quad \{6\} \quad \{7\}$$

$$5=6 \quad \{1,2\} \quad \{3\} \quad \{4\} \quad \{5,6\} \quad \{7\}$$

$$3=4 \quad \{1,2\} \quad \{3,4\} \quad \{5,6\} \quad \{7\}$$

$$1=4 \quad \{1,2,3,4\} \quad \{5,6\} \quad \{7\}$$

Se puede «resolver» el problema de equivalencia empezando con cada elemento de un conjunto determinado. Al procesar la proposición $a=b$, se ENCUENTRAN las clases de equivalencia de a y b , y después se COMBINAN. En cualquier momento se puede usar ENCUENTRA para conocer la clase de equivalencia actual de cualquier elemento.

El problema de equivalencia surge en varias áreas de las ciencias de la computación. Por ejemplo, una forma ocurre cuando un compilador de Fortran tiene que procesar «declaraciones de equivalencia» como

EQUIVALENCE (A(1), B(1, 2), C(3)), (A(2), D, E), (F, G)

Otro ejemplo, presentado en el capítulo 6, usa soluciones al problema de equivalencia para ayudar a encontrar árboles abarcadores de costo mínimo. □

Una realización simple de CONJUNTO_CE

Ahora se presenta una versión simplificada del TDA COMBINA_ENCUENTRA, al definir un TDA llamado CONJUNTO_CE, que consiste en un conjunto de subconjuntos llamados *componentes*, junto con las siguientes operaciones:

1. COMBINA(A, B), que toma la unión de los componentes A y B y al resultado lo llama A o B , arbitrariamente.

2. ENCUENTRA(x), que es una función que devuelve el nombre del componente del cual x es un miembro.
3. INICIAL(A, x), que crea un componente llamado A que contiene sólo el elemento x .

Para hacer una realización razonable de CONJUNTO_CE, se deben restringir los tipos o reconocer que CONJUNTO_CE en realidad tiene otros dos tipos como «parámetros»: el tipo de los nombres de los conjuntos y el tipo de los miembros de esos conjuntos. En muchas aplicaciones se pueden usar enteros como nombres de conjuntos. Si n es el número de elementos, también se pueden usar enteros en el intervalo [1.. n] para los miembros de los componentes. Para la implantación en cuestión, es importante que el tipo de los miembros de los conjuntos sea del tipo subintervalo, porque se desea indizar en un arreglo definido en él. El tipo de los nombres de los conjuntos no es importante, pues es del tipo de los elementos del arreglo, no de sus índices. Obsérvese, sin embargo, que si se desea que el tipo de los miembros sea distinto a un subintervalo, se puede crear una correspondencia con una tabla de dispersión, por ejemplo, que los asigne a enteros únicos de un subintervalo. Sólo es necesario conocer por adelantado el número total de elementos.

La aplicación que se está considerando es declarar

```
const
  n = {número de elementos};
type
  CONJUNTO_CE = array[1..n] of integer;
```

como un caso especial del tipo más general

```
array[subintervalo de miembros] of (tipo de nombres de los conjuntos);
```

Supóngase que se declaran *componentes* del tipo CONJUNTO_CE con la intención de que *componentes*[x] contenga el nombre del conjunto en el cual se encuentra x . Entonces, las tres operaciones para CONJUNTO_CE son fáciles de escribir. Por ejemplo, la operación COMBINA se muestra en la figura 5.22. INICIAL(A, x) simplemente hace que *componentes*[x] sea igual a A , y ENCUENTRA(x) devuelve *componentes*[x].

```
procedure COMBINA ( A, B: integer; var C: CONJUNTO_CE );
  var
    x: 1..n;
  begin
    for x := 1 to n do
      if C[x] = B then
        C[x] := A
    end; { COMBINA }
```

Fig. 5.22. El procedimiento COMBINA.

El rendimiento en tiempo de esta implantación de CONJUNTO_CE es fácil de analizar. Cada ejecución del procedimiento COMBINA lleva un tiempo $O(n)$. Por otro lado, las implantaciones obvias de INICIAL(A, x) y ENCUENTRA(x) tienen tiempos de ejecución constantes.

Realización más rápida de CONJUNTO_CE

Al utilizar el algoritmo de la figura 5.22, una secuencia de $n - 1$ operaciones COMBINA llevará un tiempo $O(n^2)$ †. Una forma de acelerar la operación COMBINA es al encadenar todos los miembros de un componente en una lista. Entonces, en vez de leer todos los miembros cuando se combina el componente B en A , basta recorrer la lista de los miembros de B . Esta organización aprovecha mejor el tiempo en el caso promedio. Sin embargo, podría suceder que la i -ésima combinación fuera de la forma COMBINA(A, B), donde A sería un componente de tamaño uno y B sería un componente de tamaño i , y que el resultado se llamara B . Esta operación COMBINA requeriría $O(i)$ pasos, y una secuencia de $n - 1$ instrucciones COMBINA lleva-

$$\text{ría un tiempo del orden de } \sum_{i=1}^{n-1} i = n(n-1)/2.$$

Una forma de evitar esta situación del peor caso es cuidar el tamaño de cada componente y combinar siempre el más pequeño dentro del más grande ‡. Así, cada vez que un miembro se combina con un componente más grande, se encuentra a sí mismo en un componente al menos dos veces más grande. De esta forma, si existen n componentes inicialmente, cada uno con un miembro, ninguno de los n miembros puede tener su componente cambiado más de $1 + \log n$ veces. Como el tiempo consumido por esta nueva versión de COMBINA es proporcional al número de miembros cuyos nombres de componentes se cambian, y el número total de tales cambios puede ser hasta $n(1 + \log n)$, el trabajo $O(n \log n)$ basta para todas las combinaciones.

Ahora, considérese la estructura de datos necesaria para esta realización. Primero se necesita una correspondencia de los nombres de conjuntos con los registros que consisten en

1. un *contador* que da el número de miembros del conjunto y
2. el índice en el arreglo del primer elemento de ese conjunto.

También se necesita otro arreglo de registros, indizados de acuerdo con los miembros, para indicar

1. el conjunto del cual cada elemento es miembro y
2. el siguiente elemento del arreglo en la lista de ese conjunto.

† Obsérvese que $n-1$ es el mayor número de combinaciones que pueden hacerse antes de que todos los elementos estén en un solo conjunto.

‡ Obsérvese que la capacidad para llamar al componente resultante de acuerdo con el nombre de cualquiera de sus elementos es importante aquí, aunque en la realización más sencilla se escoge siempre el nombre del primer argumento.

Se emplea 0 como equivalente a NIL, la marca de fin de lista. En un lenguaje que se preste para este tipo de construcciones, sería preferible el uso de apuntadores en este arreglo, pero Pascal no permite apuntadores en los arreglos.

En el caso especial donde los nombres de los conjuntos, al igual que los miembros, se escogen del subintervalo 1..n, es posible usar un arreglo para la correspondencia descrita antes. Esto es, se define

```

type
  tipo_nombre = 1..n;
  tipo_elemento = 1..n;
  CONJUNTO_CE = record
    encabezamientos_conjuntos : array[1..n] of record
      |encabezamientos para las listas de conjuntos|
      contador: 0..n;
      primer_elemento: 0..n
    end;
    nombres: array[1..n] of record
      |tabla que da los conjuntos que contienen a cada miembro|
      nombre_conjunto: tipo_nombre;
      siguiente_elemento: 0..n
    end
  end;

```

Los procedimientos INICIAL, COMBINA y ENCUENTRA se muestran en la figura 5.23.

```

procedure INICIAL ( A: tipo_nombre; x: tipo_elemento; var C: CONJUNTO_CE );
{ asigna como valor inicial de A un conjunto que sólo contiene a x }
begin
  C.nombres[x].nombre_conjunto := A;
  C.nombres[x].siguiente_elemento := 0;
  { apuntador nulo al final de la lista de los miembros de A }
  C.encabezamientos_conjuntos[A].cuenta := 1;
  C.encabezamientos_conjuntos[A].primer_elemento := x
end; { INICIAL }

procedure COMBINA ( A, B: tipo_nombre; var C: CONJUNTO_CE );
{ combina A y B y llama al resultado A o B, arbitrariamente }
var
  i: 0..n; { se usa para encontrar el fin de la lista más pequeña }
begin
  if C.encabezamientos_conjuntos[A].cuenta >
    C.encabezamientos_conjuntos[B].cuenta then begin
      { A es el conjunto más grande; combina B dentro de A }
      { encuentra el final de B, cambiando los nombres de los conjuntos
        por A conforme se avanza }
      i := C.encabezamientos_conjuntos[B].primer_elemento;
    end
  else begin
    { B es el conjunto más grande; combina A dentro de B }
    { encuentra el final de A, cambiando los nombres de los conjuntos
      por B conforme se avanza }
    i := C.encabezamientos_conjuntos[A].primer_elemento;
  end
end; { COMBINA }

```

```

repeat
    C.nombres[i].nombre_conjunto := A;
    i := C.nombres[i].siguiente_elemento
until C.nombres[i].siguiente_elemento = 0;
{ añade la lista A al final de la B y llama A al resultado }
{ ahora i es el índice del último miembro de B }
C.nombres[i].nombre_conjunto := A;
C.nombres[i].siguiente_elemento:=
    C.encabezamientos_conjuntos[A].primer_elemento;
C.encabezamientos_conjuntos[A].primer_elemento:=
    C.encabezamientos_conjuntos[B].primer_elemento;
C.encabezamientos_conjuntos[A].cuenta:=
    C.encabezamientos_conjuntos[A].cuenta+
    C.encabezamientos_conjuntos[B].cuenta;
C.encabezamientos_conjuntos[B].primer_elemento:= 0
C.encabezamientos_conjuntos[B].primer_elemento:= 0
    { los dos pasos anteriores no son realmente necesarios, pues el
    conjunto B ya no existe }

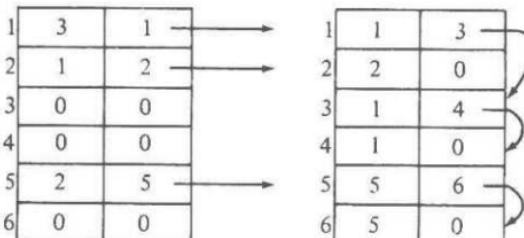
end
else { B es al menos tan grande como A }
    { código similar al del caso anterior, pero con A y B intercambiados }
end; { COMBINA }

action ENCUENTRA ( x: 1..n; var C: CONJUNTO_CE );
{ devuelve el nombre de aquel conjunto que tiene a x como miembro }
begin
    return (C.nombres[x].nombre_conjunto)
end; { ENCUENTRA }

```

Fig. 5.23. Las operaciones de un CONJUNTO_CE.

La figura 5.24 muestra un ejemplo de la estructura de datos empleada en la figura 5.23, donde el conjunto 1 es {1, 3, 4}, el conjunto 2 es {2}, y el conjunto 5 es {5, 6}.



*cuenta primer_elemento nombre_conjunto siguiente_elemento
encabezamientos_conjuntos nombres*

Fig. 5.24. Ejemplo de la estructura de datos CONJUNTO_CE.

Realización de CONJUNTO_CE con árboles

Otro enfoque completamente distinto para la realización de CONJUNTO_C usa árboles con apuntadores a los padres. Se describirá de manera informal este enfoque. La idea básica es que los nodos de los árboles se correspondan con los miembros del conjunto, con un arreglo u otra realización de una correspondencia que vaya de los miembros del conjunto a sus nodos. A excepción de la raíz del árbol, cada nodo tiene un apuntador a su parente. Las raíces tienen el nombre del conjunto, además de un elemento. Una correspondencia de los nombres del conjunto con las raíces permite el acceso a cualquier conjunto dado al hacer las combinaciones.

La figura 5.25 muestra los conjuntos $A = \{1, 2, 3, 4\}$, $B = \{5, 6\}$ y $C = \{7\}$ representados de esta forma: Se supone que los rectángulos son parte del nodo raíz, no nodos independientes.

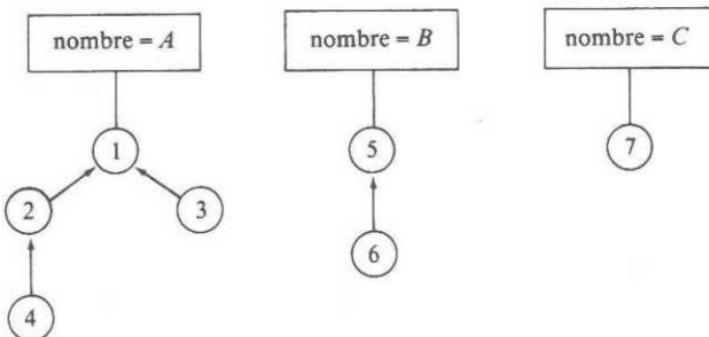


Fig. 5.25. CONJUNTO_CE representado por una colección de árboles.

Para encontrar el conjunto que contiene un elemento x , primero se consulta una correspondencia (por ejemplo, un arreglo), que no se muestra en la figura 5.25, para obtener un apuntador al nodo para x . Después, se sigue el camino de ese nodo a la raíz de su árbol para leer el nombre de ese conjunto.

La operación de combinación básica consiste en hacer que la raíz de un árbol sea el hijo de la raíz del otro. Por ejemplo, se podrían combinar A y B de la figura 5.25 y llamarle A al resultado, haciendo que el nodo 5 sea hijo del nodo 1. El resultado se muestra en la figura 5.26. Sin embargo, la combinación indiscriminada podría producir un árbol de n nodos que fuera una sola cadena. Entonces, hacer la operación ENCUENTRA en cada uno de esos nodos llevaría un tiempo $O(n^2)$. Obsérvese que aunque una combinación se puede hacer en $O(1)$ pasos, el costo de un número razonable de procedimientos ENCUENTRA dominaría en el costo total, y este enfoque no es necesariamente mejor que uno más simple para ejecutar n procedimientos combina y n encuentra.

Sin embargo, una mejora sencilla garantiza que si n es el número de elementos, ningún ENCUENTRA necesitará más de $O(\log n)$ pasos. Sólo se conserva un contador del número de elementos del conjunto en cada raíz, y al intentar combinar dos

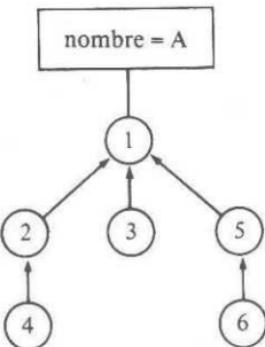


Fig. 5.26. Combinación de B dentro de A .

conjuntos, se hace que la raíz del árbol más pequeño sea un hijo de la raíz del más grande. Así, cada vez que un nodo pasa a un árbol nuevo, suceden dos cosas: la distancia del nodo a su raíz se incrementa en 1, y el nodo estará en un conjunto con por lo menos el doble de elementos que antes. Así, si n es el número total de elementos, ningún nodo puede moverse más de $\log n$ veces; la distancia a su raíz nunca puede exceder de $\log n$. Se concluye que cada ENCUENTRA requiere un máximo de tiempo $O(\log n)$.

Compresión de caminos

Otra idea que puede acelerar esta realización de CONJUNTO_CE es una *compresión de caminos*. Durante un procedimiento ENCUENTRA, al seguir un camino desde algún nodo hasta la raíz, se hace que cada nodo encontrado en el camino sea un hijo de la raíz. La forma más fácil de realizar esto es en dos pasos. Primero se encuentra la raíz y después se recorre de nuevo el mismo camino, haciendo que cada nodo sea hijo de la raíz.

Ejemplo 5.7. La figura 5.27(a) muestra un árbol antes de ejecutar una operación ENCUENTRA en el nodo del elemento 7, y la figura 5.27(b) muestra el resultado después de quedar 5 y 7 como hijos de la raíz. Los nodos 1 y 2 del camino no se mueven porque 1 es la raíz y 2 ya es hijo de la raíz. □

La compresión de caminos no afecta al costo de los procedimientos COMBINA; cada uno de ellos continúa consumiendo una cantidad constante de tiempo. Sin embargo, existe un ligero incremento en la velocidad de ENCUENTRA, ya que la compresión de caminos tiende a acortar un número grande de ellos, desde varios nodos hasta la raíz, con relativamente poco esfuerzo.

Desafortunadamente, es muy difícil analizar el costo promedio de ENCUENTRA cuando se usa la compresión de caminos. De manera que si no es necesario que los árboles más cortos se combinen dentro de los más grandes, no se requerirá

más tiempo que $O(n \log n)$ para hacer n procedimientos ENCUENTRA. Por supuesto, el primero de ellos puede llevar un tiempo $O(n)$ por sí mismo para un árbol que conste de una cadena. Pero la compresión puede cambiar muy rápido un árbol, y con independencia del orden de aplicación de ENCUENTRA a los elementos de cualquier árbol, no se necesitará un tiempo mayor que $O(n)$ para n procedimientos ENCUENTRA. Sin embargo, existen secuencias de las instrucciones COMBINA y ENCUENTRA que requiere un tiempo $\Omega(n \log n)$.

El algoritmo que usa compresión de caminos y combina los árboles más pequeños dentro de los más grandes es asintóticamente el método más eficiente conocido para aplicar CONJUNTO-CE. En particular, n procedimientos ENCUENTRA no requieren un tiempo mayor que $O(n\alpha(n))$, donde $\alpha(n)$ es una función no constante, pero que crece mucho más lentamente que $\log n$. Se definirá $\alpha(n)$ enseguida, pero el análisis que da lugar a esta cota está fuera del alcance de este libro.

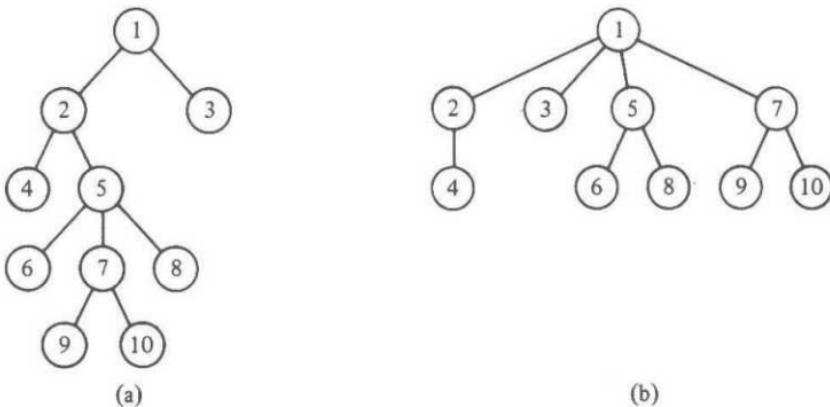


Fig. 5.27. Ejemplo de compresión de caminos.

Función $\alpha(n)$

La función $\alpha(n)$ está muy relacionada con una función de crecimiento muy rápido $A(x, y)$, conocida como *función de Ackermann*. $A(x, y)$ está definida en forma recursiva por:

$$A(0, y) = 1 \text{ para } y \geq 0$$

$$A(1, 0) = 2$$

$$A(x, 0) = x + 2 \text{ para } x \geq 2$$

$$A(x, y) = A(A(x - 1, y), y - 1) \text{ para } x, y \geq 1$$

Cada valor de y define una función de una variable. Por ejemplo, la tercera línea dice que para $y = 0$, esta función es «sumar 2». Para $y = 1$, se tiene $A(x, 1) = A(A(x - 1, 1), 0) = A(x - 1, 1) + 2$, para $x > 1$, con $A(1, 1) = A(A(0, 1), 0) = A(1, 0) =$

= 2. Así, $A(x, 1) = 2x$ para toda $x \geq 1$. En otras palabras, $A(x, 1)$ es «multiplicar por 2». Despues, $A(x, 2) = A(A(x - 1, 2), 1) = 2A(x - 1, 2)$ para $x > 1$. Tambien, $A(1, 2) = A(A(0, 2), 1) = A(1, 1) = 2$. Así, $A(x, 2) = 2^x$. De modo semejante, es posible demostrar que $A(x, 3) = 2^{2^{x-1}}$ (pila de x nmeros 2), mientras que $A(x, 4)$ crece tan rpidamente que no existe ninguna notacin matemtica aceptada para tal funcin.

Una funcin de Ackermann de una sola variable puede definirse haciendo $A(x) = A(x, x)$. La funcin $a(n)$ es una seudoinversa de esta funcin de una sola variable. Esto es, $a(n)$ es la menor x tal que $n \leq A(x)$. Por ejemplo, $A(1) = 2$, as $a(1) = a(2) = 1$. $A(2) = 4$, por lo que $a(3) = a(4) = 2$. $A(3) = 8$, de modo que $a(5) = \dots = a(8) = 3$. De lo anterior parece que $a(n)$ crece bastante uniformemente.

Sin embargo, $A(4)$ es una pila de 65 536 nmeros 2. Como $\log(A(4))$ es una pila de 65 535 nmeros 2, no cabe esperar siquiera escribir $A(4)$ en forma explcita, ya que se necesitaran $\log(A(4))$ bits. As, $a(n) \leq 4$ para todos los enteros n que sea posible encontrar. Aun as, $a(n)$ alcanzará finalmente los valores 5, 6, 7, ... en su curso inimaginablemente lento a infinito.

5.6 TDA con COMBINA y DIVIDE

Sea S un conjunto cuyos miembros estn ordenados de acuerdo con la relacin $<$. La operacin $\text{DIVIDE}(S, S_1, S_2, x)$ divide a S en dos conjuntos: $S_1 = \{a \mid a \text{ est} \text{ en } S \text{ y } a < x\}$ y $S_2 = \{a \mid a \text{ est} \text{ en } S \text{ y } a \geq x\}$. El valor de S despus de dividirse es indefinido, a menos que sea S_1 o S_2 . Existen varias situaciones donde es imprescindible la operacin de divisin de conjuntos al comparar cada miembro con un valor fijo x . Se considerar un de esos problemas aqu.

Problema de la subsecuencia comn mstica larga

Una *subsecuencia* de una secuencia x se obtiene al eliminar cero o msticos elementos de x (no necesariamente contiguos). Dadas dos secuencias x e y , una *subsecuencia comn mstica larga (SCL)* es una secuencia mstica larga que es una subsecuencia de x y de y .

Por ejemplo, una SCL de 1, 2, 3, 2, 4, 1, 2 y de 2, 4, 3, 1, 2, 1 es la subsecuencia 2, 3, 2, 1, formada como se muestra en la figura 5.28. Existen otras SCL tambin, como 2, 4, 1, 2, pero no existen subsecuencias comunes de longitud 5. Existe un mandato de UNIX, llamado *diff*, que compara archivos lnea por lnea y encuentra una subsecuencia comn mstica larga, donde una lnea de un archivo se considera como un elemento de la subsecuencia, esto es, las lneas completas son semejantes a los enteros 1, 2, 3 y 4 de la figura 5.28. La suposicin que respalda al mandato *diff* es que las lneas de cada archivo que no se encuentran en esta SCL son lneas insertadas, suprimidas o modificadas al ir de un archivo al otro. Por ejemplo, si los dos archivos son versiones del mismo programa realizadas con varios dais de diferencia, *diff* encontrará los cambios, con una alta probabilidad.

Es posible encontrar varias soluciones generales al problema SCL que funcionan en $O(n^2)$ pasos en secuencias de longitud n . El mandato *diff* usa una estrategia dife-

rente que funciona bien cuando los archivos no tienen demasiadas repeticiones de ninguna línea. Por ejemplo, los programas tenderán a tener líneas `begin` y `end` repetidas muchas veces, pero no es probable que otras líneas se repitan.

El algoritmo empleado por `diff` para encontrar una SCL hace uso de una realización eficiente de conjuntos con las operaciones COMBINA y DIVIDE, para trabajar en un tiempo $O(plogn)$, donde n es el número máximo de líneas de un archivo y p es el número de pares de posiciones, una de cada archivo, que tienen la misma línea. Por ejemplo, el valor de p para las cadenas de la figura 5.28 es 12. Los dos unos de cada cadena contribuyen con cuatro pares, los números 2 contribuyen con seis pares, y 3 y 4 contribuyen con un par cada uno. En el peor caso, p podría ser n^2 , y este algoritmo llevaría un tiempo $O(n^2logn)$. Sin embargo, en la práctica, p suele estar más cercano a n , por lo que puede esperarse una complejidad de tiempo $O(nlogn)$.

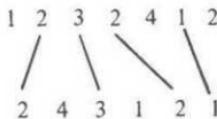


Fig. 5.28. Subsecuencia común más larga.

Para empezar la descripción del algoritmo, sean $A = a_1 a_2 \dots a_n$ y $B = b_1 b_2 \dots b_m$ las dos cadenas de las cuales se desea obtener la SCL. El primer paso es tabular las posiciones de la cadena A en las que aparezca a para cada valor de a . Esto es, se define $\text{LUGARES}(a) = \{i \mid a = a_i\}$. Se pueden calcular los conjuntos $\text{LUGARES}(a)$ al construir una correspondencia de símbolos a encabezados de las listas de posiciones. Por medio de una tabla de dispersión se pueden crear los conjuntos $\text{LUGARES}(a)$ con $O(n)$ «pasos» en promedio, donde un «paso» es el tiempo que lleva operar en un símbolo, por ejemplo, dispersarlo o compararlo con otro. Este tiempo podría ser una constante si los símbolos fueran caracteres o enteros. Sin embargo, si los símbolos de A y B son en realidad líneas de texto, los pasos llevan una cantidad de tiempo que depende de la longitud promedio de una línea de texto.

Al terminar el cálculo de $\text{LUGARES}(a)$ para cada símbolo a que aparece en una cadena A , es posible encontrar una SCL. Para simplificar las cosas, sólo se mostrará cómo encontrar la longitud de la SCL y se deja como ejercicio la construcción real de la SCL. El algoritmo considera cada b_j , para $j = 1, 2, \dots, m$, en orden. Después de considerar b_j , es necesario conocer la longitud de la SCL de las cadenas $a_1 \dots a_i$ y $b_1 \dots b_j$ para cada i entre 0 y n .

Se agrupan los valores de i en conjuntos S_k , para $k = 0, 1, \dots, n$, donde S_k consiste en todos los enteros i tales que la SCL de $a_1 \dots a_i$ y $b_1 \dots b_j$ tiene la longitud k . Obsérvese que S_k siempre será un conjunto de enteros consecutivos, y los enteros de S_{k+1} son mayores que los de S_k para toda k .

Ejemplo 5.8. Considérese la figura 5.28, con $j = 5$. Si se intenta comparar cero símbolos de la primera cadena con los cinco primeros de la segunda (24312), se tendrá una SCL de longitud 0, y así 0 está en S_0 . Si se emplea el primer símbolo de la primera cadena, es posible obtener una SCL de longitud 1, y si se manejan los dos pri-

meros símbolos, 12, una SCL de longitud 2. Sin embargo, al usar 123, los primeros tres símbolos, también se obtiene una SCL de longitud 2 cuando se comparan con 24312. Procediendo de esta manera, se descubre que $S_0 = \{0\}$, $S_1 = \{1\}$, $S_2 = \{2, 3\}$, $S_3 = \{4, 5, 6\}$ y $S_4 = \{7\}$. \square

Supóngase que se han calculado los S_k para la posición $j - 1$ de la segunda cadena y se desean modificar para aplicarlos a la posición j . Considérese el conjunto LUGARES(b_j). Para cada r en LUGARES(b_j), se ve si es posible aumentar alguna de las SCL al agregar el resultado de comparar a_r y b_j a la SCL de $a_1 \dots a_{r-1}$ y de $b_1 \dots b_j$. Esto es, si tanto $r - 1$ como r están en S_k , entonces toda $s \geq r$ en S_k pertenece en realidad a S_{k+1} cuando se toma en consideración b_j . Para ver esto, se observa que se pueden obtener k comparaciones entre $a_1 \dots a_{r-1}$ y $b_1 \dots b_{j-1}$, a las cuales se les agrega la comparación entre a_r y b_j . S_k y S_{k+1} se pueden modificar con los siguientes pasos.

1. ENCUENTRA(r) para obtener S_k .
2. Si ENCUENTRA($r - 1$) no está en S_k , entonces no se logra ninguna mejora de comparar b_j con a_r . Saltar los siguientes pasos y no modificar S_k o S_{k+1} .
3. Si ENCUENTRA($r - 1$) = S_k , aplicar DIVIDE(S_k , S_k , S'_k , r) para separar de S_k los miembros que sean mayores o iguales que r .
4. COMBINA (S'_k , S_{k+1} , S_{k+1}) para pasar esos elementos a S_{k+1} .

Es importante considerar primero los miembros más grandes de LUGARES(b_j). Para ver por qué, supóngase, por ejemplo, que 7 y 9 pertenecen a LUGARES(b_j) y que, antes de considerar b_j , $S_3 = \{6, 7, 8, 9\}$ y $S_4 = \{10, 11\}$.

Si se considera 7 antes que 9, se divide S_3 en $S_3 = \{6\}$ y $S'_3 = \{7, 8, 9\}$, entonces se hace $S_4 = \{7, 8, 9, 10, 11\}$. Si luego se considera 9, se divide S_4 en $S_4 = \{7, 8\}$ y $S'_4 = \{9, 10, 11\}$, para combinar 9, 10 y 11 en S_5 . Así, se ha pasado 9 de S_3 a S_5 considerando sólo una posición más en la segunda cadena, que representa una imposibilidad. Intuitivamente, lo que sucede es que se ha comparado en forma errónea b_j con a_7 y a_9 al crear una SCL imaginaria de longitud 5.

En la figura 5.29, se observa un bosquejo del algoritmo que genera los conjuntos S_k conforme se va analizando la segunda cadena. Para determinar la longitud de una SCL, sólo hay que ejecutar ENCUENTRA(n) al final.

```

procedure SCL;
begin
(1)    asigna valores iniciales  $S_0 = \{0, 1, \dots, n\}$  y  $S_i = \emptyset$  para  $i = 1, 2, \dots, n$ ;
(2)    for  $j := 1$  to  $n$  do { calcula los  $S_k$  en la posición  $j$  }
(3)        for  $r$  en LUGARES ( $b_j$ ), primero el mayor do begin
(4)             $k := \text{ENCUENTRA}(r)$ ;
(5)            if  $k = \text{ENCUENTRA}(r-1)$  then begin {  $r$  no es el menor en  $S_k$  }
(6)                DIVIDE( $S_k$ ,  $S_k$ ,  $S'_k$ ,  $r$ );
(7)                COMBINA ( $S'_k$ ,  $S_{k+1}$ ,  $S_{k+1}$ )
            end
        end
    end;
end; { SCL }
```

Fig. 5.29. Esbozo del programa de subsecuencia común más larga.

Análisis de tiempo del algoritmo SCL

Como se mencionó, el algoritmo de la figura 5.29 es un enfoque útil sólo si no existen demasiadas comparaciones entre símbolos de las dos cadenas. La medida de número de comparaciones es

$$p = \sum_{j=1}^m |\text{LUGARES}(b_j)|$$

donde $|\text{LUGARES}(b_j)|$ denota el número de elementos en el conjunto $\text{LUGARES}(b_j)$. En otras palabras, p es la suma sobre todas las b_j del número de posiciones de la primera cadena que coinciden con b_j . Recuérdese que en el análisis de la comparación de archivos, se esperaba que p fuera del mismo orden que m y n , las longitudes de las dos cadenas (archivos).

Esto hace que el árbol 2-3 sea una buena estructura para los conjuntos S_k . Se puede asignar valor inicial a esos conjuntos, como en la línea (1) de la figura 5.29, en $O(n)$ pasos. La operación ENCUENTRA requiere un arreglo que sirva como una correspondencia de las posiciones r a las hojas de r y también requiere apuntadores a los padres en el árbol 2-3. El nombre del conjunto, es decir, k para S_k , puede conservarse en la raíz, así que se puede ejecutar ENCUENTRA en $O(\log n)$ pasos, siguiendo los apuntadores a los padres hasta llegar a la raíz. Así, el conjunto de las ejecuciones de las líneas (4) y (5) juntas lleva un tiempo $O(p \log n)$, pues dichas líneas se ejecutan una a la vez, para cada coincidencia encontrada.

La operación COMBINA de la línea (5) tiene la propiedad especial de que cada miembro de S'_k es menor que cada miembro de S_{k+1} , y se puede aprovechar este hecho cuando se usan árboles 2-3 para la aplicación \dagger . Para empezar la operación COMBINA, se coloca el árbol 2-3 de S'_k a la izquierda del de S_{k+1} . Si ambos tienen la misma altura, se crea una raíz nueva con las raíces de los dos árboles como sus hijos. Si S'_k es más corto, se inserta la raíz de ese árbol como el hijo más a la izquierda del nodo más a la izquierda de S_{k+1} en el nivel apropiado. Si este nodo tiene ahora cuatro hijos, se modifica el árbol exactamente de la misma forma que se hizo en el procedimiento INSERTAR de la figura 5.20. En la figura 5.30 se muestra un ejemplo. En forma semejante, si S_{k+1} es más corto, se hace que su raíz quede como el hijo más a la derecha del nodo más a la derecha de S'_k en el nivel apropiado.

La operación DIVIDE en r requiere subir en el árbol a partir de una hoja r , duplicar cada nodo interior del camino y dar una copia para cada uno de los dos árboles resultantes. Los nodos sin hijos se eliminan, y los nodos con un hijo se retiran para que ese hijo quede insertado en el árbol y nivel adecuados.

Ejemplo 5.9. Supóngase que se divide el árbol de la figura 5.30(b) en el nodo 9. Los dos árboles con nodos duplicados se muestran en la figura 5.31(a). A la izquierda, el padre de 8 tiene sólo un hijo, por lo que 8 se convierte en hijo del padre de 6 y 7.

\dagger Estrictamente hablando, se debería usar un nombre diferente para la operación COMBINA, pues la realización que se propone no efectúa la unión arbitraria de conjuntos disjuntos, y hace que los elementos estén clasificados para que puedan llevarse a cabo operaciones como DIVIDE y ENCUENTRA.

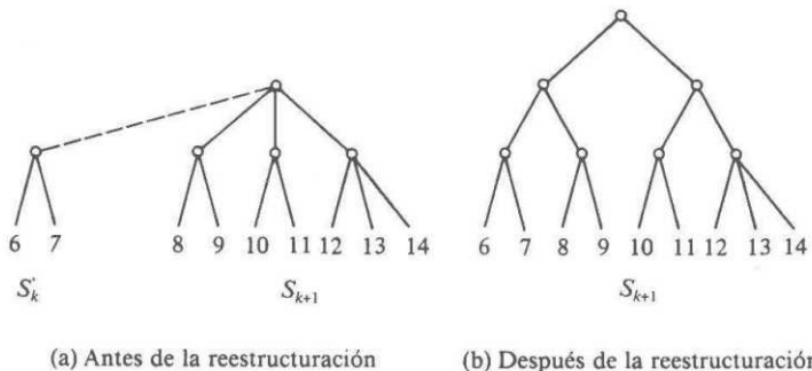


Fig. 5.30. Ejemplo de COMBINA.

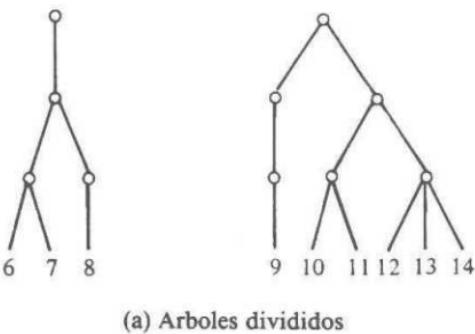


Fig. 5.31. Ejemplo de DIVIDE.

Este padre tiene ahora tres hijos, así que todo queda como debe ser; si tuviera cuatro hijos, se habría creado un nodo nuevo e insertado en el árbol. Sólo es necesario eliminar nodos con cero hijos (el padre anterior de 8) y la cadena de nodos con un

hijo que llega a la raíz. El padre de 6, 7 y 8 se convierte en la nueva raíz, como se muestra en la figura 5.31(b). En forma semejante, en el árbol del lado derecho, 9 queda como hermano de 10 y 11, y se eliminan los nodos innecesarios, como se muestra también en la figura 5.31(b). \square

Si se divide y reorganiza el árbol 2-3 de abajo hacia arriba, considerando una gran cantidad de casos es posible demostrar que $O(\log n)$ pasos son suficientes. Así, el tiempo total consumido en las líneas (6) y (7) de la figura 5.29 es $O(p \log n)$, y el algoritmo en su totalidad requiere $O(p \log n)$ pasos. Es necesario agregar el tiempo de preprocessamiento requerido para calcular y clasificar LUGARES(a) para los símbolos a . Como ya se mencionó, si los símbolos a son objetos «grandes», este tiempo puede ser mucho mayor que cualquier otra parte del algoritmo. Como se verá en el capítulo 8, si los símbolos pueden ser manipulados y comparados en «pasos» sencillos, un tiempo $O(n \log n)$ es suficiente para clasificar la primera cadena $a_1 a_2 \dots a_n$ (en realidad, para clasificar los objetos (i, a_i) en el segundo campo), así que LUGARES(a) puede leerse en esta lista en un tiempo $O(n)$. Así, la longitud de la SCL puede calcularse en un tiempo $O(\max(n, p) \log n)$, el cual puede considerarse como $O(p \log n)$, ya que $p \geq n$ es normal.

Ejercicios

- 5.1 Dibújense todos los posibles árboles binarios de búsqueda que contengan los elementos 1, 2, 3 y 4.
- 5.2 Insértense los enteros 7, 2, 9, 0, 5, 6, 8, 1 en un árbol binario de búsqueda por medio de la aplicación repetida del procedimiento INSERTA de la figura 5.3.
- 5.3 Muéstrese el resultado obtenido al suprimir 7 y después 2 del árbol final del ejercicio 5.2.
- *5.4 Cuando se eliminan dos elementos de un árbol binario de búsqueda con el procedimiento de la figura 5.5, ¿depende alguna vez el árbol final del orden en que se eliminaron?
- 5.5 Se desea tener información de todas las subcadenas de cinco caracteres que ocurren dentro de una cadena dada por medio de un trie. Muéstrese el trie que resulta de insertar las 14 subcadenas de longitud 5 de la cadena ABC-DABACDEBACADEBA.
- *5.6 Para realizar el ejercicio 5.5, se podría poner un apuntador en cada hoja, lo cual representaría la cadena $abcde$, al nodo interior que representa el sufijo $bcd e$. De esa manera, si se recibe el siguiente símbolo, por ejemplo F , no hay que insertar todo $bcd e f$, partiendo de la raíz. Más aún, habiendo visto $abcde$, es posible crear también nodos para $bcd e$, cde , de , y e , ya que, a menos que la secuencia concluya abruptamente, más tarde se necesitarán esos nodos. Modifíquense la estructura de datos del trie para colocar esos apun-

tadores, y el algoritmo de inserción en un trie, para aprovechar esta estructura de datos.

- 5.7 Muéstrese el árbol 2-3 que resulta de insertar los elementos 5, 2, 7, 0, 3, 4, 6, 1, 8, 9 en un conjunto vacío, representado como un árbol 2-3.
- 5.8 Muéstrese el resultado obtenido de eliminar el 3 del árbol 2-3 del ejercicio 5.7.
- 5.9 Muéstrense los valores sucesivos de las S_i al aplicar el algoritmo SCL de la figura 5.29 con la primera cadena *abacabada*, y la segunda *bdbacbad*.
- 5.10 Supóngase que se emplean árboles 2-3 para aplicar las operaciones COMBINA y DIVIDE de la sección 5.8.
 - a) Muéstrese el resultado de dividir el árbol del ejercicio 5.7 en el elemento 6.
 - b) Combíñese el árbol del ejercicio 5.7 con el árbol que consiste en hojas para los elementos 10 y 11.
- 5.11 Algunas estructuras estudiadas en este capítulo pueden modificarse fácilmente para manejar el TDA CORRESPONDENCIA. Escribanse los procedimientos ANULA, ASIGNA y CALCULA para operar sobre las siguientes estructuras de datos.
 - a) Árboles binarios de búsqueda. El ordenamiento «<» se aplica a los elementos del dominio.
 - b) Árboles 2-3. En los nodos interiores, colóquese sólo el campo clave de los elementos del dominio.
- 5.12 Demuéstrese que en cualquier subárbol de un árbol binario de búsqueda, el elemento mínimo es un nodo sin hijo izquierdo.
- 5.13 Empléese el ejercicio 5.12 para producir una versión no recursiva de SUPRIME-MIN.
- 5.14 Escribanse los procedimientos ASIGNA, VALOR-DE, ANULA, y TOMA-NUEVO para nodos de tries representados como listas de celdas.
- *5.15 ¿Cómo se comparan el trie (con la realización de lista de celdas), la tabla de dispersión abierta y el árbol binario de búsqueda en cuanto a velocidad y utilización de espacio cuando los elementos son cadenas de hasta diez caracteres?
- *5.16 Si los elementos de un conjunto se ordenan de acuerdo con la relación «<», se pueden guardar uno o dos elementos (no sólo sus claves) en los nodos internos de un árbol 2-3, sin tener que guardarlos en las hojas. Escribanse los procedimientos INSERTA y SUPRIME para árboles 2-3 de este tipo.
- 5.17 Otra modificación que se podría hacer a los árboles 2-3 sería guardar sólo claves en nodos internos, sin requerir que las claves k_1 y k_2 de un nodo sean

en realidad las claves mínimas del segundo y tercer subárboles, sino sólo que todas las claves k del tercer subárbol satisfagan $k \geq k_2$, todas las claves del segundo satisfagan $k_1 \leq k < k_2$, y todas las claves k del primero satisfagan $k < k_1$.

- a) ¿Cómo se simplifica SUPRIME con esta convención?
- b) ¿Qué operaciones de diccionarios o de correspondencias se hacen más complicadas o menos eficientes?

- *5.18 Otra estructura de datos que maneja diccionarios con la operación MIN es el *árbol AVL* (nombre debido a las iniciales de sus inventores) o *árbol balanceado por altura*. Son árboles binarios de búsqueda en los cuales no se permite que las alturas de dos hermanos difieran en más de uno. Escribanse los procedimientos INSERTA y SUPRIME respetando la propiedad de árbol AVL.
- 5.19 Escríbase un programa en Pascal para el procedimiento *inserta1* de la figura 5.21.
- *5.20 Un *autómata finito* consiste en un conjunto de estados, los cuales se tomarán como los enteros $1..n$, y una tabla *transiciones* [*estado, entrada*] que da el *siguiente estado* para cada *estado* y cada carácter de *entrada*. En este caso, se supondrá que la entrada es 0 ó 1; más aún, ciertos estados se designan como *estados de aceptación*. También se supondrá que todos, y únicamente los estados con numeración par son de aceptación. Dos estados p y q son *equivalentes* si son el mismo o a) ambos son de aceptación o ambos son de no aceptación, b) con la entrada 0 se transfieren a estados equivalentes, y c) con la entrada 1 se transfieren a estados equivalentes. Intuitivamente, los estados equivalentes se comportan igual en todas las secuencias de entrada. Escríbase un programa que use las operaciones de CONJUNTO_CE y que calcule los conjuntos de estados equivalentes de un autómata finito dado.
- **5.21 En la realización con árboles de CONJUNTO_CE:
- a) Demuéstrese que se necesita un tiempo $\Omega(n\log n)$ para ciertas listas de n operaciones si se usa la compresión de caminos, pero se permite la combinación de árboles grandes con otros más pequeños.
 - b) Demuéstrese que $O(na(n))$ es el tiempo de ejecución en el peor caso para n operaciones si se usa compresión de caminos, y se combina siempre el árbol más pequeño con el más grande.
- 5.22 Seleccionese una estructura de datos y escríbase un programa para calcular LUGARES (tal como se definió en la Sec. 5.6) en un tiempo promedio $O(n)$ para cadenas de longitud n .
- *5.23 Modifíquese el procedimiento SCL de la figura 5.29 para obtener la SCL, no sólo su longitud.

- *5.24 Escribase en forma detallada un procedimiento DIVIDE para trabajar con árboles 2-3.
- *5.25 Si los elementos de un conjunto representado por medio de un árbol 2-3 constaran sólo de un campo clave, un elemento cuya clave apareciera en un nodo interno no necesitaría estar en una hoja. Escribanse otra vez las operaciones de diccionario para aprovechar este hecho y evitar el almacenamiento de un elemento en dos nodos diferentes.

Notas bibliográficas

Los tries fueron propuestos por primera vez por Fredkin [1960]. Bayer y McCreight [1972] introdujeron los árboles B que, como se analizará en el capítulo 11, son una generalización de los árboles 2-3. El primer uso que se dio a los árboles 2-3 se debió a J. E. Hopcroft en 1970 (no publicado) para inserción, eliminación, concatenación y división, y a Ullman [1974] para un problema de optimización de código.

La estructura del árbol de la sección 5.5, que emplea compresión de caminos y combinación del menor con el mayor, fue utilizada primero por M. D. McIlroy y R. Morris para construir árboles abarcadores de costo mínimo. El rendimiento de la realización con árboles de los CONJUNTO_CE fue analizada por Fischer [1972] y por Hopcroft y Ullman [1973]. El ejercicio 5.21 es de Tarjan [1974].

La solución al problema de la SCL de la sección 5.6 es de Hunt y Szymanski [1975]. Una estructura de datos eficiente para ENCUENTRA, DIVIDE y COMBINA restringido (donde los elementos de un conjunto son menores que los del otro) se describe en Van Emde Boas, Kaas y Zijlstra [1975].

El ejercicio 5.6 está basado en un algoritmo eficiente para comparación de patrones desarrollado por Weiner [1973]. La variación del árbol 2-3 del ejercicio 5.16 se comenta con detalle en Wirth [1976]. La estructura de árbol AVL del ejercicio 5.18 es de Adel'son-Vel'skii y Landis [1962].

6

Grafos dirigidos

En los problemas originados en ciencias de la computación, matemáticas, ingeniería y muchas otras disciplinas, a menudo es necesario representar relaciones arbitrarias entre objetos de datos. Los grafos dirigidos y los no dirigidos son modelos naturales de tales relaciones. Este capítulo presenta las estructuras de datos básicas que pueden usarse para representar grafos dirigidos. También se presentan algunos algoritmos básicos para la determinación de conectividad en grafos dirigidos y para encontrar los caminos más cortos.

6.1 Definiciones fundamentales

Un *grafo dirigido* G consiste en un conjunto de vértices V y un conjunto de arcos A . Los vértices se denominan también *nodos* o *puntos*; los arcos pueden llamarse *arcos dirigidos* o *líneas dirigidas*. Un arco es un par ordenado de vértices (v, w) ; v es la *cola* y w la *cabeza* del arco. El arco (v, w) se expresa a menudo como $v \rightarrow w$ y se representa como



Obsérvese que la «punta de la flecha» está en el vértice llamado «cabeza», y la cola, en el vértice llamado «cola». Se dice que el arco $v \rightarrow w$ va de v a w , y que w es *adyacente* a v .

Ejemplo 6.1. La figura 6.1 muestra un grafo dirigido con cuatro vértices y cinco arcos. □

Los vértices de un grafo dirigido pueden usarse para representar objetos, y los arcos, relaciones entre los objetos. Por ejemplo, los vértices pueden representar ciudades, y los arcos, vuelos aéreos de una ciudad a otra. En otro ejemplo, como el que se presentó en la sección 4.2, un grafo dirigido puede emplearse para representar el flujo de control en un programa de computador. Los vértices representan bloques básicos, y los arcos, posibles transferencias del flujo de control.

Un *camino* en un grafo dirigido es una secuencia de vértices v_1, v_2, \dots, v_n , tal que $v_1 \rightarrow v_2, v_2 \rightarrow v_3, \dots, v_{n-1} \rightarrow v_n$ son arcos. Este camino va del vértice v_1 al vértice v_n .

pasa por los vértices v_2, v_3, \dots, v_{n-1} , y termina en el vértice v_n . La *longitud* de un camino es el número de arcos en ese camino, en este caso, $n - 1$. Como caso especial, un vértice sencillo, v , por sí mismo denota un camino de longitud cero de v a v . En la figura 6.1, la secuencia 1, 2, 4 es un camino de longitud 2 que va del vértice 1 al vértice 4.

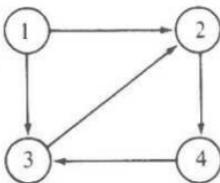


Fig. 6.1. Grafo dirigido.

Un camino es *simple* si todos sus vértices, excepto tal vez el primero y el último, son distintos. Un *ciclo simple* es un camino simple de longitud por lo menos uno, que empieza y termina en el mismo vértice. En la figura 6.1, el camino 3, 2, 4, 3 es un ciclo de longitud 3.

En muchas aplicaciones es útil asociar información a los vértices y arcos de un grafo dirigido. Para este propósito es posible usar un *grafo dirigido etiquetado*, en el cual cada arco, cada vértice o ambos pueden tener una etiqueta asociada. Una etiqueta puede ser un nombre, un costo o un valor de cualquier tipo de datos dado.

Ejemplo 6.2. La figura 6.2 muestra un grafo dirigido etiquetado en el que cada arco está etiquetado con una letra que causa una transición de un vértice a otro. Este grafo dirigido etiquetado tiene la interesante propiedad de que las etiquetas de los arcos de cualquier ciclo que sale del vértice 1 y vuelve a él producen una cadena de caminos a y b en la cual los números de a y de b son pares. \square

En un grafo dirigido etiquetado, un vértice puede tener a la vez un nombre y una etiqueta. A menudo se empleará la etiqueta del vértice como si fuera el nombre. Así, los números de la figura 6.2 pueden interpretarse como nombres o como etiquetas de vértices.

6.2 Representaciones de grafos dirigidos

Para representar un grafo dirigido se pueden emplear varias estructuras de datos; la selección apropiada depende de las operaciones que se aplicarán a los vértices y a los arcos del grafo. Una representación común para un grafo dirigido $G = (V, A)$ es la *matriz de adyacencia*. Supóngase que $V = \{1, 2, \dots, n\}$. La matriz de adyacencia para G es una matriz A de dimensión $n \times n$, de elementos booleanos, donde $A[i, j]$ es verdadero si, y sólo si, existe un arco que vaya del vértice i al j . Con frecuencia se exhibirán matrices de adyacencias con 1 para verdadero y 0 para falso; las matrices de adyacencias pueden incluso obtenerse de esa forma. En la representación con una

matriz de adyacencia, el tiempo de acceso requerido a un elemento es independiente del tamaño de V y A . Así, la representación con matriz de adyacencia es útil en los algoritmos para grafos, en los cuales suele ser necesario saber si un arco dado está presente.

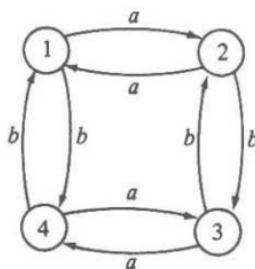


Fig. 6.2. Grafo dirigido de transiciones.

Algo muy relacionado con esto es la representación con *matriz de adyacencia etiquetada* de un grafo dirigido, donde $A[i, j]$ es la etiqueta del arco que va del vértice i al vértice j . Si no existe un arco de i a j , debe emplearse como entrada para $A[i, j]$ un valor que no pueda ser una etiqueta válida.

Ejemplo 6.3 La figura 6.3. muestra la matriz de adyacencia etiquetada para el grafo dirigido de la figura 6.2. Aquí, el tipo de la etiqueta es un carácter, y un espacio representa la ausencia de un arco. □

	1	2	3	4
1		<i>a</i>	<i>b</i>	
2	<i>a</i>		<i>b</i>	
3		<i>b</i>		<i>a</i>
4	<i>b</i>		<i>a</i>	

Fig. 6.3. Matriz de adyacencia etiquetada para el grafo dirigido de la figura 6.2.

La principal desventaja de usar una matriz de adyacencia para representar un grafo dirigido es que requiere un espacio $\Omega(n^2)$ aun si el grafo dirigido tiene menos de n^2 arcos. Sólo leer o examinar la matriz puede llevar un tiempo $O(n^2)$, lo cual invalidaría los algoritmos $O(n)$ para la manipulación de grafos dirigidos con $O(n)$ arcos.

Para evitar esta desventaja, se puede utilizar otra representación común para un grafo dirigido $G = (V, A)$ llamada representación con *lista de adyacencia*. La lista de adyacencia para un vértice i es una lista, en algún orden, de todos los vértices adyacentes a i . Se puede representar G por medio de un arreglo *CABEZA*, donde *CABEZA*[i] es un apuntador a la lista de adyacencia del vértice i . La representación con lista de adyacencia de un grafo dirigido requiere un espacio proporcional a la suma del número de vértices más el número de arcos; se usa bastante cuando

el número de arcos es mucho menor que n^2 . Sin embargo, una desventaja potencial de la representación con lista de adyacencia es que puede llevar un tiempo $O(n)$ determinar si existe un arco del vértice i al vértice j , ya que puede haber $O(n)$ vértices en la lista de adyacencia para el vértice i .

Ejemplo 6.4. La figura 6.4 muestra una representación con lista de adyacencia para el grafo dirigido de la figura 6.1, donde se usan listas enlazadas sencillas. Si los arcos tienen etiquetas, éstas podrían incluirse en las celdas de la lista ligada.

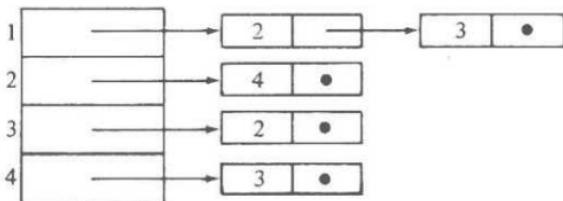


Fig. 6.4. Representación con lista de adyacencia para el grafo dirigido de la figura 6.1

Si hubo inserciones y supresiones en las listas de adyacencias, sería preferible tener el arreglo *CABEZA* apuntando a celdas de encabezamiento que no contienen vértices adyacentes †. Por otra parte, si se espera que el grafo permanezca fijo, sin cambios (o con muy pocos) en las listas de adyacencias, sería preferible que *CABEZA*[i] fuera un cursor a un arreglo *ADY*, donde *ADY*[*CABEZA*[i]], *ADY*[*CABEZA*[i] + 1], ..., y así sucesivamente, contuvieran los vértices adyacentes al vértice i , hasta el punto en *ADY* donde se encuentra por primera vez un cero, el cual marca el fin de la lista de vértices adyacentes a i . Por ejemplo, la figura 6.1 puede representarse con la figura 6.5. □

TDA grafo dirigido

Se podría definir un TDA que correspondiera formalmente al grafo dirigido y estudiar las implantaciones de sus operaciones. No se redundará demasiado en esto, porque hay poco material en verdad nuevo y las principales estructuras de datos para grafos ya han sido cubiertas. Las operaciones más comunes en grafos dirigidos incluyen la lectura de la etiqueta de un vértice o un arco, la inserción o supresión de vértices y arcos, y el recorrido de arcos desde la cola hasta la cabeza.

Las últimas operaciones requieren más cuidado. Con frecuencia, se encuentran en proposiciones informales de programas como

for cada vértice w adyacente al vértice v do
 {algunha acción sobre w } (6.1)

† Esta es otra manifestación del viejo problema de Pascal de hacer inserción y supresión en posiciones arbitrarias de listas enlazadas sencillas.

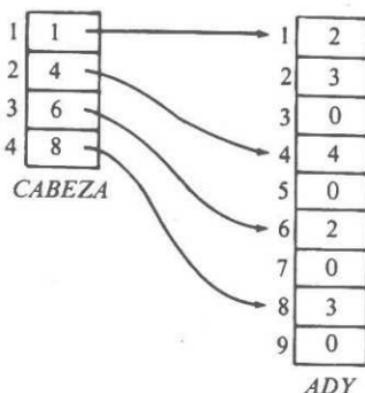


Fig. 6.5. Otra representación con lista de adyacencia de la figura 6.1.

Para obtener esto, es necesaria la noción de un tipo *índice* para el conjunto de vértices adyacentes a algún vértice v . Por ejemplo, si las listas de adyacencias se usan para representar el grafo, un índice es, en realidad, una posición en la lista de adyacencia de v . Si se usa una matriz de adyacencia, un índice es un entero que representa un vértice adyacente. Se requieren las tres operaciones siguientes en grafos dirigidos.

1. PRIMERO(v), que devuelve el índice del primer vértice adyacente a v . Se devuelve un vértice nulo Λ si no existe ningún vértice adyacente a v .
2. SIGUIENTE(v, i), que devuelve el índice posterior al índice i de los vértices adyacentes a v . Se devuelve Λ si i es el último vértice de los vértices adyacentes a v .
3. VERTICE(v, i), que devuelve el vértice cuyo índice i está entre los vértices adyacentes a v .

Ejemplo 6.5. Si se escoge la representación con matriz de adyacencia, VERTICE(v, i) devuelve i . PRIMERO(v) y SIGUIENTE(v, i) pueden escribirse como en la figura 6.6, para operar en una matriz booleana A de $n \times n$ definida de manera externa. Se supone que A está declarada como

```
array [1..n, 1..n] of boolean
```

y que 0 se emplea para Λ . Después, se obtiene la proposición (6.1) como en la figura 6.7. \square

```
function PRIMERO ( v: integer ) : integer;
var
  i: integer;
```

```

begin
    for  $i := 1$  to  $n$  do
        if  $A[v, i]$  then
            return( $i$ );
    return (0) { si se llega aquí,  $v$  no tiene vértices adyacentes }
end; { PRIMERO }

function SIGUIENTE (  $v$ : integer;  $i$ : integer ) : integer;
var
     $j$  : integer;
begin
    for  $j := i+1$  to  $n$  do
        if  $A[v, j]$  then
            return ( $j$ );
    return (0)
end; { SIGUIENTE }

```

Fig. 6.6. Operaciones para recorrer vértices adyacentes.

```

 $i := \text{PRIMERO}(v);$ 
while  $i <> \wedge$  do begin
     $w := \text{VERTICE}(v, i);$ 
    { alguna acción en  $w$  }
     $i := \text{SIGUIENTE}(v, i)$ 
end

```

Fig. 6.7. Iteración en vértices adyacentes a v .

6.3 Problema de los caminos más cortos con un solo origen

En esta sección se considera un problema común de búsqueda de caminos en grafos dirigidos. Supóngase un grafo dirigido $G = (V, A)$ en el cual cada arco tiene una etiqueta no negativa, y donde un vértice se especifica como *origen*. El problema es determinar el costo del camino más corto del origen a todos los demás vértices de V , donde la *longitud de un camino* es la suma de los costos de los arcos del camino. Esto se conoce con el nombre de problema de *los caminos más cortos con un solo origen* †. Obsérvese que se hablará de caminos con «longitud» aun cuando los costos representan algo diferente, como tiempo.

Sea G un mapa de vuelos en el cual cada vértice representa una ciudad, y cada

† Se puede esperar que un problema más natural sea encontrar el camino más corto entre el origen y un vértice *destino* particular. Sin embargo, ese problema parece tan difícil en general como el de los caminos más cortos con un solo origen (a menos que se tenga la suerte de encontrar el camino al destino antes que alguno de los otros vértices y así terminar el algoritmo un poco antes que si se buscaran los caminos hacia todos los vértices).

arco $v \rightarrow w$, una ruta aérea de la ciudad v a la ciudad w . La etiqueta del arco $v \rightarrow w$ es el tiempo que se requiere para volar de v a w [†]. La solución del problema de los caminos más cortos con un solo origen para este grafo dirigido determinaría el tiempo de viaje mínimo para ir de cierta ciudad a todas las demás del mapa.

Para resolver este problema se manejará una técnica «ávida» conocida como *algoritmo de Dijkstra*, que opera a partir de un conjunto S de vértices cuya distancia más corta desde el origen ya es conocida. En principio, S contiene sólo el vértice de origen. En cada paso, se agrega algún vértice restante v a S , cuya distancia desde el origen es la más corta posible. Suponiendo que todos los arcos tienen costo no negativo, siempre es posible encontrar un camino más corto entre el origen y v que pasa sólo a través de los vértices de S , y que se llama *especial*. En cada paso del algoritmo, se utiliza un arreglo D para registrar la longitud del camino especial más corto a cada vértice. Una vez que S incluye todos los vértices, todos los caminos son «especiales», así que D contendrá la distancia más corta del origen a cada vértice.

El algoritmo se da en la figura 6.8, donde se supone que existe un grafo dirigido $G = (V, A)$ en el que $V = \{1, 2, \dots, n\}$ y el vértice 1 es el origen. C es un arreglo bidimensional de costos, donde $C[i, j]$ es el costo de ir del vértice i al vértice j por el arco $i \rightarrow j$. Si no existe el arco $i \rightarrow j$, se supone que $C[i, j] = \infty$, un valor mucho mayor que cualquier costo real. En cada paso, $D[i]$ contiene la longitud del camino especial más corto actual para el vértice i .

Ejemplo 6.6. Aplíquese *Dijkstra* al grafo dirigido de la figura 6.9. En principio, $S = \{1\}$, $D[2] = 10$, $D[3] = \infty$, $D[4] = 30$ y $D[5] = 100$. En la primera iteración del ciclo **for** de las líneas (4) a (8), $w = 2$ se selecciona como el vértice con el mínimo valor D . Después se hace $D[3] = \min(\infty, 10 + 50) = 60$. $D[4]$ y $D[5]$ no cambian porque el camino para llegar a ellos directamente desde 1 es más corto que pasar por el vértice 2. La secuencia de valores D después de cada iteración se muestra en la figura. 6.10. □

Para reconstruir el camino más corto del origen a cada vértice, se agrega otro arreglo P de vértices, tal que $P[v]$ contenga el vértice inmediato anterior a v en el camino más corto. Se asigna $P[v]$ valor inicial 1 para toda $v \neq 1$. El arreglo P puede actualizarse después de la línea (8) de *Dijkstra*. Si $D[w] + C[w, v] < D[v]$ en la línea (8), después se hace $P[v] := w$. Al término de *Dijkstra*, el camino a cada vértice puede encontrarse regresando por los vértices predecesores del arreglo P .

```

procedure Dijkstra;
    { Dijkstra calcula el costo de los caminos más cortos entre el vértice 1
      y todos los demás de un grafo dirigido }
begin
(1)      S := { 1 };
(2)      for i := 2 to n do

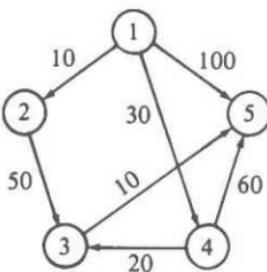
```

[†] Cabría suponer que puede usarse un grafo no dirigido, ya que la etiqueta de los arcos $v \rightarrow w$ y $w \rightarrow v$ sería la misma. Sin embargo, los tiempos de viaje son diferentes en direcciones diferentes, debido a los vientos. De cualquier forma, aunque las etiquetas $v \rightarrow w$ y $w \rightarrow v$ fueran idénticas, esto no ayudaría a resolver el problema.

```

(3)       $D[i] := C[1, i]; \{ \text{asigna valor inicial a } D \}$ 
(4)      for  $i := 1$  to  $n-1$  do begin
(5)          elige un vértice  $w$  en  $V-S$  tal que  $D[w]$  sea un mínimo;
(6)          agrega  $w$  a  $S$ ;
(7)          for cada vértice  $v$  en  $V-S$  do
(8)               $D[v] := \min(D[v], D[w] + C[w, v])$ 
end
end;  $\{ \text{Dijkstra} \}$ 

```

Fig. 6.8. Algoritmo de Dijkstra.**Fig. 6.9.** Grafo dirigido con arcos etiquetados.

Ejemplo 6.7. Para el grafo dirigido del ejemplo 6.6, el arreglo P debe tener los valores $P[2] = 1$, $P[3] = 4$, $P[4] = 1$ y $P[5] = 3$. Para encontrar el camino más corto del vértice 1 al vértice 5, por ejemplo, se siguen los predecesores en orden inverso comenzando en 5. A partir del arreglo P , se determina que 3 es el predecesor de 5, 4 el predecesor de 3 y 1 el predecesor de 4. Así, el camino más corto entre los vértices 1 y 5 es 1, 4, 3, 5. \square

Iteración	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
inicial	{1}	—	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Fig. 6.10. Cálculos de Dijkstra en el grafo dirigido de la figura 6.9.

Por qué funciona el algoritmo de Dijkstra

El algoritmo de Dijkstra es un ejemplo donde la «avidez» funciona, en el sentido de que lo que aparece localmente como lo mejor, se convierte en lo mejor de todo. En este caso, lo «mejor» localmente es encontrar la distancia al vértice w que

está fuera de S , pero tiene el camino especial más corto. Para ver por qué en este caso no puede haber un camino no especial más corto desde el origen hasta w , obsérvese la figura 6.11, en la que se muestra un camino hipotético más corto a w que primero sale de S para ir al vértice x , después (tal vez) entra y sale de S varias veces antes de llegar finalmente a w .

Pero si este camino es más corto que el camino especial más corto a w , el segmento inicial del camino entre el origen y x es un camino especial a x más corto que el camino especial más corto a w . (Obsérvese lo importante que es el hecho de que los costos no sean negativos; sin ello, este argumento no sería válido, y de hecho, el algoritmo de Dijkstra no funcionaría correctamente.) En este caso, se debió seleccionar x en vez de w en la línea (5) de la figura 6.8, porque $D[x]$ fue menor que $D[w]$.

Para completar la demostración de que la figura 6.8 funciona, se verifica que en todos los casos $D[v]$ es realmente la distancia más corta de un camino especial al vértice v . La clave de este razonamiento está en observar que al agregar un nuevo vértice w a S en la línea (6), las líneas (7) y (8) ajustan D para tener en cuenta la posibilidad de que exista ahora un camino especial más corto a v a través de w . Si ese camino va a través del anterior S a w , e inmediatamente después a v , su costo, $D[w] + C[w, v]$, será comparado con $D[v]$ en la línea (8), y $D[v]$ se reducirá si el nuevo camino especial es más corto. La otra posibilidad de un camino especial más corto se muestra en la figura 6.12, donde el camino va a w , después regresa al anterior S , a algún miembro x del S anterior, y luego a v .

Pero en realidad no puede existir tal camino; puesto que x se colocó en S antes que w , el más corto de todos los caminos entre el origen y x pasa sólo a través del S anterior. Por tanto, el camino hacia x a través de w , mostrado en la figura 6.12, no es más corto que el camino que va directo a x a través de S . Como resultado, la longitud del camino de la figura 6.12 entre el origen y w , x , y v no es menor que el anterior valor de $D[v]$, ya que $D[v]$ no fue mayor que la longitud del camino más corto hasta x a través de S y después directamente a w . Así, $D[v]$ no puede reducirse en la línea (8) por medio de un camino que pase por w y x , como el de la figura 6.12, y no es necesario considerar la longitud de esos caminos.

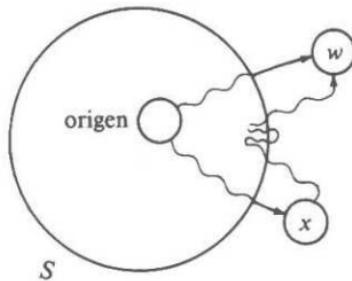


Fig. 6.11. Camino hipotético más corto a w .

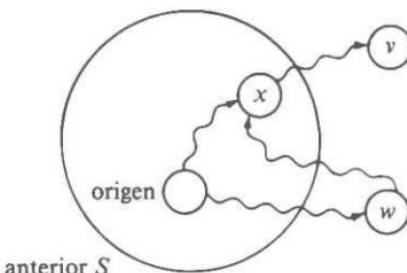


Fig. 6.12. Camino especial más corto imposible.

Tiempo de ejecución del algoritmo de Dijkstra

Supóngase que la figura 6.8 opera en un grafo dirigido con n vértices y a aristas. Si se emplea una matriz de adyacencia para representar el grafo dirigido, el ciclo de las líneas (7) y (8) lleva un tiempo $O(n)$, y se ejecuta $n - 1$ veces para un tiempo total de $O(n^2)$. El resto del algoritmo, como se puede observar, no requiere más tiempo que esto.

Si a es mucho menor que n^2 , puede ser mejor utilizar una representación con lista de adyacencia del grafo dirigido y emplear una cola de prioridad obtenida a manera de árbol parcialmente ordenado para organizar los vértices de $V - S$. El ciclo de las líneas (7) y (8) se puede realizar recorriendo la lista de adyacencia para w y actualizando las distancias en la cola de prioridad. Se hará un total de a actualizaciones, cada una con un costo de tiempo $O(\log n)$, por lo que el tiempo total consumido en las líneas (7) y (8) es ahora $O(a \log n)$, en vez de $O(n^2)$.

Las líneas (1) a (3) llevan un tiempo $O(n)$, al igual que las líneas (4) y (6). Al manejar la cola de prioridad para representar $V - S$, las líneas (5) y (6) implantan exactamente la operación SUPRIME-MIN y cada una de las $n - 1$ iteraciones de esas líneas requiere un tiempo $O(\log n)$.

Como resultado, el tiempo total consumido en esta versión del algoritmo de Dijkstra está acotado por $O(a \log n)$. Este tiempo de ejecución es mucho mejor que $O(n^2)$ si a es muy pequeña, comparada con n^2 .

6.4 Problema de los caminos más cortos entre todos los pares

Supóngase que se tiene un grafo dirigido etiquetado que da el tiempo de vuelo para ciertas rutas entre ciudades, y se desea construir una tabla que brinde el menor tiempo requerido para volar entre dos ciudades cualesquiera. Este es un ejemplo del problema de los caminos más cortos entre todos los pares (CMCP). Para plantear el problema con precisión, se emplea un grafo dirigido $G = (V, A)$ en el cual cada arco $v \rightarrow w$ tiene un costo no negativo $C[v, w]$. El problema CMCP es encontrar el camino de longitud más corta entre v y w para cada par ordenado de vértices (v, w) .

Podría resolverse este problema por medio del algoritmo de Dijkstra, tomando por turno cada vértice como vértice origen, pero una forma más directa de solución es mediante el algoritmo creado por R. W. Floyd. Por conveniencia, se supone otra vez que los vértices en v están numerados 1, 2, ..., n . El algoritmo de Floyd usa una matriz A de $n \times n$ en la que se calculan las longitudes de los caminos más cortos. Inicialmente se hace $A[i, j] = C[i, j]$ para toda $i \neq j$. Si no existe un arco que vaya de i a j , se supone que $C[i, j] = \infty$. Cada elemento de la diagonal se hace 0.

Después, se hacen n iteraciones en la matriz A . Al final de la k -ésima iteración, $A[i, j]$ tendrá por valor la longitud más pequeña de cualquier camino que vaya desde el vértice i hasta el vértice j y que no pase por un vértice con número mayor que k . Esto es, i y j , los vértices extremos del camino, pueden ser cualquier vértice, pero todo vértice intermedio debe ser menor o igual que k .

En la k -ésima iteración se aplica la siguiente fórmula para calcular A .

$$A_k[i, j] = \min \begin{cases} A_{k-1}[i, j] \\ A_{k-1}[i, k] + A_{k-1}[k, j] \end{cases}$$

El subíndice k denota el valor de la matriz A después de la k -ésima iteración; no indica la existencia de n matrices distintas. Pronto, se eliminarán esos subíndices. Esta fórmula tiene la interpretación simple de la figura 6.13.

Para obtener $A_k[i, j]$, se compara $A_{k-1}[i, j]$, el costo de ir de i a j sin pasar por k o cualquier otro vértice con numeración mayor, con $A_{k-1}[i, k] + A_{k-1}[k, j]$, el costo de ir primero de i a k y después de k a j , sin pasar a través de un vértice con número mayor que k . Si el paso por el vértice k produce un camino más económico que el de $A_{k-1}[i, j]$, se elige ese costo para $A_k[i, j]$.

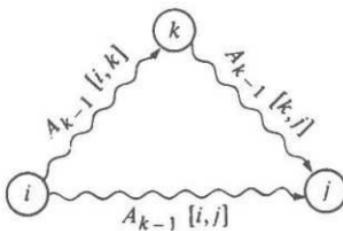


Fig. 6.13. Inclusión de k entre los vértices que van de i a j .

Ejemplo 6.8. Considérese el grafo dirigido ponderado que se muestra en la figura 6.14. En la figura 6.15 se muestran los valores iniciales de la matriz A , y después de tres iteraciones. \square

Como $A_k[i, k] = A_{k-1}[i, k]$ y $A_k[k, j] = A_{k-1}[k, j]$, ninguna entrada con cualquier subíndice igual a k cambia durante la k -ésima iteración. Por tanto, se puede realizar el cálculo sólo con una copia de la matriz A . En la figura 6.16 se muestra un programa para realizar este cálculo en matrices de $n \times n$.

Es evidente que el tiempo de ejecución de este programa es $O(n^3)$, ya que el programa está conformado por el triple ciclo anidado `for`. Para verificar que este progra-

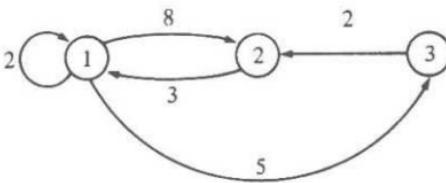


Fig. 6.14. Grafo dirigido ponderado.

ma funciona, es fácil demostrar por inducción sobre k que después de k recorridos por el triple ciclo **for**, $A[i, j]$ contiene la longitud del camino más corto desde el vértice i hasta el vértice j que no pasa a través de un vértice con número mayor que k .

	1	2	3		1	2	3	
1	0	8	5		2	3	0	8
2	3	0	∞		3	∞	2	0
3	∞	2	0					
	$A_0[i, j]$				$A_1[i, j]$			
1	0	8	5		1	0	7	5
2	3	0	8		2	3	0	8
3	5	2	0		3	5	2	0
	$A_2[i, j]$				$A_3[i, j]$			

Fig. 6.15. Valores de matrices A sucesivas.

```

procedure Floyd ( var A: array[1..n, 1..n] of real;
  C: array[1..n, 1..n] of real );
{ Floyd calcula la matriz A de caminos más cortos dada la matriz de costos
  de arcos C }
var
  i, j, k: integer;
begin
  for i := 1 to n do
    for j := 1 to n do
      A[i, j] := C[i, j];
  for i := 1 to n do
    A[i, i] := 0;
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if (A[i, k] + A[k, j]) < A[i, j] then
          A[i, j] := A[i, k] + A[k, j]
end; { Floyd }
  
```

Fig. 6.16. Algoritmo de Floyd.

Comparación entre los algoritmos de Floyd y Dijkstra

Dado que la versión de *Dijkstra* con matriz de adyacencia puede encontrar los caminos más cortos desde un vértice en un tiempo $O(n^2)$, como el algoritmo de *Floyd*, también puede encontrar todos los caminos más cortos en un tiempo $O(n^3)$. El compilador, la máquina y los detalles de realización determinarán las constantes de proporcionalidad. La experimentación y medición son la forma más fácil de descubrir el mejor algoritmo para la aplicación en cuestión.

Si a , el número de aristas, es mucho menor que n^2 , aun con el factor constante relativamente bajo en el tiempo de ejecución $O(n^3)$ de *Floyd*, cabe esperar que la versión de *Dijkstra* con lista de adyacencia, tomando un tiempo $O(na \log n)$ para resolver el CMCP, sea superior, al menos para grafos grandes y poco densos.

Recuperación de los caminos

En muchos casos se desea imprimir el camino más económico entre dos vértices. Un modo de lograrlo es usando otra matriz P , donde $P[i, j]$ tiene el vértice k que permitió a *Floyd* encontrar el valor más pequeño de $A[i, j]$. Si $P[i, j] = 0$, el camino más corto de i a j es directo, siguiendo el arco entre ambos. La versión modificada de *Floyd* de la figura 6.17 almacena los vértices intermedios apropiados en P .

```

procedure más_corto ( var A: array[1..n, 1..n] of real;
  C: array[1..n, 1..n] of real; P: array[1..n, 1..n] of integer );
{ más_corto toma una matriz de costos de arcos C de n×n y produce una matriz A
  de n × n de longitudes de caminos más cortos y una matriz P de n × n
  que da un punto en la «mitad» de cada camino más corto }

var
  i, j, k: integer;
begin
  for i := 1 to n do
    for j := 1 to n do begin
      A[i, j] := C[i, j];
      P[i, j] := 0
    end;
  for i := 1 to n do
    A[i, i] := 0;
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if A[i, k] + A[k, j] < A[i, j] then begin
          A[i, j] := A[i, k] + A[k, j];
          P[i, j] := k
        end
    end; { más_corto }
end;

```

Fig. 6.17. Programa para los caminos más cortos.

Para imprimir los vértices intermedios del camino más corto del vértice i hasta el vértice j , se invoca el procedimiento $camino(i, j)$ dado en la figura 6.18. Mientras que en una matriz arbitraria P , $camino$ puede iterar infinitamente, si P viene del procedimiento $más_corto$, no es posible tener, por ejemplo, k en el camino más corto de i a j y también tener j en el camino más corto de i a k . Obsérvese cómo la suposición de pesos no negativos es crucial otra vez.

```

procedure camino ( i, j: integer );
var
  k: integer;
begin
  k := P[i, j];
  if k = 0 then
    return;
  writeln(k);
  camino(i, k);
  writeln(k);
  camino(k, j)
end; { camino }

```

Fig. 6.18. Procedimiento para imprimir el camino más corto.

Ejemplo 6.9. La figura 6.19 muestra la matriz P final para el grafo dirigido de la figura 6.14. □

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

P

Fig. 6.19. Matriz P para el grafo dirigido de la figura 6.14.

Cerradura transitiva

En algunos problemas podría ser interesante saber sólo si existe un camino de longitud igual o mayor que uno que vaya desde el vértice i al vértice j . El algoritmo de Floyd puede especializarse para este problema; el algoritmo resultante, que antecede al de Floyd, se conoce como algoritmo de Warshall.

Supóngase que la matriz de costo C es sólo la matriz de adyacencia para el grafo dirigido dado. Esto es, $C[i, j] = 1$ si hay un arco de i a j , y 0 si no lo hay. Se desea obtener la matriz A tal que $A[i, j] = 1$ si hay un camino de longitud igual o mayor que uno de i a j , y 0 en otro caso. A se conoce a menudo como *cerradura transitiva* de la matriz de adyacencia.

Ejemplo 6.10. La figura 6.20 muestra la cerradura transitiva para la matriz de adyacencia del grafo dirigido de la figura 6.14. □

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Fig. 6.20. Cerradura transitiva.

La cerradura transitiva puede obtenerse con un procedimiento similar a *Floyd* aplicando la siguiente fórmula en el k -ésimo paso en la matriz booleana A .

$$A_k[i, j] = A_{k-1}[i, j] \vee (A_{k-1}[i, k] \text{ y } A_{k-1}[k, j])$$

Esta fórmula establece que hay un camino de i a j que no pasa por un vértice con número mayor que k si

1. ya existe un camino de i a j que no pasa por un vértice con número mayor que $k - 1$, o si
2. hay un camino de i a k que no pasa por un vértice con número mayor que $k - 1$, y un camino de k a j que no pasa por un vértice con número mayor que $k - 1$.

Igual que antes, $A_k[i, k] = A_{k-1}[i, k]$ y $A_k[k, j] = A_{k-1}[k, j]$, así que se puede realizar el cálculo con sólo una copia de la matriz A . El programa en Pascal resultante, llamado *Warshall* por su descubridor, se muestra en la figura 6.21.

```

procedure Warshall ( var A: array[1..n, 1..n] of boolean;
                     C: array[1..n, 1..n] of boolean );
  { Warshall convierte a A en la cerradura transitiva de C }
  var
    i, j, k: integer;
  begin
    for i := 1 to n do
      for j := 1 to n do
        A[i, j] := C[i, j];
    for k := 1 to n do
      for i := 1 to n do
        for j := 1 to n do
          if A[i, j] = false then
            A[i, j] := A[i, k] and A[k, j]
  end; { Warshall }

```

Fig. 6.21. Algoritmo de Warshall para cerradura transitiva.

Un ejemplo: localización del centro de un grafo dirigido

Supóngase que se desea determinar el vértice más central de un grafo dirigido. Este problema puede resolverse fácilmente con el algoritmo de Floyd. Primero, se hace

más preciso el término «vértice más central». Sea v un vértice de un grafo dirigido $G = (V, A)$. La *excentricidad* de v es

$$\max_{w \text{ en } V} \{\text{longitud mínima de un camino de } w \text{ a } v\}$$

El *centro* de G es un vértice de mínima excentricidad. Así, el centro de un grafo dirigido es un vértice más cercano al vértice más distante.

Ejemplo 6.11. Considérese el grafo dirigido ponderado de la figura 6.22.

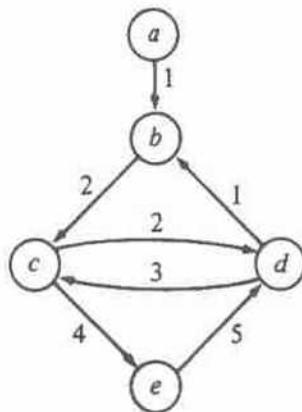


Fig. 6.22. Grafo dirigido ponderado.

Las excentricidades de los vértices son

vértice	excentricidad
a	∞
b	6
c	8
d	5
e	7

Por tanto, el centro es el vértice d . \square

Encontrar el centro de un grafo dirigido G es fácil. Supóngase que C es la matriz de costos para G .

1. Primero se aplica el procedimiento *Floyd* de la figura 6.16 a C para obtener la matriz A de los caminos más cortos entre todos los pares.
2. Se encuentra el costo máximo en cada columna i ; esto da la excentricidad del vértice i .
3. Se encuentra el vértice con excentricidad mínima; éste es el centro de G .

El tiempo de ejecución de este proceso está dominado por el primer paso, que lleva un tiempo $O(n^3)$. El paso (2) lleva $O(n^2)$ y el paso (3) lleva $O(n)$.

Ejemplo 6.12. La matriz de costo CMCP para la figura 6.22 se muestra en la figura 6.23. El valor máximo de cada columna se muestra a continuación.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>	0	1	3	5	7
<i>b</i>	∞	0	2	4	6
<i>c</i>	∞	3	0	2	4
<i>d</i>	∞	1	3	0	7
<i>e</i>	∞	6	8	5	0
máx	∞	6	8	5	7

Fig. 6.23. Matriz de costo CMCP.

6.5 Recorridos en grafos dirigidos

Para resolver con eficiencia muchos problemas relacionados con grafos dirigidos, es necesario visitar los vértices y los arcos de manera sistemática. La búsqueda en profundidad, una generalización del recorrido en orden previo de un árbol, es una técnica importante para lograrlo, y puede servir como estructura para construir otros algoritmos eficientes. Las dos últimas secciones de este capítulo contienen varios algoritmos que usan esta búsqueda como base.

Supóngase que se tiene un grafo dirigido G en el cual todos los vértices están marcados en principio como *no visitados*. La búsqueda en profundidad trabaja seleccionando un vértice v de G como vértice de partida; v se marca como *visitado*. Después, se recorre cada vértice adyacente a V no visitado, usando recursivamente la búsqueda en profundidad. Una vez que se han visitado todos los vértices que se pueden alcanzar desde v , la búsqueda de v está completa. Si algunos vértices quedan sin visitar, se selecciona alguno de ellos como nuevo vértice de partida, y se repite este proceso hasta que todos los vértices de G se hayan visitado.

Esta técnica se conoce como búsqueda en profundidad porque continúa buscando en la dirección hacia adelante (más profunda) mientras sea posible. Por ejemplo, supóngase que x es el vértice visitado más recientemente. La búsqueda en profundidad selecciona algún arco no explorado $x \rightarrow y$ que parte de x . Si se ha visitado y , el procedimiento intenta continuar por otro arco que no se haya explorado y que parte de x . Si y no se ha visitado, entonces el procedimiento marca y como visitado e inicia una nueva búsqueda a partir de y . Después de completar la búsqueda de todos los caminos que parten de y , la búsqueda regresa a x , el vértice desde el cual se visitó y por primera vez. Se continúa el proceso de selección de arcos sin explorar que parten de x hasta que todos los arcos de x han sido explorados.

Puede usarse una lista de adyacencia $L[v]$ para representar los vértices adyacentes al vértice v , y un arreglo *marca* cuyos elementos son del tipo (*visitado*, *no_visita-*

tado), puede usarse para determinar si un vértice ya fue visitado antes. El procedimiento recursivo bpf se describe en la figura 6.24. Para usarlo en un grafo con n vértices, se asigna el valor inicial *marca* a *no_visitado*, y después se comienza la búsqueda en profundidad con cada vértice que aún permanezca sin visitar cuando llegue su turno, con

```

for  $v := 1$  to  $n$  do
    marca[ $v$ ] := no_visitado;
for  $v := 1$  to  $n$  do
    if marca[ $v$ ] = no_visitado then
         $bpf(v)$ 
```

Obsérvese que la figura 6.24 es un modelo al cual se agregarán después otras acciones, al aplicar la búsqueda en profundidad. Lo único que hace el código de la figura 6.24 es actualizar el arreglo *marca*.

Análisis de la búsqueda en profundidad

Todas las llamadas a bpf en la búsqueda en profundidad de un grafo con a arcos y $n \leq a$ vértices lleva un tiempo $O(a)$. Para ver por qué, obsérvese que en ningún vértice se llama a bpf más de una vez, porque tan pronto como se llama a $bpf(v)$ se hace *marca*[v] igual a *visitado* en la línea (1), y nunca se llama a bpf en un vértice que antes tenía su *marca* igual a *visitado*. Así, el tiempo total consumido en las líneas (2) y (3) recorriendo las listas de adyacencias es proporcional a la suma de las longitudes de dichas listas, esto es, $O(a)$. De esta forma, suponiendo que $n \leq a$, el tiempo total consumido por la búsqueda en profundidad de un grafo completo es $O(a)$, lo cual es, hasta un factor constante, el tiempo necesario simplemente para recorrer cada arco.

```

procedure  $bpf$ (  $v$ : vértice );
var
     $w$ : vértice;
begin
(1)      marca[ $v$ ] := visitado;
(2)      for cada vértice  $w$  en  $L[v]$  do
(3)          if marca[ $w$ ] = no_visitado then
(4)               $bpf(w)$ 
end; |  $bpf$ |
```

Fig. 6.24. Búsqueda en profundidad.

Ejemplo 6.13. Supóngase que el procedimiento $bpf(v)$ se aplica al grafo dirigido de la figura 6.25 con $v = A$. El algoritmo marca *A* como visitado y selecciona el vértice *B* en las lista de adyacencia del vértice *A*. Puesto que *B* no se ha visitado, la búsqueda continúa llamando a $bpf(B)$. El algoritmo marca ahora *B* como visitado y selec-

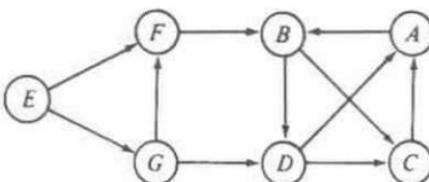


Fig. 6.25. Grafo dirigido.

ciona el primer vértice en la lista de adyacencia del vértice B . Dependiendo del orden de los vértices en la lista de adyacencia de B , la búsqueda seguirá con C o D .

Suponiendo que C aparece antes que D , se invoca a $bpf(C)$. El vértice A está en la lista de adyacencia de C . Sin embargo, en este momento ya se ha visitado A así que la búsqueda queda en C . Como ya se han visitado todos los vértices de la lista de adyacencia en C , la búsqueda regresa a B , desde donde la búsqueda prosigue a D . Los vértices A y C en la lista de adyacencia de D ya fueron visitados, por lo que la búsqueda regresa a B y después a A .

En este punto, la llamada original a $bpf(A)$ ha terminado. Sin embargo, el grafo dirigido no ha sido recorrido en su totalidad; los vértices E , F y G están sin visitar. Para completar la búsqueda, se puede llamar a $bpf(E)$.

Bosque abarcador en profundidad

Durante un recorrido en profundidad de un grafo dirigido, cuando se recorren ciertos arcos, llevan a vértices sin visitar. Los arcos que llevan a vértices nuevos se conocen como *arcos de árbol* y forman un *bosque abarcador en profundidad* para el grafo dirigido dado. Los arcos continuos de la figura 6.26 forman el bosque abarcador en profundidad del grafo dirigido de la figura 6.25. Obsérvese que los arcos de árbol deben formar realmente un bosque, ya que un vértice no puede estar sin visitar cuando se recorren dos arcos diferentes que llevan a él.

Además de los arcos de árbol, existen otros tres tipos de arcos definidos por una búsqueda en profundidad de un grafo dirigido, que se conocen como arcos de retroceso, arcos de avance y arcos cruzados. Un arco como $C \rightarrow A$ se denomina *arco de retroceso*, porque va de un vértice a uno de sus antecesores en el bosque abarcador. Obsérvese que un arco que va de un vértice hacia sí mismo, es un arco de retroceso. Un arco no abarcador que va de un vértice a un descendiente propio se llama *arco de avance*. En la figura 6.25 no hay arcos de este tipo.

Los arcos como $D \rightarrow C$ o $G \rightarrow D$, que van de un vértice a otro que no es antecesor ni descendiente, se conocen como *arcos cruzados*. Obsérvese que todos los arcos cruzados de la figura 6.26 van de derecha a izquierda, en el supuesto de que se agregan hijos al árbol en el orden en que fueron visitados, de izquierda a derecha, y que se agregan árboles nuevos al bosque de izquierda a derecha. Este patrón no es accidental. Si el arco $G \rightarrow D$ hubiera sido $D \rightarrow G$, entonces no se hubiera visitado G durante la búsqueda en D , y al encontrar el arco $D \rightarrow G$, el vértice G se haría descendiente de D , y $D \rightarrow G$ se convertiría en un arco de árbol.

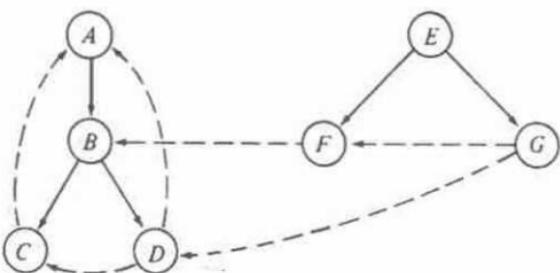


Fig. 6.26. Bosque abarcador en profundidad para la figura 6.25.

¿Cómo distinguir entre los cuatro tipos de arcos? Es obvio que los arcos de árbol son especiales, pues llevan a vértices sin visitar durante la búsqueda en profundidad. Supóngase que se numeran los vértices de un grafo dirigido de acuerdo con el orden en que se marcaron los visitados durante la búsqueda en profundidad. Esto es, se puede asignar a un arreglo.

```

númerop[v] := cont;
cont := cont + 1;

```

después de la línea (1) de la figura 6.24. A esto se le llama *numeración en profundidad* de un grafo dirigido; obsérvese que la numeración en profundidad generaliza la numeración en orden previo introducida en la sección 3.1.

La búsqueda en profundidad asigna a todos los descendientes de un vértice v , números mayores o iguales al número asignado a v . De hecho, w es un descendiente de v si, y sólo si, $númerop(v) \leq númerop(w) \leq númerop(v) + \text{el número de descendientes de } v$. Así, los arcos de avance van de los vértices de baja numeración a los de alta numeración y los arcos de retroceso van de los vértices de alta numeración a los de baja numeración.

Todos los arcos cruzados van de los vértices de alta numeración a los de baja numeración. Para ver esto, supóngase que $x \rightarrow y$ es un arco y $númerop(x) \leq númerop(y)$. Así, x se visita antes que y . Todo vértice visitado entre su invocación por primera vez a $bpf(x)$ y el momento en que $bpf(x)$ termina, se convierte en descendiente de x en el bosque abarcador en profundidad. Si y permanece sin visitar cuando se explora el arco $x \rightarrow y$, $x \rightarrow y$ se vuelve un arco de árbol. De otra forma, $x \rightarrow y$ es un arco de avance. Así, $x \rightarrow y$ no puede ser un arco cruzado con $númerop(x) \leq númerop(y)$.

En las dos secciones siguientes se analiza la forma de usar la búsqueda en profundidad para la solución de varios problemas de grafos.

6.6 Grafos dirigidos acíclicos

Un *grafo dirigido acíclico*, o *gda*, es un grafo dirigido sin ciclos. Cuantificados en función de las relaciones que representan, los *gda* son más generales que los árboles,

pero menos que los grafos dirigidos arbitrarios. La figura 6.27 muestra un ejemplo de un árbol, un gda, y un grafo dirigido con un ciclo.

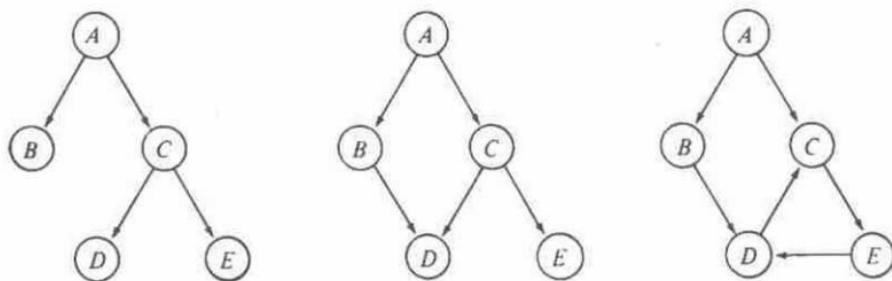


Fig. 6.27. Tres grafos dirigidos.

Entre otras cosas, los gda son útiles para la representación de la estructura sintáctica de expresiones aritméticas con subexpresiones comunes. Por ejemplo, la figura 6.28 muestra un gda para la expresión

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

Los términos $a + b$ y $(a + b) * c$ son subexpresiones comunes compartidas que se representan con vértices con más de un arco entrante.

Los gda son útiles también para la representación de órdenes parciales. Un orden parcial R en un conjunto S es una relación binaria tal que

1. para toda a en S , $a R a$ es falsa (R es irreflexivo), y
2. para toda a, b, c en S , si $a R b$ y $b R c$, entonces $a R c$ (R es transitivo).

Dos ejemplos naturales de órdenes parciales son la relación «menor que» ($<$) en enteros, y la relación de inclusión propia en conjuntos (\subset).

Ejemplo 6.14. Sea $S = \{1, 2, 3\}$ y sea $P(S)$ el conjunto exponencial de S , esto es, el conjunto de todos los subconjuntos de S . $P(S) = [\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}]$. \subset es un orden parcial en $P(S)$. Ciertamente, $A \subset A$ es falso para cualquier conjunto A (irreflexividad), y si $A \subset B$ y $B \subset C$, entonces $A \subset C$ (transitividad). \square

Los gda pueden usarse para reflejar gráficamente órdenes parciales. Para empezar, se puede considerar una relación R como un conjunto de pares (arcos) tales que (a, b) está en el conjunto si, y sólo si, $a R b$ es cierto. Si R es un orden parcial en el conjunto S , entonces el grafo dirigido $G = (S, R)$ es un gda. Del mismo modo, supóngase que $G = (S, R)$ es un gda y R^* es la relación definida por $a R^* b$ si, y sólo si, existe un camino de longitud uno o más que va de a a b . (R^* es la cerradura transitiva de la relación R .) Entonces, R^* es un orden parcial en S .

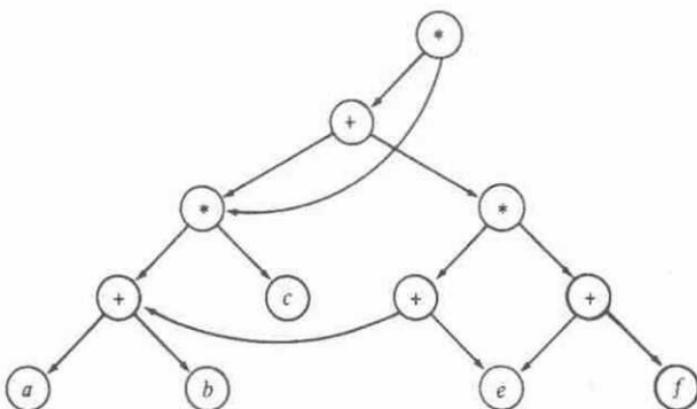


Fig. 6.28. Gda para expresiones aritméticas.

Ejemplo 6.15. La figura 6.29 muestra un gda $(P(S), R)$, donde $S = \{1, 2, 3\}$. La relación R^* es la inclusión propia en el conjunto exponencial $P(S)$. \square

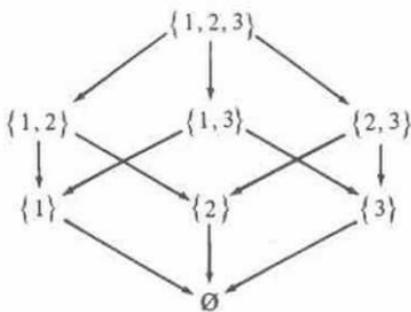


Fig. 6.29. Gda para inclusiones propias.

Prueba de aciclicidad

Se tiene un grafo dirigido $G = (V, A)$, para determinar si G es acíclico, esto es, si G no contiene ciclos. La búsqueda en profundidad puede usarse para responder a esta pregunta. Si se encuentra un arco de retroceso durante la búsqueda en profundidad de G , el grafo tiene un ciclo. Si, al contrario, un grafo dirigido tiene un ciclo, entonces siempre habrá un arco de retroceso en la búsqueda en profundidad del grafo.

Para ver este hecho, supóngase que G es cíclico. Si se efectúa una búsqueda en profundidad en G , habrá un vértice v que tenga el número de búsqueda en profundidad menor en un ciclo. Considérese un arco $u \rightarrow v$ en algún ciclo que contenga

a v . Ya que u está en el ciclo, debe ser un descendiente de v en el bosque abarcador en profundidad. Así, $u \rightarrow v$ no puede ser un arco cruzado. Puesto que el número en profundidad de u es mayor que el de v , $u \rightarrow v$ no puede ser un arco de árbol ni un arco de avance. Así que $u \rightarrow v$ debe ser un arco de retroceso, como se ilustra en la figura 6.30.



Fig. 6.30. Todo ciclo contiene un arco de retroceso.

Clasificación topológica

Un proyecto grande suele dividirse en una colección de tareas más pequeñas, algunas de las cuales se han de realizar en ciertos órdenes específicos, de modo que se pueda culminar el proyecto total. Por ejemplo, una carrera universitaria puede tener cursos que requieran otros como prerrequisitos. Los gda pueden emplearse para modelar de manera natural estas situaciones. Por ejemplo, podría tenerse un arco del curso C al curso D si C fuera un prerrequisito de D .

Ejemplo 6.16. La figura 6.31 muestra un gda con la estructura de prerrequisitos de cinco cursos. El curso C_3 , por ejemplo, requiere los cursos C_1 y C_2 como prerrequisitos. □

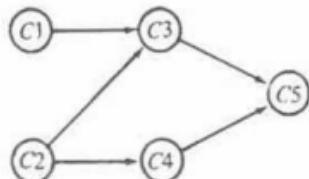


Fig. 6.31. Gda de prerrequisitos.

La *clasificación topológica* es un proceso de asignación de un orden lineal a los vértices de un gda tal que si existe un arco del vértice i al vértice j , i aparece antes que j en el ordenamiento lineal. Por ejemplo, $C1, C2, C3, C4, C5$ es una clasificación topológica del gda de la figura 6.31. Al tomar los cursos en esta secuencia, se puede satisfacer la estructura de prerequisitos dada en la figura.

La clasificación topológica puede efectuarse con facilidad si se agrega una instrucción de impresión después de la línea (4) al procedimiento de búsqueda en profundidad de la figura 6.24:

```
procedure clasificación_topológica(v: vértice);
  {imprime los vértices accesibles desde v en orden topológico invertido}
  var
    w: vértice;
  begin
    marca[v] := visitado;
    for cada vértice w en  $L[v]$  do
      if marca[w] = no_visitado then
        clasificación_topológica(w);
      writeln(v)
    end; {clasificación_topológica}
```

Cuando *clasificación_topológica* termina de buscar en todos los vértices adyacentes a un vértice dado x , imprime x . El efecto de llamar a *clasificación_topológica*(v) es imprimir en orden topológico inverso todos los vértices de un gda accesibles desde v por medio de un camino en el gda.

Esta técnica funciona porque no existen arcos de retroceso en un gda. Considerese lo que sucede cuando la búsqueda en profundidad deja un vértice x por última vez. Los únicos arcos que emanan de v son arcos de árbol, de avance y cruzados. Pero todos esos arcos están dirigidos hacia vértices que ya se han visitado completamente y que, por tanto, preceden a x en el orden que se están construyendo.

6.7 Componentes fuertes

Un componente fuertemente conexo de un grafo dirigido es un conjunto maximal de vértices en el cual existe un camino que va desde cualquier vértice del conjunto hasta cualquier otro vértice también del conjunto. La búsqueda en profundidad puede usarse para determinar con eficiencia los componentes fuertemente conexos de un grafo dirigido.

Sea $G = (V, A)$ un grafo dirigido; se puede dividir V en clases de equivalencia V_i , $1 \leq i \leq r$, tales que los vértices v y w son equivalentes si, y sólo si, existe un camino de v a w y otro de w a v . Sea A_i , $1 \leq i \leq r$, el conjunto de los arcos con cabeza y cola en V_i . Los grafos $G_i = (V_i, A_i)$ se denominan *componentes fuertemente conexos* (o sólo *componentes fuertes*) de G . Un grafo dirigido con sólo un componente fuerte, se dice que está *fuertemente conexo*.

Ejemplo 6.17. La figura 6.32 ilustra un grafo dirigido con los dos componentes fuertes mostrados en la figura 6.33. □

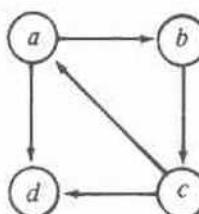


Fig. 6.32. Grafo dirigido.

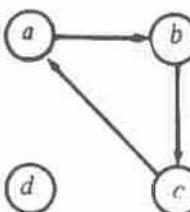


Fig. 6.33. Componentes fuertes del grafo de la figura 6.32.

Obsérvese que todo vértice de un grafo dirigido G está en algún componente fuerte, pero que ciertos arcos pueden no estarlo. Tales arcos, llamados arcos de *cruce de componentes*, van de un vértice de un componente a un vértice de otro. Se pueden representar las interconexiones entre los componentes construyendo un *grafo reducido* de G , cuyos vértices son los componentes fuertemente conexos de G . Hay un arco de un vértice C a un vértice diferente C' de este tipo de grafos, si existe un arco en G que vaya de algún vértice del componente C a algún otro del componente C' . El grafo reducido siempre es un gda, porque si existiera algún ciclo, todos los componentes del *ciclo* serían en realidad un solo componente fuerte, lo cual significaría que no se calcularon en forma adecuada los componentes fuertes. La figura 6.34 muestra el grafo reducido del grafo dirigido de la figura 6.32.



Fig. 6.34. Grafo reducido.

Ahora se presenta un algoritmo para encontrar los componentes fuertemente conexos de un grafo dirigido G dado.

1. Efectúese una búsqueda en profundidad de G y numérense los vértices en el orden de terminación de las llamadas recursivas; esto es, asígnese un número al vértice v después de la línea (4) de la figura 6.24.
2. Construyase un grafo dirigido nuevo G_r invirtiendo las direcciones de todos los arcos de G .
3. Realícese una búsqueda en profundidad en G_r , partiendo del vértice con numeración más alta de acuerdo con la numeración asignada en el paso (1). Si la búsqueda en profundidad no llega a todos los vértices, iníciense la siguiente búsqueda a partir del vértice restante con numeración más alta.
4. Cada árbol del bosque abarcador resultante es un componente fuertemente conexo de G .

Ejemplo 6.18. Se aplica este algoritmo al grafo dirigido de la figura 6.32, partiendo de a y continuando primero hasta b . Después del paso (1) se numeran los vértices como se muestra en la figura 6.35. Al invertir la dirección de los arcos, se obtiene el grafo G_r de la figura 6.36.

Cuando se realiza la búsqueda en profundidad en G_r , surge el bosque abarcador en profundidad de la figura 6.37. Se comienza con a como raíz, porque a tiene el número más alto. Desde a sólo se alcanza c y después b . El siguiente árbol tiene raíz d , ya que es el siguiente (y único) vértice restante con numeración más alta. Cada árbol del bosque forma un componente fuertemente conexo del grafo dirigido original. □

Se ha pretendido que los vértices de un componente fuertemente conexo se correspondan con los vértices de un árbol del bosque abarcador de la segunda búsqueda en profundidad. Para ver por qué, obsérvese que si v y w son vértices del mismo componente fuertemente conexo, existen caminos en G desde v hasta w y desde w hasta v . Así, existen también caminos desde v hasta w y desde w hasta v en G_r .

Supóngase que en la búsqueda en profundidad de G , se inicia la búsqueda en alguna raíz x y se llega hasta v o w . Como v y w se alcanzan uno al otro, ambos terminarán formando parte del árbol abarcador con raíz x .

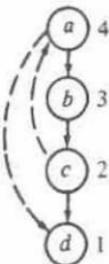
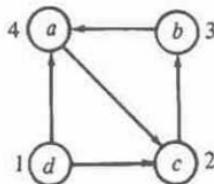
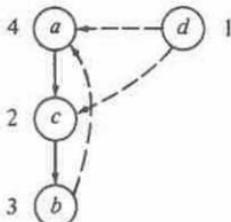


Fig. 6.35. Despues del paso 1.

Fig. 6.36. G_r .Fig. 6.37. Bosque abarcador en profundidad para G_r .

Ahora, supóngase que v y w están en el mismo árbol abarcador del bosque abarcador en profundidad de G_r . Se debe demostrar que v y w están en el mismo componente fuertemente conexo. Sea x la raíz del árbol abarcador que contiene v y w . Puesto que v es descendiente de x , existe un camino en G , que va de x a v . Así, existe un camino en G de v a x .

En la construcción del bosque abarcador en profundidad de G_r , el vértice v quedó sin visitarse cuando se inició la búsqueda en x . De aquí que x tiene un número mayor que v , por lo que en la búsqueda en profundidad de G , la llamada recursiva en v terminó antes que la llamada recursiva en x . Pero en la búsqueda en profundidad de G , la búsqueda en v no pudo haberse iniciado antes que la de x , ya que el camino en G de v a x implicaría que la búsqueda en x empezaría y terminaría antes de terminar la búsqueda en v .

Se concluye que en la búsqueda de G , v se visita durante la búsqueda de x , por lo que, v es descendiente de x en el primer bosque abarcador en profundidad de G . Así, existe un camino de x a v en G . Por tanto, x y v están en el mismo componente fuertemente conexo. Un razonamiento idéntico muestra que x y w están en el mismo componente fuertemente conexo y, por tanto, v y w también lo están, como muestran los caminos que van de v a x a w , y de w a x a v .

Ejercicios

6.1 Represéntese el grafo dirigido de la figura 6.38

- por medio de una matriz de adyacencia dando los costos de los arcos, y
- por medio de una lista enlazada de adyacencia con indicación de los costos de los arcos.

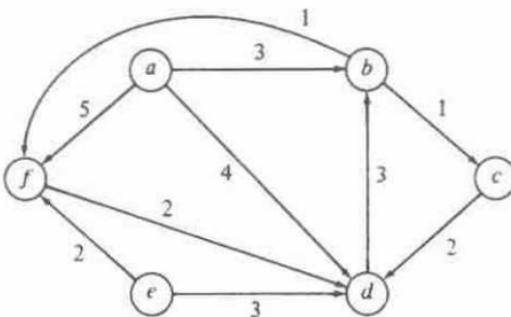


Fig. 6.38. Grafo dirigido con costos de los arcos.

- 6.2 Describase un modelo matemático para el siguiente problema de horarios. Dadas las tareas T_1, T_2, \dots, T_n , que requieren tiempos t_1, t_2, \dots, t_n para ejecutarse, y un conjunto de restricciones, cada una de la forma « T_i debe terminar antes del inicio de T_j », encuéntrese el tiempo mínimo necesario para ejecutar todas las tareas.
- 6.3 Realicense las operaciones PRIMERO, SIGUIENTE y VERTICE para grafos dirigidos representados por
- matrices de adyacencia,
 - listas enlazadas de adyacencias, y
 - listas de adyacencias como la representada en la figura 6.5.
- 6.4 En el grafo dirigido de la figura 6.38,
- empleése el algoritmo *Dijkstra* para encontrar los caminos más cortos que van del vértice a a los otros vértices.
 - utilícese el algoritmo *Floyd* para encontrar las distancias más cortas entre todos los pares de puntos. Constrúyase también la matriz P que permita recuperar los caminos más cortos.
- 6.5 Escríbase un programa completo para el algoritmo de Dijkstra usando árboles parcialmente ordenados como colas de prioridad y listas enlazadas de adyacencias.
- *6.6 Demuéstrese que el programa *Dijkstra* no funciona bien si los arcos tienen costos negativos.
- **6.7 Muéstrese que el programa *Floyd* funciona si alguno de los arcos, pero ningún ciclo, tienen costo negativo.
- 6.8 Suponiendo que el orden de los vértices es a, b, \dots, f en la figura 6.38, constrúyase un bosque abarcador en profundidad; indíquese los arcos de árbol, de retroceso, de avance y cruzados, y la numeración en profundidad de los vértices.

- *6.9 Supóngase que se tiene un bosque abarcador en profundidad, y se lista en orden posterior cada uno de los árboles abarcadores (los árboles que están formados por aristas abarcadoras), de izquierda a derecha. Demuéstrese que este orden es el mismo en el que terminaron las llamadas a *bpf* cuando se construyó el bosque abarcador.
- 6.10 Una *raíz* de un gda es un vértice *r* tal que todo vértice del gda puede alcanzarse por un camino dirigido desde *r*. Escribábase un programa para determinar si un gda posee raíz.
- *6.11 Considérese un gda con *a* arcos y con dos vértices distintos *s* y *t*. Constrúyase un algoritmo *O(a)* para encontrar el conjunto maximal de caminos disjuntos de *s* a *t*. Por maximal se entiende que ya no se pueden añadir caminos adicionales, pero eso no significa que sea el tamaño más grande para ese conjunto.
- 6.12 Constrúyase un algoritmo para convertir un árbol de expresiones con los operadores + y * en un gda al compartir subexpresiones comunes. ¿Cuál es la complejidad de tiempo de ese algoritmo?
- 6.13 Constrúyase un algoritmo para la evaluación de expresiones aritméticas representadas con un gda.
- 6.14 Escribábase un programa para encontrar el camino más largo en un gda. ¿Cuál es la complejidad de tiempo de este programa?
- 6.15 Encuéntrense los componentes fuertes de la figura 6.38.
- *6.16 Pruébese que el grafo reducido de los componentes fuertes de la sección 6.7 debe ser un gda.
- 6.17 Dibújese el primer bosque abarcador, el grafo invertido y el segundo bosque abarcador que se obtiene al aplicar el algoritmo de componentes fuertes al grafo dirigido de la figura 6.38.
- 6.18 Obténgase el algoritmo de componentes fuertes analizado en la sección 6.7.
- *6.19 Muéstrese que el algoritmo de componentes fuertes requiere un tiempo *O(a)* en un grafo dirigido de *a* arcos y *n* vértices, suponiendo que *n* ≤ *a*.
- *6.20 Escribábase un programa que tome como entrada un grafo dirigido y dos de sus vértices. El programa debe imprimir todos los caminos simples que vayan de un vértice al otro. ¿Cuál es la complejidad de tiempo de este programa?
- *6.21 Una *reducción transitiva* de un grafo dirigido *G* = (*V, A*) es cualquier grafo *G'* con los mismos vértices pero con la menor cantidad de arcos posible, de modo que el cierre transitivo *G'* es el mismo que el de *G*. Demuéstrese que si *G* es un gda, la reducción transitiva de *G* es única.

- *6.22 Escribase un programa para obtener la reducción transitiva de un grafo dirigido. ¿Cuál es la complejidad de tiempo de este programa?
- *6.23 $G' = (V, A')$ se conoce como el *grafo dirigido equivalente minimal* de un grafo dirigido $G = (V, A)$, si A' es el subconjunto más pequeño de A y la cerradura transitiva de G y G' es el mismo. Demuéstrese que si G es acíclico, sólo hay un grafo dirigido equivalente minimal, es decir, la reducción transitiva.
- *6.24 Escribase un programa para encontrar un grafo dirigido equivalente minimal para un grafo dirigido dado. ¿Cuál es la complejidad de tiempo de ese programa?
- *6.25 Escribase un programa para encontrar el camino simple más largo de un vértice dado de un grafo dirigido. ¿Cuál es la complejidad de tiempo del programa?

Notas bibliográficas

Berge [1985] y Harary [1969] son dos buenas fuentes de material suplementario sobre teoría de grafos. Algunos libros que tratan algoritmos sobre grafos son Deo [1975], Even [1980] y Tarjan [1983].

El algoritmo para caminos más cortos con un solo origen de la sección 6.3 se debe a Dijkstra [1959]. El algoritmo de los caminos más cortos entre todos los pares es de Floyd [1962] y el de cerradura transitiva es de Warshall [1962]. Johnson [1977] analiza algoritmos eficientes para encontrar caminos más cortos en grafos dispersos. Knuth [1968] contiene material adicional sobre clasificación topológica.

El algoritmo de componentes fuertes de la sección 6.7 es similar al sugerido por R. Kosaraju en 1978 (sin publicar), y al publicado por Sharir [1981]. Tarjan [1972] contiene otro algoritmo de componentes fuertes que sólo necesita un recorrido con búsqueda en profundidad.

Coffman [1976] contiene muchos ejemplos de cómo se pueden usar los grafos dirigidos para los problemas de modelado de horarios, como en el ejercicio 6.2. Aho, Garey y Ullman [1972] muestran que la reducción transitiva de un gda es única, y que el cálculo de la reducción transitiva de un grafo dirigido es, computacionalmente, equivalente al cálculo de la cerradura transitiva (Ejercicios 6.21 y 6.22). La obtención del grafo dirigido equivalente minimal (Ejercicios 6.23 y 6.24), por otro lado, parece ser mucho más difícil desde el punto de vista computacional; este problema es NP-completo [Sahni (1974)].

7

Grafos no dirigidos

Un grafo no dirigido $G = (V, A)$ consta de un conjunto finito de vértices V y de un conjunto de aristas A . Se diferencia de un grafo dirigido en que cada arista en A es un par no ordenado de vértices †. Si (v, w) es una arista no dirigida, entonces $(v, w) = (w, v)$. De ahora en adelante se hará referencia a los grafos no dirigidos tan sólo como grafos.

Los grafos se emplean en distintas disciplinas para modelar relaciones simétricas entre objetos. Los objetos se representan por los vértices del grafo, y dos objetos están conectados por una arista si están relacionados entre sí. En este capítulo se presentan varias estructuras de datos que pueden usarse para representar grafos, y los algoritmos para tres problemas comunes que se relacionan con grafos no dirigidos: construcción de árboles abarcadores minimales, componentes biconexos y comparaciones maximales.

7.1 Definiciones

Buena parte de la terminología para grafos dirigidos es aplicable también a los no dirigidos. Por ejemplo, los vértices v y w son *adyacentes* si (v, w) es una arista [o, en forma equivalente, si (w, v) lo es]. Se dice que la arista (v, w) es *incidente* sobre los vértices v y w .

Un *camino* es una secuencia de vértices v_1, v_2, \dots, v_n tal que (v_i, v_{i+1}) es una arista para $1 \leq i < n$. Un camino es *simple* si todos sus vértices son distintos, con excepción de v_1 y v_n , que pueden ser el mismo. La longitud del camino es $n - 1$, el número de aristas a lo largo del camino. Se dice que el camino v_1, v_2, \dots, v_n *conecta* v_1 y v_n . Un grafo es *conexo* si todos sus pares de vértices están conectados.

Sea $G = (V, A)$ un grafo con conjunto de vértices V y conjunto de aristas A . Un *subgrafo* de G es un grafo $G' = (V', A')$ donde

1. V' es un subconjunto de V .
2. A' consta de las aristas (v, w) en A tales que v y w están en V' .

Si A' consta de todas las aristas (v, w) en A , tal que v y w están en V' , entonces G' se conoce como un *subgrafo inducido* de G .

† A menos que se especifique lo contrario, aquí se supondrá que una arista siempre es un par de vértices distintos.

Ejemplo 7.1. En la figura 7.1(a) se observa un grafo $G = (V, A)$ con $V = \{a, b, c, d\}$ y $A = \{(a, b), (a, d), (b, c), (b, d), (c, d)\}$, y en la figura 7.1(b), uno de sus subgrafos inducidos, definido por el conjunto de vértices $\{a, b, c\}$ y todas las aristas de la figura 7.1(a) que no inciden sobre el vértice d . \square

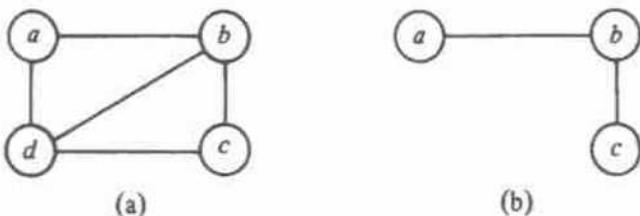


Fig. 7.1. Grafo con uno de sus subgrafos.

Un *componente conexo* de un grafo G es un subgrafo conexo inducido maximal, esto es, un subgrafo conexo inducido que por sí mismo no es un subgrafo propio de ningún otro subgrafo conexo de G .

Ejemplo 7.2. La figura 7.1 es un grafo conexo que tiene sólo un componente conexo, y que es él mismo. La figura 7.2 es un grafo con dos componentes conexos. \square

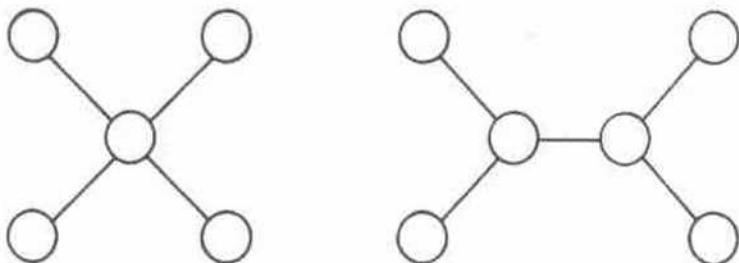


Fig. 7.2. Grafo no conexo.

Un *ciclo* (simple) de un grafo es un camino (simple) de longitud mayor o igual a tres, que conecta un vértice consigo mismo. No se consideran ciclos los caminos de la forma v (camino de longitud 0), v, v (camino de longitud 1), o v, w, v (camino de longitud 2). Un grafo es *cíclico* si contiene por lo menos un ciclo. Un grafo conexo acíclico algunas veces se conoce como *árbol libre*. La figura 7.2 muestra un grafo que consta de dos componentes conexos, cada uno de los cuales es un árbol libre. Un árbol libre puede convertirse en ordinario si se elige cualquier vértice deseado como raíz y se orienta cada arista desde ella.

Los árboles libres tienen dos propiedades importantes que se usarán en la siguiente sección.

1. Todo árbol libre con $n \geq 1$ vértices contiene exactamente $n - 1$ aristas.
2. Si se agrega cualquier arista a un árbol libre, resulta un ciclo.

Se puede probar (1) por inducción en n , o en forma equivalente, con un argumento basado en el "contraejemplo más pequeño". Supóngase que $G = (V, A)$ es un contraejemplo de (1) con un mínimo de vértices n , por ejemplo n no puede valer uno, porque el único árbol libre con un vértice tiene cero aristas, y (1) se satisface. Por tanto, n debe ser mayor que uno.

Ahora se pretende que en el árbol libre exista algún vértice con exactamente una arista incidente. En la demostración, ningún vértice puede tener cero aristas incidentes, o G no sería conexo. Supóngase que todo vértice tiene por lo menos dos aristas incidentes. Después, pártese de algún vértice v_1 , y sígase cualquier arista desde v_1 . En cada paso, se abandona un vértice por una arista diferente a la que se utilizó para llegar, formando un camino v_1, v_2, v_3, \dots .

Dado que sólo se tiene un número finito de vértices en V , no es posible que todos los vértices en el camino sean diferentes; en un momento dado, se encuentra $v_i = v_j$ para alguna $i < j$. No se puede tener $i = j - 1$, porque no hay ciclos de un vértice a sí mismo, y tampoco $i = j - 2$, ya que se llegaría y se abandonaría el vértice v_{i+1} por la misma arista. Así, $i \leq j - 3$, y se tiene un ciclo $v_i, v_{i+1}, \dots, v_j = v_i$, con lo que se contradice la hipótesis de que G no tiene vértices con sólo una arista incidente y, por tanto, se concluye que existe tal vértice v con arista (v, w) .

Ahora, considérese el grafo G' formado al eliminar el vértice v y la arista (v, w) de G . G' no puede contradecir (1), porque si lo hiciera podría ser un contraejemplo más pequeño que G . Por tanto, G' tiene $n - 1$ vértices y $n - 2$ aristas. Pero G tiene un vértice y una arista más que G' , es decir, tiene $n - 1$ aristas, probando que G satisface realmente (1). Como no hay un contraejemplo más pequeño para (1), se concluye que no existe ese contraejemplo, y (1) es cierto.

Ahora, es posible probar con facilidad la proposición (2) de que la adición de una arista a un árbol libre forma un ciclo. De no ser así, el resultado de agregar la arista a un árbol libre de n vértices sería un grafo con n vértices y n aristas. Este grafo aún sería conexo, y se ha supuesto que agregando la arista quedaría un grafo acíclico. Con esto, se tendría un árbol libre cuyas cantidades de vértices y de aristas no satisfarían la condición (1).

Métodos de representación

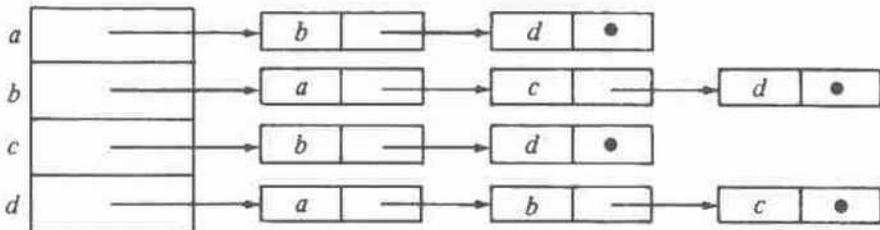
Los métodos de representación de grafos dirigidos se pueden emplear también para representar los no dirigidos. Una arista no dirigida entre v y w se representa simplemente con dos aristas dirigidas, una de v a w , y otra de w a v .

Ejemplo 7.3. Las representaciones con matriz y lista de adyacencia para el grafo de la figura 7.1(a) se muestran en la figura 7.3. □

Es notorio que la matriz de adyacencia para un grafo es simétrica. En la representación con lista de adyacencia, si (i, j) es una arista, el vértice j estará en la lista del vértice i y el vértice i estará en la lista del vértice j .

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>	0	1	0	1
<i>b</i>	1	0	1	1
<i>c</i>	0	1	0	1
<i>d</i>	1	1	1	0

(a) Matriz de adyacencia



(b) Lista de adyacencia

Fig. 7.3. Representaciones.

7.2 Arboles abarcadores de costo mínimo

Supóngase que $G = (V, A)$ es un grafo conexo en donde cada arista (u, v) de A tiene un costo asociado $c(u, v)$. Un *árbol abarcador* para G es un árbol libre que conecta todos los vértices de V ; su *costo* es la suma de los costos de las aristas del árbol. En esta sección se muestra cómo obtener el árbol abarcador de costo mínimo para G .

Ejemplo 7.4. La figura 7.4 muestra un grafo ponderado y su árbol abarcador de costo mínimo. □

Una aplicación típica de los árboles abarcadores de costo mínimo tiene lugar en el diseño de redes de comunicación. Los vértices del grafo representan ciudades, y las aristas, las posibles líneas de comunicación entre ellas. El costo asociado a una arista representa el costo de seleccionar esa línea para la red. Un árbol abarcador de costo mínimo representa una red que comunica todas las ciudades a un costo mínimo.

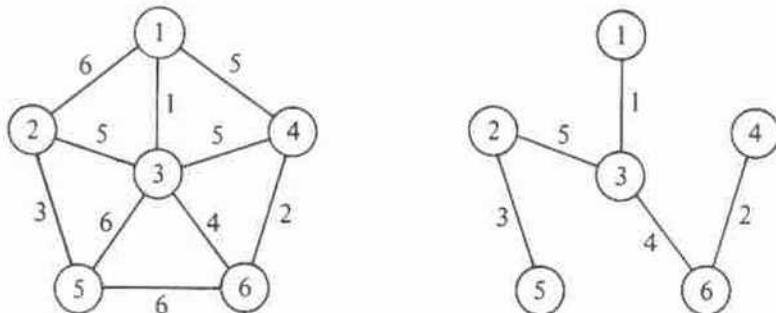


Fig. 7.4. Grafo y árbol abarcador.

La propiedad AAM (Árbol Abarcador de costo Mínimo)

Hay distintas formas de construir un árbol abarcador de costo mínimo. Muchos de esos métodos utilizan la siguiente propiedad de los árboles abarcadores de costo mínimo, que se denomina *propiedad AAM*. Sea $G = (V, A)$ un grafo conexo con una función de costo definida en las aristas. Sea U algún subconjunto propio del conjunto de vértices V . Si (u, v) es una arista de costo mínimo tal que $u \in U$ y $v \in V - U$, existe un árbol abarcador de costo mínimo que incluye (u, v) entre sus aristas.

La demostración de que todo árbol abarcador de costo mínimo satisface la propiedad AAM no es muy difícil. Supóngase, por el contrario, que no existe el árbol abarcador de costo mínimo para G que incluya (u, v) . Sea T cualquier árbol abarcador de costo mínimo para G . Agregar (u, v) a T debe formar un ciclo, ya que T es un árbol libre y, por tanto, satisface la propiedad (2) de los árboles libres. Este ciclo incluye la arista (u, v) . Así, debe haber otra arista (u', v') en T tal que $u' \in U$ y $v' \in V - U$, como se ilustra en la figura 7.5. Si no, no habría forma de que el ciclo fuera de u a v sin pasar por segunda vez por la arista (u, v) .

Al eliminar la arista (u', v') se rompe el ciclo y se obtiene un árbol abarcador T' cuyo costo en realidad no es mayor que el costo de T , ya que, por suposición, $c(u, v) \leq c(u', v')$. Así T' contradice la suposición de que no hay un árbol abarcador de costo mínimo que incluya (u, v) .

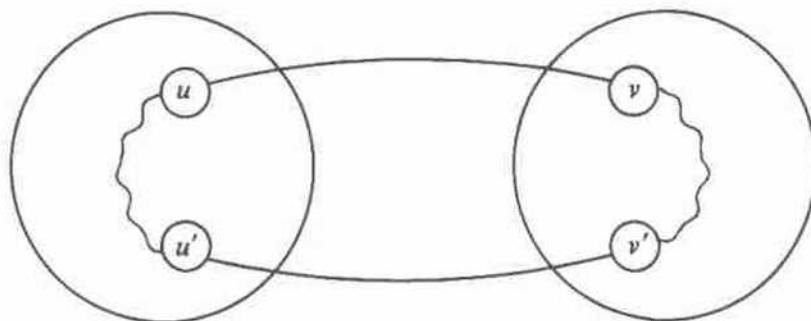


Fig. 7.5. Ciclo resultante.

Algoritmo de Prim

Existen dos técnicas populares que explotan la propiedad AAM para construir un árbol abarcador de costo mínimo a partir de un grafo ponderado $G = (V, A)$; una de ellas se conoce como algoritmo de Prim. Supóngase que $V = \{1, 2, \dots, n\}$. El algoritmo de Prim comienza cuando se asigna a un conjunto U un valor inicial $\{1\}$, en el cual «crece» un árbol abarcador, arista por arista. En cada paso localiza la arista más corta (u, v) que conecta U y $V - U$, y después agrega v , el vértice en $V - U$, a U . Este paso se repite hasta que $U = V$. El algoritmo se resume en la figura 7.6, y la secuencia de aristas agregadas a T para el grafo de la figura 7.4(a) se muestra en la figura 7.7.

```

procedure Prim (G: grafo; var T: conjunto de aristas);
  [Prim construye un árbol abarcador de costo mínimo T para G]
  var
    U: conjunto de vértices;
    u, v: vértice;
  begin
    T := Ø;
    U := {1};
    while U ≠ V do begin
      sea (u, v) una arista de costo mínimo tal que u está en U y
      v en V-U;
      T := T ∪ {(u, v)};
      U := U ∪ {v}
    end
  end; [Prim]

```

Fig. 7.6. Esbozo del algoritmo de Prim.

Una forma sencilla de encontrar la arista de menor costo entre U y $V - U$ en cada paso es por medio de dos arreglos; uno, *MAS_CERCANO[i]*, da el vértice en U que esté más cercano a i en $V - U$. El otro, *MENOR_COSTO[i]*, da el costo de la arista $(i, \text{MAS_CERCANO}[i])$.

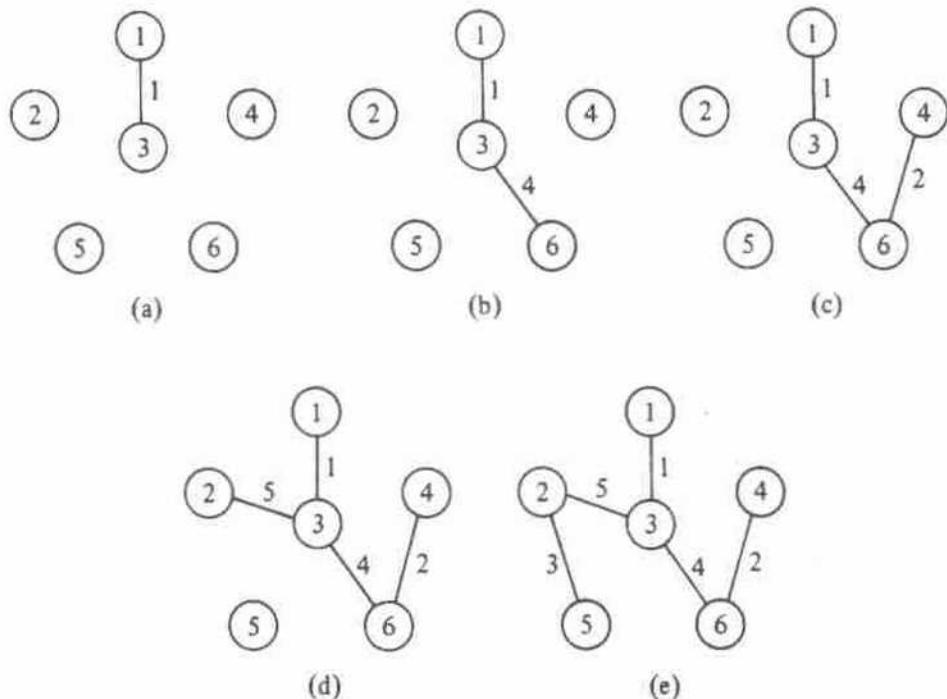


Fig. 7.7. Secuencias de aristas añadidas por el algoritmo de Prim.

En cada paso se revisa *MENOR_COSTO* para encontrar algún vértice, como k , en $V - U$ que esté más cercano a U . Se imprime la arista $(k, MAS_CERCANO[k])$. Entonces se actualizan los arreglos *MENOR_COSTO* y *MAS_CERCANO*, teniendo en cuenta el hecho de que k ha sido agregada a U . En la figura 7.8 se da una versión en Pascal de este algoritmo. Se supone que C es un arreglo de $n \times n$ tal que $C[i, j]$ es el costo de la arista (i, j) . Si la arista (i, j) no existe, se supone que $C[i, j]$ tiene un valor grande apropiado.

Si encuentra otro vértice k para el árbol abarcador, se hace que *MENOR_COSTO*[k] sea *infinito*, un valor muy grande, de modo que este vértice ya no se considera en los recorridos siguientes para incluirlo en U . El valor *infinito* es mayor que el costo de cualquier arista o que el costo asociado a una arista no existente.

La complejidad de tiempo del algoritmo de Prim es $O(n^2)$, ya que se efectúan $n - 1$ iteraciones del ciclo de las líneas (4) a (16) y cada iteración del ciclo lleva un tiempo $O(n)$, debido a los ciclos más internos de las líneas (7) a (10) y (13) a (16). Conforme n crece, el rendimiento de este algoritmo puede dejar de ser satisfactorio. Ahora se presenta otro algoritmo, debido a Kruskal, para encontrar árboles abarcadores de costo mínimo cuyo rendimiento puede ser como máximo $O(a \log a)$, donde a es el número de aristas del grafo dado. Si a es mucho menor que n^2 , el algoritmo de Kruskal es superior, pero si es cercano a n^2 , se debe optar por el algoritmo de Prim.

```

procedure Prim (C: array[1..n, 1..n] of real);
  | Prim imprime las aristas de un árbol abarcador de costo mínimo
    para un grafo con vértices [1, 2, ..., n] y matriz de costo C
    definida en las aristas |
  var
    MENOR_COSTO: array[1..n] of real;
    MAS_CERCANO: array[1..n] of integer;
    i, j, k, min: integer;
    | i y j son índices. Durante una revisión del arreglo MENOR_COSTO,
      k es el índice del vértice más cercano encontrado hasta
      ese punto, y min = MENOR_COSTO[k] |

  begin
    (1)   for i := 2 to n do begin
          | asigna valor inicial al conjunto U sólo con el vértice 1 |
          (2)   MENOR_COSTO[i] := C[1, i];
          (3)   MAS_CERCANO[i] := 1
          end;
    (4)   for i := 2 to n do begin
          | encuentra el vértice k fuera de U más cercano a algún vértice
            en U |
          (5)   min := MENOR_COSTO[2];
          (6)   k := 2;
          (7)   for j := 3 to n do
                if MENOR_COSTO[j] < min then begin

```

```

(9)           min := MENOR_COSTO[j];
(10)          k := j
(11)          end;
(12)          writeln(k, MAS_CERCANO[k]); | imprime la arista |
(13)          MENOR_COSTO[k] := infinito; | se añade k a U |
(14)          for j := 2 to n do | ajusta los costos de U |
(15)            if (C[k, j] < MENOR_COSTO[j]) and
(16)              (MENOR_COSTO[j] < infinito) then begin
               MENOR_COSTO[j] := C[k, j];
               MAS_CERCANO[j] := k
end
end; | Prim |

```

Fig. 7.8. Algoritmo de Prim.

Algoritmo de Kruskal

Supóngase de nuevo que se tiene un grafo conexo $G = (V, A)$, con $V = \{1, 2, \dots, n\}$ y una función de costo c definida en las aristas de A . Otra forma de construir un árbol abarcador de costo mínimo para G es empezar con un grafo $T = (V, \emptyset)$ constituido sólo por los vértices de G y sin aristas. Por tanto, cada vértice es un componente conexo por sí mismo. Conforme el algoritmo avanza, habrá siempre una colección de componentes conexos, y para cada componente se seleccionarán las aristas que formen un árbol abarcador.

Para construir componentes cada vez mayores, se examinan las aristas a partir de A , en orden creciente de acuerdo con el costo. Si la arista conecta dos vértices que se encuentran en dos componentes conexos distintos, entonces se agrega la arista T . Se descartará la arista si conecta dos vértices contenidos en el mismo componente, ya que puede provocar un ciclo si se la añadiera al árbol abarcador para ese componente conexo. Cuando todos los vértices están en un solo componente, T es un árbol abarcador de costo mínimo para G .

Ejemplo 7.5. Considérese el grafo ponderado de la figura 7.4(a). La secuencia de aristas agregadas a T se muestra en la figura 7.9. Las aristas de costo 1, 2, 3 y 4 se consideran primero, y todas son aceptadas, ya que ninguna de ellas causa un ciclo. Las aristas (1, 4) y (3, 4) de costo 5 no pueden aceptarse, porque conectan vértices que están dentro del mismo componente en la figura 7.9(d) y, por tanto, pueden completar un ciclo. Sin embargo, la arista restante de costo 5, o sea (2, 3), no crea ciclos. Una vez que se acepta, el proceso termina. \square

Es posible aplicar este algoritmo mediante los conjuntos y sus operaciones asociadas de los capítulos 4 y 5. Primero se necesita un conjunto formado por las aristas de A . Al conjunto se le aplica en forma repetida el operador *SUPRIME-MIN* para seleccionar aristas en orden creciente de acuerdo con el costo. El conjunto de aristas, por tanto, forma una cola de prioridad, y entonces un árbol parcialmente ordenado es la estructura de datos más apropiada para usar aquí.

También se requiere mantener un conjunto de componentes conexos C . Las operaciones que se le aplican son:

1. COMBINA(A, B, C), para combinar los componentes A y B en C y llamar al resultado A o B en forma arbitraria †.
2. ENCUENTRA(v, C), para devolver el nombre del componente de C , del cual el vértice v es miembro. Esta operación se usará para determinar si los dos vértices de una arista se encuentran en dos componentes distintos o en el mismo.
3. INICIAL(A, v, C), para que A sea el nombre de un componente que pertenece a C , y que inicialmente contiene sólo el vértice v .

Estas son las operaciones del TDA COMBINA-ENCUENTRA llamado CONJUNTO-CE, estudiado en la sección 5.5. En la figura 7.10 se muestra un esbozo de un programa llamado *Kruskal* para encontrar árboles abarcadores de costo mínimo con estas operaciones.

Se pueden emplear las técnicas desarrolladas en la sección 5.5 para implantar las operaciones utilizadas en este programa. El tiempo de ejecución de este programa depende de dos factores. Si hay a aristas, lleva un tiempo $O(a \log a)$ insertar las aristas en la cola de prioridad ‡. En cada iteración del ciclo while, la obtención de la

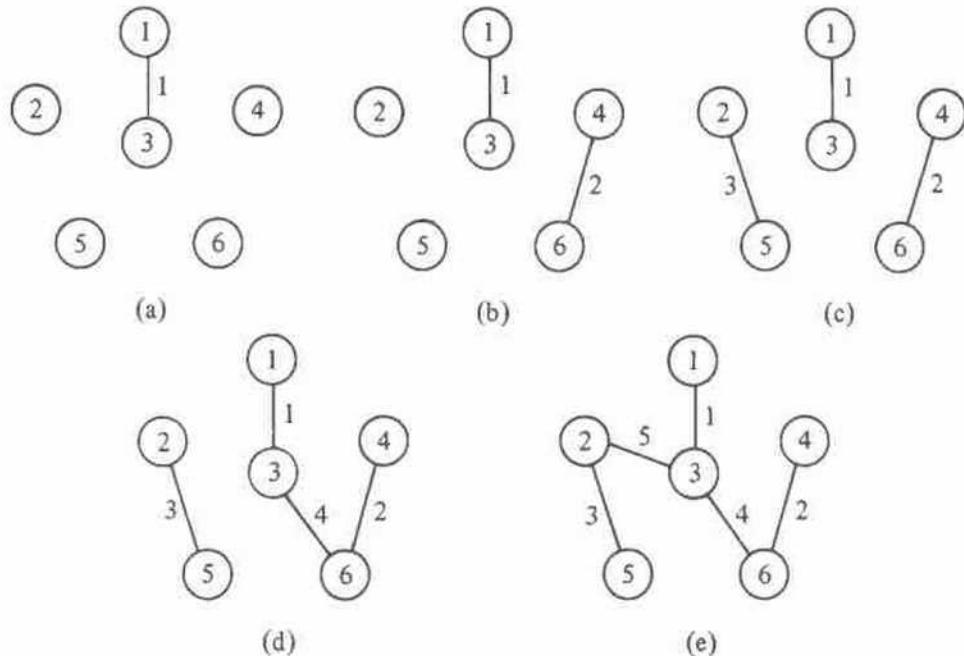


Fig. 7.9. Secuencia de aristas añadidas por el algoritmo de Kruskal.

† Obsérvese que COMBINA y ENCUENTRA son ligeramente distintas a las definiciones de la sección 5.5, ya que C es un parámetro que indica dónde se pueden encontrar A y B .

‡ Se puede asignar valor inicial a un árbol parcialmente ordenado de a elementos en un tiempo $O(a)$, si se hace todo de una vez. Esta técnica se analiza en la sección 8.4, aunque tal vez se debería utilizar aquí, ya que si se examinan menos de a aristas antes de encontrar el árbol abarcador de costo mínimo, se puede ahorrar un tiempo significativo.

arista de menor costo en *aristas* lleva un tiempo $O(\log a)$. Así, las operaciones en la cola de prioridad llevan un tiempo $O(a \log a)$ en el peor caso. El tiempo total requerido para realizar las operaciones COMBINA y ENCUENTRA depende del método usado para implantar el CONJUNTO_CE. Como se muestra en la sección 5.5, hay métodos $O(a \log a)$ y $O(a \alpha(a))$. En cualquiera de los casos, el algoritmo de Kruskal se puede implantar para que se ejecute en tiempo $O(a \log a)$.

7.3 Recorridos

En un gran número de problemas con grafos, es necesario visitar sistemáticamente los vértices del grafo. Las búsquedas en profundidad y en amplitud, temas de esta sección, son dos técnicas importantes para hacerlo. Ambas técnicas pueden usarse para determinar de manera eficiente todos los vértices que están conectados a un vértice dado.

```

procedure Kruskal ( V: CONJUNTO de vértices;
                    A: CONJUNTO de aristas;
                    var T: CONJUNTO de aristas );
var
  comp_n: integer; { número actual de componentes }
  aristas: COLA_DE_PRIORIDAD; { el conjunto de aristas }
  componentes: CONJUNTO_CE; { el conjunto V agrupado en
                            un conjunto de componentes COMBINA_ENCUENTRA }
  u, v: vértice;
  a: arista;
  comp_siguiente: integer; { nombre para el nuevo componente }
  comp_u, comp_v; { nombres de componentes }

begin
  ANULA(T);
  ANULA(aristas);
  comp_siguiente := 0;
  comp_n := número de miembros de V;
  for v en V do begin { asigna valor inicial a un componente
                        para que contenga un vértice de V }
    comp_siguiente := comp_siguiente + 1;
    INICIAL(comp_siguiente, v, componentes)
  end;
  for a en A do { asigna valor inicial a la cola de prioridad de aristas }
    INSERTA(a, aristas);
  while comp_n > 1 do begin { considera la siguiente arista }
    a := SUPRIME_MIN(aristas)
    sea a = (u, v);
    comp_u := ENCUENTRA(u, componentes);
    comp_v := ENCUENTRA(v, componentes);
    if comp_u ≠ comp_v then
      union(comp_u, comp_v, componentes);
      T := T ∪ {a};
    end;
  end;
end;
  
```

```

if comp_u <> comp_v then begin
    { a conecta dos componentes diferentes }
    COMBINA(comp_u, comp_v, componentes);
    comp_n := comp_n-1;
    INSERTA(a, T)
end
end
end; { Kruskal }

```

Fig. 7.10. Algoritmo de Kruskal.

Búsqueda en profundidad

Recuérdese de la sección 6.5 el algoritmo *bpf* para búsquedas en grafos dirigidos. El mismo algoritmo puede emplearse para búsqueda en grafos no dirigidos, puesto que la arista no dirigida (v, w) puede considerarse como el par de aristas dirigidas $v \rightarrow w$ y $w \rightarrow v$.

De hecho, los bosques abarcadores en profundidad, construidos para grafos no dirigidos, son más simples que para los dirigidos. Primero, se debe observar que cada árbol del bosque es un componente conexo del grafo, y que si el grafo fuera conexo, tendría sólo un árbol en su bosque. Segundo, para grafos dirigidos se identifican cuatro clases de arcos: de árbol, de avance, de retroceso y cruzado. Para grafos no dirigidos sólo hay dos clases: aristas de árbol y de retroceso.

Dado que en grafos no dirigidos no existe distinción entre las aristas de retroceso y las de avance, se denominarán arcos *de retroceso*. En un grafo no dirigido no existen las aristas cruzadas, esto es, aristas (v, w) donde v no es antecesor ni descendiente de w en el árbol abarcador. Supóngase que las hubiera; entonces, sea v un vértice alcanzado antes que w en la búsqueda. La llamada a $bpf(v)$ no puede terminar hasta haber buscado w , así que w se introduce en el árbol como descendiente de v . De modo semejante, si $bpf(w)$ se llama antes que $bpf(v)$, v se convierte en descendiente de w .

Como resultado, durante una búsqueda en profundidad en un grafo no dirigido G , todas las aristas pueden ser,

1. *aristas de árbol*, aquellas aristas (v, w) tales que $bpf(v)$ llama directamente a $bpf(w)$ o viceversa, o bien
2. *aristas de retroceso*, aquellas aristas (v, w) tales que ni $bpf(v)$ ni $bpf(w)$ se llaman directamente, pero una llamó indirectamente a la otra (es decir, $bpf(w)$ llama a $bpf(x)$, que llama a $bpf(v)$, de modo que w es antecesor de v).

Ejemplo 7.6. Considérese el grafo conexo G de la figura 7.11(a). Un árbol abarcador en profundidad T resultante de una búsqueda en profundidad de G se muestra en la figura 7.11(b). Se supuso que la búsqueda empezó en el vértice a , y se adoptó la convención de mostrar las aristas del árbol con líneas de trazo continuo y las aristas de retroceso con líneas de puntos. El árbol se dibujó con la raíz en la parte

superior, y los hijos de cada vértice, en el orden de izquierda a derecha en que fueron visitados por el procedimiento *bpf*.

Para seguir unos cuantos pasos de la búsqueda, el procedimiento *bpf(a)* llama a *bpf(b)* y añade la arista (a, b) a T , ya que b no ha sido visitado. En b , *bpf* llama a *bpf(d)* y agrega la arista (b, d) a T . En d , *bpf* llama a *bpf(e)* y añade la arista (d, e) a T . En e , los vértices a , b y d ya están marcados como visitados, de modo que *bpf(e)* regresa sin incorporar ninguna arista a T . En d , *bpf* encuentra los vértices a y b marcados como visitados, así que *bpf(d)* regresa también sin agregar más aristas a T . En b , *bpf* encuentra los vértices adyacentes restantes a y e marcados como visitados, así que *bpf(b)* regresa. La búsqueda continúa después con c , f y g . \square

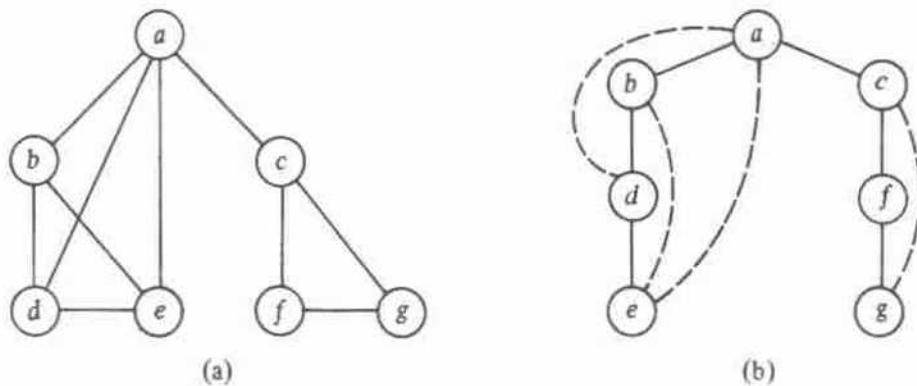


Fig. 7.11. Un grafo y su búsqueda en profundidad.

Búsqueda en amplitud

Otra forma sistemática de visitar los vértices se conoce como *búsqueda en amplitud*. Este enfoque se denomina «en amplitud» porque desde cada vértice v que se visita se busca en forma tan amplia como sea posible, visitando todos los vértices adyacentes a v . Esta estrategia de búsqueda también se puede aplicar a grafos dirigidos.

Igual que en la búsqueda en profundidad, al realizar una búsqueda en amplitud se puede construir un bosque abarcador. En este caso, se considera la arista (x, y) como una arista de árbol si el vértice y es el que se visitó primero partiendo del vértice x del ciclo interno del procedimiento de búsqueda *bea* de la figura 7.12.

Resulta que para la búsqueda en amplitud en un grafo no dirigido, toda arista que no es de árbol es una arista cruzada; esto es, conecta dos vértices ninguno de los cuales es antecesor del otro.

El algoritmo de búsqueda en amplitud de la figura 7.12 inserta las aristas de árbol en un conjunto T , que se supone inicialmente vacío. Se presume que cada entrada en el arreglo *marca* tiene asignado el valor inicial *no_visitado*; la figura 7.12 opera en un componente conexo. Si el grafo no es conexo, *bea* debe llamarse desde un vértice de cada componente. Obsérvese que en una búsqueda en amplitud se debe

marcar cada vértice como visitado antes de meterlo en la cola, y así evitar que se coloque en la cola más de una vez.

Ejemplo 7.7. El árbol abarcador en amplitud del grafo G de la figura 7.11(a) se muestra en la figura 7.13. Se supone que la búsqueda empieza en el vértice a . Como antes, las aristas de árbol se muestran con líneas de trazo continuo y las otras con líneas de puntos. También se ha dibujado la raíz del árbol en la parte superior, y los hijos, de izquierda a derecha, de acuerdo con el orden en que fueron visitados. \square

```

procedure bea (v);
    { bea visita todos los vértices conectados a v usando búsqueda en
      amplitud }
    var
        C: COLA de vértice;
        x, y: vértice;
    begin
        marca[v] := visitado;
        PONE_EN_COLA(v, C);
        while not VACIA(C) do begin
            x := FRENTE(C);
            QUITA_DE_COLA (C);
            for cada vértice y adyacente a x do
                if marca[y] = no_visitado then begin
                    marca[y] := visitado;
                    PONE_EN_COLA(y, C);
                    INSERTA((x, y), T)
                end
            end
        end; { bea }
    
```

Fig. 7.12. Búsqueda en amplitud.

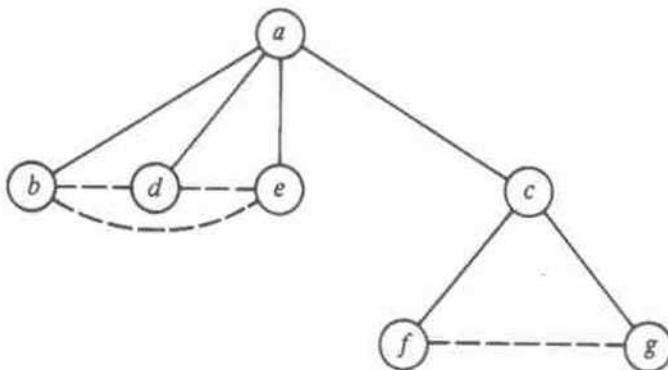


Fig. 7.13. Búsqueda en amplitud para G .

La complejidad de tiempo de la búsqueda en amplitud es la misma que para la búsqueda en profundidad. Cada vértice visitado se coloca en la cola una vez, así que el ciclo `while` se ejecuta una sola vez para cada vértice. Cada arista (x, y) se examina dos veces, desde x y desde y . Así, si el grafo tiene n vértices y a aristas, el tiempo de ejecución de `bea` es $O(\max(n, a))$ si se utiliza una representación con lista de adyacencia para las aristas. Dado que es típico que $a \geq n$, en general se hará referencia al tiempo de ejecución de la búsqueda en amplitud con $O(a)$, como ocurrió para la búsqueda en profundidad.

Las búsquedas en profundidad y en amplitud se pueden usar como marcos de trabajo, alrededor de los cuales se diseñan eficientes algoritmos para grafos. Por ejemplo, se puede emplear cualquiera de los dos métodos para encontrar los componentes conexos de un grafo, ya que aquéllos son los árboles de los dos bosques abarcadores.

Se puede verificar la existencia de ciclos por medio de la búsqueda en amplitud en un tiempo $O(n)$, donde n es el número de vértices, independientemente del número de aristas. Como se vio en la sección 7.1, cualquier grafo con n vértices y n o más aristas debe tener un ciclo. Sin embargo, un grafo puede tener $n - 1$ o menos aristas y de todos modos tener un ciclo, si tiene dos o más componentes conexos. Una forma segura de encontrar los ciclos es construir un bosque abarcador en amplitud. Así, toda arista cruzada (v, w) debe completar un ciclo simple con las aristas de árbol que conducen a v y w desde su antecesor común más cercano, como se muestra en la figura 7.14.

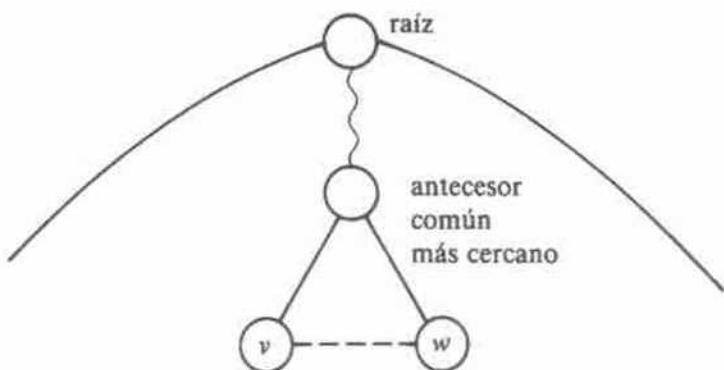


Fig. 7.14. Ciclo encontrado por la búsqueda en amplitud.

7.4 Puntos de articulación y componentes biconexos

Un *punto de articulación* de un grafo es un vértice v tal que cuando se elimina junto con todas las aristas incidentes sobre él, se divide un componente conexo en dos o más partes. Por ejemplo, los puntos de articulación de la figura 7.11(a) son a y c . Si se elimina a , el grafo, que es un componente conexo, se divide en dos triángulos: $\{b, d, e\}$ y $\{c, f, g\}$; si se elimina c , se divide en $\{a, b, d, e\}$ y $\{f, g\}$. Sin embargo, al eli-

minar cualquier otro vértice del grafo de la figura 7.11(a), el componente conexo no se dividirá. A un grafo sin puntos de articulación se le llama *biconexo*. La búsqueda en profundidad es muy útil para encontrar los componentes biconexos de un grafo.

El problema de encontrar los puntos de articulación es el más simple de muchos problemas importantes relacionados con la conectividad de grafos. Como ejemplo de las aplicaciones de los algoritmos de conectividad, se puede presentar una red de comunicaciones como un grafo en el que los vértices son lugares que hay que mantener comunicados entre sí. Un grafo tiene *conectividad k* si la eliminación de $k - 1$ vértices cualesquiera no lo desconecta. Por ejemplo, un grafo tiene conectividad dos o más si, y sólo si, no tiene puntos de articulación, es decir, si, y sólo si, es biconexo. Cuanto mayor sea su conectividad, tanto más fácil será que sobreviva al fallo de alguno de sus vértices, sea por fallo de las unidades de procesamiento colocadas en los vértices o por motivos externos.

Ahora se presenta un algoritmo simple de búsqueda en profundidad para encontrar todos los puntos de articulación de un grafo, y probar por medio de su ausencia, si el grafo es biconexo.

1. Realizar una búsqueda en profundidad del grafo, calculando $número_bp[v]$ para todo vértice v , como se analizó en la sección 6.5. En esencia, $número_bp$ ordena los vértices como en un recorrido en orden previo del árbol abarcador en profundidad.
2. Para cada vértice v , obtener $bajo[v]$, que es el $número_bp$ más pequeño de v o de cualquier otro vértice w accesible desde v , siguiendo cero o más aristas de árbol hasta un descendiente x de v (x puede ser v) y después seguir una arista de retroceso (x, w). Se calcula $bajo[v]$ para todos los vértices v , visitándolos en un recorrido en orden posterior. Cuando se procesa v , se ha calculado $bajo[y]$ para todo hijo y de v . Se toma $bajo[v]$ como el mínimo de
 - a) $número_bp[v]$,
 - b) $número_bp[z]$ para cualquier vértice z para el cual haya una arista de retroceso (v, z), y
 - c) $bajo[y]$ para cualquier hijo y de v .
3. Ahora se encuentran los puntos de articulación como sigue.
 - a) La raíz es un punto de articulación si, y sólo si, tiene dos o más hijos. Puesto que no hay aristas cruzadas, la eliminación de la raíz debe desconectar los subárboles cuyas raíces se encuentren en sus hijos, como a desconecta $\{b, d, e\}$ de $\{c, f, g\}$ en la figura 7.11(b).
 - b) Un vértice v distinto de la raíz es un punto de articulación si, y sólo si, hay un hijo w de v tal que $bajo[w] \geq número_bp[v]$. En este caso, v desconecta w y sus descendientes del resto del grafo. A la inversa, si $bajo[w] < número_bp[v]$, debe haber un camino para descender desde w en el árbol y regresar hasta un antecesor propio de v (el vértice cuyo $número_bp$ es $bajo[w]$) y, por tanto, la eliminación de v no desconecta w ni sus descendientes del resto del grafo.

Ejemplo 7.8. *número_bp* y *bajo* se calculan para el grafo de la figura 7.14(a) en la figura 7.15. Como ejemplo de la obtención de *bajo*, el recorrido en orden posterior visita *e* primero. En *e*, hay aristas de regreso (*e*, *a*) y (*e*, *b*), así que *bajo[e]* se iguala a $\min(\text{número_bp}[e], \text{número_bp}[a], \text{número_bp}[b]) = 1$. Después se visita *d*, y *bajo[d]* se hace igual al mínimo de *número_bp[d]*, *bajo[e]* y *número_bp[a]*. El segundo de éstos surge porque *e* es un hijo de *d*, y el tercero, por la existencia de la arista de retroceso (*d*, *a*).

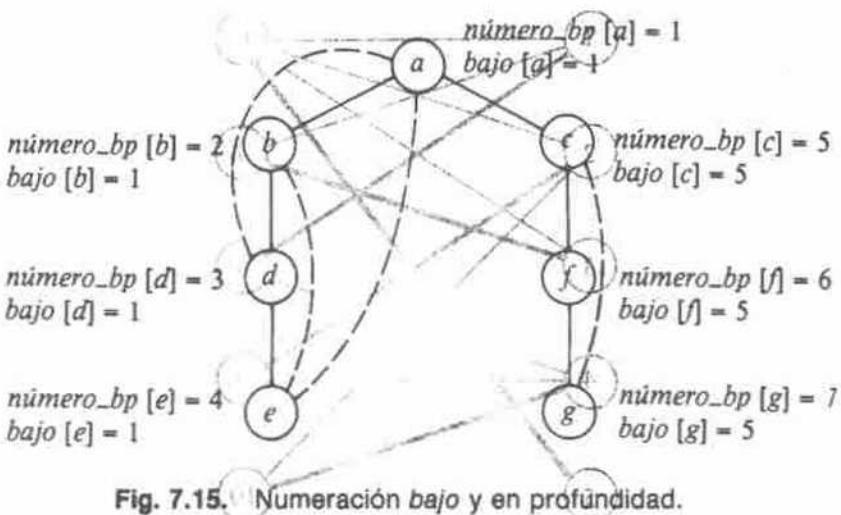


Fig. 7.15. Numeración *bajo* y en profundidad.

Después de obtener *bajo*, se considera cada vértice. La raíz *a* es un punto de articulación porque tiene dos hijos. El vértice *c* es un punto de articulación porque tiene un hijo *f* con *bajo[f] ≥ número_bp[c]*. Los otros vértices no son puntos de articulación. □

El tiempo que consume el algoritmo anterior en un grafo de a aristas y $n \leq a$ vértices es $O(a)$. Es recomendable comprobar que el tiempo empleado en cada una de las tres fases puede atribuirse al vértice visitado o a una arista que parte de ese vértice, y sólo se le puede atribuir una cantidad constante de tiempo a cualquier arista o vértice en cualquier paso. Así, el tiempo total es $O(n+a)$, el cual es $O(a)$ en el supuesto de que $n \leq a$.

7.5 Pareamiento de grafos

En esta sección se bosquejará un algoritmo para resolver «problemas de pareamiento» en grafos. Un ejemplo simple de problema de pareamiento ocurre cuando se tiene un conjunto de profesores para distribuir en un conjunto de cursos. Cada profesor es competente para impartir ciertos cursos, pero no otros. Se desea asignar un curso al profesor adecuado, pero sin asignar dos profesores al mismo curso. Para ciertas distribuciones de profesores y cursos, es imposible asignar un curso a cada profesor; en tal situación, es deseable asignar tantos profesores como sea posible.

Esto se representa con un grafo como el de la figura 7.16, donde los vértices están divididos en dos conjuntos V_1 y V_2 , de modo que los vértices del conjunto V_1 representan a los profesores, y los vértices en V_2 , los cursos. Que el profesor v sea adecuado para impartir el curso w se representa con una arista (v, w) . Un grafo como éste, cuyos vértices se pueden dividir en dos grupos disjuntos y las aristas presentan un extremo en cada grupo, se conoce como *bipartido*. Asignar un profesor a un curso es equivalente a seleccionar una arista entre un vértice profesor y un vértice curso.

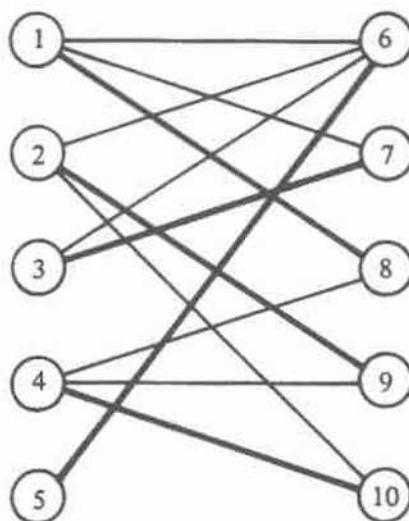


Fig. 7.16. Grafo bipartido.

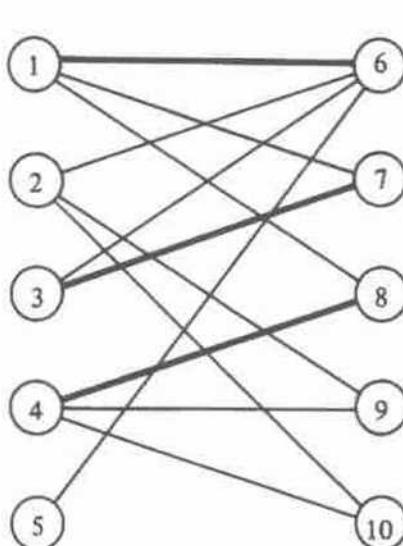
El problema del pareamiento se puede formular en términos generales como sigue. Dado un grafo $G = (V, A)$, el subconjunto de las aristas de A en el que ningún par de aristas es incidente sobre el mismo vértice de V , se conoce como *pareamiento*. La tarea de la selección de subconjuntos máximos de tales aristas se denomina *problema de pareamiento maximal*, y un ejemplo son las aristas más gruesas de la figura 7.16. Un *pareamiento completo* es aquel en el que todo vértice es un punto final de alguna arista en ella. Claramente, todo pareamiento completo es maximal.

Un modo directo de encontrar pareamientos maximales, es generar en forma sistemática todos los pareamientos y luego marcar uno que tenga el mayor número de aristas. La dificultad de este método radica en que tiene un tiempo de ejecución que es una función exponencial del número de aristas.

Existen algoritmos más eficientes para la obtención de pareamientos maximales. Esos algoritmos usan de ordinario una técnica conocida como «caminos aumentados». Sea C un pareamiento en un grafo G . Un vértice v está *pareado* si es el punto final de una arista de C . Un camino que conecte dos vértices no pareados, cuyas aristas alternas estén en C , se conoce como *caminos aumentados relativos a C*. Obsérvese que un camino aumentado debe tener longitud impar, y debe empezar y terminar con aristas que no estén en C . Obsérvese también que a partir de un camino aumentado A siempre es posible encontrar un pareamiento más amplio, suprimiendo de C

aquellas aristas que estén en A , y añadiendo después a C las aristas de A que no estaban inicialmente en C . Este pareamiento nuevo es $C \oplus A$, donde \oplus denota «o exclusivo» en conjuntos, esto es, el nuevo pareamiento que consta de aquellas aristas que están en C o en A , pero no en ambos.

Ejemplo 7.9. La figura 7.1(a) muestra un grafo y un pareamiento C que consta de las aristas gruesas $(1, 6)$, $(3, 7)$ y $(4, 8)$. El camino $2, 6, 1, 8, 4, 9$ de la figura 7.17(b) es un camino aumentado relativo a C . La figura 7.18 muestra el pareamiento $(1, 8)$, $(2, 6)$, $(3, 7)$, $(4, 9)$ obtenido al eliminar aquellas aristas de C que están en el camino, y al agregar después a C las otras aristas del camino. \square



(a) Pareamiento



(b) Camino aumentado



Fig. 7.17. Pareamiento y camino aumentado.

La observación clave es que C es un pareamiento maximal si, y sólo si, no existe un camino aumentado relativo a C . Esta observación es la base del algoritmo del pareamiento maximal.

Supóngase que C y D son pareamientos con $|C| < |D|$. ($|C|$ representa el número de aristas en C .) Para ver que $C \oplus D$ contiene un camino aumentado relativo a C , considérese el grafo $G' = (V, C \oplus D)$. Ya que C y D son pareamientos, cada vértice de V es un punto final de hasta una arista de C y un punto final de hasta una arista de D . Así, cada componente conexa de G' forma un camino simple (quizá un ciclo) con aristas alternando entre C y D . Cada camino que no sea un ciclo puede ser un camino aumentado relativo a C o un camino aumentado relativo a D , según

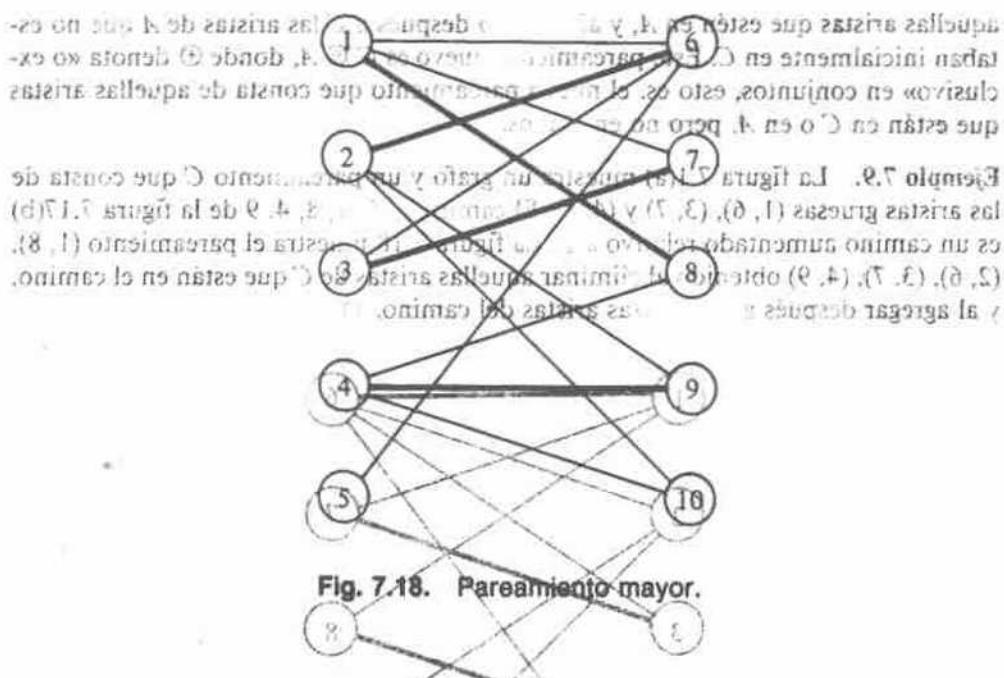


Fig. 7.18. Pareamiento mayor.

tenga más aristas de D o de C . Cada ciclo tiene un número igual de aristas de C y D . Ya que $|C| < |D|$, $C \oplus D$ tiene más aristas de D que de C , y de aquí que tenga por lo menos un camino aumentado relativo a C .

Ahora se puede plantear el procedimiento para encontrar un pareamiento maximal C para el grafo $G = (V, A)$.

1. Iniciar con $C = \emptyset$.
2. Encontrar un camino aumentado relativo a C y reemplazar C por $C \oplus A$.
3. Repetir el paso (2) hasta que ya no existan más caminos aumentados, punto en el que C es un pareamiento maximal.

Sólo falta mostrar cómo encontrar un camino aumentado relativo a un pareamiento C , y se hará para el caso más simple, en el que G es un grafo bipartido. Se construirá un grafo de *caminos aumentados* para G en los niveles $i = 0, 1, 2, \dots$ por medio de un proceso similar a la búsqueda en amplitud. En el nivel $i = 0$ se inicia con algún vértice sin pareamiento. En un nivel impar i , se agregan vértices nuevos adyacentes a un vértice en un nivel $i - 1$, a través de una arista no pareada, y también se agrega esa arista. En un nivel par i , se añaden vértices nuevos adyacentes a un vértice en un nivel $i - 1$ gracias a una arista del pareamiento C , junto con esa arista.

Se continúa construyendo el grafo de caminos aumentados nivel a nivel hasta que se agregue un vértice sin pareamiento en un nivel impar, o hasta que no se puedan agregar más vértices. Si un vértice sin pareamiento se agrega en un nivel impar, el camino existente entre v y el vértice inicial del nivel cero será un camino aumentado relativo a C . Si no hay más vértices para agregar al camino aumentado del grafo, se construye otro camino aumentado empezando en un nuevo vértice inicial sin pareamiento. Si no existen más vértices nuevos sin pareamiento, entonces no ha-

brá camino aumentado relativo a C . Si hay un camino aumentado, tarde o temprano se construirá un grafo de caminos aumentados que termina en un vértice sin pareamiento en un nivel impar.

Ejemplo 7.10. La figura 7.19 ilustra el grafo de caminos aumentados para la figura 7.17(a) relativo al pareamiento de la figura 7.18, en la cual se ha escogido 5 como el vértice sin pareamiento en el nivel 0. En el nivel 1 se agrega la arista sin pareamiento $(5, 6)$. En el nivel 2 se agrega la arista de pareamiento $(6, 2)$. En el nivel 3 se puede agregar cualquiera de las aristas sin pareamiento $(2, 9)$ o $(2, 10)$. Dado que los vértices 9 y 10 están actualmente sin pareamiento, se puede terminar la construcción del grafo de caminos aumentados después de incorporar cualquiera de esos vértices. Los caminos $9; 2, 6, 5$ y $10, 2, 6, 5$ son caminos aumentados relativos al pareamiento de la figura 7.18. \square

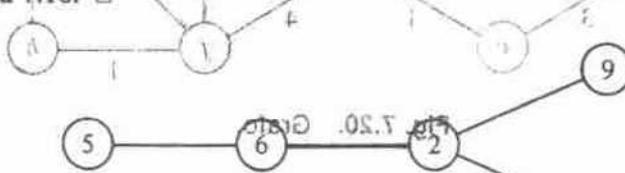


Figura 7.19. El grafo de caminos aumentados para el problema de pareamiento de la figura 7.17(a).

Figura 7.19. El grafo de caminos aumentados para el problema de pareamiento de la figura 7.17(a).

Supóngase que G tiene n vértices y m aristas. La construcción de los grafos de caminos aumentados para un pareamiento dado, lleva un tiempo $O(n)$ si se representan las aristas por medio de una lista de adyacencia. Así, encontrar cada camino aumentado nuevo lleva un tiempo $O(n)$. Para encontrar un pareamiento maximal, se construyen hasta $n/2$ caminos aumentados, ya que cada uno aumenta por lo menos en una arista el pareamiento actual. Por tanto, puede encontrarse un pareamiento maximal en un tiempo $O(n^2)$ para un grafo bipartido. \square

7.1 Describase un algoritmo para insertar y eliminar aristas en un grafo no dirigido representado por medio de una lista de adyacencia. Recuérdese que una arista (i, j) aparece en la lista de adyacencia de los vértices i y j .

7.2 Modifíquese la representación con lista de adyacencia de un grafo no dirigido, para que la primera arista de la lista de un vértice se pueda eliminar en un tiempo constante. Escribase un algoritmo para eliminar la primera arista en un vértice utilizando esta representación. *Sugerencia:* ¿Cómo se puede hacer que los dos enlaces que representan la arista (i, j) se encuentren una a la otra con rapidez?

7.3 Considérese el grafo de la figura 7.20. Encuéntrese,

- Un árbol abarcador de costo mínimo con el algoritmo de Prim.
- Un árbol abarcador de costo mínimo con el algoritmo de Kruskal.
- Un árbol abarcador en profundidad empezando en a y en d .
- Un árbol abarcador en amplitud empezando en a y en d .

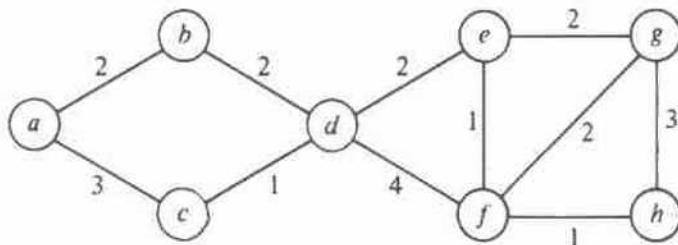


Fig. 7.20. Grafo.

7.4 Sea T un árbol abarcador en profundidad, y B las aristas de retroceso de un grafo conexo no dirigido $G = (V, A)$.

- *a) Demuéstrese que cuando cada arista de retroceso de B se agrega a T , se obtiene un ciclo único. Llámese tal ciclo un ciclo *básico*.
- **b) La *combinación lineal* de ciclos C_1, C_2, \dots, C_n es $C_1 \oplus C_2 \oplus \dots \oplus C_n$. Pruébese que se obtiene un ciclo de la combinación lineal de dos ciclos no disjuntos distintos.
- **c) Demuéstrese que todo ciclo en G puede expresarse como una combinación lineal de ciclos básicos.

*7.5 Sea $G = (V, A)$ un grafo, y R una relación en V tal que $u R v$ si, y sólo si, u y v están dentro de un ciclo común (no necesariamente simple). Pruébese que R es una relación de equivalencia en V .

7.6 Implántense los algoritmos de Prim y de Kruskal. Compárense los tiempos de ejecución de sus programas en una colección de grafos «aleatorios».

7.7 Escribábase un programa para encontrar todos los componentes conexos de un grafo.

7.8 Escribábase un programa de orden $O(n)$ para determinar si un grafo de n vértices contiene algún ciclo.

7.9 Escribábase un programa para enumerar todos los ciclos simples de un grafo. ¿Cuántos ciclos de este tipo puede haber? ¿Cuál es la complejidad en tiempo del programa?

7.10 Muéstrese que todas las aristas de una búsqueda en amplitud son de árbol o cruzadas.

- 7.11 Implántese el algoritmo para encontrar los puntos de articulación, analizado en la sección 7.4.
- *7.12 Sea $G = (V, A)$ un grafo *completo*, esto es, un grafo en el cual existe una arista entre todo par de vértices distintos. Sea $G' = (V, A')$ un grafo dirigido en el cual A' es A que da a cada arista una orientación arbitraria. Muéstrese que G' tiene un camino dirigido que incluye todos los vértices exactamente una vez.
- **7.13 Muéstrese que un grafo completo de n vértices tiene n^{n-2} árboles abarcadores.
- 7.14 Encuéntrense todos los pareamientos maximales para la figura 7.16.
- 7.15 Escribábase un programa para encontrar un pareamiento maximal para un grafo bipartido.
- 7.16 Sea C un pareamiento y c el número de aristas en un pareamiento maximal.
- Pruébese que existe un camino aumentado relativo a C cuya longitud es $2(|C|/c - |C|) + 1$ como máximo.
 - Pruébese que si A es el camino aumentado más corto relativo a C y si A' es un camino aumentado relativo a $C \oplus A$, entonces $|A'| \geq |A| + |A \cap A'|$.
- *7.17 Pruébese que un grafo es bipartido si, y sólo si, no tiene ciclos de longitud impar. Dése un ejemplo de un grafo no bipartido para el cual la técnica del grafo de caminos aumentados no funcione.
- 7.18 Sean C y D los pareamientos de un grafo bipartido. Pruébese que $C \oplus D$ tiene al menos $|C| - |D|$ vértices disjuntos en los caminos aumentados relativos a C .

Notas bibliográficas

Los métodos para construcción de árboles abarcadores minimales se han estudiado por lo menos desde Boruvka [1926]. Los dos algoritmos dados en este capítulo están basados en Kruskal [1956] y Prim [1957]. Johnson [1975] muestra cómo los árboles k -arios parcialmente ordenados se pueden usar para realizar el algoritmo de Prim. Cheriton y Tarjan [1978] y Yao [1975] presentan algoritmos $O(\text{alog } \log n)$ para árboles abarcadores. Tarjan [1981] presenta una historia y perspectiva completas de los algoritmos para árboles abarcadores.

Hopcroft y Tarjan [1973] y Tarjan [1972] popularizaron el uso de la búsqueda en profundidad en algoritmos para grafos. De ahí procede el algoritmo para componentes biconexos.

El pareamiento de grafos lo estudió Hall [1948], y los caminos aumentados, Berge [1957] y Edmonds [1965]. Hopcroft y Karp [1973] dan un algoritmo $O(n^{2.5})$ para pareamientos maximales en grafos bipartidos, y Micali y Vazirani [1980] dan un algoritmo $O(\sqrt{|V|} \cdot A)$ para pareamiento maximal en grafos generales. En Papadimitriou y Steiglitz [1982] hay un buen análisis del pareamiento general.

8

Clasificación

El proceso de clasificación u ordenamiento de una lista de objetos de acuerdo con algún orden lineal, como para números, es tan fundamental, y se realiza con tanta frecuencia, que justifica una revisión cuidadosa del tema. La clasificación se dividirá en dos partes: **interna** y **externa**. La clasificación interna se produce en la memoria principal del computador, donde es posible aprovechar la capacidad del acceso aleatorio en distintas formas. La clasificación externa es necesaria cuando el número de objetos a ordenar es demasiado grande para caber en la memoria principal. En ese caso, el «embotellamiento» suele ser la transferencia de datos entre el almacenamiento principal y el secundario, y es imprescindible mover grandes bloques de datos para lograr mayor eficiencia. El hecho de que sea más conveniente mover en un solo bloque los datos físicamente contiguos restringe las clases de algoritmos de clasificación externa posibles. La clasificación externa se cubrirá en el capítulo 11.

8.1 El modelo de clasificación interna

En este capítulo, se presentarán los principales algoritmos de clasificación interna. Los algoritmos más simples de ordinario requieren un tiempo $O(n^2)$ para clasificar n objetos y sólo son útiles para listas pequeñas. Uno de los algoritmos de clasificación más populares es la clasificación rápida (*quicksort*), que lleva en promedio un tiempo $O(n \log n)$. La clasificación rápida funciona muy bien para la mayor parte de las aplicaciones comunes; sin embargo, en el peor caso lleva un tiempo $O(n^2)$. Existe otro método, como la clasificación por montículos (*heapsort*) y la clasificación por intercalación (*mergesort*) que llevan un tiempo $O(n \log n)$ en el peor caso, aunque su comportamiento en el caso promedio quizá no sea tan bueno como el de la clasificación rápida. La clasificación por intercalación, sin embargo, es una buena elección para clasificación externa. Se considerarán otros algoritmos llamados «clasificación por cubetas» o «clasificación por urnas». Esos algoritmos operan sólo con clases especiales de datos, como los enteros elegidos de un intervalo limitado, pero cuando son aplicables son muy rápidos, pues requieren sólo un tiempo $O(n)$ en el peor caso.

zar cualquier tipo de clave para la cual estén definidas las relaciones «menor o igual que» o «menor que».

El problema de la *clasificación* consiste en ordenar una secuencia de registros de tal forma que los valores de sus claves formen una secuencia no decreciente. Esto es, dados los registros r_1, r_2, \dots, r_n , con valores de clave k_1, k_2, \dots, k_n , respectivamente, debe resultar la misma secuencia de registros en orden $r_{i_1}, r_{i_2}, \dots, r_{i_n}$, tal que $k_{i_1} \leq k_{i_2} \leq \dots \leq k_{i_n}$. No es necesario que todos los registros tengan valores distintos, ni que los registros con claves iguales aparezcan en un orden particular.

Se emplearán varios criterios para evaluar el tiempo de ejecución de un algoritmo de clasificación interna. La primera medición, y también la más común, es el número de pasos requeridos por el algoritmo para clasificar n registros. Otra medición frecuente es el número de comparaciones entre claves que debe efectuarse para clasificar n registros; esto resulta muy útil cuando la comparación entre un par de claves es una operación relativamente costosa, como sucede cuando las claves son grandes cadenas de caracteres. Si el tamaño del registro es grande, puede ser también conveniente contar las veces que debe moverse. La aplicación manual hace evidente la medida del costo apropiada.

8.2 Algunos esquemas simples de clasificación

Tal vez uno de los métodos de clasificación más simples que pueda haber es un algoritmo llamado «clasificación de burbuja» (*bubblesort*). La idea básica de la clasificación de burbuja es imaginar que los registros a ordenar están almacenados en un arreglo vertical. Los registros con claves menores son «ligeros» y suben. Se recorre varias veces el arreglo de abajo hacia arriba, y al hacer esto, si hay dos elementos adyacentes que no están en orden, esto es, si el «más ligero» está abajo, se invierten. El efecto producido por esta operación es que en el primer recorrido, el registro «más ligero» de todos, es decir, el registro que posee la clave con menor valor, sube hasta la superficie (o parte superior del arreglo). En el segundo recorrido, la segunda clave menor sube hasta la segunda posición, y así sucesivamente. En este recorrido no es necesario subir hasta la posición uno, porque ya se sabe que la clave menor se encuentra ahí. En general, el recorrido i no intenta pasar más allá de la posición i . Se presenta el esbozo del algoritmo en la figura 8.1, con el supuesto de que el arreglo A es un array[1.. n] of tipo_registro, y n es el número de registros. Aquí y en el resto del capítulo se supondrá que un campo llamado *clave* contiene el valor de la clave de cada registro.

(1) for i := 1 to n-1 do
(2) for j := n downto i+1 do
(3) if A[j].clave < A[j-1].clave then
(4) intercambia(A[j], A[j-1])

Fig. 8.1. Algoritmo de clasificación de burbuja.

El procedimiento *intercambia* se utiliza en varios algoritmos de clasificación y se define en la figura 8.2, si bien se obvió que *A* tiene que ser un array de registros.

```

procedure intercambia ( var x, y: tipo_registro )
{ intercambia cambia los valores de x e y }
var
    temp: tipo_registro;
begin
    temp := x;
    x := y;
    y := temp
end; { intercambia }

```

Fig. 8.2. El procedimiento *intercambia*.

Ejemplo 8.1. La figura 8.3 muestra una lista de volcanes famosos, y el año en que hicieron erupción.

NOMBRE	AÑO
Pelée	1902
Etna	1669
Krakatoa	1883
Agung	1963
Vesubio	79
Santa Elena	1980

Fig. 8.3. Volcanes famosos.

Para este ejemplo, se emplean las siguientes definiciones de tipos:

```

type
    tipo_clave = array[1..10] of char;
    tipo_registro = record
        clave: tipo_clave; {nombre del volcán}
        año: integer
    end;

```

El algoritmo de clasificación de burbuja de la figura 8.1, aplicado a la lista de la figura 8.3, clasifica la lista en orden alfabético de nombres, si la relación \leq en objetos con este tipo de claves es el orden lexicográfico habitual. En la figura 8.4, se muestran los cinco pasos dados por el algoritmo cuando $n=6$. Las líneas indican el punto sobre el cual se sabe que los nombres son los más pequeños (en orden alfabético) y ocupan el lugar correcto. Sin embargo, después de $i=5$, cuando todos excepto el último registro se han colocado en su lugar, el último también debe estar en el lugar correcto, y el algoritmo termina.

Al principio del primer recorrido, el Santa Elena sobrepasa a Vesubio, pero no a Agung. En el resto de este recorrido, Agung sube hasta la parte superior. En el se-

gundo recorrido, Etna sube a la posición 2. En el tercer recorrido, Krakatoa sobrepasa a Pelée, y la lista queda en orden lexicográfico; sin embargo, de acuerdo con el algoritmo de la figura 8.1, se hacen dos recorridos adicionales. □

Pelée	<u>Agung</u>	Agung	Agung	Agung	Agung
Etna	Pelée	<u>Etna</u>	Etna	Etna	Etna
Krakatoa	Etna	Pelée	<u>Krakatoa</u>	Krakatoa	Krakatoa
Agung	Krakatoa	Krakatoa	Pelée	<u>Pelée</u>	Pelée
Vesubio	Santa Elena	Santa Elena	Santa Elena	Santa Elena	<u>Santa Elena</u>
Santa Elena	Vesubio	Vesubio	Vesubio	Vesubio	Vesubio
inicial	después	después	después	después	después
	de $i=1$	de $i=2$	de $i=3$	de $i=4$	de $i=5$

Fig. 8.4. Recorridos de la clasificación de burbuja.

Clasificación por inserción

El segundo método de clasificación a considerar se denomina «clasificación por inserción», porque en el i -ésimo recorrido se «inserta» el i -ésimo elemento $A[i]$ en el lugar correcto, entre $A[1], A[2], \dots, A[i-1]$, los cuales fueron ordenados previamente. Despues de hacer esta inserción, se encuentran clasificados los registros colocados en $A[1], \dots, A[i]$. Esto es, se ejecuta

```
for  $i := 2$  to  $n$  do
    mover  $A[i]$  hacia la posición  $j \leq i$  tal que
         $A[i] < A[k]$  para  $j \leq k < i$ , y
         $A[i] \geq A[j-1]$  o  $j = 1$ 
```

Para facilitar el proceso de mover $A[i]$, es útil introducir un elemento $A[0]$, cuya clave tiene un valor menor que el de cualquier clave existente en $A[1], \dots, A[n]$. Se postulará la existencia de una constante $-\infty$ de tipo tipo_clave que es menor que la clave de cualquier registro que pueda aparecer en la práctica. Si ninguna constante $-\infty$ se puede utilizar con seguridad, se debe probar primero si $j = 1$ al decidir la inserción de $A[i]$ antes de la posición j , y si no, comparar $A[i]$ (que se encuentra ahora en la posición j) con $A[j-1]$. El programa completo se muestra en la figura 8.5.

```
(1)    $A[0].clave := -\infty;$ 
(2)   for  $i := 2$  to  $n$  do begin
(3)        $j := i;$ 
(4)       while  $A[j] < A[j-1]$  do begin
(5)           intercambia( $A[j], A[j-1]$ );
(6)            $j := j-1$ 
    end
end
```

Fig. 8.5. Clasificación por inserción.

Ejemplo 8.2. En la figura 8.6 se muestra la lista inicial de la figura 8.3 y el resultado de los recorridos sucesivos de la clasificación por inserción para $i = 2, 3, \dots$. Despues de cada recorrido está garantizado que los elementos programados de la línea estarán ordenados entre sí, aunque su orden no tenga relación con los registros encontrados bajo la línea, los cuales se insertarán despues. □

Fig. 8.6. Recorridos de la clasificación por inserción.

Clasificación por selección. Deben ser pasadas $n-1$ veces las n entradas de la lista para que se produzca la clasificación. La idea en que se sustenta la «claseificación por selección» también es elemental. En el i -ésimo recorrido se selecciona el registro con la clave más pequeña, entre $A[1], \dots, A[n]$, y se intercambia con $A[i]$. Como resultado, después de i pasadas, los i registros menores ocuparán $A[1], \dots, A[i]$, en el orden clasificado. Esto es, la clasificación por selección puede describirse por

for $i := 1$ to $n-1$ do

Ejemplo 8.3. En la figura 8.8 se muestran los recorridos de la clasificación por selección de la lista de la figura 8.3. Por ejemplo, en el recorrido 1, el último valor de indice_menor es 4, la posición de Agung, que se intercambia con Vesuvio en $A[1]$.

Las líneas de la figura 8.8 indican el punto sobre el cual se sabe que los elementos menores aparecen clasificados. Después de $n-1$ recorridos, el registro $A[n]$, Ve-subio en la figura 8.8, está también en el lugar correcto, ya que es el elemento que se sabe que no está entre los $n-1$ más pequeños.

Complejidad de tiempo de los métodos

Las clasificaciones de burbuja, por inserción y por selección llevan un tiempo $O(n^2)$, y llevarán un tiempo $O(n^2)$ en buena parte de las secuencias de entrada de n elementos.

as que sucede no se necesita intercambios (es decir, si la sucesión ya está ordenada).
clave_menor, tipo_clave; Si la clave menor encontrada actualmente en un
recorrido a través de $A[0:n-1]$ es menor que el valor inicial de i , se establece el valor
indice_menor igual a la posición de **clave_menor**.
begin
for $i = 0$ **to** $n - 1$ **do begin**
 elegir el menor entre $A[i:n-1]$ e intercambiárselo con $A[i]$
(1) **indice_menor** := i ; $i = i + 1$
(2) **indice_menor** := i ; $i = i + 1$
(3) **clave_menor** := $A[i].clave$;
(4) **for** $j = i + 1$ **to** $n - 1$ **do**
 comparar cada clave con la actual **clave_menor** de $A[j]$ a (b) , así que es
if $A[j].clave < clave_menor$ **then begin**
(5) **clave_menor** := $A[j].clave$;
(6) **end;**
(7) **indice_menor** := j ; $i = i + 1$
end;
end;

Fig. 8.7. Clasificación por selección.

Pelé	Agung	Agung	Agung	Agung	Agung
Etna	Etna	Etna	Etna	Etna	Etna
Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa
Agung	Pelé	Pelé	Pelé	Pelé	Pelé
Vesubio	Vesubio	Vesubio	Vesubio	Vesubio	Santa Elena
Santa Elena	Santa Elena	Santa Elena	Santa Elena	Santa Elena	Vesubio
inicial	después de $i=1$	después de $i=2$	después de $i=3$	después de $i=4$	después de $i=5$

Fig. 8.8. Recorridos de la clasificación por selección.

Considerese la clasificación de burbuja de la figura 8.1. Sin importar qué tipo de registro sea, *intercambia* lleva un tiempo constante. Así, las líneas (3) y (4) de la figura 8.1 consumen hasta c_1 unidades de tiempo para alguna constante c_1 . Por tanto, para un valor fijo de i , el ciclo de las líneas (2) a (4) lleva hasta $c_2(n-i)$ pasos, para alguna constante c_2 ; la última constante es algo mayor que c_1 para justificar los decrementos y pruebas de j . En consecuencia, el programa completo requiere

$$\Omega(1 + n^2) = \Omega(n^2)$$

que ($\sum_{i=1}^{n-1} c_2(i+1) = c_2(n+1) - c_2$) es $\frac{1}{2}c_2n^2 + (c_2 - \frac{1}{2}c_2)n + c_2$, o, lo más cercano, $\frac{1}{2}c_2n^2 + c_2n + c_2$. La constante c_2 es menor que c_1 (ya que $c_2 = c_1 + \text{constante}$), así que $\frac{1}{2}c_2n^2 + c_2n + c_2 < \frac{1}{2}c_1n^2 + c_1n + c_1$. De modo similar, la constante c_1 es menor que c_3 (ya que $c_1 = c_3 - \text{constante}$), así que $c_1n < c_3n$. Por tanto, la complejidad de tiempo de la clasificación de burbuja es $O(n^2)$. El algoritmo requiere $\Omega(n^2)$ pasos,

ya que aunque no se necesiten intercambios (es decir, si la entrada ya estuviera clasificada), la prueba de la línea (3) se ejecutaría $n(n - 1)/2$ veces.

A continuación, considérese la clasificación por inserción de la figura 8.5. El ciclo **while** de las líneas (4) a (6) de la figura 8.5 no puede llevar más de $O(i)$ pasos, ya que j toma un valor inicial i en la línea (3) y decrece en cada ciclo. El ciclo termina cuando $j = 1$, ya que $A[0]$ es $-\infty$, lo cual hace que la prueba de la línea (4) sea falsa cuando $j = 1$. Se puede concluir que el ciclo **for** de las líneas (2) a (6) lleva hasta $c \sum_{i=2}^n i$ pasos para alguna constante c . Esta suma es $O(n^2)$.

Sería aconsejable comprobar que si en un inicio el arreglo está clasificado en orden inverso, se ejecutará $i - 1$ veces el ciclo **while** de las líneas (4) a (6), así que la línea (4) se ejecuta $\sum_{i=2}^n (i - 1) = n(n - 1)/2$ veces. Por tanto, la clasificación por inserción requiere un tiempo $\Omega(n^2)$ en el peor caso. Se puede demostrar que este límite menor interno vale también para el caso promedio.

Por último, considérese la clasificación por selección de la figura 8.7. Se puede comprobar que el ciclo **for** interno de las líneas (4) a (7) lleva un tiempo $O(n - i)$, puesto que j va desde $i + 1$ hasta n . Así, el tiempo total que requiere el algoritmo es $c \sum_{i=1}^{n-1} (n - i)$, para alguna constante c . Esta suma, que es $cn(n - 1)/2$, se observa que es $O(n^2)$. Por el contrario, se puede mostrar que al menos la línea (5) se ejecuta $\sum_{i=1}^{n-1} \sum_{j=i+1}^n (1) = n(n - 1)/2$ veces, sin importar el valor inicial del arreglo A , así que la clasificación por selección lleva un tiempo $\Omega(n^2)$, tanto en el peor caso, como en el caso promedio.

Cuenta de intercambios

Si el tamaño de los registros es grande, el procedimiento *intercambia*, único lugar en los tres algoritmos donde se copian registros, llevará más tiempo que cualquier otro paso, como la comparación de claves o los cálculos en los índices del arreglo. Así, mientras que los tres algoritmos llevan un tiempo proporcional a n^2 , se pueden comparar con más detalle al contar las veces que se usa *intercambia*.

Para comenzar, la clasificación de burbuja ejecuta el paso de intercambio de la línea (4) en la figura 8.1

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

veces a lo sumo, o unas $n^2/2$ veces. Pero dado que la ejecución de la línea (4) depende del resultado obtenido en la prueba de la línea (3), cabe esperar que el número real de intercambios sea mucho menor que $n^2/2$.

De hecho, la clasificación de burbuja, en promedio, intercambia elementos exactamente la mitad de veces, así que el número de intercambios esperado, si todas las

secuencias iniciales tienen igual probabilidad, será aproximadamente $n^2/4$. Para corroborar esto, considérense dos listas iniciales de claves mutuamente inversas: $L_1 = k_1, k_2, \dots, k_n$ y $L_2 = k_n, k_{n-1}, \dots, k_1$. Un intercambio es la única forma de que k_i y k_j puedan cruzarse si están inicialmente fuera de orden; pero, k_i y k_j lo están sólo en una de las listas L_1 o L_2 . Así, el número total de intercambios ejecutados cuando se aplica la clasificación de burbuja a L_1 y L_2 es igual al número de pares de elementos, esto es, $\binom{n}{2}$ o $n(n - 1)/2$. Por tanto, el número de intercambios promedio para L_1 y L_2 es $n(n - 1)/4$ o unas $n^2/4$. Ya que todas las clasificaciones posibles pueden aparecerse con sus inversas, como sucedió con L_1 y L_2 , el número promedio de intercambios en todas las clasificaciones también será aproximadamente $n^2/4$.

El número de intercambios efectuados en la clasificación por inserción es, en promedio, idéntico al de la clasificación de burbuja. Aplicando el mismo argumento, cada par de elementos se intercambiará en una lista L o en su inversa, pero nunca en ambas.

Sin embargo, en caso de que *intercambia* sea una operación costosa, es fácil observar que la clasificación por selección es superior a las clasificaciones de burbuja y por inserción. La línea (8) de la figura 8.7 se encuentra fuera del ciclo interno del algoritmo de clasificación por selección, por lo que se ejecuta exactamente $n-1$ veces en un arreglo de longitud n . Como la línea (8) contiene la única llamada a *intercambia*, la velocidad de crecimiento en el número de intercambios en la clasificación por selección, que es $O(n)$, es menor que las tasas de crecimiento del número de intercambios de los otros dos algoritmos, que es $O(n^2)$. A diferencia de las clasificaciones de burbuja y por inserción, la clasificación por selección permite a los elementos «saltar» sobre grandes cantidades de elementos sin necesidad de intercambiarlos entre sí individualmente.

Cuando los registros sean grandes y los intercambios costosos, una estrategia muy útil es mantener un arreglo de apuntadores a registros por medio de cualquier algoritmo. Entonces se pueden intercambiar apuntadores en lugar de registros. Una vez que los apuntadores a registros se han dispuesto en el orden apropiado, los registros pueden disponerse en el orden clasificado final en un tiempo $O(n)$.

Limitaciones de los algoritmos simples

Se debe tener presente que los algoritmos mencionados en esta sección tienen un tiempo de ejecución $O(n^2)$, tanto en el peor caso como en el promedio. Así, para una n grande, ninguno de esos algoritmos se compara de modo favorable con los algoritmos $O(n\log n)$ que se analizarán en las siguientes secciones. El valor de n para el cual esos algoritmos más complejos se hacen mejores que los simples, depende de diversos factores, como la calidad del código objeto generado por el compilador, la máquina con que se ejecutan los programas y el tamaño de los registros que se deben intercambiar. La experimentación con un programa que registre tiempos de ejecución (*profiler*) es una buena forma de determinar el punto de ruptura. Una regla razonable es que a menos que n sea aproximadamente 100, puede ser una pérdida de tiempo implantar un algoritmo más complicado que los estudiados en esta sección. La clasificación de Shell, una generalización de la clasificación de burbuja, es un algo-

ritmo de clasificación $O(n^{1.5})$ simple, muy sencillo de ampliar y razonablemente eficiente para valores modestos de n . La clasificación de Shell se presenta en el ejercicio 8.3.

8.3 Clasificación rápida (quicksort)

El primer algoritmo $O(n\log n)$ que se estudia †, y tal vez el más eficiente para clasificación interna, recibe el nombre de «clasificación rápida» (*quicksort*). La esencia del método consiste en clasificar un arreglo $A[1], \dots, A[n]$ tomando en el arreglo un valor clave v como elemento *pivote*, alrededor del cual reorganizar los elementos del arreglo. Es de esperar que el pivote esté cercano al valor medio de la clave del arreglo, de forma que esté precedido por una mitad de las claves y seguido por la otra mitad. Se permutan los elementos del arreglo con el fin de que para alguna j , todos los registros con claves menores que v aparezcan en $A[1], \dots, A[j]$, y todos aquellos con claves v o mayores aparezcan en $A[j+1], \dots, A[n]$. Despues, se aplica recursivamente la clasificación rápida a $A[1], \dots, A[j]$ y a $A[j+1], \dots, A[n]$ para clasificar ambos grupos de elementos. Dado que todas las claves del primer grupo preceden a todas las claves del segundo grupo, todo el arreglo quedará ordenado.

Ejemplo 8.4. En la figura 8.9 se muestran los pasos recursivos que la clasificación rápida puede realizar para clasificar la secuencia de enteros 3, 1, 4, 1, 5, 9, 2, 6, 5, 3. En cada caso, se ha escogido como valor v al mayor de los dos valores distintos que se encuentran más a la izquierda. La recursión termina al descubrir que la porción del arreglo que se debe clasificar consta de claves idénticas. Se ha mostrado que cada nivel consta de dos pasos, uno antes de dividir cada subarreglo, y el segundo, despues. La reorganización de los registros que tiene lugar durante la división se explicará en seguida. □

Ahora se inicia el diseño del procedimiento recursivo *quicksort*(i, j) que opera en un arreglo A con elementos $A[1], \dots, A[n]$, definido de manera externa al procedimiento. *quicksort*(i, j) ordena desde $A[i]$ hasta $A[j]$ en el mismo lugar. En la figura 8.10 se muestra un esbozo del procedimiento. Obsérvese que si todos los elementos $A[i], \dots, A[j]$ tienen la misma clave, el procedimiento no afecta a A .

Se empieza desarrollando una función *encuentra_pivote* que obtiene la prueba de la línea (1) de la figura 8.10, para determinar si todas las claves $A[i], \dots, A[j]$ tienen el mismo valor. Si *encuentra_pivote* no encuentra dos claves distintas, devuelve 0. De otro modo, devuelve el índice de la mayor de las dos primeras claves diferentes, la cual se convierte en el elemento pivote. La función *encuentra_pivote* está escrita en la figura 8.11.

Luego, se aplica la línea (3) de la figura 8.10, donde se enfrenta el problema de la permutación de $A[i], \dots, A[j]$, en el mismo lugar ††, de manera que todas las cla-

† Técnicamente, la clasificación rápida es $O(n\log n)$ sólo en el caso promedio; en el peor caso es $O(n^2)$.

†† Podrían copiarse $A[i], \dots, A[j]$ y ordenarlos conforme se toman, para copiar el resultado otra vez en $A[i], \dots, A[j]$. No se hace así porque se malgasta espacio y llevará más tiempo que el método utilizado aquí.

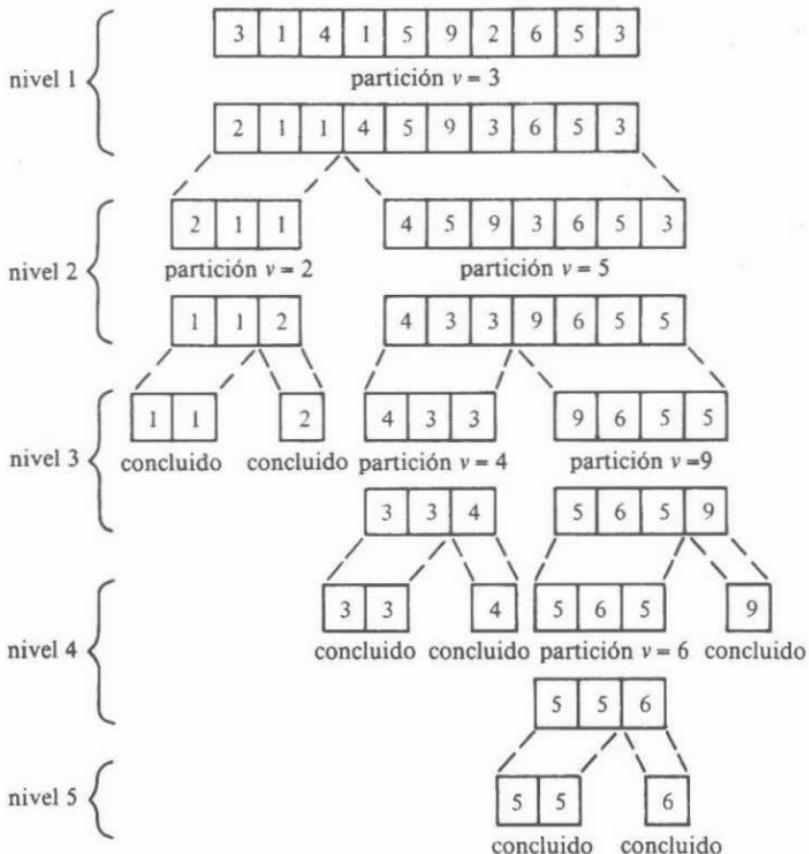


Fig. 8.9. Operación de la clasificación rápida.

ves menores que el valor del pivote aparecen a la izquierda de las demás. Para realizar esto, se introducen dos cursores, z y d , en un principio, en los extremos izquierdo y derecho, respectivamente, de la porción de A que se está clasificando. Los elementos a la izquierda de z , esto es, $A[i], \dots, A[z-1]$, siempre tendrán claves menores que el pivote. Los elementos a la derecha de d , esto es, $A[d+1], \dots, A[j]$, tendrán claves mayores o iguales al pivote, y los elementos del centro estarán mezclados, como se sugiere en la figura 8.12.

En un comienzo, $i = z$ y $j = d$ para que la proposición anterior se siga cumpliendo, ya que no existe nada a la izquierda de z ni a la derecha de d . Se efectúan varias veces los pasos siguientes, los cuales mueven z a la derecha y d a la izquierda, hasta que terminen cruzándose, momento en el que $A[i], \dots, A[z-1]$ tendrán todas las claves menores que el pivote y $A[d+1], \dots, A[j]$ todas las claves mayores o iguales que el pivote.

```

(1) if de  $A[i]$  a  $A[j]$  existen al menos dos claves distintas then begin
(2)     sea  $v$  la mayor de las dos claves distintas encontradas;
(3)     permutar  $A[i], \dots, A[j]$  de manera que para alguna  $k$  entre
            $i+1$  y  $j$ ,  $A[i], \dots, A[k-1]$  tengan claves menores que
            $v$  y los elementos  $A[k], \dots, A[j]$  tengan claves  $\geq v$ 
(4)     quicksort( $i, k-1$ );
(5)     quicksort( $k, j$ )
end

```

Fig. 8.10. Esbozo de la clasificación rápida.

```

function encuentra_pivote (  $i, j$ : integer ) : integer;
{ devuelve 0 si  $A[i], \dots, A[j]$  tienen claves idénticas; de otra forma, devuel-
ve el índice de la mayor de las dos claves diferentes a la izquierda }
var
    primera_clave: tipo_clave; { valor de la primera clave encontrada,
        es decir,  $A[i].clave$  }
     $k$ : integer; { va de izquierda a derecha buscando una clave diferente }
begin
    primera_clave :=  $A[i].clave$ ;
    for  $k := i + 1$  to  $j$  do { rastrea en busca de una clave distinta }
        if  $A[k].clave > primera\_clave$  then { selecciona la clave mayor }
            return ( $k$ )
        else if  $A[k].clave > primera\_clave$  then
            return ( $i$ );
    return (0) { nunca se encontraron dos claves diferentes }
end; { encuentra_pivote }

```

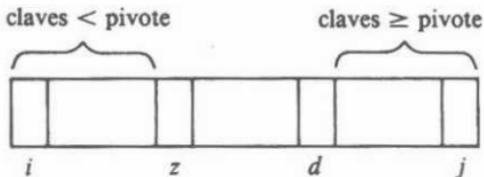
Fig. 8.11. Procedimiento *encuentra_pivote*.

Fig. 8.12. Situación durante el proceso de permutación.

1. *Rastrear.* Mueve z a la derecha en los registros cuyas claves sean menores que el pivote. Mueve d a la izquierda en las claves mayores o iguales que pivot. Obsérvese que esta selección del pivote por *encuentra_pivote* garantiza que por lo menos existe una clave menor y una no menor que el pivote, de modo que z y d con seguridad se detendrán antes de quedar fuera del intervalo de i a j .

2. *Probar.* Si $z > d$ (lo que en la práctica significa que $z = d + 1$), entonces se ha dividido $A[i], \dots, A[j]$ en forma satisfactoria, lo cual basta.
3. *Desviar.* Si $z < d$ (obsérvese que no se puede parar durante el rastreo con $z = d$, porque uno u otro se moverá más allá de una clave dada), se intercambia $A[z]$ con $A[d]$. Después de hacerlo, $A[z]$ tiene una clave menor que el pivote y $A[d]$ tiene una clave por lo menos igual que el pivote, y se sabrá que en la siguiente fase de rastreo z se moverá al menos una posición a la derecha, en la $A[d]$ anterior y d se moverá al menos una posición a la izquierda.

El ciclo anterior es poco apropiado, ya que la prueba que lo termina está justo en la mitad. Para ponerla en la forma de un ciclo **repeat**, se pasa la fase *desviar* al principio. El efecto es que inicialmente, cuando $i = z$ y $j = d$, se intercambia $A[i]$ con $A[j]$. Esto puede ser o no correcto, pero no es muy importante, pues se supone que en un principio no existe ningún orden en particular para las claves entre $A[i], \dots, A[j]$. Sin embargo, es necesario comprender este «truco» y no dejarse inquietar por él. La función *partición*, que realiza las operaciones anteriores y devuelve z , el punto en el cual empieza la mitad superior del arreglo dividido, se muestra en la figura 8.13.

En este momento ya es posible esbozar el programa para la clasificación rápida de la figura 8.10; el programa final se muestra en la figura 8.14. Para clasificar un arreglo A de tipo **array[1..n]** of tipo_registro sólo se llama a *quicksort(1, n)*.

```

function partición ( i, j: integer; pivot: tipo_clave ): integer;
  { divide  $A[i], \dots, A[j]$  para que las claves menores que pivot estén a la
    izquierda y las claves mayores o iguales que pivot estén a la derecha.
    Devuelve el lugar donde se inicia el grupo de la derecha. }

var
  z,d: integer; { cursores, como se describieron antes }

begin
  (1)   z := i;
  (2)   d := j;
  repeat
    (3)     intercambia(A[z], A[d]);
            { ahora se inicia la fase de rastreo }
    (4)     while A[z].clave < pivot do
      (5)       z := z + 1;
    (6)     while A[d].clave >= pivot do
      (7)       d := d - 1
    until
    (8)     z > d;
    return (z)
end; { partición }

```

Fig. 8.13. Procedimiento *partición*.

Tiempo de ejecución de la clasificación rápida

Ahora se mostrará que el algoritmo lleva en promedio un tiempo $O(n \log n)$ para clasificar n elementos, y que en el peor caso lleva $O(n^2)$. El primer paso en la demostración de ambas proposiciones es probar qué *partición* lleva un tiempo proporcional al número de elementos que deberá separar, es decir, un tiempo $O(j - i + 1)$.

```

procedure quicksort ( i, j: integer );
  { clasifica los elementos  $A[i], \dots, A[j]$  del arreglo externo  $A$  }
  var
    pivot: tipo_clave; { el valor del pivote }
    indice_pivot: integer; { el índice de un elemento de  $A$  donde clave es
                           el pivote }
    k: integer; { índice al inicio del grupo de elementos  $\geq$  pivot }
  begin
    (1)   indice_pivot := encuentra_pivot(i,j);
    (2)   if indice_pivot <> 0 then begin { no hacer nada si todas las claves
                                         son iguales }
    (3)     pivot := A[indice_pivot].clave;
    (4)     k := partición(i, j, pivot);
    (5)     quicksort(i, k - 1);
    (6)     quicksort(k, j)
  end
end; { quicksort }

```

Fig. 8.14. El procedimiento *quicksort*.

Para ver por qué la proposición es cierta, es preciso usar un truco muy frecuente en el análisis de algoritmos: encontrar ciertos «artículos» a los cuales se les pueda «cargar» el tiempo, y mostrar cómo cargar cada paso del algoritmo para que ningún artículo se cargue más que alguna constante. El tiempo total consumido, entonces, no será mayor que el producto de esta constante por el número de artículos.

En este caso, los artículos son los elementos desde $A[i]$ hasta $A[j]$, y a cada uno se le carga todo el tiempo gastado por *partición* desde el momento en que z o d apuntan hacia él por primera vez, hasta que dejan de hacerlo. Obsérvese primero que z y d nunca regresan a un elemento. A causa de que por lo menos existe un elemento en el grupo inferior y uno en el superior, y *partición* termina tan pronto como z excede a d , se sabe que cada elemento será cargado sólo una vez.

Un elemento se deja en los ciclos de las líneas (4) y (6) de la figura 8.13, ya sea por incremento de z o por decremento de d . ¿Cuánto tiempo puede transcurrir entre cada ejecución de $z := z + 1$ o $d := d - 1$? Lo peor que puede suceder es en el comienzo. Las líneas (1) y (2) asignan un valor inicial a z y d . Ahí se puede pasar por el ciclo sin hacer nada sobre z o d . En el segundo paso y en los siguientes, el intercambio de la línea (3) garantiza que los ciclos while de las líneas (4) y (6) se ejecutarán por lo menos una vez cada uno; así, lo peor que puede cargarse a una ejecución de $z := z + 1$ o $d := d - 1$ es el costo de las líneas (1), (2), dos veces (3), y las pruebas de

las líneas (4), (6), (8) y (4) otra vez. Esta es sólo una cantidad constante, independiente de i o j , y las siguientes ejecuciones de $z := z + 1$ o $d := d - 1$ se cargan menos: a lo sumo, una ejecución de las líneas (3) y (8) y un recorrido de los ciclos de las líneas (4) o (6).

Al final, también existen dos pruebas no satisfechas en las líneas (4), (6) y (8), que pueden no estar cargadas a ningún artículo, pero representan sólo una cantidad constante y pueden cargarse a cualquier artículo. Después de haber hecho todos los cargos, aún se tiene alguna constante c , por lo que ningún artículo ha sido cargado con más de c unidades de tiempo. Dado que existen $j - i + 1$ artículos, esto es, elementos en la porción del arreglo que se va a clasificar, el tiempo total empleado por $\text{partición}(i, j, \text{pivot})$ es $O(j - i + 1)$.

Ahora, considérese el tiempo de ejecución de $\text{quicksort}(i, j)$. Es fácil comprobar que el tiempo consumido por llamada a *encuentra_pivote* de la línea (1) de la figura 8.14 es $O(j - i + 1)$, y en muchos casos es bastante menor. La prueba de la línea (2) requiere una cantidad constante de tiempo, al igual que el paso (3) cuando se ejecuta. Se ha mostrado que la línea (4), que llama a *partición*, llevará un tiempo $O(j - i + 1)$. Así, con excepción de las llamadas recursivas que hace a *quicksort*, cada llamada individual de *quicksort* lleva un tiempo como máximo proporcional al número de elementos que se le pide clasificar.

En otras palabras, el tiempo total consumido por *quicksort* es la suma, en todos los elementos, de las veces que el elemento forma parte del subarreglo en el que se hizo la llamada a *quicksort*. Recuérdese la figura 8.9, donde se observan las llamadas a *quicksort* organizadas por niveles. Es evidente que ningún elemento puede incluirse en dos llamadas del mismo nivel, así que el tiempo consumido por *quicksort* puede expresarse como la suma en todos los elementos de la *profundidad*, o máximo nivel, en el cual se encuentra ese elemento. Por ejemplo, los unos de la figura 8.9 son de profundidad 3 y el 6 es de profundidad 5.

En el peor caso, podría suceder que en cada llamada a *quicksort* se seleccionara el peor pivote posible, por ejemplo, el mayor valor de las claves en el subarreglo que se está clasificando. Entonces se dividiría el subarreglo en dos subarreglos más pequeños, uno con sólo un elemento (el elemento que tuviera al pivote como clave), y el otro con todos los demás. Esa secuencia de particiones forma un árbol como el de la figura 8.15, donde r_1, r_2, \dots, r_n es la secuencia de registros en el orden creciente de las claves.

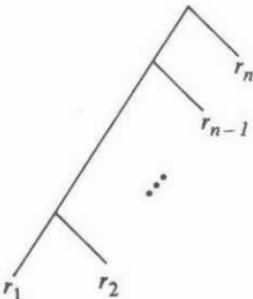


Fig. 8.15. Peor secuencia posible de selecciones de pivotes.

La profundidad de r_i es $n - i + 1$ para $2 \leq i \leq n$, y la profundidad de r_1 es $n - 1$. Así, la suma de las profundidades es

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

la cual es $\Omega(n^2)$. En el peor caso, la clasificación rápida requiere un tiempo proporcional a n^2 para clasificar n elementos.

Análisis del caso promedio de la clasificación rápida

Como siempre, se interpreta «caso promedio» para un algoritmo de clasificación como el promedio sobre todas las clasificaciones iniciales, dando igual probabilidad a cualquier clasificación posible. Por simplicidad, se supondrá que no existen dos elementos con claves iguales. En general, las igualdades entre elementos hacen la tarea de clasificación más fácil, nunca más difícil.

Una segunda suposición que hace más fácil el análisis del algoritmo de clasificación rápida es que, cuando se llama a *quicksort*(i, j), todas las clasificaciones para $A[i], \dots, A[j]$ son igualmente probables. La justificación es que antes de la llamada, no había pivotes con los cuales $A[i], \dots, A[j]$ se pudieran comparar para distinguirlos entre sí; es decir, para que todos fueran menores que el pivote v , o para que todos fueran mayores. Una revisión cuidadosa del programa desarrollado muestra la probabilidad de que cada elemento pivote concluya cerca del extremo derecho del subarreglo de elementos mayores o iguales que él, pero para subarreglos grandes, el hecho de que el elemento mínimo (el pivote previo) pueda aparecer cerca del extremo derecho no marca una diferencia considerable †.

Ahora, sea $T(n)$ el tiempo promedio consumido por la clasificación rápida para ordenar n elementos. Es evidente que $T(1)$ es alguna constante c_1 , ya que en un elemento esta clasificación no hace llamadas recursivas a sí mismas. Cuando $n > 1$, como se supone que todos los elementos tienen claves distintas, se sabe que la clasificación rápida tomará un pivote y dividirá el subarreglo, consumiendo un tiempo $c_2 n$ para hacerlo, para alguna constante c_2 , y después llamará a la clasificación en los dos subarreglos. Sería bueno poder pedir que el pivote tuviera la misma probabilidad de ser el primero, segundo, ..., n -ésimo elemento en el orden clasificado, para el subarreglo que se esté ordenando. Sin embargo, para garantizar que la clasificación rápida encuentre por lo menos una clave menor que cada pivote y al menos una igual o mayor que el pivote (de modo que cada fragmento sea menor que el total y, por tanto, no sean posibles los ciclos infinitos), siempre se escoge el mayor de los dos primeros elementos encontrados. Resulta que esta selección no afecta a la distribución de tamaños de los subarreglos, pero tiende a hacer los grupos izquierdos (aquejlos que son menores que el pivote) más grandes que los grupos derechos.

† Si hay una razón para creer que los ordenamientos no aleatorios de elementos pueden hacer que *quicksort* se ejecute más lentamente de lo esperado, el programa *quicksort* debería permutar aleatoriamente los elementos del arreglo antes de la clasificación.

Se hará el desarrollo de una fórmula para la probabilidad de que el grupo izquierdo tenga i de los n elementos, en el supuesto de que todos los elementos son distintos. Para que el grupo izquierdo tenga i elementos, el pivote debe ser el $(i+1)$ -ésimo elemento entre los n . El pivote, por el método de selección, podía haber estado en la primera posición, con alguno de los i menores como segundo, o bien pudo haber sido el segundo, con uno de los i menores como primero. La probabilidad de que cualquier elemento en particular, tal como el $(i+1)$ -ésimo, aparezca primero en una secuencia aleatoria, es $1/n$. Dado que apareció primero, la probabilidad de que el segundo elemento sea uno de los i elementos más pequeños, de los $n-1$ elementos restantes es $i/(n-1)$. Así, la probabilidad de que el pivote aparezca en la primera posición y sea el número $i+1$ de los n en el orden apropiado, es $i/n(n-1)$. En forma semejante, la probabilidad de que el pivote aparezca en la segunda posición y sea el número $i+1$ de los n en el orden clasificado es $i/n(n-1)$, así que la probabilidad de que el grupo izquierdo sea de tamaño i es $2i/n(n-1)$, para $1 \leq i < n$.

Ahora, se puede escribir una ecuación de recurrencia para $T(n)$.

$$T(n) \leq \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} [T(i) + T(n-i)] + c_2 n \quad (8.1)$$

Según la ecuación (8.1), el tiempo promedio requerido por la clasificación rápida es el tiempo, $c_2 n$, empleado fuera de las llamadas recursivas, más el tiempo promedio utilizado en las mismas. Este último tiempo se expresa en (8.1) como la suma, en todas las posibles i , de la probabilidad de que el grupo izquierdo tenga tamaño i (y, por tanto, el grupo derecho tiene tamaño $n-i$), multiplicado por el costo de las dos llamadas recursivas: $T(i)$ y $T(n-i)$, respectivamente.

La primera tarea es transformar la ecuación (8.1) de manera que se simplifique la sumatoria y, de hecho, tome la forma que tendría si se hubiera elegido un pivote verdaderamente aleatorio en cada paso. Para hacer la transformación, obsérvese que para una función $f(i)$ cualquiera, sustituyendo i por $n-i$ se puede probar

$$\sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i) \quad (8.2)$$

Al sustituir la mitad del lado izquierdo de (8.2) por el lado derecho, queda

$$\sum_{i=1}^{n-1} f(i) = \frac{1}{2} \sum_{i=1}^{n-1} (f(i) + f(n-i)) \quad (8.3)$$

Aplicando (8.3) a (8.1) con $f(i)$ igual a la expresión interna de la sumatoria de (8.1), se obtiene

$$\begin{aligned} T(n) &\leq \frac{1}{2} \sum_{i=1}^{n-1} \left\{ \frac{2i}{n(n-1)} [T(i) + T(n-i)] + \frac{2(n-i)}{n(n-1)} [T(n-i) + T(i)] \right\} + c_2 n \\ &\leq \frac{1}{n-1} \sum_{i=1}^{n-1} [T(i) + T(n-i)] + c_2 n \end{aligned} \quad (8.4)$$

A continuación, se aplica (8.3) a (8.4), con $f(i) = T(i)$. Esta transformación da

$$T(n) \leq \frac{2}{n-1} \sum_{i=1}^{n-1} T(i) + c_2 n \quad (8.5)$$

Obsérvese que (8.4) es la ecuación de recurrencia que se obtendría si todos los tamaños entre 1 y $n-1$ para el grupo izquierdo fueran igualmente probables. Así, tomar como pivote la mayor de dos claves en realidad no afecta a la distribución de tamaños. Se estudiarán recurrencias de esta forma con mayor detalle en el capítulo 9. Aquí se resolverá la recurrencia (8.5) proponiendo una solución y demostrando que funciona. La solución propuesta es que $T(n) \leq cn\log n$ para alguna constante c y toda $n \geq 2$.

Para demostrar que esta suposición es correcta, se efectúa una inducción sobre n . Para $n = 2$, sólo se observa que para alguna constante c , $T(2) \leq 2c\log 2 = 2c$. Para comprobar la inducción, se supone que $T(i) \leq ci\log i$ para $i < n$, y se sustituye esta fórmula por $T(i)$ en el lado derecho de (8.5) para demostrar que la cantidad resultante no es mayor que $cn\log n$. Así, (8.5) se convierte en

$$T(n) \leq \frac{2c}{n-1} \sum_{i=1}^{n-1} i \log i + c_2 n \quad (8.6)$$

Aquí es necesario dividir la sumatoria de (8.6) en términos menores, donde $i \leq n/2$, y, por tanto, $\log i$ no es mayor que $\log(n/2)$, que es $(\log n)-1$, y en términos mayores, donde $i > n/2$, y $\log i$ puede ser tan grande como $\log n$. Entonces (8.6) se convierte en

$$\begin{aligned} T(n) &\leq \frac{2c}{n-1} \left[\sum_{i=1}^{n/2} i \log i + \sum_{i=n/2+1}^{n-1} i \log i \right] + c_2 n \\ &\leq \frac{2c}{n-1} \left[\sum_{i=1}^{n/2} i(\log n - 1) + \sum_{i=n/2+1}^{n-1} i \log n \right] + c_2 n \\ &\leq \frac{2c}{n-1} \left[\frac{n}{4} \left(\frac{n}{2} + 1 \right) \log n - \frac{n}{4} \left(\frac{n}{2} + 1 \right) + \frac{3}{4} n \left(\frac{n}{2} - 1 \right) \log n \right] + c_2 n \\ &\leq \frac{2c}{n-1} \left[\left(\frac{n^2}{2} - \frac{n}{2} \right) \log n - \left(\frac{n^2}{8} + \frac{n}{4} \right) \right] + c_2 n \\ &\leq cn \log n - \frac{cn}{4} - \frac{cn}{2(n-1)} + c_2 n \end{aligned} \quad (8.7)$$

Al tomar $c \geq 4c_2$, la suma del segundo y cuarto términos de (8.7) no es mayor que cero. El tercer término de (8.7) hace una contribución negativa, así que de (8.7) es posible confirmar que $T(n) \leq cn\log n$, si $c = 4c_2$. Esto completa la demostración de que la clasificación rápida requiere un tiempo $O(n\log n)$ en el caso promedio.

Mejoras a la clasificación rápida

Quicksort es muy rápido, su tiempo promedio de ejecución es menor que el de todos los algoritmos de clasificación $O(n\log n)$ conocidos en la actualidad (en un factor constante, por supuesto). Es factible mejorar aún más el factor constante al tomar pivotes que dividan cada subarreglo en partes similares. Por ejemplo, al dividir siempre los subarreglos en partes iguales, cada elemento será de profundidad exactamente $\log n$, en el árbol de particiones semejante al de la figura 8.9. En comparación, la profundidad promedio de un elemento para *quicksort*, como se constituyó en la figura 8.14, es de cerca de $1.4\log n$. Así, cabe esperar un incremento en la velocidad de *quicksort* seleccionando los pivotes con cuidado.

Por ejemplo, se pueden escoger tres elementos de un subarreglo al azar y tomar el elemento medio como pivote. Se pueden tomar k elementos al azar para cualquier k , clasificarlos por una llamada recursiva a la clasificación rápida o por uno de los algoritmos más simples de la sección 8.2, y elegir el elemento medio, es decir, el elemento $[(k+1)/2]$ -ésimo como pivote †. Es un ejercicio interesante determinar el mejor valor de k como una función del número de elementos del subarreglo a clasificar. Si k es muy pequeña, se malgasta el tiempo, porque, en promedio, el pivote dividirá los elementos de forma desigual. Si k es muy grande, llevará demasiado tiempo encontrar el elemento medio de los k elementos.

Otra mejora de la clasificación rápida está relacionada con lo que sucede cuando se toman subarreglos pequeños. Recuérdese de la sección 8.2 que los métodos simples $O(n^2)$ son mejores que los métodos $O(n\log n)$ para n pequeñas. La pequeñez de n depende de muchos factores, como el tiempo empleado en una llamada recursiva, la cual es una propiedad de la arquitectura de la máquina y de la estrategia utilizada por el compilador para realizar las llamadas a procedimientos en el lenguaje en que se escribió el método de clasificación. Knuth [1973] sugiere 9 como el tamaño del subarreglo en el que *quicksort* debe llamar a un algoritmo de clasificación más simple.

Existe otra forma de «acelerar» *quicksort*, que en realidad es una forma de canjear espacio por tiempo; la misma idea es válida para cualquier otro algoritmo de clasificación. Si se tiene espacio disponible, se crea un arreglo de apuntadores a los registros del arreglo A . Se efectúan las comparaciones entre las claves de los registros apuntados, pero sin mover los registros; en vez de eso, se mueven los apuntadores a los registros de la misma forma que la clasificación rápida mueve los registros. Al final, los apuntadores, leídos de izquierda a derecha, apuntan a los registros

† Dado que sólo se desea la mediana y no la lista completa clasificada de k elementos, puede ser mejor usar uno de los algoritmos de la sección 8.7, que encuentran la mediana con rapidez.

en el orden deseado, y será relativamente fácil reordenar los registros de A en el orden correcto.

De esta forma, sólo se hacen n intercambios de registros, en lugar de $O(n\log n)$, lo cual significa una diferencia sustancial si los registros son grandes. Por el lado negativo, se requiere espacio adicional para el arreglo de apuntadores, y el acceso a las claves para efectuar las comparaciones es más lento que antes, ya que se debe seguir primero el apuntador, y luego ir al registro, para conseguir el campo de la clave.

8.4 Clasificación por montículos (*heapsort*)

En esta sección se desarrolla un algoritmo de clasificación llamado *clasificación por montículos* (*heapsort*) cuyo peor caso y el caso promedio son $O(n\log n)$. Este algoritmo puede expresarse en forma abstracta por medio de las cuatro operaciones de conjuntos INSERTA, SUPRIME, VACIA y MIN, presentadas en los capítulos 4 y 5. Supóngase que L es la lista de elementos que se va a clasificar y S es un conjunto de elementos de tipo tipo_registro que se usará para guardar los elementos conforme se clasifican. El operador MIN se aplica al campo de la clave de los registros; esto es, MIN(S) devuelve el registro en S cuya clave tiene el valor más pequeño. La figura 8.16 presenta el algoritmo de clasificación abstracto que se transformará en una clasificación por montículos.

```
(1)   for  $x$  en la lista  $L$  do
(2)     INSERTA( $x$ ,  $S$ );
(3)   while not VACIA( $S$ ) do begin
(4)      $y := \text{MIN}(S)$ ;
(5)     writeln( $y$ );
(6)     SUPRIME( $y$ ,  $S$ )
end
```

Fig. 8.16. Algoritmo abstracto de clasificación.

Los capítulos 4 y 5 analizaron varias estructuras de datos, como los árboles 2-3, que manejan cada operación en un tiempo $O(\log n)$ por operación, si los conjuntos nunca crecen más allá de n elementos. Si se supone que la lista L es de longitud n , el número de operaciones realizadas será n veces INSERTA, n veces SUPRIME, n veces MIN, y $n + 1$ pruebas VACIA. El tiempo total consumido por el algoritmo de la figura 8.16 es $O(n\log n)$, si se emplea una estructura de datos adecuada.

La estructura de datos de árbol parcialmente ordenado que se estudió en la sección 4.11, es adecuada para la realización de este algoritmo. Recuérdese que un árbol parcialmente ordenado puede representarse por un *montículo*, un arreglo $A[1], \dots, A[n]$, cuyos elementos tienen la propiedad de árbol parcialmente ordenado: $A[i].clave \leq A[2*i].clave$ y $A[i].clave \leq A[2*i + 1].clave$. Al considerar los elementos $2i$ y $2i + 1$ como los «hijos» del elemento en i , el arreglo forma un árbol binario equilibrado en el cual la clave del padre nunca excede a las claves de los hijos.

En la sección 4.11, se vio que el árbol parcialmente ordenado puede manejar las

operaciones INSERTA y SUPRIME_MIN en un tiempo $O(\log n)$ por operación. Mientras que el árbol parcialmente ordenado no puede manejar la operación general SUPRIME en un tiempo $O(\log n)$ (sólo encontrar un elemento arbitrario lleva un tiempo lineal en el peor caso), debe hacerse notar que en la figura 8.16, los únicos elementos que se eliminan son los encontrados como minimales. Así, las líneas (4) y (6) de la figura 8.16 pueden combinarse en una función SUPRIME_MIN que devuelve el elemento y . Con esto se puede obtener el algoritmo de la figura 8.16 con la estructura de datos árbol parcialmente ordenado de la sección 4.11.

Es necesaria una modificación más al algoritmo de la figura 8.16 para evitar imprimir los elementos conforme se eliminan. El conjunto S siempre estará almacenado como un montículo en la parte superior del arreglo A , como $A[1], \dots, A[i]$ si S tiene i elementos. Por la propiedad de árbol parcialmente ordenado, el elemento más pequeño estará siempre en $A[1]$. Los elementos que se van eliminando de S pueden almacenarse en $A[i+1], \dots, A[n]$, clasificados en orden inverso, es decir, con $A[i+1] \geq \dots \geq A[i+2] \geq \dots \geq A[n]$.[†] Como $A[1]$ debe ser el menor de $A[1], \dots, A[i]$, puede efectuarse la operación SUPRIME_MIN simplemente intercambiando $A[1]$ con $A[i]$. Ya que el nuevo $A[i]$ (el anterior $A[1]$) no es menor que $A[i+1]$ (o el anterior se habría eliminado de S antes que el último), se tienen $A[i], \dots, A[n]$ clasificados en orden decreciente. Se puede considerar que S se encuentra ocupando $A[1], \dots, A[i-1]$.

Dado que el $A[1]$ nuevo ($A[i]$ anterior) viola la propiedad de árbol parcialmente ordenado, debe descender en el árbol como en el procedimiento SUPRIME_MIN de la figura 4.23. Aquí se utiliza el procedimiento *empuja*, que se muestra en la figura 8.17, que opera sobre el arreglo A definido en forma externa. Mediante una secuencia de intercambios, *empuja* lleva el elemento $A[\text{primero}]$ hasta su lugar adecuado entre sus descendientes en el árbol. Para restaurar la propiedad de árbol parcialmente ordenado en el montículo, *empuja* se llama con *primero* = 1.

Las líneas (4) a (6) de la figura 8.16 funcionan de la siguiente manera. La selección del mínimo en la línea (4) es fácil: siempre se encuentra en $A[1]$ gracias a la propiedad de árbol parcialmente ordenado. En vez de imprimir en la línea (5), se intercambia $A[1]$ con $A[i]$, el último elemento del montículo actual. Esto simplifica la eliminación del elemento mínimo en el árbol parcialmente ordenado; sólo se disminuye i , el cursor que indica el fin del montículo actual. Entonces, se invoca *empuja* (1, i) para restituir la propiedad de árbol parcialmente ordenado al montículo, $A[1], \dots, A[i]$.

La prueba de ausencia de elementos de S realizada en la línea (3) de la figura 8.16, se hace probando el valor de i , el cursor que marca el fin del montículo actual. Sólo resta ahora considerar cómo trabajan las líneas (1) y (2). Se puede suponer que L está presente originalmente en $A[1], \dots, A[n]$, en algún orden dado. Para establecer inicialmente la propiedad de árbol ordenado, se llama *empuja* (j, n) para toda $j = n/2, n/2-1, \dots, 1$. Obsérvese que después de las llamadas a *empuja* (j, n), no se viola la propiedad de árbol parcialmente ordenado en $A[j], \dots, A[n]$, porque al empujar en el árbol un registro no se introducen nuevas violaciones, pues sólo se inter-

[†] Al final, se podría invertir el arreglo A , pero si se desea que A termine clasificado de menor a mayor, simplemente se aplica un operador SUPRIME_MAX en lugar de SUPRIME_MIN, y se ordena A parcialmente, de modo que un padre no tenga claves menores (en vez de mayores) que sus hijos.

cambia un registro violador con su hijo más pequeño. El procedimiento completo, llamado *heapsort*, se muestra en la figura 8.18.

Análisis de la clasificación por montículos

Examíñese ahora el procedimiento *empuja* para saber el tiempo que requiere. Una inspección de la figura 8.17 confirma que el cuerpo del ciclo *while* lleva un tiempo constante. Además, después de cada iteración, *r* tiene por lo menos el doble del valor que tenía. Así, puesto que *r* empieza igual a *primero*, después de *i* iteraciones se tiene, $r \geq \text{primero} \cdot 2^i$. Es seguro que $r > \text{último}/2$ si $\text{primero} \cdot 2^i > \text{último}/2$, esto es si

$$i > \log(\text{último}/\text{primero}) - 1 \quad (8.8)$$

```

procedure empuja (primero, último: integer);
  {supone que A[primero], ..., A[último] obedece la propiedad de árbol
  parcialmente ordenado, excepto, quizás, para el hijo de A[primero]. El
  procedimiento empuja A[primero] hasta que se restituye la propiedad
  de árbol parcialmente ordenado}

  var
    r: integer; { indica la posición actual de A[primero] }
  begin
    r := primero; { asignación de valores iniciales }
    while r <= último div 2 do
      if último = 2*r then begin { r tiene un hijo en 2*r }
        if A[r].clave > A[2*r].clave then
          intercambia (A[r], A[2*r]);
        r := último { fuerza la terminación del ciclo while }
      end
      else { r tiene dos hijos, los elementos ubicados en 2*r y 2*r+1 }
        if A[r].clave > A[2*r].clave and
            A[2*r].clave < A[2*r+1].clave then begin
              { intercambia r con su hijo izquierdo }
              intercambia (A[r], A[2*r]);
              r := 2*r
            end
        else if A[r].clave > A[2*r+1].clave and
            A[2*r+1].clave < A[2*r].clave then begin
              { intercambia r con su hijo derecho }
              intercambia (A[r], A[2*r+1]);
              r := 2*r+1
            end
        else { r no viola la propiedad de árbol parcialmente
              ordenado }
        r := último { para forzar la terminación del ciclo while }
    end; { empuja }
  
```

Fig. 8.17. El procedimiento *empuja*.

Aquí, el número de iteraciones del ciclo `while` en *empuja* es $\log(\text{último}/\text{primero})$ a lo sumo.

Dado que $\text{primero} \geq 1$ y $\text{último} \leq n$ en cada llamada que el algoritmo de la figura 8.18 hace a *empuja*, (8.8) dice que cada llamada a *empuja*, en la línea (2) o (5) de la figura 8.18, lleva un tiempo $O(\log n)$. Es evidente que el ciclo de las líneas (1) y (2) se ejecuta $n/2$ veces, así que el tiempo dedicado es $O(n \log n)$ †. También el ciclo de las líneas (3) a (5) se ejecuta $n - 1$ veces. Así un tiempo total $O(n)$ se consume en todas las repeticiones de *intercambia* en la línea (4), y $O(n \log n)$ durante las repeticiones de la línea (5). Así, el tiempo total gastado en el ciclo de las líneas (3) a (5) es $O(n \log n)$, y todo *heapsort* lleva un tiempo $O(n \log n)$.

```

procedure heapsort;
    {clasifica el arreglo  $A[1], \dots, A[n]$  en orden decreciente}

    var
        i: integer; {cursor hacia  $A$ }
    begin
        {establece inicialmente la propiedad de árbol
         parcialmente ordenado}
        for i := n div 2 downto 1 do
            empuja(i, n);
        for i := n downto 2 do begin
            intercambia(A[1], a[i]);
            {elimina el mínimo del frente del montículo}
            empuja(1, i-1)
            {restablece la propiedad de árbol parcialmente ordenado}
        end
    end; {heapsort}

```

Fig. 8.18. El procedimiento *heapsort*.

A pesar de su tiempo $O(n \log n)$ en el peor caso, *heapsort* llevará en promedio más tiempo que *quicksort*, en un pequeño factor constante. La clasificación por montículos tiene interés intelectual porque es el primer algoritmo $O(n \log n)$ en el peor caso que se ha estudiado. Es de utilidad práctica cuando no se desea clasificar los n elementos, sino sólo las k menores de entre ellos, con k mucho menor que n . Como se mencionó antes, las líneas (1) a (2) en realidad sólo requieren un tiempo $O(n)$. Si se realizan sólo k iteraciones de las líneas (3) a (5), el tiempo empleado en

† De hecho, este tiempo es $O(n)$, por un argumento más cuidadoso. Para j dentro del intervalo $n/2$ a $n/4 + 1$, (8.8) dice que sólo se necesita una iteración en el ciclo `while` de *empuja*. Para j entre $n/4$ y $n/8 + 1$, sólo dos iteraciones, y así sucesivamente. El número total de iteraciones cuando j está comprendida entre $n/2$ y 1 está acotado por $n/4*1 + n/8*2 + n/16*3 + \dots$. Obsérvese que el límite mejorado para las líneas (1) a (2) no implica un límite mejorado para *heapsort*; todo el tiempo se consume en las líneas (3) a (5).

ese ciclo es $O(k\log n)$. Así, *heapsort*, modificado para producir sólo los primeros k elementos, lleva un tiempo $O(n + k\log n)$. Si $k \leq n/\log n$, es decir, se desea como máximo una fracción $(1/\log n)$ de la lista completa clasificada, entonces el tiempo requerido es $O(n)$.

8.5 Clasificación por urnas (*binsort*)

Se plantea la cuestión de si son necesarios $\Omega(n\log n)$ pasos para clasificar n elementos. En la siguiente sección se mostrará que ése es el caso de los algoritmos de clasificación que no suponen algo acerca del tipo de datos de las claves, excepto que pueden ordenarse mediante alguna función que indica si el valor de alguna clave es «menor que» otro. En muchas ocasiones es posible clasificar en tiempos menores a $O(n\log n)$, siempre que se conozca algo especial acerca de las claves que se están clasificando.

Ejemplo 8.5. Supóngase que *tipo_clave* es entero, y que se sabe que los valores de las claves se encuentran en el intervalo de 1 a n , sin duplicados, donde n es el número de elementos. Entonces, si A y B son del tipo *array [1..n] of tipo_registro*, y los n elementos a clasificar están almacenados inicialmente en A , es posible colocarlos en orden en el arreglo B , por medio de

```
for i := 1 to n do
  B[A[i]. clave] := A[i];
```

(8.9)

Este código calcula el lugar que pertenece al registro $A[i]$ y lo coloca en él. El ciclo completo lleva un tiempo $O(n)$. Trabaja bien sólo cuando existe un único registro con clave v para todo valor de v entre 1 y n . Un segundo registro que tenga la clave v puede ubicarse también en $B[v]$, destruyendo el registro anterior con clave v .

Hay otras formas de clasificar en su sitio un arreglo A con claves 1, 2, ..., n en sólo un tiempo $O(n)$. Se visitan $A[1], \dots, A[n]$ por turno; si el registro de $A[i]$ tiene clave $j \neq i$, se intercambian $A[i]$ con $A[j]$. Si después del intercambio el registro con la clave k se encuentra en $A[i]$, y $k \neq i$, se intercambia $A[i]$ con $A[k]$, y así sucesivamente. Cada intercambio coloca algún registro donde le corresponde, después de lo cual no vuelve a moverse. Así, el siguiente algoritmo ordena A en su sitio, en un tiempo $O(n)$, a condición de que exista un registro con cada una de las claves 1, 2, ..., n .

```
for i := 1 to n do
  while A[i]. clave <> i do
    intercambia (A[i], A[A[i]. clave]);
```

□

El programa (8.9) dado en el ejemplo 8.5 es un caso simple de una «clasificación por urnas» (*binsort*), un proceso de clasificación donde se crea una urna para contener todos los registros con cierta clave. Se examina cada registro r a clasificar y se

coloca en la urna de acuerdo con el valor de la clave de r . En el programa (8.9), las urnas son los elementos del arreglo $B[1], \dots, B[n]$, y $B[i]$ es la urna para la clave cuyo valor es i . Se pueden usar elementos del arreglo como urnas en este caso simple, porque se sabe que nunca habrá más de un elemento en una urna. Más aún, no es necesario ensamblar las urnas en una lista clasificada, porque B sirve como tal lista.

En el caso general, sin embargo, a veces puede ser necesario almacenar más de un registro en una urna y enlazarlas (o *concatenarlas*) en el orden apropiado. En otras palabras, supóngase que, como siempre, $A[1], \dots, A[n]$ es un arreglo del tipo `tipo_registro`, y que las claves de los registros son del tipo `tipo_clave`. Sólo a efectos de esta sección, se supondrá que `tipo_clave` es un tipo enumerado, tal como `1..m` o `char`. Sea `tipo_lista` un tipo que representa listas de elementos de tipo `tipo_registro`; `tipo_lista` puede ser cualquiera de los tipos para listas mencionados en el capítulo 2, pero una lista enlazada es más efectiva, ya que se generarán listas de tamaño impredecible en cada urna; con todo, las longitudes totales de las listas estarán fijadas en n , y, por tanto, un arreglo de n celdas puede proporcionar las listas para las urnas según sea necesario.

Por último, sea B un arreglo del tipo `array[tipo_clave] of tipo_lista`. Entonces, B es un arreglo de urnas, que son listas (o, si se usa la representación enlazada de listas, encabezado de listas). B está indexada por `tipo_clave`, así que existe una urna para cada posible valor de clave. De esta forma se puede efectuar la primera generalización de (8.9): las urnas tienen capacidad arbitraria.

Ahora, es necesario considerar cómo se concatenarán las urnas. De manera abstracta, partiendo de las listas a_1, a_2, \dots, a_i y b_1, b_2, \dots, b_j , debe formarse la *concatenación*, que será $a_1, a_2, \dots, a_i b_i, b_1, b_2, \dots, b_j$. La realización de esta operación `CONCATENA(L_1, L_2)`, que reemplaza la lista L_1 por la concatenación $L_1 L_2$, puede efectuarse en cualquiera de las representaciones de listas estudiadas en el capítulo 2.

Sin embargo, por eficiencia, es útil tener un apuntador al último elemento en cada lista (o al encabezado si la lista se encuentra vacía), además de uno de encabezamiento. Esta modificación facilita la búsqueda del último elemento en la lista L_1 sin

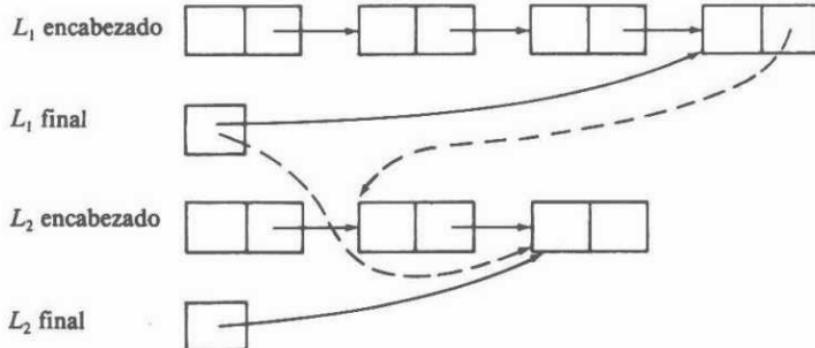


Fig. 8.19. Concatenación de listas enlazadas.

necesidad de recorrer toda la lista. La figura 8.19 muestra los apuntadores revisados, con líneas de puntos necesarios para concatenar L_1 y L_2 y obtener el resultado L_1 . Se supone que la lista L_2 «desaparece» después de la concatenación, en el sentido de que el encabezado y el apuntador final de L_2 se vuelven nulos.

Ahora se puede escribir un programa para clasificar colecciones arbitrarias de registros por medio de urnas, donde el campo de la clave es de un tipo enumerado. El programa de la figura 8.20 está escrito en función de primitivas de procesamiento de listas. Como ya se ha mencionado, una lista enlazada es la aplicación preferible, pero existen otras opciones. Recuérdese también que el ambiente para el procedimiento es que un arreglo A de tipo $\text{array}[1..n]$ of tipo_registro contiene los elementos a clasificar, y el arreglo B , de tipo $\text{array}[\text{tipo_clave}]$ of tipo_lista representa las urnas. Se supone que tipo_clave es expresable como $\text{clave_menor}..\text{clave_mayor}$, como debe ser cualquier tipo enumerado, para algunas cantidades clave_menor y clave_mayor .

```

procedure clasificación_por_urnas;
    { clasifica el arreglo A mediante urnas, dejando la lista ordenada en
      B[clave_menor] }

    var
        i: integer;
        v: tipo_clave;

    begin
        { coloca los registros en las urnas }
        for i := 1 to n do
            { mete A[i] al frente de la urna destinada a su clave }
            INSERTA(A[i], PRIMERO(B[A[i].clave]), B[A[i].clave]);
        for v := succ(lowkey) to clave_mayor do
            { concatena todas las urnas al final de B [clave_menor] }
            CONCATENA(B[clave_menor], B[v])
    end; { clasificación_por_urnas }

```

Fig. 8.20. Programa abstracto de clasificación por urnas.

Análisis de la clasificación por urnas

Se pretende que si hay n elementos a clasificar y m valores distintos de claves (y, por tanto, m urnas distintas), el programa de la figura 8.20 lleva un tiempo $O(n + m)$, si la estructura de datos empleada para las urnas es la adecuada. En particular, si $m \leq n$, la clasificación por urnas lleva un tiempo $O(n)$. La estructura de datos en cuestión es una lista enlazada. Los apuntadores a los finales de las listas, como se indica en la figura 8.19, son útiles, pero no indispensables.

El ciclo de las líneas (1) a (2) de la figura 8.20, que coloca los registros en las urnas, lleva un tiempo $O(n)$, debido a que la operación INSERTA de la línea (2) requiere un tiempo constante, pues la inserción siempre se hace al principio de la lista. Para

el ciclo de las líneas (3) a (4), que concatena las urnas, se supone temporalmente que existen apuntadores a los finales de las listas. El paso (4) emplea un tiempo constante, por lo que el ciclo lleva un tiempo $O(m)$. De aquí que el programa de clasificación por urnas completo lleve un tiempo $O(n + m)$.

Si los apuntadores a los finales de las listas no existen, en la línea (4) se debe consumir cierto tiempo para llegar al final de $B[v]$ antes de la concatenación con $B[clave_menor]$. De esta manera, el extremo de $B[clave_menor]$ siempre estará disponible para la siguiente concatenación. El tiempo adicional dedicado a llegar al final de cada urna, suma un tiempo de $O(n)$, debido a que la suma de las longitudes de las urnas es n . Este tiempo extra no afecta al orden de magnitud del tiempo de ejecución para el algoritmo, debido a que $O(n)$ no es mayor que $O(n + m)$.

Clasificación de grandes conjuntos de claves

Si m , el número de claves, no es mayor que n , el número de elementos, el tiempo de ejecución $O(n + m)$ del procedimiento de la figura 8.20 es en realidad $O(\cdot)$. Pero ¿qué sucedería si $m = n^2$? Evidentemente, la figura 8.20 necesitará $O(n + n^2)$, lo cual es $O(n^2)$. ¿Será posible aprovechar el hecho de que el conjunto de claves está limitado para optimizar el algoritmo? La respuesta sorprendente es que aun si el conjunto de claves posibles es 1, 2, ..., n^k , para cualquier k fija, existe una generalización de la técnica de clasificación por urnas, que requiere un tiempo de sólo $O(n)$.

Ejemplo 8.6. Considérese el problema específico de clasificación de n enteros en el intervalo de 0 a $n^2 - 1$. Se clasificarán los n enteros en dos fases; la primera parece que no es de mucha ayuda, pero es esencial. Se emplean n urnas, una para cada entero entre 0 y $n - 1$. Se coloca cada entero i de la lista a clasificar dentro de la urna numerada $i \bmod n$. Sin embargo, a diferencia de la figura 8.20, es importante agregar cada entero al final de la lista de la urna, no al principio. Si se quiere que la agregación sea eficiente, hace falta la representación de listas enlazadas para cada urna con apuntadores a los finales de las listas.

Por ejemplo, supóngase que $n = 10$, y que la lista a clasificar está constituida por los cuadrados perfectos desde 0^2 hasta 9^2 en el orden aleatorio 36, 9, 0, 25, 1, 49, 64, 16, 81, 4. En este caso, donde $n = 10$, la urna para el entero i sólo es el dígito más a la derecha de i escrito en decimal. La figura 8.21(a) muestra la colocación de esta lista en las urnas. Obsérvese que los enteros aparecen en las urnas en el mismo orden en el cual aparecen en la lista original; por ejemplo, la urna 6 contiene 36, 16, y no 16, 36, debido a que 36 precede a 16 en la lista original.

Ahora, se concatenan en orden las urnas, produciendo la lista

$$0, 1, 81, 64, 4, 25, 36, 16, 9, 49 \quad (8.10)$$

de la figura 8.21(a). Si se utiliza la estructura de datos de lista enlazada con apuntadores a los finales de las listas, la colocación de los n -enteros en las urnas y su concatenación pueden llevar un tiempo $O(n)$ cada una.

En la lista creada por la concatenación de las urnas, los enteros se redistribuyen en urnas, pero con una estrategia de selección diferente. Ahora se coloca el entero i dentro de la urna $[i/n]$, esto es, el máximo entero menor o igual que i/n . De nuevo, los enteros se agregan al final de las listas de las urnas. Al concatenar en orden las urnas, se observa que la lista está clasificada.

En el presente ejemplo, la figura 8.21 (b) muestra la lista (8.10) distribuida en las urnas con i contenido en la urna $[i/10]$.

Para ver por qué este algoritmo funciona, hay que observar que cuando varios enteros se colocan en una urna, como sucedió con 0, 1, 4 y 9, que se colocaron en la urna 0, deben estar en orden creciente, ya que la lista (8.10) resultante del primer recorrido los ordenó de acuerdo con el dígito más a la derecha. Así, en cualquier urna los dígitos de más a la derecha deben formar una secuencia creciente. Por supuesto, cualquier entero colocado en la urna i debe preceder a un entero colocado en una urna mayor que i , y al concatenar en orden las urnas se produce la lista ordenada.

<i>Urna</i>	<i>Contenido</i>	<i>Urna</i>	<i>Contenido</i>
0	0	0	0, 1, 4, 9
1	1, 81	1	16
2		2	25
3		3	36
4	64, 4	4	49
5	25	5	
6	36, 16	6	64
7		7	
8		8	81
9	9, 49	9	

(a)
(b)

Fig. 8.21. Clasificación por urnas en dos recorridos.

En forma más general, pueden considerarse los enteros entre 0 y $n^2 - 1$ como números de dos dígitos con base n y usar el mismo argumento para comprobar que la estrategia de clasificación funciona. Sean los enteros $i = an + b$ y $j = cn + d$, donde a, b, c y d se encuentran en el intervalo 0 a $n - 1$; esto es, son dígitos de base n . Supóngase que $i < j$, entonces $a > c$ no es posible, y debe suponerse que $a \leq c$. Si $a < c$, i aparece en una urna menor que j después del segundo recorrido, por lo que i precederá a j en el orden final. Si $a = c$, b debe ser menor que d . Después del primer recorrido, i precede a j , ya que i fue colocado en la urna b , y j , en la d . Así, aunque i y j se colocan en la misma urna a (la mitad que c), i se inserta antes que j en la urna. \square

Clasificación general por residuos (*radix sort*)

Supóngase que tipo_clave es una secuencia de campos, como en

```
type
  tipo_clave = record
    dia : 1..31;
    mes : (ene,...,dic);
    año : 1900..1999;
  end;
```

(8.11)

o un arreglo de elementos del mismo tipo, como en

```
type
  tipo_clave = array[1..10] of char;
```

(8.12)

Se supondrá de aquí en adelante que tipo_clave está constituido por k elementos, f_1, f_2, \dots, f_k de tipos t_1, t_2, \dots, t_k . Por ejemplo, en (8.11) $t_1 = 1..31$, $t_2 = (\text{ene}, \dots, \text{dic})$, y $t_3 = 1900..1999$. En (8.12), $k = 10$, y $t_1 = t_2 = \dots = t_k = \text{char}$.

También se supondrá de que se desea clasificar registros en orden *lexicográfico* de acuerdo con sus claves. Esto es, el valor de la clave (a_1, a_2, \dots, a_k) es menor que el valor de la clave (b_1, b_2, \dots, b_k) , donde a_i y b_i son los valores del campo f_i , para $i = 1, 2, \dots, k$, si

1. $a_1 < b_1$, o bien
2. $a_1 = b_1$ y $a_2 < b_2$, o bien

\vdots

- $k.$ $a_1 = b_1, a_2 = b_2, \dots, a_{k-1} = b_{k-1}$, y $a_k < b_k$.

Esto es, para alguna j entre 0 y $k - 1$, $a_1 = b_1, \dots, a_j = b_j$ †, y $a_{j+1} < b_{j+1}$.

Se pueden considerar las claves del tipo antes definido como si los valores de las claves fueran enteros expresados en alguna notación de residuos extraña. Por ejemplo, (8.12), donde cada campo es un carácter, puede considerarse como la expresión de enteros en base 128 o de tantos caracteres como haya en el conjunto de caracteres de la máquina empleada. La definición de tipo (8.11) puede considerarse como si el lugar de más a la derecha estuviera en base 100 (correspondiente a los valores entre 1900 y 1999), el siguiente lugar en base 12, y el tercero en base 31. Desde este punto de vista, la clasificación por urnas generalizada se conoce como clasificación *por residuos* (*radix sorting*). En último caso, se puede emplear para clasificar enteros hasta cualquier límite fijo, tomándolos como arreglos de dígitos de base 2 u otra.

La idea clave de la clasificación por residuos es ordenar por urnas todos los registros, primero en f_k , el «dígito menos significativo», después concatenar las urnas,

† Obsérvese que una secuencia que va desde 1 hasta 0 (o más generalmente, desde x hasta y , donde $y < x$) se considera una secuencia vacía.

primero el menor valor, clasificar de nuevo en f_{k-1} , y así sucesivamente. Como en el ejemplo 8.6, al insertar en las urnas hay que asegurarse de que cada registro se agrega al final de la lista, no al principio. El algoritmo de clasificación por residuos se esboza en la figura 8.22; la razón de su funcionamiento se ilustró en el ejemplo 8.6. En general, después de la clasificación por urnas en f_0, f_{k-1}, \dots, f_p , los registros aparecerán en orden lexicográfico si las claves constan sólo de los campos f_0, \dots, f_k .

Análisis de la clasificación por residuos

Primero se debe emplear la estructura de datos adecuada para hacer una clasificación eficiente. Obsérvese que se supone que la lista a clasificar ya está en forma de lista enlazada, no de arreglo. En la práctica, sólo es necesario agregar un campo adicional, el campo de enlace, al tipo tipo_registro, para poder enlazar $A[i]$ a $A[i+1]$ para $i = 1, 2, \dots, n-1$ y así hacer una lista enlazada del arreglo A en un tiempo $O(n)$. Obsérvese también que si se presentan en esta forma los elementos a clasificar, nunca se copia un registro. Sólo se cambian registros de una lista a otra.

```

procedure radixsort;
    { clasifica la lista A de n registros con claves que consisten en campos
       $f_1, \dots, f_k$  de tipos  $t_1, \dots, t_k$ , respectivamente. El procedimiento usa los
      arreglos  $B_i$  de tipo array [ $t_i$ ], of tipo_lista para  $1 \leq i \leq k$ , donde
      tipo_lista es una lista enlazada de registros. }

begin
    for  $i := k$  downto 1 do begin
        for cada valor  $v$  de tipo  $t_i$  do { limpia las urnas }
            vaciar  $B_i[v]$ 
        for cada registro  $r$  de la lista  $A$  do
            mover  $r$  desde  $A$  hasta el final de la urna  $B_i[v]$ , donde  $v$  es
            el valor del campo  $f_i$  de la clave de  $r$ 
        for cada valor  $v$  de tipo  $t_i$ , desde el menor hasta el mayor do
            concatena  $B_i[v]$  en el extremo de  $A$ 
    end
end; { radixsort }

```

Fig. 8.22. Clasificación por residuos.

Como antes, para hacer la concatenación con rapidez, se necesitan apuntadores al final de cada lista. Después, el ciclo de las líneas (2) y (3) de la figura 8.22 lleva un tiempo $O(s_i)$, donde s_i es el número de diferentes valores del tipo t_i . El ciclo de las líneas (4) y (5) lleva un tiempo $O(n)$, y el de las líneas (6) y (7), $O(s_i)$. Así, el tiempo total requerido por la clasificación por residuos es $\sum_{i=1}^k O(s_i + n)$, lo cual es $O(kn + \sum_{i=1}^k s_i)$, o $O(n + \sum_{i=1}^k s_i)$, si se toma k como una constante.

Ejemplo 8.7. Si las claves son enteros en el intervalo $0 \text{ a } n^k - 1$, para alguna constante k , se puede generalizar el ejemplo 8.6 y considerar las claves como enteros en base n con k dígitos de longitud. Entonces, s_i es $0..(n-1)$ para toda i entre 1 y k , así que $s_i = n$. La expresión $O(n + \sum_{i=1}^k s_i)$ se vuelve $O(n + kn)$ que, como k es una constante, es $O(n)$.

Como otro ejemplo, si las claves son cadenas de caracteres de longitud k , para la constante k , entonces $s_i = 128$, por ejemplo, para toda i , y $\sum_{i=1}^k s_i$ es una constante. Así, la clasificación por residuos en cadenas de caracteres de longitud fija también toma $O(n)$. De hecho, siempre que k es constante y los s_i son constantes, o simplemente $O(n)$, la clasificación por residuos lleva un tiempo $O(n)$. Sólo si k crece con n , puede ocurrir que no tome $O(n)$. Por ejemplo, si las claves se consideran como cadenas binarias de longitud $\log n$, entonces $k = \log n$, y $s_i = 2$ para $1 \leq i \leq k$. Así, la clasificación por residuos tomaría $O(kn + \sum_{i=1}^k s_i)$, lo cual es $O(n \log n)$ †. □

8.6 Cota inferior para la clasificación por comparaciones

Existe un «teorema heurístico» que dice que la clasificación de n elementos «requiere un tiempo $n \log n$ ». Se estudió en la sección anterior que esta proposición no siempre es cierta; si el tipo de la clave es tal que la clasificación por urnas o la clasificación por residuos puedan usarse con ventaja, el tiempo $O(n)$ es suficiente. Sin embargo, estos algoritmos de clasificación se basan en claves de tipos especiales: un tipo con un conjunto limitado de valores. Todos los demás algoritmos de clasificación estudiados cuentan sólo con el hecho de que se puede probar si una clave es menor que otra.

Se debe tener en cuenta que en todos los algoritmos de clasificación anteriores a la sección 8.5, la determinación del orden apropiado de los elementos se hace al comparar dos claves, de modo que el flujo de control del algoritmo siga uno de los dos caminos. En contraste, un algoritmo como el del ejemplo 8.5 hace que uno de n diferentes eventos suceda sólo en un paso, al almacenar un registro con una clave entera en una de las n urnas, dependiendo del valor de ese entero. Todos los programas de la sección 8.5 usan una posibilidad de los lenguajes de programación y de las máquinas que es mucho más poderosa que una simple comparación de valores; es la posibilidad de encontrar en un paso una localidad de un arreglo, dado el índice de esa localidad. Pero este poderoso tipo de operación no es factible si tipo-clave fuera, por ejemplo, real. No es posible, en Pascal ni en muchos otros lenguajes, declarar un arreglo indizado por números reales, y si lo fuera, no se podrían concatenar, en una cantidad de tiempo razonable, todas las urnas correspondientes a los números reales representables en la máquina.

† Pero en este caso, si los enteros de $\log n$ bits pueden ocupar una palabra, será mejor tratar las claves como si estuvieran compuestas de un solo campo, de tipo $1..n$, usando una clasificación por urnas ordinaria.

Arboles de decisión

Se tratarán ahora los algoritmos de clasificación que sólo usan los elementos a clasificar cuando comparan dos claves. Se puede dibujar un árbol binario en el cual los nodos representan el «estado» del programa después hacer un número de comparaciones de claves. También se puede considerar que un nodo representa las disposiciones iniciales de los elementos que llevarán el programa a este «estado». Así, un «estado» del programa es, en esencia, el conocimiento de la disposición inicial conseguida hasta ese punto por el programa.

Si cualquier nodo representa dos o más disposiciones iniciales posibles, el programa todavía no puede conocer el orden correcto, por lo que debe hacer otra comparación de claves, como «¿es $k_1 < k_2?$ ». Entonces se pueden crear dos hijos para el nodo; el hijo izquierdo representa aquellas disposiciones iniciales consistentes con el hecho de $k_1 < k_2$, y el hijo derecho representa las disposiciones consistentes con el hecho de que $k_1 > k_2$.[†] Así, cada hijo representa un «estado» que contiene la información conocida en el padre, más el hecho de que $k_1 < k_2$ o que $k_1 > k_2$, dependiendo de si el hijo es izquierdo o derecho.

Ejemplo 8.8. Considérese el algoritmo de clasificación por inserción con $n = 3$. Supóngase que al principio, $A[1]$, $A[2]$ y $A[3]$ tienen claves con valores a , b y c , respectivamente. Cualquiera de las seis disposiciones de a , b y c puede ser la correcta, así que se empieza la construcción del árbol de decisión con el nodo etiquetado (1) de la figura 8.23, el cual representa todas las posibles disposiciones. El algoritmo de clasificación por inserción compara primero $A[2]$ con $A[1]$, esto es, b con a . Si b resultara ser el más pequeño, la disposición correcta sólo puede ser bac , bca o cba , las tres disposiciones en las cuales b precede a a . Esas tres disposiciones están representadas por el nodo (2) de la figura 8.23. Las otras tres disposiciones están representadas por el nodo (3), el hijo derecho de (1), y son las disposiciones para las cuales a precede a b .

Ahora considérese lo que sucede si el dato inicial es tal que se alcanza el nodo (2). Sólo se ha intercambiado $A[1]$ con $A[2]$, y se encontró que $A[2]$ no puede subir más ya que se encuentra en la «cima». La disposición actual de los elementos es bac . La clasificación por inserción empieza a continuación insertando $A[3]$ en el lugar adecuado, y comparando $A[3]$ con $A[2]$. Dado que $A[2]$ contiene ahora a , y $A[3]$ contiene c , y comparando c con a ; el nodo (4) representa las dos disposiciones del nodo (2) en las cuales c precede a a , mientras que (5) representa la disposición donde esto no sucedió.

El algoritmo terminará si alcanza el estado del nodo (5), ya que ha movido a $A[3]$ lo más arriba posible. Por otra parte, en el estado del nodo (4), se tiene que $A[3] < A[2]$, por lo que se intercambiaron, dejando b en $A[1]$ y c en $A[2]$. La clasificación por inserción compara esos dos elementos y los intercambia si a continuación $c < b$. Los nodos (8) y (9) representan las disposiciones consistentes con $c < b$ y su opuesto, respectivamente, así como la información recogida de los nodos (1) a

[†] También es posible suponer que todas las claves son diferentes, ya que si se clasifica una colección de claves distintas, con seguridad se producirá un orden correcto cuando alguna clave esté repetida.

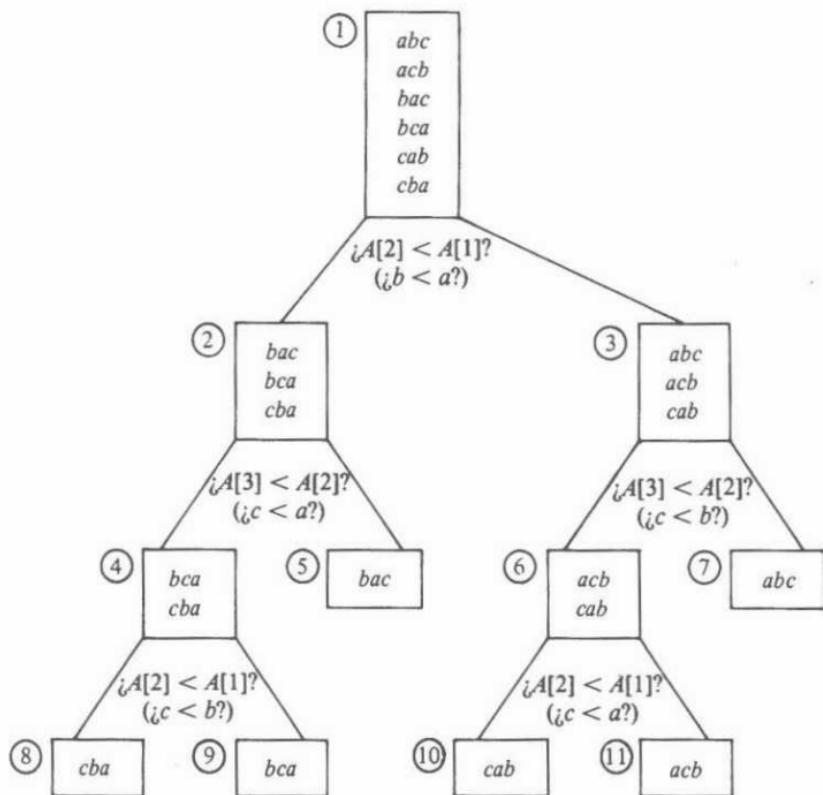


Fig. 8.23. Árbol de decisión para la clasificación por inserción con $n = 3$.

(2) a (4), es decir, $b < a$ y $c < a$. La clasificación por inserción termina, se haya intercambiado $A[2]$ con $A[1]$ o no, y no se efectúan más comparaciones. Por fortuna, cada hoja (5), (8) y (9) representa disposiciones individuales, así que se ha acumulado suficiente información para determinar la clasificación correcta.

La descripción del árbol que desciende del nodo (3) es bastante simétrica con lo que se ha visto, por lo que se omite. La figura 8.23 se emplea como guía para determinar el orden correcto de las llaves a , b y c , puesto que todas sus hojas están asociadas con una disposición individual.

Tamaño de los árboles de decisión

La figura 8.23 tiene seis niveles, que corresponden a las seis posibles disposiciones de la lista inicial a , b , c . En general, si se clasifica una lista de n elementos, existen $n! = n(n - 1)(n - 2)\dots(2)(1)$ resultados posibles, los cuales se encuentran en el orden

correcto para la lista inicial a_1, a_2, \dots, a_n . Esto es, cualquiera de los n elementos puede estar en primer lugar, cualquiera de los $n - 1$ elementos puede estar en segundo lugar, cualquiera de los $n - 2$ puede estar en tercer lugar, y así sucesivamente. De esta forma, cualquier árbol de decisión que describa un algoritmo de clasificación correcto y que trabaje en una lista de n elementos, debe tener por lo menos $n!$ hojas, ya que cada disposición posible debe estar sola en una hoja. De hecho, si se eliminan nodos correspondientes a comparaciones innecesarias y hojas que no corresponden a disposiciones posibles (ya que las hojas pueden alcanzarse sólo por una serie inconsistente de resultados de comparaciones), habrá exactamente $n!$ hojas.

Los árboles binarios con muchas hojas deben tener caminos largos. La longitud de un camino desde la raíz hasta una hoja proporciona el número de comparaciones hechas cuando la disposición representada por esa hoja es el orden clasificado para cierta lista de entrada L . Así, la longitud del camino más largo desde la raíz hasta una hoja es una cota inferior en el número de pasos efectuados por el algoritmo en el peor caso. En especial, si L es la lista de entrada, el algoritmo efectuará al menos tantas comparaciones como la longitud del camino, probablemente en adición a otros pasos que no sean comparaciones de claves.

Por tanto, es necesario preguntarse lo cortos que pueden ser todos los caminos en un árbol binario con k hojas. Un árbol binario en el que todos los caminos son de longitud p o menor, pueden tener una raíz, dos nodos en el nivel 1, cuatro nodos en el nivel 2 y, en general, 2^i nodos en el nivel i . Así, el número mayor de hojas de un árbol sin nodos en los niveles más altos que p es 2^p . Es decir, un árbol binario con k hojas debe tener un camino de longitud no menor que $\log k$. Si $k = n!$, entonces cualquier algoritmo de clasificación que sólo utilice comparaciones para determinar el orden clasificado, en el peor caso debe requerir un tiempo $\Omega(\log(n!))$.

Pero ¿con qué rapidez crece $\log(n!)$? Una aproximación cercana a $n!$ es $(n/e)^n$, donde $e = 2.7183\dots$ es la base de los logaritmos naturales. Dado que $\log((n/e)^n) = n\log n - n\log e$, se observa que $\log(n!)$ es de orden $n\log n$. Es posible tomar una cota inferior precisa observando que $n(n-1)\dots(2)$ (1) es el producto de por lo menos $n/2$ factores, que a su vez son cada uno al menos $n/2$. Así, $n! \geq (n/2)^{n/2}$; de aquí que $\log(n!) \geq (n/2)\log(n/2) = (n/2)\log n - n/2$, por lo que la clasificación por comparaciones requiere un tiempo $\Omega(n\log n)$ en el peor caso.

Análisis del caso promedio

¿Puede haber algún algoritmo que sólo use comparaciones para clasificar, y requiera un tiempo $\Omega(n\log n)$ en el peor caso, como todos los algoritmos de este tipo, pero que el tiempo promedio requerido sea $O(n)$ o algo menor que $O(n\log n)$? La respuesta es no, y sólo se mencionará cómo probar la aseveración, dejando los detalles al lector.

Lo que se desea probar es que en cualquier árbol binario con k hojas, la profundidad promedio de una hoja es por lo menos $\log k$. Supóngase que no fuera así, y el árbol T fuera el contraejemplo con menos nodos. T no puede ser un solo nodo, porque la aseveración dice que los árboles de una hoja tienen profundidad promedio

por lo menos de 0. Ahora, supóngase que T tiene k hojas. Un árbol binario con $k \geq 2$ hojas es semejante a los árboles de las figuras 8.24(a) o (b).

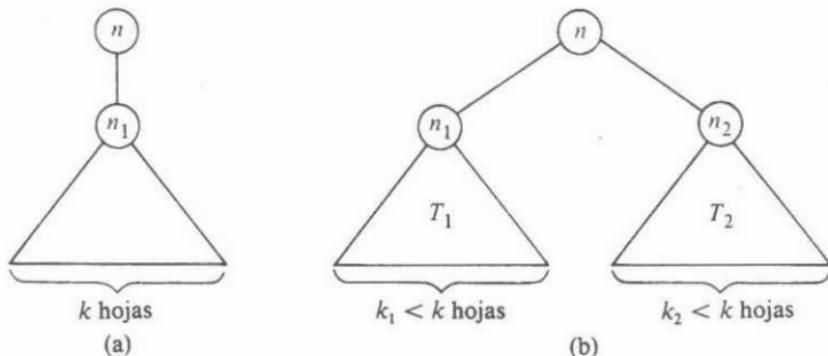


Fig. 8.24. Formas posibles del árbol binario T .

La figura 8.24(a) no puede ser el contraejemplo más pequeño, porque el árbol cuya raíz es n_1 tiene tantas hojas como T , pero una profundidad promedio más pequeña. Si la figura 8.24(b) fuera T , los árboles cuyas raíces son n_1 y n_2 , al ser más pequeños que T , no violarían la suposición. Esto es, la profundidad promedio de las hojas de T_1 es al menos $\log(k_1)$, y la profundidad promedio de T_2 es por lo menos $\log(k_2)$. Entonces, la profundidad promedio de T es

$$\left(\frac{k_1}{k_1+k_2} \right) \log(k_1) + \left(\frac{k_2}{k_1+k_2} \right) \log(k_2) + 1.$$

Como $k_1 + k_2 = k$, la profundidad promedio se expresa como

$$\frac{1}{k} (k_1 \log(2k_1) + k_2 \log(2k_2)) \quad (8.13)$$

Se puede comprobar que cuando $k_1 = k_2 = k/2$, (8.13) tiene valor $k \log k$, y se debe demostrar que (8.13) tiene un mínimo cuando $k_1 = k_2$, dada la restricción $k_1 + k_2 = k$. Se deja esta demostración como ejercicio a quien tenga experiencia en cálculo diferencial. Si se asume que (8.13) tiene un valor mínimo de $k \log k$, se observa que T no fue un contraejemplo.

8.7 Estadísticas de orden

El problema del cálculo de *estadísticas de orden* consiste en encontrar la clave del k -ésimo registro en el orden clasificado de los registros, dada una lista de n registros y un entero k . En general, este problema se plantea como «encontrar el k -ésimo en-

tre n . Ocurren casos especiales cuando $k = 1$ (encontrar el mínimo), $k = n$ (encontrar el máximo), y el caso en que n es impar y $k = (n + 1)/2$, que es encontrar la *mediana*.

Ciertos casos del problema son muy fáciles de resolver en tiempo lineal. Por ejemplo, encontrar el mínimo de n elementos en un tiempo $O(n)$ no requiere nada especial. Como se mencionó en relación con la clasificación por montículos, si $k \leq n/\log n$, es posible encontrar el k -ésimo de n construyendo un montículo, que lleva un tiempo $O(n)$, y después seleccionar los k elementos más pequeños en un tiempo $O(n + k\log n) = O(n)$. Sistématicamente, se puede encontrar el k -ésimo de n en un tiempo $O(n)$ cuando $k \geq n - n/\log n$.

Una variante de la clasificación rápida

Tal vez la forma más rápida para encontrar el k -ésimo entre n , en promedio, es usar un procedimiento recursivo similar a la clasificación rápida, que se puede llamar *selecciona* (i, j, k), y que encuentra el k -ésimo elemento entre $A[i], \dots, A[j]$ dentro de un arreglo más grande $A[i], \dots, A[n]$. Los pasos básicos de *selecciona* son:

1. Tomar un elemento pivot, como v .
2. Usar el procedimiento *partición* de la figura 8.13 para dividir $A[i], \dots, A[j]$ en dos grupos: $A[i], \dots, A[m - 1]$ con claves menores que v , y $A[m], \dots, A[j]$ con claves v o mayores.
3. Si $k \leq m - i$, el k -ésimo entre $A[i], \dots, A[j]$ está en el primer grupo, se llama a *selecciona* ($i, m - 1, k$); si $k > m - i$, se llama a *selecciona* ($m, j, k - m + i$).

Con el tiempo se encuentra que se llama a *selecciona* (i, j, k), cuando todos los elementos $A[i], \dots, A[j]$ tienen la misma clave (en general, porque $j = i$). Entonces se sabe que la clave deseada es cualquiera de las encontradas en esos registros.

Como con la clasificación rápida, la función *selecciona* descrita antes puede llevar un tiempo $\Omega(n^2)$ en el peor caso. Por ejemplo, supóngase que se busca el primer elemento, pero por mala suerte el pivot siempre es la mayor clave disponible. Sin embargo, en promedio, *selecciona* es aún más rápido que la clasificación rápida; lleva un tiempo $O(n)$. El motivo es que mientras esta última se llama a sí misma dos veces, *selecciona* lo hace sólo una. Se puede analizar *selecciona* como se hizo con la clasificación rápida, pero de nuevo las matemáticas son complejas, y un simple concepto intuitivo debe ser convincente. En promedio, *selecciona* se llama a sí mismo en un subarreglo la mitad de largo que el subarreglo dado. Supóngase que, en forma conservadora, se dice que cada llamada es en un arreglo cuyo tamaño es $9/10$ de la llamada previa. Entonces, si $T(n)$ es el tiempo empleado por *selecciona* en un arreglo de longitud n , se sabe que para alguna constante c ,

$$T(n) \leq T\left(\frac{9}{10}n\right) + cn \quad (8.14)$$

Usando técnicas del próximo capítulo, se puede mostrar que la solución de (8.14) es $T(n) = O(n)$.

Método lineal en el peor caso para encontrar estadísticas de orden

Para garantizar que una función como *selecciona* tenga en el peor caso, en vez de en el caso promedio, complejidad $O(n)$, es suficiente demostrar que en un tiempo lineal se puede encontrar algún pivote que con certeza esté a una fracción positiva de la distancia desde cualquier extremo. Por ejemplo, la solución a (8.14) muestra que si el pivote de n elementos nunca es menor que el elemento $(n/10)$ -ésimo, ni mayor que el elemento $(9n/10)$ -ésimo, de forma que la llamada recursiva a *selecciona* sea da como máximo en nueve décimas del arreglo, esta variante de *selecciona* será $O(n)$ en el peor caso.

La clave para encontrar un buen pivote está contenida en los dos pasos siguientes.

1. Dividir los n elementos en grupos de cinco, dejando a un lado entre 0 y 4 elementos que no puedan colocarse en un grupo. Clasificar cada grupo de cinco con cualquier algoritmo y tomar el elemento central de cada grupo, hasta un total de $[n/5]$ elementos.
2. Usar *selecciona* para encontrar la mediana de esos $[n/5]$ elementos, o si $[n/5]$ es par, un elemento en la posición más cercana al centro. Tanto si $[n/5]$ es par como si es impar, el elemento deseado estará en la posición $[(n+5)/10]$.

Este pivote está lejos de los extremos, siempre que no sean muchos los registros que tengan el pivote como clave \dagger . Para simplificar, se supondrá por el momento que todas las claves son diferentes. Entonces, se dice que el pivote elegido, el elemento $[(n+5)/10]$ -ésimo de los $[n/5]$ elementos centrales obtenidos de los grupos de cinco, es mayor que al menos $3[(n-5)/10]$ de los n elementos, puesto que excede $[(n-5)/10]$ de los elementos centrales, y cada uno de éstos excede dos más, de los cinco de los cuales es el centro. Si $n \geq 75$, $3[(n-5)/10]$ es por lo menos $n/4$. En forma semejante, se puede probar que el pivote escogido es menor o igual que al menos $3[(n-5)/10]$ elementos, así que para $n \geq 75$, el pivote cae entre los puntos $1/4$ y $3/4$ en el orden clasificado. Más importante, cuando se escoge el pivote para dividir los n elementos, el elemento k -ésimo se aislará dentro de un intervalo de hasta $3n/4$ de los elementos. Un esbozo del algoritmo completo se proporciona en la figura 8.25; igual que para los algoritmos de clasificación, éste supone un arreglo $A[i], \dots, A[n]$ de tipo_registro, y ese tipo_registro tiene un campo clave del tipo tipo_clave. El algoritmo para encontrar el k -ésimo elemento es sólo una llamada a *selecciona* ($1, n, k$).

Para analizar el tiempo de ejecución de *selecciona* de la figura 8.25, sea $n = j - i + 1$. Las líneas (2) y (3) se ejecutan sólo si n es 74 o menor. Así, aunque el paso (2) en general puede llevar $O(n^2)$ pasos, hay una constante c_1 , con independencia de su tamaño, tal que para $n \leq 74$, las líneas (1) a (3) no llevan más de un tiempo c_1 .

\dagger En el caso extremo, cuando todas las claves son iguales, el pivote no ofrece ninguna separación. Obviamente, el pivote es el k -ésimo elemento para cualquier k , y se hace necesario otro enfoque.

```

function selecciona ( i, j, k: integer ) : tipo_clave;
  { devuelve la clave del k-ésimo elemento de acuerdo con el orden
    de clasificación entre A[i],...,A[j] }

var
  m: integer; { utilizado como índice }

begin
(1)   if j-i < 74 then begin { muy pocos para usar selecciona recursivamente }
(2)     clasifica A[i],...,A[j] mediante algún algoritmo simple;
(3)     return (A[i+k-1].clave)
(4)   end
(5)   else begin { aplica selecciona recursivamente }
(6)     for m := 0 to (j-i-4) div 5 do
      { toma los elementos medios de los grupos de cinco
        en A[i], A[i+1],... }
      encuentra el tercer elemento entre A[i+5*m] y
        A[i+5*m+4] y lo intercambia con A[i+m];
(7)     pivot := selecciona(i, i+(j-i-4) div 5, (j-i-4) div 10);
      { encuentra la mediana de los elementos medios. Obsérvese
        que j-i-4 aquí es n-5 de la descripción informal
        anterior }
(8)     m := partición(i, j, pivot);
(9)     if k <= m-i then
(10)       return (selecciona(i, m-1, k))
      else
        return(selecciona(m, j, k-(m-i)))
(11)   end
end; { selecciona }

```

Fig. 8.25. Algoritmo lineal en el peor caso para encontrar el k-ésimo elemento.

Ahora, considérese las líneas (4) a (10). La línea (7), el paso de la partición, se mostró en conexión con la clasificación rápida, y lleva tiempo $O(n)$. El ciclo de las líneas (4) a (5) se ejecuta unas $n/5$ veces, y cada ejecución de la línea (5), requiriendo la clasificación de 5 elementos, lleva un tiempo constante, de manera que el ciclo lleva un tiempo $O(n)$ en total.

Sea $T(n)$ el tiempo requerido por una llamada a *selecciona* con n elementos. Entonces, la línea (6) lleva como máximo un tiempo $T(n/5)$. Ya que $n \geq 75$ cuando se alcanza la línea (10), y se ha especificado que si $n \geq 75$, a lo sumo $3n/4$ elementos son menores que el pivote, y hasta $3n/4$ son iguales o mayores, se sabe que las líneas (9) o (10) requieren como máximo un tiempo $T(3n/4)$. Así, para algunas constantes c_1 y c_2 ,

$$T(n) \leq \begin{cases} c_1 \text{ si } n \leq 74 \\ c_2 n + T(n/5) + T(3n/4) \text{ si } n \geq 75 \end{cases} \quad (8.15)$$

El término c_2n de (8.15) representa las líneas (1), (4), (5) y (7); el término $T(n/5)$ se deriva de la línea (6), y $T(3n/4)$ representa las líneas (9) y (10).

Se mostrará que $T(n)$ de (8.15) es $O(n)$. Antes de proseguir, debe apreciarse que el «número mágico» 5, el tamaño de los grupos de la línea (5), y la elección de $n = 75$ en el punto de equilibrio debajo del cual no se usa *selecciona* en forma recursiva, fueron diseñados para que los argumentos $n/5$ y $3n/4$ de T en (8.15) sumaran algo menos que n . Se podrían hacer otras elecciones de esos parámetros, pero obsérvese que al resolver (8.15) se necesita saber que $1/5 + 3/4 < 1$ para probar la linealidad.

La ecuación (8.15) puede resolverse suponiendo una solución y verificando por inducción que se cumple para toda n . Se seleccionará una solución de la forma cn , para alguna constante c . Si se escoge $c \geq c_1$, es sabido que $T(n) \leq cn$ para toda n entre 1 y 74, así que se considerará el caso $n \geq 75$. Supóngase por inducción que $T(m) \leq cm$ para $m < n$. Entonces, por (8.15),

$$T(n) \leq c_2n + cn/5 + 3cn/4 \leq c_2n + 19cn/20 \quad (8.16)$$

Al elegir $c = \max(c_1, 20c_2)$, por (8.16) se tiene $T(n) \leq cn/20 + cn/5 + 3cn/4 = cn$, lo cual se pretendía demostrar. Así, $T(n)$ es $O(n)$.

Caso en el que existen algunas igualdades entre claves

Recuérdese que en la figura 8.25 se supuso que no había dos claves iguales. La razón de esta suposición es que en otro caso no puede mostrarse que la línea (7) divide A en bloques de tamaño $3n/4$ como máximo. La modificación requerida para manipular igualdades entre claves es agregar, después del paso (7), otro paso tipo partición, que agrupe todos los registros con claves iguales al pivote. Por ejemplo, existen $p \geq 1$ de estas claves. Si $m - i \leq k \leq m - i + p$, entonces la recursión no es necesaria; simplemente se devuelve $A[m]. clave$. De otra forma, la línea (8) no cambia, pero la línea (10) llama a *selecciona*($m + p, j, k - (m - i) - p$).

Ejercicios

- 8.1 Dados los ocho enteros 1, 7, 3, 2, 0, 5, 0, 8. Clasifíquense por medio de a) clasificación de burbuja, b) clasificación por inserción, y c) clasificación por selección.
- 8.2 Dados los diecisésis enteros 22, 36, 6, 79, 26, 45, 75, 13, 31, 62, 27, 76, 33, 16, 62, 47. Clasifíquense usando a) clasificación rápida, b) clasificación por inserción, c) clasificación por montículos y d) clasificación por urnas, tratándolos como pares de dígitos en el intervalo 0 – 9.
- 8.3 El procedimiento *Shellsort* (clasificación de Shell) de la figura 8.26, algunas veces llamado *clasificación de incremento decreciente*, clasifica un arreglo $A[1..n]$ de enteros, al clasificar $n/2$ pares ($A[i], A[n/2 + i]$) para $1 \leq i \leq n/2$ en el primer recorrido, $n/4$ cuádruplos ($A[i], A[n/4 + i], A[n/2 + i], A[3n/4 + i]$,

$A[3n/4 + i]$) para $1 \leq i \leq n/4$ en el segundo recorrido, $n/8$ óctuplos en el tercer recorrido, y así sucesivamente. En cada recorrido, el ordenamiento se realiza con la clasificación por inserción, la cual termina cuando encuentra dos elementos en el orden apropiado.

```

procedure Shellsort ( var A: array[1..n] of integer );
var
    i, j, incr: integer;
begin
    incr := n div 2;
    while incr > 0 do begin
        for i := incr + 1 to n do begin
            j := i - incr;
            while j > 0 do
                if A[j] > A[j+incr] then begin
                    intercambia(A[j], A[j+incr]);
                    j := j - incr
                end
                else
                    j := 0 { fuerza la terminación del ciclo}
            end;
            incr := incr div 2
        end
    end; { Shellsort }

```

Fig. 8.26. Clasificación de Shell (Shellsort).

- Clasifíquense las secuencias de enteros de los ejercicios 8.1 y 8.2 usando *Shellsort*.
 - Muéstrese que si $A[i]$ y $A[n/2^k + i]$ quedan clasificados en el recorrido k (es decir, fueron intercambiados), entonces esos dos elementos permanecen clasificados en el recorrido $k + 1$.
 - Las distancias entre elementos comparados e intercambiados en un recorrido, disminuyen como $n/2, n/4, \dots, 2, 1$ en la figura 8.26. Demuéstrese que *Shellsort* trabajará con cualquier secuencia de distancias siempre que la última distancia sea 1.
 - Muéstrese que *Shellsort* trabaja en un tiempo $O(n^{1.5})$.
- 8.4 Supóngase que se está clasificando una lista L que consta de una lista clasificada seguida de unos cuantos elementos «aleatorios». ¿Cuál de los métodos de clasificación analizados en este capítulo será especialmente apto para tal tarea?
- 8.5 Un algoritmo de clasificación es *estable* si conserva el orden original de los registros cuyas claves son iguales. ¿Cuáles de los algoritmos de clasificación de este capítulo son estables?

- *8.6 Supóngase que se emplea una variante de la clasificación rápida, donde siempre se elige como pivote el primer elemento del subáreglo que se está clasificando.
- ¿Qué modificaciones deben hacerse al algoritmo de la figura 8.11 para evitar ciclos infinitos cuando haya una secuencia de elementos iguales?
 - Demuéstrese que el algoritmo modificado tiene un tiempo de ejecución de $O(n \log n)$ en el caso promedio.
- 8.7 Muéstrese que cualquier algoritmo de clasificación que mueva los elementos sólo una posición a la vez, debe tener una complejidad de tiempo de $\Omega(n^2)$ al menos.
- 8.8 En la clasificación por montículos, el procedimiento *empuja* de la figura 8.17 establece la propiedad de árbol parcialmente ordenado en tiempo $O(n)$. En vez de empezar en las hojas empujando elementos hasta formar un montículo, se podría empezar en la raíz y empujar elementos hacia arriba. ¿Cuál es la complejidad de tiempo de este método?
- *8.9 Supóngase que se tiene un conjunto de palabras, por ejemplo, cadenas de letras *a* a *z*, cuya longitud total es n . Muéstrese cómo clasificar esas palabras en un tiempo $O(n)$. Obsérvese que si la longitud máxima de las palabras es constante, funcionará la clasificación por urnas. Sin embargo, debe considerarse el caso en que algunas palabras sean muy largas.
- *8.10 Muéstrese que el tiempo de ejecución de la clasificación por inserción es $\Omega(n^2)$ en el caso promedio.
- **8.11 Considérese el siguiente algoritmo *clasif_aleatoria* para clasificar un arreglo $A[1..n]$ de enteros: si los elementos $A[1], A[2], \dots, A[n]$ están en el orden adecuado, se para; en otro caso, selecciona un número aleatorio i entre 1 y n , intercambia $A[1]$ y $A[i]$, y se repite. ¿Cuál es el tiempo de ejecución esperado para *clasif_aleatoria*?
- *8.12 Previamente se mostró que la clasificación por comparaciones requiere $\Omega(n \log n)$ comparaciones en el peor caso. Demuéstrese que esta cota inferior persiste también en el caso promedio.
- *8.13 Pruébese que el procedimiento *selecciona*, descrito de manera informal al principio de la sección 8.7, tiene un tiempo de ejecución $O(n)$ en el caso promedio.
- 8.14 Obténgase CONCATENA para la estructura de datos de la figura 8.19.
- 8.15 Escribábase un programa para encontrar los k elementos más pequeños de un arreglo de longitud n . ¿Cuál es la complejidad de tiempo del programa? ¿Para qué valor de k es ventajoso clasificar el arreglo?
- 8.16 Escribábase un programa para encontrar los elementos mayor y menor de un arreglo. ¿Puede hacerse esto con menos de $2n - 3$ comparaciones?

- 8.17 Escríbase un programa para encontrar el elemento más frecuente de una lista de elementos. ¿Cuál es la complejidad de tiempo del programa?
- *8.18 Muéstrese que cualquier algoritmo para eliminar duplicados de una lista requiere por lo menos un tiempo $\Omega(n \log n)$ con el modelo de computación del árbol de decisión de la sección 8.6.
- *8.19 Supóngase que se tienen k conjuntos, S_1, S_2, \dots, S_k , y cada uno con n números reales. Escríbase un programa para listar todas las sumas de la forma $s_1 + s_2 + \dots + s_k$, donde s_i está en S_i , de acuerdo con algún orden de clasificación. ¿Cuál es la complejidad de tiempo del programa?
- 8.20 Supóngase que existe un arreglo clasificado de cadenas s_1, s_2, \dots, s_n . Escríbase un programa para determinar si una cadena x dada es miembro de esta secuencia. ¿Cuál es la complejidad de tiempo del programa como una función de n y la longitud de x ?

Notas bibliográficas

Knuth [1973] es una referencia completa sobre métodos de clasificación. La clasificación rápida (*quicksort*) se debe a Hoare [1962] y las modificaciones posteriores fueron publicadas por Singleton [1969], y Frazer y McKellar [1970]. La clasificación por montículos (*heapsort*) fue descubierta por Williams [1964] y mejorada por Floyd [1964]. El árbol de decisión de la complejidad de la clasificación fue estudiado por Ford y Johnson [1959]. El algoritmo de selección lineal de la sección 8.7 es de Blum, Floyd, Pratt, Rivest y Tarjan [1972].

Shellsort se debe a Shell [1959] y su rendimiento ha sido analizado por Pratt [1979]. Véase Aho, Hopcroft y Ullman [1974] sobre una solución al ejercicio 8.9.

9

Técnicas de análisis de algoritmos

¿Qué es un buen algoritmo? No es fácil responder a esta pregunta. Muchos criterios para un buen algoritmo incluyen cuestiones muy subjetivas como simplicidad, claridad y adecuación a los datos manejados. Una meta más objetiva, lo cual no significa que sea más importante, es la eficiencia en tiempo de ejecución. En la sección 1.5 se abordaron las técnicas básicas para establecer el tiempo de ejecución de programas simples. Sin embargo, en casos más complejos se requieren técnicas nuevas, como cuando los programas son recursivos. Este corto capítulo presenta algunas técnicas generales para resolver ecuaciones de recurrencia que se presentan en el análisis de los tiempos de ejecución de algoritmos recursivos.

9.1 Eficiencia de los algoritmos

Una forma de determinar la eficiencia del tiempo de ejecución de un algoritmo es programarlo y medir el tiempo que lleva la versión en particular, en un computador específico, para un conjunto seleccionado de entradas. Aunque es popular y útil, este enfoque tiene algunos problemas inherentes. Los tiempos de ejecución dependen no sólo del algoritmo base, sino también del conjunto de instrucciones del computador, de la calidad del compilador y de la destreza del programador. El programa también puede adaptarse para trabajar correctamente sobre el conjunto particular de entradas de prueba. Estas dependencias pueden ser muy notorias con un computador, un compilador, un programador o un conjunto de entradas de prueba distinto. Para superar esos inconvenientes, los expertos en computación han adoptado la complejidad de tiempo asintótica como una medida fundamental del rendimiento de un algoritmo. El término *eficiencia* se referirá a esta medida y, en especial, a la complejidad de tiempo en el peor caso (en contraposición al promedio).

Recuérdense del capítulo 1 las definiciones de $O(f(n))$ y $\Omega(f(n))$. La eficiencia o complejidad en el peor caso de un algoritmo es $O(f(n))$, o $f(n)$ para abreviar, si $O(f(n))$ es la función de n que da el máximo, en todas las entradas de longitud n , del número de pasos dados por el algoritmo en esas entradas. En otras palabras, existe alguna constante c tal que para una n suficientemente grande, $cf(n)$ es una cota superior del número de pasos dados por el algoritmo con cualquier entrada de longitud n .

En la aseveración de que «la eficiencia de un algoritmo dado es $f(n)$ », existe la implicación de que la eficiencia también es $\Omega(f(n))$, de forma que $f(n)$ es la función

de crecimiento más lento de n que acota el tiempo de ejecución en el peor caso por arriba. Sin embargo, este último requisito no es parte de la definición de $O(f(n))$, y algunas veces no es posible asegurar que se tenga la cota de crecimiento superior más lenta.

Esta definición de eficiencia ignora factores constantes en el tiempo de ejecución y existen varias razones prácticas para ello. Primero, dado que la mayoría de los algoritmos están escritos en lenguajes de alto nivel, se deben describir en función de «pasos», los cuales emplean una cantidad constante de tiempo cuando se traducen al lenguaje de máquina de cualquier computador. Sin embargo, exactamente cuánto tiempo requiere un paso depende no sólo del paso mismo, sino del proceso de traducción y del conjunto de instrucciones de la máquina. Así, intentar tener más precisión que para decir que el tiempo de ejecución de un algoritmo es «del orden de $f(n)$ », esto es, $O(f(n))$, podría empantanar al usuario en detalles de máquinas específicas y sólo sería aplicable a esas máquinas.

Una segunda razón importante para tratar con la complejidad asintótica e ignorar factores constantes es que, más que estos factores, es la complejidad asintótica lo que determina para qué tamaño de entradas puede usarse el algoritmo para producir soluciones en un computador. En el capítulo 1 se analizó con detalle este aspecto. Sin embargo, es necesario tener cautela acerca de la posibilidad de que para problemas muy importantes, como los de clasificación, se pueda considerar adecuado analizar los algoritmos con tal detalle que sean factibles ciertas proposiciones como «el algoritmo A debe ejecutarse con el doble de rapidez que el algoritmo B en un computador típico».

Una segunda situación en la cual conviene desviarse de la noción de eficiencia en el peor caso ocurre cuando se conoce la distribución esperada de las entradas a un algoritmo. En estas situaciones, el análisis del caso promedio puede ser mucho más significativo que el análisis del peor caso. Por ejemplo, en el capítulo previo se analizó el tiempo de ejecución promedio de la clasificación rápida en el supuesto de que todas las permutaciones del orden de clasificación correcto tienen la misma probabilidad de presentarse como entradas.

9.2 Análisis de programas recursivos

En el capítulo 1 se mostró cómo analizar el tiempo de ejecución de un programa que no se llama a sí mismo en forma recursiva. El análisis de un programa recursivo es bastante distinto, y suele implicar la solución de una ecuación de diferencias. Las técnicas para la solución de ecuaciones de diferencias algunas veces son sutiles, y tienen una considerable semejanza con los métodos de solución de ecuaciones diferenciales, alguna de cuya terminología se utiliza.

Considérese el programa de clasificación presentado en la figura 9.1. Ahí, el procedimiento *mergesort* (clasificación por intercalación) toma una lista de longitud n como entrada, y devuelve una lista ordenada como salida. El procedimiento *combi-na*(L_1, L_2) toma como entrada las listas clasificadas L_1 y L_2 , y recorre cada una, elemento por elemento, desde el inicio. En cada paso, el mayor de los dos elementos

delanteros se borra de esta lista y se emite como salida. El resultado es una sola lista clasificada con los elementos de L_1 y L_2 . Los detalles de *combina* no tienen importancia alguna en este momento, puesto que este algoritmo de clasificación se analizará con detalle en el capítulo 11. Lo que importa es que el tiempo empleado por *combina* en listas de longitud $n/2$ es $O(n)$.

```

function mergesort (  $L$ : LISTA;  $n$ : integer ) : LISTA;
{  $L$  es una lista de longitud  $n$ . Se devuelve una versión clasificada de  $L$ .
  Se supone que  $n$  es una potencia de 2. }
var
   $L_1, L_2$ : LIST
begin
  if  $n = 1$  then
    return ( $L$ )
  else begin
    partir  $L$  en dos mitades,  $L_1$  y  $L_2$ , cada una de longitud  $n/2$ ;
    return (combina(mergesort( $L_1, n/2$ ), mergesort( $L_2, n/2$ )))
  end
end; { mergesort }
```

Fig. 9.1. Procedimiento recursivo *mergesort*.

Sea $T(n)$ el tiempo de ejecución en el peor caso del procedimiento *mergesort* de la figura 9.1. Se escribe una ecuación de *recurrencia* (o *diferencias*) que acote $T(n)$ por arriba, como sigue

$$T(n) \leq \begin{cases} c_1 & \text{si } n = 1 \\ 2T(n/2) + c_2n & \text{si } n > 1 \end{cases} \quad (9.1)$$

El término c_1 en (9.1) representa el número constante de pasos dados cuando L tiene longitud 1. En el caso de que $n > 1$, el tiempo requerido por *mergesort* puede dividirse en dos partes. Las llamadas recursivas a *mergesort* con listas de longitud $n/2$ cada una llevan un tiempo $T(n/2)$, de aquí el término $2T(n/2)$. La segunda parte consiste en la prueba para descubrir que $n \neq 1$, la división de la lista L en dos partes iguales y el procedimiento *combina*. Esas tres operaciones requieren un tiempo que puede ser una constante, en el caso de la prueba, o ser proporcional a n al dividir y combinar. Así, la constante c_2 puede escogerse de modo que el término c_2n sea una cota superior del tiempo requerido por *mergesort* para hacer todo excepto las llamadas recursivas. La ecuación (9.1) representa todo lo anterior.

Obsérvese que (9.1) se aplica sólo cuando n es par, por lo que generará una cota superior en forma cerrada (esto es, como una fórmula para $T(n)$ que no implica ningún $T(m)$ para $m < n$) sólo cuando n es una potencia de 2. Sin embargo, aunque sólo se conozca $T(n)$ cuando n es una potencia de 2, se tiene una buena idea de $T(n)$ para toda n . En particular, para casi todos los algoritmos, se puede suponer que $T(n)$ está entre $T(2^i)$ y $T(2^{i+1})$ si n está entre 2^i y 2^{i+1} . Y con un poco más de esfuerzo

para encontrar la solución, se puede reemplazar el término $2T(n/2)$ de (9.1) por $T((n+1)/2) + T((n-1)/2)$ para $n > 1$ impar. Después, se podría resolver la nueva ecuación de diferencia para obtener una solución cerrada para toda n .

9.3 Resolución de ecuaciones de recurrencia

Existen tres enfoques distintos para resolver una ecuación de recurrencia.

1. Suponer una solución $f(n)$ y usar la recurrencia para mostrar que $T(n) \leq f(n)$. Algunas veces sólo se supone la forma de $f(n)$, dejando algunos parámetros sin especificar (por ejemplo, supóngase $f(n) = an^2$ para alguna a) y deduciendo valores adecuados para los parámetros al intentar demostrar que $T(n) \leq f(n)$ para toda n .
2. Utilizar la recurrencia misma para sustituir $m < n$ por cualquier $T(m)$ en la derecha, hasta que todos los términos $T(m)$ para $m > 1$ se hayan reemplazado por fórmulas que impliquen sólo $T(1)$. Como $T(1)$ siempre es constante, se tiene una fórmula para $T(n)$ en función de n y de algunas constantes. A esta fórmula se ha denominado «forma cerrada» para $T(n)$.
3. Emplear la solución general para ciertas ecuaciones de recurrencia de tipos comunes de esta sección o de otra (véanse las notas bibliográficas).

En esta sección se examinan los dos primeros métodos.

Suposición de una solución

Ejemplo 9.1. Considérese el método (1) aplicado a la ecuación (9.1). Supóngase que para alguna a , $T(n) = an\log n$. Al sustituir $n = 1$, se observa que esta suposición no funcionará, porque $an\log n$ tiene valor 0, independiente del valor de a . Así, se intenta a continuación $T(n) = an\log n + b$. Ahora, $n = 1$ requiere que $b \geq c_1$.

Para la inducción, se supone que

$$T(k) \leq ak\log k + b \quad (9.2)$$

para toda $k < n$ y se intenta establecer que

$$T(n) \leq an\log n + b$$

Para iniciar la demostración, se supone que $n \geq 2$. De (9.1),

$$T(n) \leq 2T(n/2) + c_2n$$

De (9.2), con $k = n/2$, se obtiene

$$T(n) \leq 2[a\frac{n}{2}\log\frac{n}{2} + b] + c_2n \quad (9.3)$$

$$\begin{aligned} &\leq an \log n - an + c_2 n + 2b \\ &\leq an \log n + b \end{aligned}$$

siempre que $a \geq c_2 + b$.

Así, $T(n) \leq an \log n + b$ siempre y cuando se satisfagan dos restricciones: $b \geq c_1$ y $a \geq c_2 + b$. Por fortuna, existen valores que se pueden escoger para a que satisfacen ambas restricciones. Por ejemplo, al elegir $b = c_1$ y $a = c_1 + c_2$, por inducción sobre n , se concluye que para toda $n \geq 1$

$$T(n) \leq (c_1 + c_2)n \log n + c_1 \quad (9.4)$$

En otras palabras, $T(n)$ es $O(n \log n)$. \square

Son útiles dos observaciones acerca del ejemplo 9.1. Si se supone que $T(n)$ es $O(f(n))$, y si el intento de probar que $T(n) \leq cf(n)$ por inducción falla, se sigue que $T(n)$ no sea $O(f(n))$. De hecho, puede funcionar una hipótesis inductiva de la forma $T(n) \leq cf(n) - 1$.

En segundo lugar, aún no se ha determinado la tasa exacta de crecimiento asintótica para $f(n)$, aunque se ha demostrado que no es peor que $O(n \log n)$. Si se conjectura una solución más lenta, como $f(n) = an$, o $f(n) = an \log \log n$, no se puede demostrar la validez de $T(n) \leq f(n)$. La cuestión sólo se puede establecer concluyentemente examinando mergesort y demostrando que necesita realmente un tiempo $\Omega(n \log n)$; de hecho, requiere un tiempo proporcional a $n \log n$ en todas las entradas, no sólo en las peores entradas posibles. Se deja esta observación como ejercicio.

El ejemplo 9.1 expone una técnica general para demostrar que alguna función es una cota superior en el tiempo de ejecución de un algoritmo. Supóngase la ecuación de recurrencia

$$\begin{aligned} T(1) &= c \\ T(n) &\leq g(T(n/2), n), \text{ para } n > 1 \end{aligned} \quad (9.5)$$

Obsérvese que (9.5) generaliza (9.1), donde $g(x, y)$ es $2x + c_2y$. También obsérvese que podrían imaginarse ecuaciones más generales que (9.5). Por ejemplo, la fórmula g puede comprender todos los $T(n-1)$, $T(n-2)$, ..., $T(1)$, y no sólo $T(n/2)$. También, es posible tener valores para $T(1)$, $T(2)$, ..., $T(k)$, y la recurrencia sólo sería aplicable para $n > k$. A manera de ejercicio, sería interesante considerar cómo resolver estas recurrencias más generales por el método (1), suponiendo una solución y verificándola.

Considérese ahora (9.5), en vez de sus generalizaciones. Supóngase una función $f(a_1, \dots, a_j, n)$, donde a_1, \dots, a_j son parámetros, e inténtese demostrar por inducción en n que $T(n) \leq f(a_1, \dots, a_j, n)$. Por ejemplo, la suposición del ejemplo 9.1 fue $f(a_1, a_2, n) = a_1 n \log n + a_2$, pero se utilizaron a y b por a_1 y a_2 . Para corroborar que para algunos valores de a_1, \dots, a_j se tiene $T(n) \leq f(a_1, \dots, a_j, n)$ para toda $n \geq 1$, se debe cumplir

$$\begin{aligned} f(a_1, \dots, a_j, 1) &\geq c \\ f(a_1, \dots, a_j, n) &\geq g(f(a_1, \dots, a_j, n/2), n) \end{aligned} \quad (9.6)$$

Esto es, por la hipótesis inductiva, se puede sustituir f por T en el lado derecho de la recurrencia (9.5) para obtener

$$T(n) \leq g(f(a_1, \dots, a_j, n/2), n) \quad (9.7)$$

Cuando se cumple la segunda línea de (9.6), se combina con (9.7) para demostrar que $T(n) \leq f(a_1, \dots, a_n, n)$, que es lo que se deseaba demostrar por inducción en n .

Por ejemplo, en el ejemplo 9.1, $g(x, y) = 2x + c_2y$ y $f(a_1, a_2, n) = a_1 n \log n + a_2$. Aquí se intenta satisfacer

$$\begin{aligned} f(a_1, a_2, 1) &= a_2 \geq c_1 \\ f(a_1, a_2, n) &= a_1 n \log n + a_2 \geq 2(a_1 \left(\frac{n}{2}\right) \log \left(\frac{n}{2}\right) + a_2) + c_2 n \end{aligned}$$

Como se analizó, $a_2 = c_1$ y $a_1 = c_1 + c_2$ son una elección satisfactoria.

Expansión de recurrencias

Si no se puede suponer una solución, o no existe la seguridad de tener la mejor cota en $T(n)$, se usa un método que, en principio, siempre es adecuado para la solución exacta de $T(n)$, aunque en la práctica a menudo surgen problemas al sumar series y hay que recurrir al cálculo de una cota superior en la suma. La idea general es tomar una recurrencia como (9.1), que indica que $T(n) \leq 2T(n/2) + c_2n$, y emplearla para obtener una cota en $T(n/2)$ sustituyendo $n/2$ por n . Esto es,

$$T(n/2) \leq 2T(n/4) + c_2n/2 \quad (9.8)$$

Al sustituir el lado derecho de (9.8) por $T(n/2)$ en (9.1), se obtiene

$$T(n) \leq 2(2T(n/4) + c_2n/2) + c_2n = 4T(n/4) + 2c_2n \quad (9.9)$$

Del mismo modo, se sustituye $n/4$ por n en (9.1) y se emplea para obtener una cota superior en $T(n/4)$. Esa cota, $2T(n/8) + c_2n/4$, se sustituye en el lado derecho de (9.9) para obtener

$$T(n) \leq 8T(n/8) + 3c_2n \quad (9.10)$$

Ahora es necesario percibir un patrón. Por inducción en i se obtiene la relación

$$T(n) \leq 2^i T(n/2^i) + i c_2 n \quad (9.11)$$

para cualquier i . Si se supone que n es una potencia de 2, como 2^k , este proceso de expansión terminará tan pronto como se alcance $T(1)$ en el lado derecho de (9.11), lo que ocurrirá cuando $i = k$, quedando (9.11) como

$$T(n) \leq 2^k T(1) + k c_2 n \quad (9.12)$$

Entonces, como $2^k = n$, se sabe que $k = \log n$. Como $T(1) \leq c_1$, (9.12) es

$$T(n) \leq c_1 n + c_2 n \log n \quad (9.13)$$

La ecuación (9.13) es en realidad la mejor cota que se pudo colocar en $T(n)$, y demuestra que $T(n)$ es $O(n \log n)$.

9.4 Solución general para una clase grande de recurrencias

Considérese la recurrencia que surge al dividir un problema de tamaño n en a subproblemas de tamaño n/b . Por conveniencia, se supone que un problema de tamaño 1 requiere una unidad de tiempo y que el tiempo para reunir las soluciones de los subproblemas y obtener una solución del problema de tamaño n es $d(n)$, en las mismas unidades de tiempo. Para el ejemplo de *mergesort*, se tienen $a = b = 2$ y $d(n) = -c_2 n / c_1$, en unidades de c_1 . Entonces, si $T(n)$ es el tiempo para resolver un problema de tamaño n , se tiene

$$\begin{aligned} T(1) &= 1 \\ T(n) &= aT(n/b) + d(n) \end{aligned} \quad (9.14)$$

Obsérvese que (9.14) sólo se aplica a las n que sean una potencia entera de b , pero si se presume que $T(n)$ es continua, al tomar una cota superior ajustada sobre $T(n)$ para aquellos valores de n , nos indica cómo crece $T(n)$ en general.

Obsérvese también que se utiliza la igualdad en (9.14), mientras que en (9.1) hay desigualdades. La razón es que aquí $d(n)$ puede ser arbitraria y, por tanto, exacta, mientras que en (9.1) la suposición de que $c_2 n$ fue el peor caso en tiempo de combinación, para una constante c_2 y toda n , fue sólo una cota superior; el peor caso real del tiempo de ejecución con entradas de tamaño n pudo haber sido menor que $2T(n/2) + c_2 n$. En realidad, hay muy poca diferencia entre utilizar $=$ o \leq en la recurrencia, ya que de cualquier modo se obtiene una cota superior para el peor caso del tiempo de ejecución.

Para resolver (9.14) se aplica la técnica de sustituciones repetidas para T en el lado derecho, igual que se hizo para un ejemplo específico en la exposición anterior de la expansión de recurrencias. Esto es, la sustitución n/b^i por n en la segunda línea de (9.14) da

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right) + d\left(\frac{n}{b^i}\right) \quad (9.15)$$

Así, al empezar con (9.14) y sustituir (9.15) para $i = 1, 2, \dots$, se tiene

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\ &= a[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)] + d(n) = a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \end{aligned}$$

$$\begin{aligned}
 &= a^2[aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)] + ad\left(\frac{n}{b}\right) + d(n) = a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\
 &= \dots \\
 &= a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right)
 \end{aligned}$$

Ahora, suponiendo que $n = b^k$, se puede utilizar el hecho de que $T(n/b^k) = T(1) = 1$, para tener de lo anterior, con $i = k$, la fórmula

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}) \quad (9.16)$$

Si se aplica el hecho de que $k = \log_b n$, el primer término de (9.16) puede escribirse como $a^{\log_b n}$ o, de forma equivalente, $n^{\log_b a}$ (se toman logaritmos de base b de ambas expresiones para ver que son lo mismo). Esta expresión es n a una potencia constante. Por ejemplo, en el caso de *mergesort*, donde $a = b = 2$, el primer término es n . En general, cuanto mayor sea a , es decir, cuantos más subproblemas haya que resolver, tanto mayor será el exponente; cuanto mayor sea b , es decir, cuanto menor sea cada subproblema, tanto menor será el exponente.

Soluciones homogéneas y particulares

Es interesante ver los diferentes papeles que desempeñan los dos términos de (9.16). El primero, a^k o $n^{\log_b a}$, se conoce como *solución homogénea*, en analogía con la terminología de las ecuaciones diferenciales. La solución homogénea es la solución exacta cuando $d(n)$, conocida como *función motriz*, es 0 para toda n . En otras palabras, la solución homogénea representa el costo de resolver todos los subproblemas, aunque puedan combinarse sin costo.

Por otra parte, el segundo término de (9.16) comprende el costo de creación de los subproblemas y la combinación de sus resultados. Este término se denomina *solución particular*, que está afectada tanto por la función motriz como por el número y tamaño de los subproblemas. Como regla general, si la solución homogénea es mayor que la función motriz, la solución particular tendrá la misma tasa de crecimiento que la solución homogénea. Si la función motriz crece más rápido que la solución homogénea en más que n^ϵ para alguna $\epsilon > 0$, la solución particular tendrá la misma tasa de crecimiento que la función motriz. Si la función motriz tiene la misma tasa de crecimiento que la solución homogénea, o crece más rápido que $\log^k n$ como mucho para alguna k , entonces la solución particular crecerá $\log n$ veces la función motriz.

Es importante reconocer que cuando se buscan mejoras en un algoritmo es necesario saber si la solución homogénea es mayor que la función motriz. Por ejemplo, si la solución homogénea es mayor, prácticamente no tendrá efecto encontrar una forma más rápida de combinar los subproblemas en la eficiencia de todo el al-

goritmo. Lo mejor, en este caso, es encontrar una forma de dividir el problema en menos o menores subproblemas. Eso afectará a la solución homogénea y reducirá el tiempo total de ejecución.

Si la función motriz excede a la solución homogénea, entonces es necesario tratar de reducir la función motriz. Por ejemplo, en el caso de *mergesort*, donde $a = b = 2$, y $d(n) = cn$, la solución particular es $O(n\log n)$. Sin embargo, reducir $d(n)$ a una función ligeramente sublineal, por ejemplo, $n^{0.9}$, hará que la solución particular sea también menos que lineal y que se reduzca el tiempo de ejecución total a $O(n)$, que es la solución homogénea †.

Funciones multiplicativas

La solución particular de (9.16) es difícil de evaluar, aun sabiendo lo que es $d(n)$. Sin embargo, para ciertas funciones $d(n)$ comunes, se puede resolver (9.16) con exactitud, y hay otras para las cuales se puede conseguir una buena cota superior. Se dice que una función f en enteros es *multiplicativa* si $f(xy) = f(x)f(y)$ para todos los enteros positivos x e y .

Ejemplo 9.2. Las funciones multiplicativas de mayor interés son de la forma n^a para cualquier a positiva. Para demostrar que $f(n) = n^a$ es multiplicativa, sólo hay que observar que $(xy)^a = x^a y^a$. □

Ahora, si $d(n)$ de (9.16) es multiplicativa, entonces $d(b^{k-j}) = (d(b))^{k-j}$, y la solución particular de (9.16) es

$$\begin{aligned} \sum_{j=0}^{k-1} a^j (d(b))^{k-j} &= d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)} \right)^j \\ &= d(b)^k \frac{\left(\frac{a}{d(b)} \right)^k - 1}{\frac{a}{d(b)} - 1} \\ &= \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1} \end{aligned} \tag{9.17}$$

Hay tres casos a considerar, dependiendo de si a es mayor, menor o igual que $d(b)$.

† Pero no debe esperarse descubrir una manera de combinar dos listas clasificadas de $n/2$ elementos en un tiempo menor que el lineal; en ese caso, no sería posible siquiera ver todos los elementos de la lista.

- Si $a > d(b)$, la fórmula (9.17) es $O(a^k)$, que como se recuerda es $n^{\log_b a}$, ya que $k = \log_b n$. En este caso, las soluciones particular y homogénea son iguales, y sólo dependen de a y b , y no de la función motriz d . Así, las mejoras en el tiempo de ejecución deben proceder de disminuir a o aumentar b ; la disminución de $d(n)$ no es muy útil.
- Si $a < d(b)$, (9.17) es $O(d(b)^k)$ o, en forma equivalente, $O(n^{\log_b d(b)})$. En este caso, la solución particular excede a la homogénea, y se puede dirigir la atención también hacia la función motriz $d(n)$, además de a y b , para obtener mejoras. Obsérvese el importante caso especial en que $d(n) = n^a$. Entonces, $d(b) = b^a$, y $\log_b(b^a) = a$. Así, la solución particular es $O(n^a)$ o bien $O(d(n))$.
- Si $a = d(b)$, se reconsideran los cálculos implicados en (9.17), pues la fórmula para la suma de una serie geométrica no es ahora apropiada. En este caso,

$$\begin{aligned} \sum_{j=0}^{k-1} a^j (d(b))^{k-j} &= d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)} \right)^j \\ &= d(b)^k \sum_{j=0}^{k-1} 1 \\ &= d(b)^k k \\ &= n^{\log_b d(b)} \log_b n \end{aligned} \quad (9.18)$$

Como $a = d(b)$, la solución particular dada por (9.18) es $\log_b n$ veces la solución homogénea, y de nuevo la solución particular excede a la homogénea. En el caso especial $d(n) = n^a$, (9.18) se reduce a $O(n^a \log n)$, por observaciones similares a las del caso (2).

Ejemplo 9.3. Considérense las siguientes recurrencias, con $T(1) = 1$.

- $T(n) = 4T(n/2) + n$
- $T(n) = 4T(n/2) + n^2$
- $T(n) = 4T(n/2) + n^3$

En cada caso, $a = 4$, $b = 2$, y la solución homogénea es n^2 . En la ecuación (1), con $d(n) = n$, se tiene $d(b) = 2$. Como $a = 4 > d(b)$, la solución particular también es n^2 , y $T(n)$ es $O(n^2)$ en (1).

En la ecuación (3), $d(n) = n^3$, $d(b) = 8$, y $a < d(b)$. Así, la solución particular es $O(n^{\log_2 d(b)}) = O(n^3)$, y $T(n)$ de la ecuación (3) es $O(n^3)$. Se puede deducir que la solución particular es del mismo orden que $d(n) = n^3$, aplicando las observaciones anteriores acerca de los $d(n)$ de la forma n^a y analizando el caso $a < d(b)$ de (9.17). En la ecuación (2), se tiene $d(b) = 4 = a$, con lo que se aplica (9.18). Como $d(n)$ es de la forma n^a , la solución particular y, por tanto, $T(n)$, es $O(n^2 \log n)$. \square

Otras funciones motrices

Existen otras funciones motrices no multiplicativas, por medio de las cuales se obtienen soluciones para (9.16) e incluso para (9.17). Se considerarán dos ejemplos. El primero generaliza cualquier función que sea el producto de una función multiplicativa y una constante mayor o igual que uno. La segunda es típica de un caso donde es preciso examinar (9.16) con detalle y obtener una cota superior ajustada a la solución particular.

Ejemplo 9.4. Considérese

$$\begin{aligned}T(1) &= 1 \\T(n) &= 3T(n/2) + 2n^{1.5}\end{aligned}$$

Ahora, $2n^{1.5}$ no es multiplicativa, pero $n^{1.5}$ sí lo es. Sea $U(n) = T(n)/2$ para toda n . Entonces,

$$\begin{aligned}U(1) &= 1/2 \\U(n) &= 3U(n/2) + n^{1.5}\end{aligned}$$

La solución homogénea, si $U(1)$ fuera 1, sería $n^{\log 3} = n^{1.59}$; como $U(1) = 1/2$, se demuestra con facilidad que la solución homogénea es $n^{1.59}/2$; y es $O(n^{1.59})$. Para la solución particular se ignora el hecho de que $U(1) \neq 1$, dado que al incrementar $U(1)$ seguramente no se reducirá la solución particular. Como $a = 3$, $b = 2$ y $b^{1.5} = 2.82 < a$, la solución particular también es $O(n^{1.59})$, que es la tasa de crecimiento de $U(n)$. Como $T(n) = 2U(n)$, $T(n)$ también es $O(n^{1.59})$ o $O(n^{\log 3})$. \square

Ejemplo 9.5. Considérese

$$\begin{aligned}T(1) &= 1 \\T(n) &= 2T(n/2) + n\log n\end{aligned}$$

Es fácil observar que la solución homogénea es n , pues $a = b = 2$. Sin embargo, $d(n) = n\log n$ no es multiplicativa, y se debe sumar la fórmula de la solución particular de (9.16) por medios apropiados. Esto es, se desea evaluar

$$\begin{aligned}\sum_{j=0}^{k-1} 2^{j-k} \log(2^{k-j}) &= 2^k \sum_{j=0}^{k-1} (k-j) \\&= 2^{k-1}k(k+1)\end{aligned}$$

Como $k = \log n$, la solución particular es $O(n\log^2 n)$, y esta solución, al ser mayor que la homogénea, también es el valor que se obtiene de $T(n)$. \square

Ejercicios

- 9.1 Escribanse algunas de las ecuaciones de recurrencia para las complejidades de tiempo y espacio correspondientes al siguiente algoritmo, suponiendo que n es una potencia de 2.

```

function camino (s, t, n: integer): boolean;
begin
  if n = 1 then
    if arista (s, t) then
      return (true)
    else
      return (false);
  { si se llega aquí es que n > 1 }
  for i := 1 to n do
    if camino (s, i, n div 2) and camino (i, t, n div 2) then
      return (true);
    return (false)
  end; { camino }

```

La función *arista*(*i, j*) devuelve verdadero si los vértices *i* y *j* de un grafo de *n* vértices están conectados por una arista o si *i* = *j*; de lo contrario, *arista*(*i, j*) devuelve falso. ¿Qué hace el programa?

- 9.2 Resuélvanse las siguientes recurrencias, donde $T(1) = 1$ y $T(n)$ para $n \geq 2$ satisface:
- $T(n) = 3T(n/2) + n$
 - $T(n) = 3T(n/2) + n^2$
 - $T(n) = 8T(n/2) + n^3$
- 9.3 Resuélvanse las siguientes recurrencias, donde $T(1) = 1$ y $T(n)$ para $n \geq 2$ satisface:
- $T(n) = 4T(n/3) + n$
 - $T(n) = 4T(n/3) + n^2$
 - $T(n) = 9T(n/3) + n^2$
- 9.4 Obténgase las cotas o mayúscula y omega mayúscula en los $T(n)$ definidos por las siguientes recurrencias. Supóngase que $T(1) = 1$
- $T(n) = T(n/2) + 1$
 - $T(n) = 2T(n/2) + \log n$
 - $T(n) = 2T(n/2) + n$
 - $T(n) = 2T(n/2) + n^2$

*9.5 Resuélvanse las siguientes recurrencias suponiendo una solución y comprobando la respuesta

a) $T(1) = 2$

$$T(n) = 2T(n - 1) + 1 \text{ para } n \geq 2$$

b) $T(1) = 1$

$$T(n) = 2T(n-1) + n \text{ para } n \geq 2$$

9.6 Verifíquense las respuestas del ejercicio 9.5 resolviendo las recurrencias por sustitución repetida.

9.7 Generalícese el ejercicio 9.6 resolviendo todas las recurrencias de la forma

$$T(1) = 1$$

$$T(n) = aT(n - 1) + d(n) \text{ para } n \geq 1$$

en función de a y $d(n)$.

*9.8 Supóngase en el ejercicio 9.7 que $d(n) = c^n$ para alguna constante $c \geq 1$. ¿Qué dependencia tiene la solución de $T(n)$ de la relación entre a y c ? ¿Qué es $T(n)$?

**9.9 Resuélvase para $T(n)$:

$$T(1) = 1$$

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \text{ para } n \geq 2$$

9.10 Encuéntrense expresiones en su forma cerrada para las siguientes sumas.

a) $\sum_{i=0}^n i$

b) $\sum_{i=0}^n i^k$

c) $\sum_{i=0}^n 2^i$

d) $\sum_{i=0}^n \binom{n}{i}$

*9.11 Muéstrese que el número de órdenes distintos en que se puede multiplicar una secuencia de n matrices, está dado por la recurrencia

$$T(1) = 1$$

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

Muéstrese que $T(n+1) = \frac{1}{n+1} \binom{2n}{n}$. Los $T(n)$ se conocen como *números de Catalán*.

**9.12 Muéstrese que el número de comparaciones requeridas para clasificar n elementos con clasificación por intercalación (*mergesort*) está dado por

$$T(1) = 0$$

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$$

donde $\lfloor x \rfloor$ denota la parte entera de x , y $\lceil x \rceil$, el entero más pequeño $\geq x$. Muéstrese que la solución a esta recurrencia es

$$T(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

- 9.13** Muéstrese que el número de funciones booleanas de n variables está dado por la recurrencia

$$T(1) = 4$$

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

Resuélvase para $T(n)$.

- **9.14** Muéstrese que el número de árboles binarios de altura $\leq n$ está dado por la recurrencia

$$T(1) = 1$$

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

Muéstrese que $T(n) = \lfloor k^{2^n} \rfloor$ para alguna constante k . ¿Cuál es el valor de k ?

Notas bibliográficas

Bentley, Haken, y Saxe [1978], Greene y Knuth [1983], Liu [1968] y Lueker [1980] contienen material adicional sobre la solución de recurrencias. En Aho y Sloane [1973] se demuestra que muchas recurrencias no lineales de la forma $T(n) = (T(n-1))^2 + g(n)$ tienen una solución de la forma $T(n) = \lfloor k^{2^n} \rfloor$ donde k es una constante, como en el ejercicio 9.14.

10

Técnicas de diseño de algoritmos

A través de los años, los científicos de la computación han identificado diversas técnicas generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas. Este capítulo presenta algunas de las técnicas más importantes como dividir para vencer, programación dinámica, técnicas ávidas, el método de retroceso y búsqueda local. En el intento de diseñar un algoritmo para resolver un problema dado, a menudo es útil plantear cuestiones como «¿qué clase de solución se puede obtener de la programación dinámica, del enfoque ávido, de la técnica dividir para vencer o de otra técnica estándar?».

Sin embargo, debe subrayarse que hay algunos problemas, como los NP completos, para los cuales ni éstas ni otras técnicas conocidas producirán soluciones eficientes. Cuando se encuentra algún problema de este tipo, suele ser útil determinar si las entradas al problema tienen características especiales que se puedan explotar en la búsqueda de una solución, o si puede usarse alguna solución aproximada sencilla, en vez de la solución exacta, difícil de calcular.

10.1 Algoritmos dividir para vencer

Tal vez la técnica más importante y aplicada para el diseño de algoritmos eficientes sea la estrategia llamada dividir para vencer, que consiste en la descomposición de un problema de tamaño n en problemas más pequeños, de modo que a partir de la solución de dichos problemas sea posible construir con facilidad una solución al problema completo. Ya se han visto varias aplicaciones de esta técnica, como la clasificación por intercalación o los árboles binarios de búsqueda.

Para ilustrar el método, considérese el conocido acertijo de las «torres de Hanói». Consta de tres postes A , B y C . Inicialmente, el poste A tiene cierta cantidad de discos de distintos tamaños, con el más grande en la parte inferior y otros sucesivamente más pequeños encima, como se muestra en la figura 10.1. El objeto del acertijo es pasar un disco a la vez de un poste a otro, sin colocar nunca un disco grande sobre otro más pequeño, hasta que todos los discos estén en el poste B .

Pronto se descubre que el acertijo puede resolverse con el sencillo algoritmo siguiente. Se imaginan los postes dispuestos en un triángulo. Con un número de movimientos impar, se mueve el disco más pequeño hacia un poste en el sentido de las manecillas del reloj. Con un número de movimientos pares, se hace el único movimiento válido que no implique al disco más pequeño.

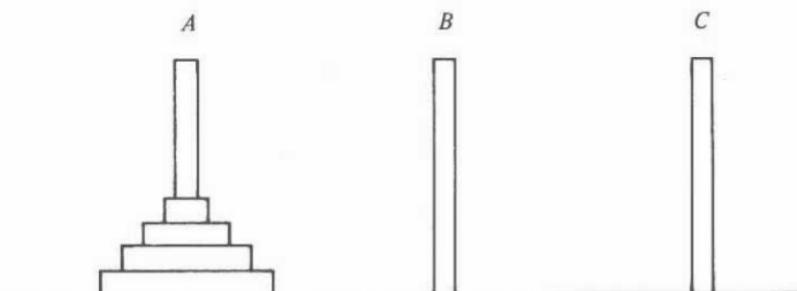


Fig. 10.1. Posición inicial en el acertijo de las torres de Hanoi.

El algoritmo anterior es conciso y correcto, pero es difícil entender por qué funciona y de descubrir intuitivamente. En cambio, considérese el siguiente enfoque de dividir para vencer. El problema de pasar los n discos más pequeños de A a B puede considerarse compuesto de dos problemas de tamaño $n - 1$. Primero se mueven los $n - 1$ discos más pequeños del poste A al C , dejando el n -ésimo disco más pequeño en el poste A . Se mueve ese disco de A a B . Después se pasan los $n - 1$ discos más pequeños de C a B . El movimiento de los $n - 1$ discos más pequeños se efectuará por medio de la aplicación recursiva del método. Como los n discos implicados en los movimientos son más pequeños que los demás discos, no es necesario preocuparse de los que se encuentran debajo de ellos en los postes A , B o C . Aunque el movimiento real de los discos individuales no es obvio y la simulación a mano es difícil debido al apilamiento de llamadas recursivas, el algoritmo es conceptualmente fácil de entender, de probar que es correcto y, tal vez, de considerarlo como primera opción. Es probable que la facilidad de descubrir los algoritmos de división haga que esta técnica sea tan importante, aunque en muchos casos los algoritmos son también más eficientes que otros más convencionales †.

Problema de la multiplicación de enteros grandes

Considérese el problema de la multiplicación de dos enteros X e Y de n bits. Recuérdese que el algoritmo para la multiplicación de enteros de n bits (o n dígitos), que se suele enseñar en la escuela primaria, implica el cálculo de n productos parciales de tamaño n , de aquí que sea un algoritmo $O(n^2)$, si se toman las multiplicaciones y las adiciones de un sólo bit o dígito como un paso. Un enfoque de clasificación del resultado por división en la multiplicación de enteros dividiría X e Y en dos enteros de $n/2$ bits cada uno, como se muestra en la figura 10.2. (Por simplicidad, se supone que n es una potencia de 2.)

† En el caso de las torres de Hanoi, el algoritmo dividir para vencer es en realidad el mismo que el dado inicialmente.

$$X := \boxed{A \quad B}$$

$$X = A2^{n/2} + B$$

$$Y := \boxed{C \quad D}$$

$$Y = C2^{n/2} + D$$

Fig. 10.2. División de un entero de n bits en segmentos de $n/2$ bits.

El producto de X e Y puede escribirse como

$$XY = AC2^n + (AD+BC)2^{n/2} + BD \quad (10.1)$$

Si se evalúa XY de esta forma directa, es necesario realizar cuatro multiplicaciones de enteros de $n/2$ bits (AC, AD, BC, BD), tres sumas de enteros con un máximo de $2n$ bits (correspondientes a los tres signos + de (10.1)), y dos desplazamientos (multiplicaciones por 2^n y por $2^{n/2}$). Como esas sumas y desplazamientos requieren $O(n)$ pasos, se puede escribir la siguiente recurrencia para $T(n)$, el número total de operaciones de bits necesarias para multiplicar enteros de n bits de acuerdo con (10.1).

$$T(1) = 1$$

$$T(n) = 4T(n/2) + cn \quad (10.2)$$

Por un razonamiento similar al del ejemplo 9.4, se toma la constante c de (10.2) como 1, de modo que la función motriz $d(n)$ es tan sólo n , y se deduce que la solución homogénea y la particular son $O(n^2)$.

En caso de que la fórmula (10.1) se utilice para multiplicar enteros, la eficiencia asintótica no es mayor que para el método de la escuela primaria. Pero recuérdese que para ecuaciones como (10.2) se consigue una mejora asintótica si se reduce el número de subproblemas. Puede ser sorprendente que se haga eso, pero considérese la siguiente fórmula para multiplicar X por Y .

$$XY = AC2^n + [(A-B)(D-C) + AC + BD]2^{n/2} + BD \quad (10.3)$$

Aunque (10.3) parece más complicada que (10.1), sólo requiere tres multiplicaciones de enteros con $(n/2)$ bits, AC, BD y $(A-B)(D-C)$, seis sumas o restas y dos desplazamientos. Como todas las operaciones, excepto las multiplicaciones, requieren $O(n)$ pasos, el tiempo $T(n)$ para multiplicar enteros de n bits por (10.3) está dado por

$$T(1) = 1$$

$$T(n) = 3T(n/2) + cn$$

cuya solución es $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

El algoritmo completo, incluyendo los detalles implicados por el hecho de que (10.3) requiere multiplicaciones de enteros de $(n/2)$ bits tanto negativos como positivos, está dado en la figura 10.3. Obsérvese que las líneas (8) a (11) se realizan co-

piando bits, y la multiplicación por 2^n y $2^{n/2}$ de la línea (16), por desplazamientos. Del mismo modo, la multiplicación por s de la línea (16) tan sólo introduce el signo adecuado en el resultado.

```

function mult ( X, Y, n: integer ): integer;
  { X e Y son enteros con signo  $\leq 2^n$ .
    n es una potencia de 2. La función devuelve XY }
  var
    s: integer; { contiene el signo de XY }
    m1, m2, m3: integer; { contiene los tres productos }
    A, B, C, D: integer; { contiene las mitades izquierda y derecha de X e Y }
  begin
(1)    s := sign(X) * sign(Y);
(2)    X := abs(X);
(3)    Y := abs(Y); { hace positivos a X e Y }
(4)    if n = 1 then
(5)      if (X = 1) and (Y = 1) then
(6)        return (s)
      else
(7)        return (0)
    else begin
(8)      A := n/2 bits izquierdos de X;
(9)      B := n/2 bits derechos de X;
(10)     C := n/2 bits izquierdos de Y;
(11)     D := n/2 bits derechos de Y;
(13)     m1 := mult(A, C, n/2);
(14)     m2 := mult(A - B, D - C, n/2);
(15)     m3 := mult(B, D, n/2);
(16)     return (s * (m1 * 2n + (m1 + m2 + m3) * 2n/2 + m3))
  end
end; { mult }

```

Fig. 10.3. Algoritmo dividir para vencer de la multiplicación de enteros.

Obsérvese que el algoritmo de la figura 10.3 es asintóticamente más rápido que el método estudiado en la escuela primaria, al requerir $O(n^{1.59})$ pasos en vez de $O(n^2)$. Puede plantearse la pregunta de por qué no se imparte en la escuela primaria si es mejor. Existen dos respuestas: primero, aunque es fácil de implantar en un computador, la descripción del algoritmo es lo bastante compleja que si se intenta enseñarlo en la escuela primaria, los estudiantes no aprenderán a multiplicar. Más aún, se han ignorado las constantes de proporcionalidad. Mientras que el procedimiento *mult* de la figura 10.3 es asintóticamente superior al método usual, las constantes son tales que para problemas pequeños (de hecho hasta de 500 bits) el método de la escuela primaria es mejor, y en raras ocasiones se pide a estudiantes de ese nivel que multipliquen números de esa magnitud.

Programación de torneos de tenis

La técnica de dividir para vencer tiene varias aplicaciones, no sólo en el diseño de algoritmos, sino también en el diseño de circuitos, construcción de demostraciones matemáticas y en otros campos del quehacer humano. Se brinda un ejemplo para ilustrar esto. Considérese el diseño del programa de un torneo de tenis para $n = 2^k$ jugadores. Cada jugador debe enfrentarse a todos los demás, y debe tener un encuentro diario durante $n - 1$ días, que es el número mínimo de días necesarios para completar el torneo.

El programa del torneo es una tabla de n filas por $n - 1$ columnas cuya posición en fila i y columna j es el jugador con quien debe contender i el j -ésimo día.

El enfoque dividir para vencer construye un programa para la mitad de los jugadores, que está diseñado por una aplicación recursiva del algoritmo buscando un programa para una mitad de esos jugadores, y así sucesivamente. Cuando se llega hasta dos jugadores, se tiene el caso base y simplemente se emparejan.

Supóngase que hay ocho jugadores. El programa para los jugadores 1 a 4 ocupa la esquina superior izquierda (4 filas por 3 columnas) del programa que se está construyendo. La esquina inferior izquierda (4 filas por 3 columnas) debe enfrentar a los jugadores con numeración más alta (5 a 8). Este subprograma se obtiene agregando 4 a cada entrada de la esquina superior izquierda.

Ahora se ha simplificado el problema, y lo único que falta es enfrentar a los jugadores de baja numeración con los de alta numeración. Esto es fácil si los jugadores 1 a 4 contienen con los 5 a 8, respectivamente, en el día 4, y se permutan cíclicamente 5 a 8 en los días siguientes. El proceso se ilustra en la figura 10.4. El lector debe ser capaz de generalizar las ideas de esta figura para obtener un programa de 2^k jugadores para cualquier k .

Balanceo de subproblemas

En el diseño de algoritmos siempre hay que afrontar varios compromisos. Ha surgido la regla de que suele ser ventajoso balancear los costos de competencia siempre que sea posible. Por ejemplo, en el capítulo 5 se vio que los árboles 2-3 balanceaban los costos de búsqueda con los de inserción, mientras que los métodos más directos requieren $O(n)$ pasos por cada búsqueda o por cada inserción, aun cuando la otra operación pueda efectuarse en un número constante de pasos.

En forma semejante, para los algoritmos de dividir para vencer, casi siempre es mejor que los subproblemas tengan un tamaño aproximadamente igual. Por ejemplo, la clasificación por inserciones puede considerarse como la división de un problema en dos subproblemas, uno de tamaño 1 y otro de tamaño $n - 1$, con un costo máximo de n pasos para combinarlos. Esto da la recurrencia

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

que tiene una solución $O(n^2)$. La clasificación por intercalación, por otra parte, divide el problema en dos subproblemas de tamaño $n/2$ y tiene un rendimiento

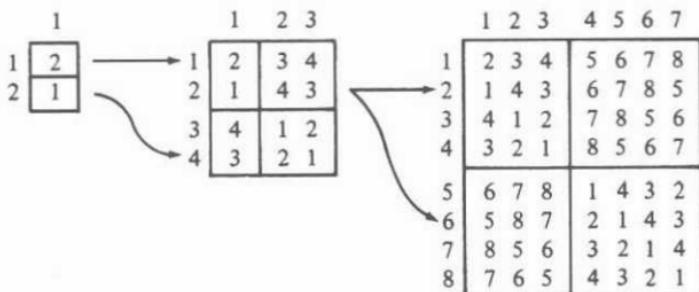


Fig. 10.4. Torneo para ocho jugadores.

$O(n \log n)$. Como principio general, a menudo resulta que dividir un problema en subproblemas iguales o casi iguales es un factor crucial para la obtención de un buen rendimiento.

10.2 Programación dinámica

A menudo sucede que no hay manera de dividir un problema en un pequeño número de subproblemas cuya solución pueda combinarse para resolver el problema original. En tales casos, se puede intentar dividir el problema en tantos subproblemas como sea necesario, dividir cada subproblema en subproblemas más pequeños, y así sucesivamente. Si eso es todo lo que se hace, quizás se produzca un algoritmo de tiempo exponencial.

No obstante, con frecuencia, sólo hay un número polinomial de subproblemas, de aquí que se deba resolver algún subproblema muchas veces. Si, en cambio, se conserva la solución a cada subproblema resuelto, y tan sólo se toma la respuesta cuando se requiere, se obtiene un algoritmo de tiempo polinomial.

Desde el punto de vista de la realización, algunas veces es más fácil crear una tabla de las soluciones de todos los subproblemas que se tengan que resolver. Se rellena la tabla sin tener en cuenta si se necesita realmente un subproblema particular en la solución total. La formación de la tabla de subproblemas para alcanzar una solución a un problema dado se denomina *programación dinámica*, nombre procedente de la teoría de control.

La forma de un algoritmo de programación dinámica puede variar, pero hay un esquema común: una tabla a llenar y un orden en el cual se hacen las entradas. Se ilustrarán las técnicas mediante dos ejemplos, el cálculo de apuestas en un encuentro como la Serie Mundial de Béisbol y el «problema de la triangulación».

Apuestas en la Serie Mundial de Béisbol

Supóngase que dos equipos, A y B , tienen un enfrentamiento para ver quién es el primero en ganar n partidos para una n en particular. La Serie Mundial es uno de

tales enfrentamientos, con $n = 4$. Se supone que A y B son igualmente competentes, de modo que cada uno tiene un 50% de oportunidades de ganar cualquier partido. Sea $P(i, j)$ la probabilidad de que A gane el encuentro, considerando que A necesita i partidos para ganar, y B necesita j . Por ejemplo, si los Dodgers han ganado dos partidos y los Yankees uno, entonces $i = 2$, $j = 3$, y se observa que $P(2, 3)$ es 11/16.

Para calcular $P(i, j)$, se emplea una ecuación de recurrencia en dos variables. Primero, si $i = 0$ y $j > 0$, el equipo A ya ha ganado el encuentro, por lo que $P(0, j) = 1$. Igualmente, $P(i, 0) = 0$ para $i > 0$. Si i y j son mayores que cero, deberá jugarse al menos otro partido más y los dos equipos ganan la mitad de las veces. Así, $P(i, j)$ debe ser el promedio de $P(i - 1, j)$ y $P(i, j - 1)$, siendo la primera de ellas la probabilidad de que A gane el encuentro si gana el siguiente partido, y la segunda, la probabilidad de que A gane el encuentro aun cuando pierda el siguiente partido. En resumen:

$$\begin{aligned} P(i, j) &= 1 \quad \text{si } i = 0 \text{ y } j > 0 \\ &= 0 \quad \text{si } i > 0 \text{ y } j = 0 \\ &= (P(i - 1, j) + P(i, j - 1))/2 \quad \text{si } i > 0 \text{ y } j > 0 \end{aligned} \quad (10.4)$$

Si se utiliza (10.4) recursivamente como una función, se demuestra que $P(i, j)$ requiere un tiempo no mayor que $O(2^{i+j})$. Sea $T(n)$ el tiempo máximo requerido por una llamada a $P(i, j)$, donde $i + j = n$. Entonces, de (10.4),

$$T(1) = c$$

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

para algunas constantes c y d . Es útil verificar por los medios analizados en el capítulo anterior, que $T(n) \leq 2^{n-1}c + (2^{n-1} - 1)d$, lo cual es $O(2^n)$ u $O(2^{i+j})$.

Así, se ha obtenido una cota superior exponencial para el tiempo requerido por el cálculo recursivo de $P(i, j)$. Sin embargo, para tener la convicción de que la fórmula recursiva para $P(i, j)$ es una mala forma de calcularla, se toma una cota inferior omega mayúscula. Se deja como ejercicio demostrar que al llamar a $P(i, j)$, el número total de llamadas a P que se hace es $\binom{i+j}{i}$, el número de formas de escoger i objetos a partir de $i + j$. Si $i = j$, ese número es $\Omega(2^n/\sqrt{n})$, donde $n = i + j$. Así, $T(n)$ es $\Omega(2^n/\sqrt{n})$ y, de hecho, se demuestra que también es $O(2^n/\sqrt{n})$. Mientras $2^n/\sqrt{n}$ crece asintóticamente más lento que 2^n , la diferencia no es muy grande, y $T(n)$ crece demasiado rápido como para que el cálculo recursivo de $P(i, j)$ sea práctico.

El problema con el cálculo recursivo es que se calcula la misma $P(i, j)$ muchas veces. Por ejemplo, si se desea obtener $P(2, 3)$, se calcula, por (10.4), $P(1, 3)$ y $P(2, 2)$. $P(1, 3)$ y $P(2, 2)$ requieren el cálculo de $P(1, 2)$, por lo que se calcula ese valor dos veces.

Una forma mejor de calcular $P(i, j)$ es llenar la tabla sugerida en la figura 10.5. La fila inferior está llena de ceros y la columna más a la derecha está llena de unos, debido a las dos primeras filas de (10.4). Por la última fila de (10.4), cada una de las otras entradas es el promedio de la entrada de abajo y de la entrada de la derecha. Así, una forma apropiada de llenar la tabla es proceder en diagonales a partir

la esquina inferior derecha, y procediendo hacia arriba y hacia la izquierda en diagonales que representan las posiciones con valor constante de $i + j$, como se sugiere en la figura 10.6. Este programa se presenta en la figura 10.7, suponiendo que opera en un arreglo bidimensional P de tamaño apropiado.

1/2	21/32	13/16	15/16	1	4	
11/32	1/2	11/16	7/8	1	3	t
3/16	5/16	1/2	3/4	1	2	j
1/16	1/8	1/4	1/2	1	1	
0	0	0	0		0	
4	3	2	1	0		
			$\leftarrow i$			

Fig. 10.5. Tabla de apuestas.

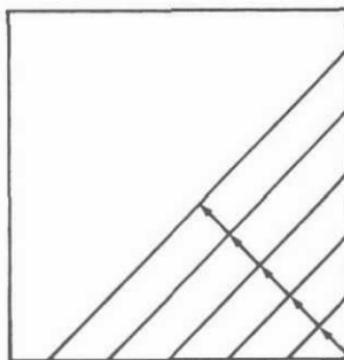


Fig. 10.6. Patrón de computación.

El análisis de la función *apuestas* es fácil. El ciclo de las líneas (4) a (5) requiere un tiempo $O(s)$, y domina el tiempo $O(1)$ de las líneas (2) a (3). Así, el ciclo externo requiere un tiempo $O(\sum_{s=1}^n s)$ u $O(n^2)$, donde $i + j = n$. Con ello, la programación dinámica requiere un tiempo $O(n^2)$, comparado con $O(2^n / \sqrt{n})$ para el enfoque directo. Puesto que $2^n / \sqrt{n}$ crece mucho más rápido que n^2 , casi siempre es preferible utilizar la programación dinámica que el enfoque recursivo.

```
function apuestas ( i, j: integer ) : real;
var
  s, k: integer;
begin
```

```

(1)      for s := 1 to i + j do begin
          { calcular la diagonal de las posiciones cuyos índices se suman a s }
          P[0, s] := 1.0;
(2)      P[s, 0] := 0.0;
(3)      for k := 1 to s - 1 do
(4)          P[k, s - k] := (P[k - 1, s - k] + P[k, s - k - 1])/2.0
(5)
end;
(6)      return (P[i, j])
end; { apuestas }

```

Fig. 10.7. Cálculo de apuestas.

Problema de la triangulación

Como otro ejemplo de programación dinámica, considérese el problema de la *triangulación* de un polígono. Se dan los vértices de un polígono y una medida de la distancia entre cada par de vértices. La distancia puede ser la normal (euclidiana) en el plano, o puede ser una función de costo arbitraria dada por una tabla. El problema consiste en seleccionar un conjunto de *cuerdas* (líneas entre vértices no adyacentes) de modo que dos cuerdas no se crucen entre sí y que todo el polígono quede dividido en triángulos. La longitud total (distancia entre puntos finales) de las cuerdas seleccionadas debe ser mínima. Ese conjunto de cuerdas se llama *triangulación mínimal*.

Ejemplo 10.1. La figura 10.8 muestra un polígono de siete lados y las coordenadas (x, y) de sus vértices. La función de distancia es la distancia normal euclidiana. Una triangulación no minimal, se muestra por medio de las líneas de puntos. Su costo es la suma de las longitudes de las cuerdas (v_0, v_2) , (v_0, v_3) , (v_0, v_5) y (v_3, v_5) , o $\sqrt{8^2+16^2} + \sqrt{15^2+16^2} + \sqrt{22^2+2^2} + \sqrt{7^2+14^2} = 77.56$. \square

Además de ser interesante por sí mismo, el problema de la triangulación tiene diversas aplicaciones útiles. Por ejemplo, Fuchs, Kedem, y Uselton [1977] usaron una generalización del problema de la triangulación para el propósito siguiente. Considerese el problema de sombrear la imagen bidimensional de un objeto cuya superficie está definida por una colección de puntos en un espacio tridimensional. La fuente de luz proviene de una dirección dada, y el brillo de un punto en la superficie depende de los ángulos entre la dirección de la luz, la dirección de la vista del observador y una perpendicular a la superficie en ese punto. Para estimar la dirección de la superficie en un punto, se calcula una triangulación mínima de los puntos que definen la superficie.

Cada triángulo define un plano en el espacio tridimensional, y dado que se encontró una triangulación mínima, es de esperarse que los triángulos sean muy pequeños. Es fácil encontrar la dirección de una perpendicular a un plano, así que se calcula la intensidad de la luz para los puntos de cada triángulo en el supuesto de que la superficie puede considerarse como un plano triangular en una región dada. Si los triángulos no son suficientemente pequeños como para hacer que la intensidad de la luz aparezca suave, entonces obtener el promedio local puede mejorar la imagen.

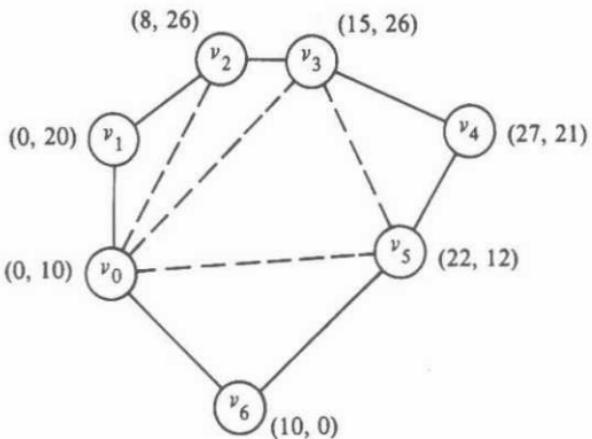


Fig. 10.8. Heptágono y triangulación.

Antes de proseguir con la solución de la programación dinámica al problema de la triangulación, se dejarán establecidas dos observaciones acerca de las triangulaciones que ayudarán a diseñar el algoritmo. Se supone que hay un polígono con n vértices v_0, v_1, \dots, v_{n-1} , en el sentido de las manecillas del reloj.

Hecho 1. En cualquier triangulación de un polígono con más de tres vértices, cada par de vértices adyacentes es tocado al menos por una cuerda. Para ver esto, supóngase que ni v_i ni v_{i+1} † fueron tocados por una cuerda. Entonces la región que la arista (v_i, v_{i+1}) limita debe incluir las aristas (v_{i-1}, v_i) , (v_{i+1}, v_{i+2}) y al menos una arista adicional. Esta región no sería un triángulo.

Hecho 2. Si (v_i, v_j) es una cuerda de una triangulación, debe haber algún v_k tal que (v_i, v_k) y (v_k, v_j) sean aristas del polígono o cuerdas. De otra forma, (v_i, v_j) puede limitar una región que no sea un triángulo.

Para iniciar la búsqueda de una triangulación minimal, se escogen dos vértices adyacentes, como v_0 y v_1 . Por los dos hechos, se sabe que en cualquier triangulación y, por tanto, en la triangulación mínima, debe haber un vértice v_k tal que (v_1, v_k) y (v_k, v_0) sean cuerdas o aristas en la triangulación. Así pues, se debe considerar la bondad de la triangulación que se obtendrá después de seleccionar cada valor posible de k . Si el polígono tiene n vértices, se puede hacer un total de $(n - 2)$ selecciones.

Cada selección de k da lugar como máximo a dos subproblemas, que son polígonos formados por una cuerda y las aristas del polígono original que van de un extremo a otro de la cuerda. Por ejemplo, la figura 10.9 muestra los dos subproblemas que resultan al seleccionar el vértice v_3 .

A continuación, se deben encontrar triangulaciones mínimas para los polígonos de la figura 10.9(a) y (b). Lo primero que hay que hacer es considerar otra vez todas

† En lo sucesivo, se considerará que todos los subíndices están calculados módulo n . Así, en la figura 10.8, v_i y v_{i+1} pueden ser v_6 y v_0 , respectivamente, puesto que $n = 7$.

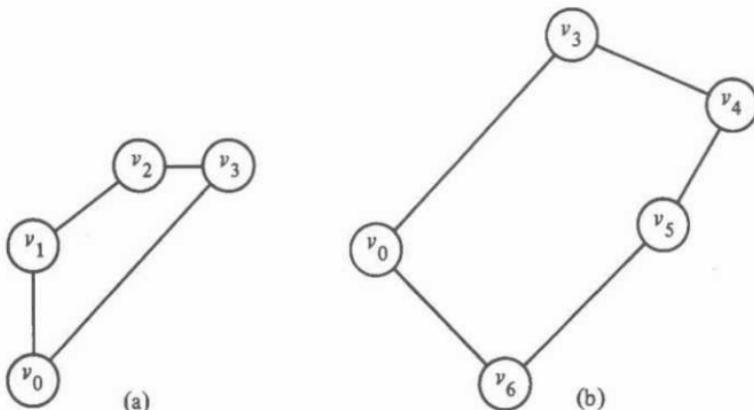


Fig. 10.9. Los dos subproblemas después de seleccionar v_3 .

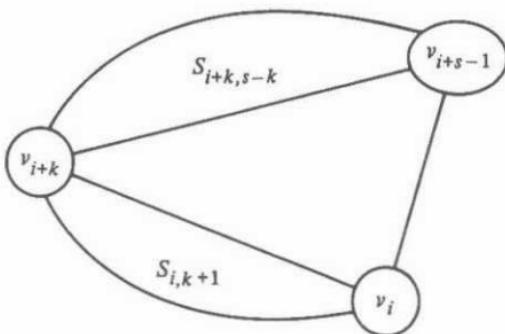
las cuerdas que parten de dos vértices adyacentes. Por ejemplo, en la solución de la figura 10.9(b), se considera la selección de la cuerda (v_3, v_5) , la cual deja el subproblema (v_0, v_3, v_5, v_6) , un polígono en el cual dos lados, (v_0, v_3) y (v_3, v_5) , son cuerdas del polígono original. Este camino conduce a un algoritmo de tiempo exponencial.

Sin embargo, considerando el triángulo que incluye la cuerda (v_0, v_k) nunca se tienen que considerar polígonos en los cuales más de un lado sean cuerdas del polígono original. El hecho 2 dice que, en la triangulación mínima, la cuerda del subproblema, como (v_0, v_3) de la figura 10.9(b), debe formar un triángulo con uno de los otros vértices. Por ejemplo, si se selecciona v_4 , se forma el triángulo (v_0, v_3, v_4) y el subproblema (v_0, v_4, v_5, v_6) , que sólo tiene una cuerda del polígono original. Al probar con v_5 , se forman los subproblemas (v_3, v_4, v_5) y (v_0, v_5, v_6) , con las cuerdas (v_3, v_5) y (v_0, v_5) nada más.

En general, se define el *subproblema de tamaño s* partiendo del vértice v_i , denotado S_{is} , como el problema de triangulación minimal para el polígono formado por los s vértices que comienzan en v_i y siguen en el sentido de las manecillas del reloj, esto es, $v_i, v_{i+1}, \dots, v_{i+s-1}$. La cuerda en S_{is} es (v_i, v_{i+s-1}) . Por ejemplo, en la figura 10.9(a) es S_{04} y en la figura 10.9 (b) es S_{35} . Para resolver S_{is} debemos considerar las tres opciones siguientes.

1. Tomar el vértice v_{i+s-2} para formar un triángulo con las cuerdas (v_i, v_{i+s-1}) , (v_i, v_{i+s-2}) y con el tercer lado (v_{i+s-2}, v_{i+s-1}) , y después resolver el subproblema $S_{i,s-1}$.
2. Tomar el vértice v_{i+1} para formar un triángulo con las cuerdas (v_i, v_{i+s-1}) , (v_{i+1}, v_{i+s-1}) y con el tercer lado (v_i, v_{i+1}) , y después resolver el subproblema $S_{i+1,s-1}$.
3. Para alguna k entre 2 y $s - 3$, tomar el vértice v_{i+k} y formar un triángulo con los lados (v_i, v_{i+k}) , (v_{i+k}, v_{i+s-1}) , y (v_i, v_{i+s-1}) , y después resolver los subproblemas $S_{i,k+1}$ y $S_{i+k,s-k}$.

Si se recuerda que la «solución» a cualquier subproblema de tamaño tres o menor no requiere acción alguna, se puede resumir (1) a (3) diciendo que se toma al-

Fig. 10.10. División de S_{is} en subproblemas.

gún vértice k entre 1 y $s - 2$ y se resuelven los subproblemas $S_{i,k+1}$ y $S_{i+k,s-k}$. La figura 10.10 ilustra esta división en subproblemas.

Si se emplea el algoritmo recursivo implícito en las reglas anteriores para resolver subproblemas de tamaño 4 o mayores se podrá mostrar que cada llamada en un subproblema de tamaño s provoca un total de 3^{s-4} llamadas recursivas, si se «resuelven» directamente los subproblemas de tamaño 3 o menores y sólo se cuentan las llamadas en subproblemas de tamaño 4 o mayores. Así, el número de subproblemas a resolver es exponencial en s . Dado que el problema inicial es de tamaño n , donde n es el número de vértices del polígono dado, el número total de pasos realizados por este procedimiento recursivo es exponencial en n .

Es evidente que todavía hay algo erróneo en este análisis, porque se sabe que sin contar el problema original, sólo hay $n(n - 4)$ subproblemas diferentes que necesitan solución, y se representan por S_{is} , donde $0 \leq i < n$ y $4 \leq s < n$. Es obvio que no todos los subproblemas resueltos por el procedimiento recursivo son diferentes. Por ejemplo, si en la figura 10.8 se escoge la cuerda (v_0, v_3) , y después, en el subproblema de la figura 10.9(b), v_4 , hay que resolver el subproblema S_{44} . Pero también se tendría que resolver este problema si se tomara primero la cuerda (v_0, v_4) , o la cuerda (v_1, v_4) , y entonces, al resolver el subproblema S_{45} , se tomaría el vértice v_0 para completar un triángulo con v_1 y v_4 .

Esto sugiere una forma eficiente de resolver el problema de la triangulación. Se prepara una tabla que contenga el costo C_{is} de la triangulación S_{is} para toda i y s . Dado que la solución a cualquier problema dado sólo depende de la solución a problemas de tamaño menor, el orden lógico en el cual se debe llenar la tabla es de acuerdo con el tamaño. Esto es, para tamaños $s = 4, 5, \dots, n - 1$ se introduce el costo mínimo para los subproblemas S_{is} , para todos los vértices i . Es conveniente incluir también los problemas de tamaño $0 \leq s < 4$, pero recuérdese que S_{is} tiene costo 0 si $s < 4$.

De las reglas anteriores (1) a (3) para encontrar subproblemas, la fórmula para calcular C_{is} para $s \geq 4$ es:

$$C_{is} = \min_{1 \leq k \leq s-2} [C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})] \quad (10.5)$$

donde $D(v_p, v_q)$ es la longitud de la cuerda entre los vértices v_p y v_q , si v_p y v_q no son puntos adyacentes en el polígono; $D(v_p, v_q) = 0$ si v_p y v_q son adyacentes.

Ejemplo 10.2. La figura 10.11 contiene la tabla de costos de S_{is} para $0 \leq i \leq 6$ y $4 \leq s \leq 6$, basada en el polígono y las distancias de la figura 10.8. Los costos de las filas con $s < 3$ son todos cero. Se ha llenado la posición C_{07} , en la columna 0 y la fila para $s = 7$. Este valor, como todos los de la misma fila, representa la triangulación del polígono completo. Para ver esto, obsérvese que se puede, si se desea, considerar la arista (v_0, v_6) como una cuerda de un polígono mayor y el polígono de la figura 10.8 como un subproblema de este polígono, el cual tiene una serie de vértices adicionales que se extienden en el sentido de las manecillas del reloj desde v_6 hasta v_0 . Obsérvese que toda la fila para $s = 7$ tiene el mismo valor que C_{07} , para conservar la exactitud de cálculo.

Como ejemplo, se proporciona la demostración de cómo se introdujo el valor 38.09 de la columna para $i = 6$ y la fila para $s = 5$. De acuerdo con (10.5), el valor de esta posición, C_{65} , es el mínimo de tres sumas correspondientes a $k = 1, 2$ ó 3 . Esas sumas son:

$$C_{62} + C_{04} + D(v_6, v_0) + D(v_0, v_3)$$

$$C_{63} + C_{13} + D(v_6, v_1) + D(v_1, v_3)$$

$$C_{64} + C_{22} + D(v_6, v_2) + D(v_2, v_3)$$

7	$C_{07} = 75.43$						
6	$C_{06} = 53.54$	$C_{16} = 55.22$	$C_{26} = 57.58$	$C_{36} = 64.69$	$C_{46} = 59.78$	$C_{56} = 59.78$	$C_{66} = 63.62$
5	$C_{05} = 37.54$	$C_{15} = 31.81$	$C_{25} = 35.49$	$C_{35} = 37.74$	$C_{45} = 45.50$	$C_{55} = 39.98$	$C_{65} = 38.09$
4	$C_{04} = 16.16$	$C_{14} = 16.16$	$C_{24} = 15.65$	$C_{34} = 15.65$	$C_{44} = 22.69$	$C_{54} = 22.69$	$C_{64} = 17.89$
s	$i=0$	1	2	3	4	5	6

Fig. 10.11. Tabla de C_{is} .

Las distancias requeridas se calculan a partir de las coordenadas de los vértices:

$$D(v_2, v_3) = D(v_6, v_0) = 0$$

(puesto que son aristas del polígono, no cuerdas, y se presentan en forma «gratuita»)

$$D(v_6, v_2) = 26.08$$

$$D(v_1, v_3) = 16.16$$

$$D(v_6, v_1) = 22.36$$

$$D(v_0, v_3) = 21.93$$

Las tres sumas anteriores dan 38.09, 38.52 y 43.97, respectivamente. Se concluye que el costo mínimo del subproblema S_{65} es 38.09. Más aún, como la primera suma fue la menor, se sabe que para alcanzar este valor es necesario utilizar los subproblemas S_{62} y S_{04} , esto es, seleccionar la cuerda (v_0, v_3) y resolver S_{64} como se pude; la cuerda (v_1, v_3) es la elección preferida para este subproblema. \square

Existe un truco útil para llenar la tabla de la figura 10.11 de acuerdo con la fórmula (10.5). Cada término de la operación \min de (10.5) requiere un par de valores. El primer par, para $k = 1$, puede encontrarse en la tabla a) en la parte «inferior» (la fila de $s = 2$) † de la columna del elemento que se está calculando, y b) justo abajo y a la derecha ‡ del elemento calculado. El segundo par está a) junto a la parte inferior de la columna, y b) dos posiciones más abajo y a la derecha. La figura 10.12 muestra las dos líneas de posiciones que se siguen para obtener todos los pares de valores que es necesario considerar simultáneamente. El patrón —subir por la columna y bajar por la diagonal— es común para el llenado de tablas durante la programación dinámica.

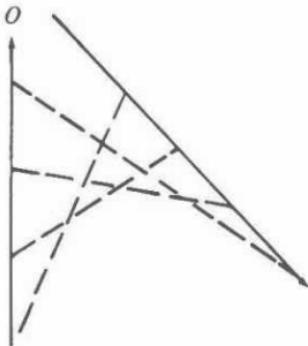


Fig. 10.12. Patrón de seguimiento de la tabla para calcular un elemento.

Para encontrar soluciones a partir de la tabla

Aunque la figura 10.11 ofrece el costo de la triangulación mínima, no proporciona de inmediato la triangulación misma. Para cada posición se necesita el valor de k que produjo el mínimo en (10.5). Luego se puede deducir que la solución consta de las cuerdas (v_i, v_{i+k}) y (v_{i+k}, v_{i+s-1}) (a menos que una de ellas no sea cuerda, porque $k = 1$ o $k = s - 2$), más las cuerdas que estén implicadas por las soluciones a $S_{i,k+1}$ y $S_{i+k,s-k}$. Al calcular un elemento de la tabla, es útil incluir el valor de k que brinde la mejor solución.

Ejemplo 10.3. En la figura 10.11, el valor C_{07} , que representa la solución al problema completo de la figura 10.8, procede de los términos de $k = 5$ en (10.5). Esto es,

† Recuérdese que la tabla de la figura 10.11 tiene filas de ceros por debajo de las que se muestran.

‡ Por «a la derecha» se quiere decir que la tabla se lee en forma circular. Así, si uno se encuentra en la columna de más a la derecha, la columna «a la derecha» es la que está más a la izquierda.

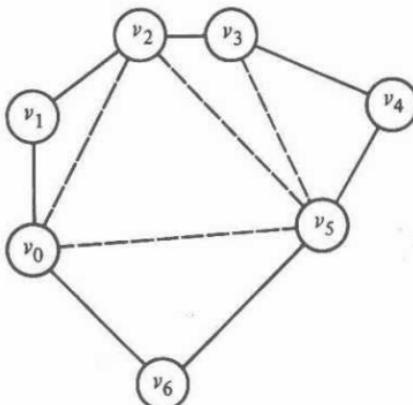


Fig. 10.13. Triangulación de costo minimal.

el problema S_{07} se divide en S_{06} y S_{52} ; el primero es el problema con seis vértices v_0, v_1, \dots, v_5 , y el último es un «problema» trivial de costo 0. Así, se introduce la cuerda (v_0, v_5) de costo 22.09 y se debe resolver S_{06} .

El costo mínimo de C_{06} procede de los términos para $k = 2$ en (10.5), en tanto el problema S_{06} se divide en S_{03} y S_{24} . El primero es el triángulo con vértices v_0, v_1 y v_2 , mientras que el último es un cuadrilátero definido por v_2, v_3, v_4 y v_5 . S_{03} no necesita ser resuelto, pero S_{24} sí, y deben incluirse los costos de las cuerdas (v_0, v_2) y (v_2, v_5) , los cuales son 17.89 y 19.80, respectivamente. El valor mínimo para C_{24} se supone cuando $k = 1$ en (10.5), dando los subproblemas C_{22} y C_{33} , que tienen tamaño menor o igual que tres y, por tanto, costo 0. Se introduce la cuerda (v_3, v_5) , con un costo de 15.65. □

10.3 Algoritmos ávidos

Considérese el problema de dar un cambio. Supónganse monedas de 25 € (un cuarto) 10 € (un décimo), 5 € (un vigésimo) y 1 € (un centavo), y que se desea dar un cambio de 63 €. Sin pensar, se convierte esta cantidad a dos cuartos, un décimo y tres centavos. No sólo se determinó rápidamente una lista de monedas con el valor correcto, sino que se produjo la lista más corta de monedas con ese valor.

El algoritmo empleado probablemente fue seleccionar la moneda mayor cuyo valor no excedía de 63 € (un cuarto), agregarla a la lista y sustraer su valor de 63, quedando 38 €. De nuevo, se seleccionó la moneda más grande cuyo valor no fuera mayor de 38 € (otro cuarto) y se añadió a la lista, y así sucesivamente.

Este método de dar cambio es un *algoritmo ávido*. En cualquier etapa individual, un algoritmo ávido selecciona la opción que sea «localmente óptima» en algún sentido particular. Obsérvese que el algoritmo ávido para dar cambio produce una solución óptima general debido a las propiedades de las monedas. Si las monedas tuvieran valores 1 €, 5 € y 11 € y se deseara dar un cambio de 15 €, el

algoritmo ávido seleccionaría primero la moneda de 11 € y después cuatro de 1 €, para un total de cinco monedas. Sin embargo, tres monedas de 5 € bastan.

Ya se han visto varios algoritmos ávidos, como el algoritmo del camino más corto de Dijkstra y el algoritmo de árboles abarcadores de costo mínimo de Kruskal. El algoritmo de Dijkstra es «ávido» en el sentido de que siempre escoge el vértice más cercano al origen de entre aquellos cuyo camino más corto todavía se desconoce. El algoritmo de Kruskal también es «ávido»: de las aristas restantes, elige la más corta de entre aquellas que no crean un ciclo.

Es necesario subrayar el hecho de que no todos los enfoques ávidos llegan a dar mejor resultado. Como sucede en la vida real, una estrategia ávida puede dar un buen resultado durante un tiempo, pero el resultado general puede ser pobre. Como ejemplo, se considera lo que sucede al permitir aristas ponderadas negativas en los algoritmos de Dijkstra y de Kruskal. Resulta que el algoritmo de árboles abarcadores de Kruskal no se ve afectado y continuará produciendo el árbol de costo mínimo, pero el algoritmo de Dijkstra en algunos casos no produce los caminos más cortos.

Ejemplo 10.4. En la figura 10.14 se presenta un grafo con una arista entre b y c cuyo costo es negativo. Al aplicar el algoritmo de Dijkstra con origen s , primero se descubrirá, correctamente, que el camino mínimo hacia a tiene longitud 1. Ahora, considerando sólo las aristas de s o a a b o c , se espera que b tenga el camino más corto desde s , es decir, $s \rightarrow a \rightarrow b$, de longitud 3. Así, se descubre que c tiene un camino más corto desde s de longitud 1.

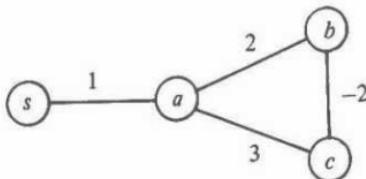


Fig. 10.14. Grafo con una arista ponderada negativa.

Sin embargo, esta selección «ávida» de b antes que c fue errónea desde un punto de vista general. Dado que el camino $s \rightarrow a \rightarrow c \rightarrow b$ tiene una longitud de sólo 2, la distancia mínima de 3 para b fue errónea †. □

Algoritmos ávidos como procedimientos heurísticos

Para algunos problemas, no existen algoritmos ávidos conocidos que produzcan soluciones óptimas; con todo, existen algunos que pueden llegar a producir soluciones «buenas» con una alta probabilidad. A menudo, una solución semióptimal

† De hecho, se debe tener cuidado con el significado de «camino más corto» cuando existen aristas negativas. Si se permiten ciclos con costo negativo, entonces se pueden recorrer repetidamente ese ciclo para alcanzar distancias negativas arbitrariamente grandes, por lo que se desea ceñirse a caminos acíclicos.

con un costo de un pequeño porcentaje sobre la óptima es muy satisfactoria. En esos casos, un algoritmo ávido con frecuencia brinda la forma más rápida para llegar a una solución «buena». De hecho, si el problema es tal que la única forma de alcanzar una solución óptima es mediante el uso de una técnica de búsqueda exhaustiva, un algoritmo ávido u otro procedimiento heurístico para llegar a una buena solución, pero no necesariamente óptima, puede ser la única opción real.

Ejemplo 10.5. Se presenta un conocido problema donde los únicos algoritmos conocidos que producen soluciones óptimas son del tipo «intentar todas las posibilidades» y pueden tener tiempos de ejecución con entradas de tamaño exponencial. El problema, llamado *problema del agente viajero*, o *PAV*, es encontrar, en un grafo no dirigido con pesos en las aristas, un *recorrido* (un ciclo simple que incluya todos los vértices) donde la suma de todos los pesos de las aristas sea un mínimo. Un recorrido se suele denominar *ciclo de Hamilton* (o *ciclo hamiltoniano*).

La figura 10.15(a) muestra un ejemplo del problema del agente viajero, un grafo con seis vértices (a veces llamados «ciudades»). Se dan las coordenadas de cada vértice, y se toma el peso de cada arista como su longitud. Obsérvese que, por convención con el PAV, se supone que todas las aristas existen, esto es, el grafo es completo. En ejemplos más generales, donde los pesos de las aristas no estén basados en la distancia euclíadiana, se puede encontrar un peso infinito sobre aristas que en realidad no están presentes.

Las figuras 10.15(b) a (e) muestran cuatro recorridos de las seis «ciudades» de la figura 10.15(a). El lector puede preguntarse cuál de los cuatro es el óptimo, si lo es alguno. Las longitudes de los cuatro recorridos son 50.00, 49.73, 48.39 y 49.78, respectivamente; (d) es el más corto de todos los recorridos posibles.

El PAV tiene varias aplicaciones prácticas. Como su nombre indica, puede usarse para planear la ruta de una persona que debe visitar varios sitios y re-

$$\begin{array}{ll}
 c \bullet (1, 7) & d \bullet (15, 7) \\
 b \bullet (4, 3) & e \bullet (15, 4) \\
 a \bullet (0, 0) & f \bullet (18, 0) \\
 \end{array}$$

(a) seis «ciudades»

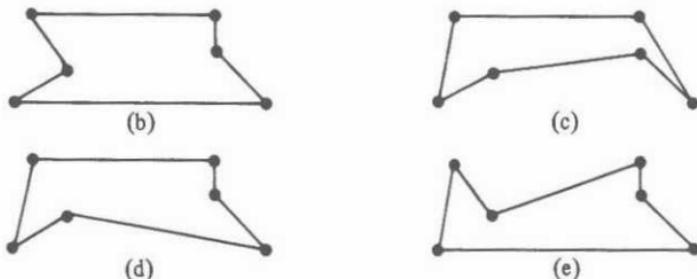


Fig. 10.15. Ejemplo del problema del agente viajero.

gresar al punto inicial. Por ejemplo, el PAV se ha usado para seleccionar la ruta de los recolectores de dinero de teléfonos públicos. Los vértices son los teléfonos y la oficina de cobro. El costo de cada arista es el tiempo del viaje entre los dos sitios en cuestión.

Otra aplicación del PAV es la resolución del *problema del recorrido del caballo* para encontrar una secuencia de movimientos de modo que el caballo pueda visitar cada cuadro del tablero de ajedrez sólo una vez y regrese al punto de partida. Se dan como vértices los cuadros del tablero de ajedrez y las aristas entre dos cuadros que son un movimiento del caballo con peso 0; las demás aristas tienen peso infinito. Un recorrido óptimal tiene peso 0 y debe ser un recorrido del caballo. Sorprendentemente, los procedimientos heurísticos buenos para el PAV no tienen dificultad en encontrar los recorridos del caballo, pero encontrar uno «a mano» es un reto.

El algoritmo ávido para el PAV que se tiene en mente es una variante del algoritmo de Kruskal. Como en dicho algoritmo, primero se considerarán las aristas más cortas. En el algoritmo de Kruskal, se acepta una arista en su turno si no forma un ciclo con las que ya han sido tomadas; en caso de no cumplir esto, rechaza la arista. Para el PAV, el criterio de aceptación es que una arista sometida a consideración, junto con las que ya se han seleccionado,

1. no haga que un vértice tenga grado 3 o mayor, y
2. no forme un ciclo, a menos que el número de aristas seleccionadas sea igual al número de vértices del problema.

Las colecciones de aristas seleccionadas con ese criterio, formarán una colección de caminos no conexos, hasta el último paso, cuando el único camino existente esté cerrado y forme un recorrido.

En la figura 10.15(a), se puede tomar primero la arista (d, e) , puesto que es la más corta, con longitud 3. Después, las aristas (b, c) , (a, b) y (e, f) , que tienen longitud 5. No importa en qué orden se tomen, todas cumplen con las condiciones de selección, y han de seleccionarse si se quiere seguir el enfoque ávido. La siguiente arista más corta es (a, c) , con longitud 7.08. Sin embargo, esta arista puede formar un ciclo con (a, b) y (b, c) , por lo que se rechaza. La arista (d, f) es la siguiente que se rechaza por motivos similares. La arista (b, e) es la siguiente a considerar, pero debe rechazarse porque puede aumentar los grados de los vértices b y e a 3, y nunca formaría un recorrido con lo seleccionado. De modo similar, se rechaza (b, d) . A continuación se considera (c, d) , que sí se acepta. Ahora existe un camino, $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$, y por último se acepta (a, f) para completar el recorrido. El recorrido resultante está en la figura 10.15(b), y es el cuarto mejor de todos los posibles, pero es más costoso que el óptimo en menos del 4%. □

10.4 Método de retroceso (*backtracking*)

Algunas veces surge el problema de encontrar una solución óptima a un subproblema, pero sucede que no existe una teoría que pueda aplicarse para ayudar a encontrar lo óptimo, si no es recurriendo a una búsqueda exhaustiva. Esta sección se de-

dica a una técnica de búsqueda exhaustiva sistemática conocida como método de retroceso (*backtracking*) y una técnica llamada poda alfa-beta, que suele reducir mucho la búsqueda.

Considérese un juego como el ajedrez, damas o tres en raya (gato), donde hay dos jugadores. Los jugadores alternan las jugadas, y el estado del juego puede representarse por una posición del tablero. Se hace la suposición de que hay un número finito de posiciones del tablero y que el juego tiene algún tipo de regla para asegurar la terminación. Con cada uno de esos juegos se asocia un árbol llamado *árbol de juego*. Cada nodo del árbol representa una posición en el tablero, y se asocia la raíz con la posición de partida. Si la posición del tablero x se asocia con el nodo n , los hijos de n corresponderán al conjunto de movimientos posibles desde la posición x del tablero, y cada hijo se asociará con la posición resultante. Por ejemplo, la figura 10.16 muestra parte del árbol del juego de tres en raya.

Las hojas del árbol corresponden a las posiciones del tablero donde ya no existen movimientos posibles, porque uno de los jugadores ha ganado o porque todos los cuadros están ocupados y se produjo un empate. Se asocia un valor con cada nodo del árbol, y primero se asignan valores a las hojas. Si el juego es tres en raya, una hoja tendrá un valor -1 , 0 ó 1 , dependiendo de si la posición del tablero corresponde a una derrota, empate o victoria para el jugador 1 (jugando con X).

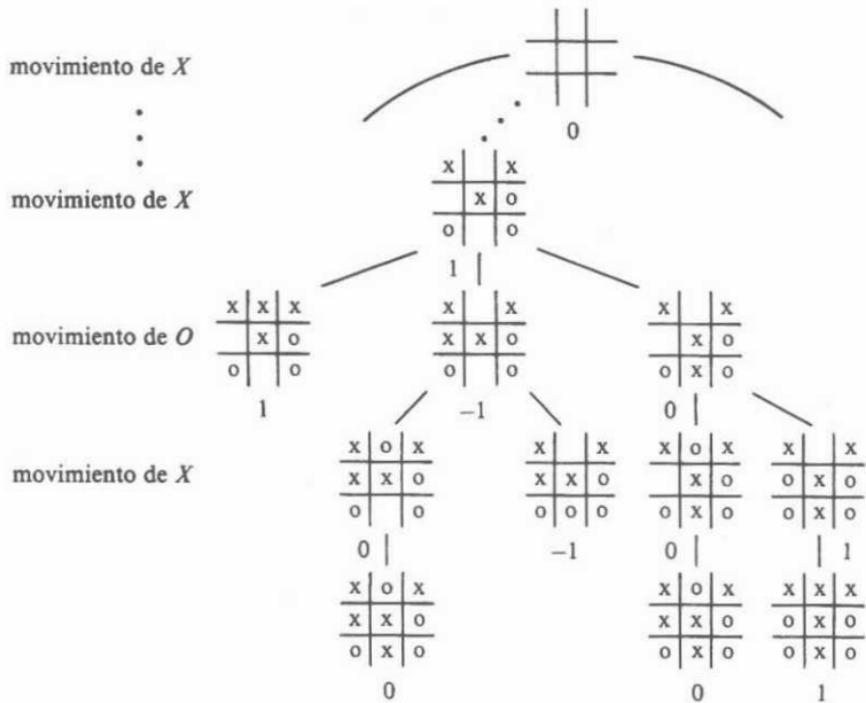


Fig. 10.16. Parte del árbol del juego de tres en raya.

Los valores se propagan hacia arriba en el árbol de acuerdo con la siguiente regla: si un nodo corresponde a una posición del tablero donde hay movimiento del jugador 1, el valor será el máximo de los valores de los hijos de ese nodo. Esto es, se supone que el jugador 1 hará el movimiento más favorable para él, o sea, el que produce el resultado con mayor valor. Si el nodo corresponde al movimiento del jugador 2, entonces el valor será el mínimo de los valores de los hijos. Es decir, se presume que el jugador 2 hará su movimiento más favorable, consiguiendo, de ser posible, una derrota para el jugador 1, o bien un empate como siguiente preferencia.

Ejemplo 10.6. Los valores de los tableros se encuentran en la figura 10.16. La hojas que significan triunfos para O tienen el valor -1 , los empates toman el 0 , y los triunfos de X tienen $+1$. En seguida se procede hacia arriba en el árbol. En el nivel 8, donde sólo queda un cuadro vacío, y es un movimiento de X , los valores de los tableros no resueltos son el «máximo» del hijo en el nivel 9.

En el nivel 7, correspondiente a un movimiento de O , hay dos opciones y se toma como valor para un nodo interno el mínimo de los valores de sus hijos. El tablero de la extrema izquierda, mostrado en el nivel 7, es una hoja y tiene un valor 1 , porque corresponde a un triunfo de X . El segundo tablero del mismo nivel tiene un valor $\min(0, -1) = -1$, mientras que el tercero tiene valor $\min(0, 1) = 0$. El único tablero mostrado en el nivel 6, que corresponde a un movimiento de X , tiene un valor $\max(1, -1, 0) = 1$, lo que significa que existe una selección que X puede hacer para ganar; en tal caso, el triunfo es inmediato. \square

Nótese que si la raíz tiene valor 1 , el jugador 1 tiene una estrategia ganadora; cualquiera que sea su turno, tiene la garantía de seleccionar un movimiento que le dé una posición del tablero con valor 1 . Cuando el turno corresponde al jugador 2, no tiene más opción que seleccionar un movimiento que deje una posición del tablero con valor 1 , lo cual significa una derrota para él. El hecho de que se suponga que el juego termina, garantiza que el primer jugador podrá ganar. Si la raíz tiene valor 0 , como sucede en el juego tres en raya, ningún jugador tiene una estrategia ganadora, pero por lo menos puede garantizarse un empate si juega lo mejor posible. Si la raíz tiene valor -1 , entonces el jugador 2 tiene una estrategia ganadora.

Funciones de utilidad

La idea de un árbol de juego, donde los nodos tienen valores -1 , 0 y 1 , puede generalizarse a árboles donde a las hojas se les da un número cualquiera (llamado la *utilidad*) como valor, y se aplican las mismas reglas de evaluación de nodos interiores: toma el máximo de los hijos en los niveles donde corresponda un movimiento del jugador 1, y el mínimo de los hijos en los niveles donde mueve el jugador 2.

Como ejemplo de donde son provechosas las utilidades generales, considérese un juego complejo como el ajedrez, donde el árbol de juego, aunque finito, es tan inmenso que la evaluación de abajo arriba no es posible. Los programas de ajedrez operan construyendo el árbol de juego con ese tablero como raíz para cada posición del tablero desde la cual se debe mover extendiéndose hacia abajo por varios niveles; el número exacto de niveles depende de la velocidad con que el computador puede tra-

bajar. Puesto que la mayor parte de las hojas del árbol son ambiguas, no indican triunfos, derrotas, ni empates, cada programa usa una función de las posiciones del tablero que intenta evaluar la probabilidad de que el computador gane desde esa posición. Por ejemplo, la diferencia entre las piezas afecta en gran medida tal estimación, al igual que algunos factores, como el poder defensivo alrededor de los reyes. Al usar esta función de utilidad, el computador puede estimar la probabilidad de un triunfo después de hacer cada uno de sus siguientes movimientos posibles en el supuesto de que se hará la mejor jugada siguiente de cada lado, y se escogerá el movimiento de mayor utilidad †.

Realización de la búsqueda con retroceso

Supóngase que se proporcionan las reglas de un juego ‡, esto es, sus movimientos válidos y reglas de terminación. Se desea construir su árbol de juego y evaluar la raíz, en la manera más obvia, y después visitar los nodos en orden final. El recorrido en orden final asegura que se visita un nodo interior n después de sus hijos, y entonces es posible evaluar n , tomando el mínimo o el máximo de los valores de sus hijos, lo que sea más apropiado.

El espacio de almacenamiento del árbol puede ser demasiado grande, pero con cuidado nunca se necesitará almacenar más de un camino, desde la raíz hasta un nodo, en cualquier momento. En la figura 10.17 se presenta el esbozo de un programa recursivo que representa el camino en el árbol por medio de la secuencia de las llamadas a procedimientos activos en cualquier instante. Ese programa supone lo siguiente:

1. Las utilidades son números reales en un intervalo limitado, por ejemplo -1 a $+1$.
2. La constante ∞ es mayor que cualquier utilidad positiva y su negativo es menor que cualquier utilidad negativa.
3. El tipo tipo_modo se declara

```
type
  tipo_modo = (MIN, MAX)
```

4. Existe un tipo tipo_tablero declarado de alguna forma adecuada para la representación de posiciones en el tablero.

† Inicialmente, otras cosas que hacen los programas para jugar al ajedrez son las siguientes.

1. Usar técnicas heurísticas para eliminar la consideración de ciertos movimientos que tal vez no serán buenos. Esto ayuda a expandir el árbol a más niveles en un tiempo dado.
2. Expandir «cadenas de captura» que son secuencias de movimientos de captura más allá del último nivel al que se expande el árbol normalmente. Esto ayuda a estimar con mayor exactitud la fuerza material relativa de las posiciones.
3. Podar el árbol de búsqueda por medio de la técnica alfa-beta, como se analizará más adelante en esta sección.

‡ Esto no implica que de esta forma sólo se puedan resolver «juegos». Como se verá en ejemplos posteriores, el «juego» puede representar en realidad la solución de un problema práctico.

5. Existe una función UTILIDAD que calcula la utilidad de cualquier tablero que sea una *hoja* (esto es, situación de triunfo, derrota o empate).

```

function busca ( B: tipo_tablero; modo: tipo_modo ) : real;
  { evalúa la utilidad del tablero B, en el supuesto
    de que es el movimiento del jugador 1 si modo = MAX, y es
    el movimiento del jugador 2 si modo = MIN. Devuelve la utilidad }
  var
    C: tipo_tablero; { un hijo del tablero B }
    valor: real; { valor mínimo o máximo temporal }
  begin
    (1)   if B es una hoja then
        return (utilidad(B))
    (2)   else begin
        { asigna el valor inicial mínimo o máximo de los hijos }
        (3)   if modo = MAX then
            valor := -∞
        (4)   else
            valor := ∞;
        (5)   for cada hijo C del tablero B do
            (6)     if modo = MAX then
                (7)       valor := máx(valor, busca(C, MIN))
            (8)     else
                (9)       valor := mín(valor, busca(C, MAX));
        (10)   return (valor)
    end
  end; { busca }

```

Fig. 10.17. Programa de búsqueda recursiva con el método de retroceso.

Otra implantación a considerar es por medio de un programa no recursivo que contenga una pila de tableros correspondientes a la secuencia de llamadas activas de *busca*. Las técnicas estudiadas en la sección 2.6 pueden usarse para construir dicho programa.

Poda alfa-beta

Hay una observación simple que permite dejar de considerar buena parte de un árbol de juego típico. Partiendo de la figura 10.17, el ciclo **for** de la línea (6) puede saltar ciertos hijos, con frecuencia muchos. Supóngase que se tiene un nodo *n*, como en la figura 10.18, y ya se ha determinado que *c*₁, el primero de los hijos de *n*, tiene valor 20. Como *n* es un nodo máx, se sabe que su valor es por lo menos 20. Ahora, supóngase que al continuar con la búsqueda se observa que *d*, un hijo de *c*₂, tiene valor 15. Como *c*₂, otro hijo de *n*, es un nodo mín, se sabe que el valor de *c*₂ no pue-

de exceder de 15. Así, cualquiera que sea el valor c_2 no afecta al valor de n ni de cualquier parente de n .

Así, en la situación de la figura 10.18, es posible ignorar la consideración de los hijos de c_2 que aún no se han examinado. Las reglas generales para pasar por alto o «podar» nodos implican la noción de valores tentativos y finales para los nodos. El valor *final* es el que se ha denominado hasta ahora simplemente «valor». Un valor

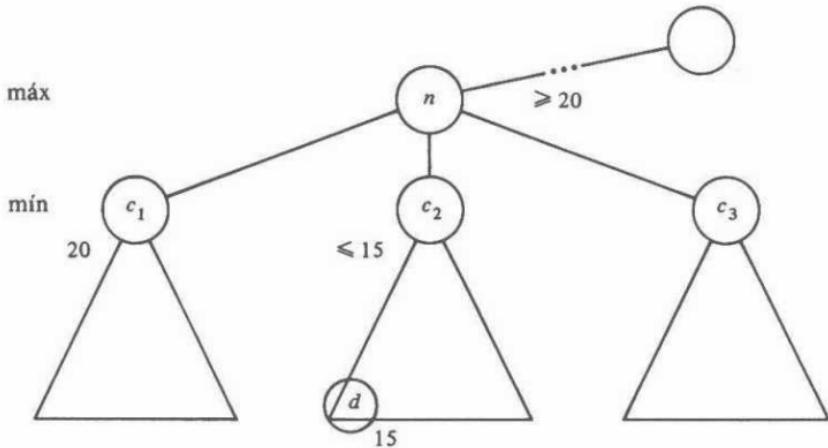


Fig. 10.18. Poda de los hijos de un nodo.

tentativo es una cota superior al valor del nodo *mín*, o una cota inferior al valor del nodo *máx*. Las reglas para el cálculo de los valores finales y tentativos son las siguientes.

1. Si todos los hijos de un nodo n se consideraron o se podaron, conviértase el valor tentativo de n en final.
2. Si un nodo *máx* n tiene valor tentativo v_1 y un hijo con valor final v_2 , entonces el valor tentativo de n se hace $\max(v_1, v_2)$. Si n es un nodo *mín*, se asigna su valor tentativo a $\min(v_1, v_2)$.
3. Si p es un nodo *mín* con parente q (un nodo *máx*), y p y q tienen valores tentativos v_1 y v_2 , respectivamente, con $v_1 \leq v_2$, entonces se pueden podar todos los hijos no considerados de p . También se pueden podar los hijos no considerados de p si p es un nodo *máx* (y por tanto, q es un nodo *mín*) y $v_2 \leq v_1$.

Ejemplo 10.7. Considérese el árbol de la figura 10.19. Suponiendo los valores mostrados en las hojas, se desea calcular el valor de la raíz; se empieza con un recorrido en orden final. Después de alcanzar el nodo D , por la regla (2) se asigna un valor tentativo de 2 al nodo C , que es el valor final de D . Entonces se explora E y se vuelve a C y después a B . Por la regla (1), el valor final de C se fija en 2 y el valor de B se hace tentativamente 2. La búsqueda desciende hasta G y después regresa a F . El valor de F se hace tentativamente 6. Por la regla (3), con p y q iguales a F y B , respec-

tivamente, se puede podar H . Esto es, no se necesita explorar el nodo H , ya que el valor tentativo de F nunca puede decrecer y ya es mayor que el valor de B , que no puede aumentar.

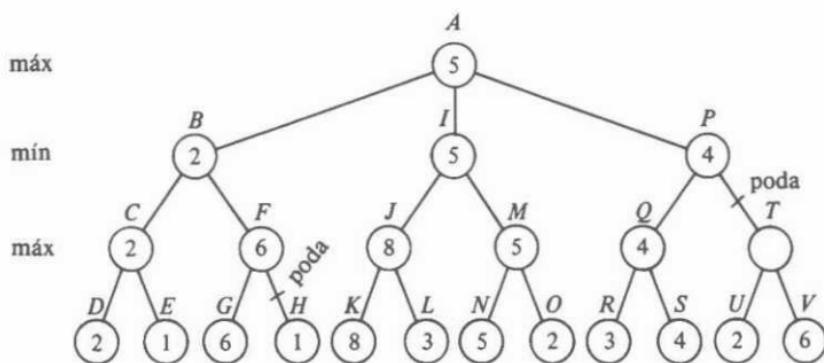


Fig. 10.19. Árbol de juego.

Continuando con el ejemplo, A tiene asignado un valor tentativo de 2 y la búsqueda prosigue hasta K . J tiene asignado un valor tentativo de 8. L no determina el valor del nodo máx J . I recibe un valor tentativo de 8. La búsqueda sigue hacia abajo hasta N , y a M se le asigna un valor tentativo de 5. El nodo O debe ser explorado, ya que 5, el valor tentativo de M , es menor que el valor tentativo de I . Se revisan los valores tentativos de I y A y la búsqueda continúa hasta R . Finalmente, se exploran R y S , y P recibirá un valor tentativo de 4. No es necesario buscar en T ni más abajo, ya que eso sólo disminuiría el valor de P , que ya es demasiado pequeño para afectar al valor de A . \square

Búsqueda de ramificación y acotamiento

Los juegos no son la única clase de «problemas» que pueden resolverse explorando exhaustivamente un árbol completo de posibilidades. Hay muchos problemas que exigen encontrar una configuración mínima o máxima de alguna clase que son susceptibles de resolución por medio de la búsqueda de retroceso de un árbol de todas las posibilidades. Los nodos del árbol pueden considerarse como conjuntos de configuraciones y los hijos de un nodo n representan cada uno un subconjunto de las configuraciones que n representa. Finalmente, cada hoja representa configuraciones sencillas o soluciones al problema, y se puede evaluar cada una de las configuraciones con el fin de saber si es la mejor solución encontrada hasta ese momento.

Si se es hábil en el diseño de la búsqueda, los hijos de un nodo representarán muchas menos configuraciones que el nodo mismo, así que no es necesario profundizar mucho para alcanzar las hojas. A fin de evitar que esta noción de búsqueda parezca muy vaga, he aquí un ejemplo concreto.

Ejemplo 10.8. Recuérdese de la sección anterior el problema del agente viajero, en el que se dio un «algoritmo ávido» para encontrar un buen recorrido, aunque no necesariamente el óptimo. Ahora se analiza cómo encontrar el recorrido óptimo por medio de la consideración sistemática de todos los recorridos. Una forma es tener en cuenta todas las permutaciones de los nodos y evaluar los recorridos que visitan los nodos en ese orden, recordando el mejor encontrado hasta el momento. El tiempo para esta aproximación es $O(n!)$ en un grafo con n nodos, ya que se deben considerar $(n-1)!$ permutaciones distintas †, y cada permutación toma un tiempo $O(n)$.

Se presentará un enfoque un poco distinto que no es mejor que el anterior en el peor caso, pero en el promedio, cuando se acopla con una técnica llamada «ramificación y acotamiento» —que se estudiará a continuación— produce la respuesta mucho más rápido que el método de «intentar con todas las permutaciones». Se empieza construyendo un árbol, con una raíz que representa todos los recorridos. Los recorridos son lo que se denominó «configuraciones» en el material preliminar. Cada nodo tiene dos hijos, y los recorridos que representa un nodo están divididos por esos dos hijos en dos grupos: los que tienen una arista particular y los que no. Por ejemplo, la figura 10.20 muestra porciones del árbol para el caso del PAV de la figura 10.15(a).

En la figura 10.20 se han considerado las aristas en orden lexicográfico (a, b) , (a, c) , ..., (a, f) , (b, c) , ..., (b, f) , (c, d) , y así sucesivamente. Claro está que se puede tomar cualquier otro orden. Obsérvese que no todos los nodos del árbol tienen dos hijos. Se pueden eliminar algunos hijos, porque las aristas seleccionadas no forman parte de un recorrido. Así, no hay nodo para los «recorridos que contienen (a, b) , (a, c) y (a, d) », porque a tendría grado 3 y el resultado no sería un recorrido. De forma semejante, al ir descendiendo en el árbol, se eliminan algunos nodos, porque alguna ciudad puede tener grado menor que 2. Por ejemplo, no habrá nodos para recorridos sin (a, b) , (a, c) , (a, d) o (a, e) . □

Procedimientos heurísticos necesarios para ramificación y acotamiento

Al utilizar ideas similares a las de la poda alfa-beta, se eliminan muchos más nodos del árbol de búsqueda que los mostrados en el ejemplo 10.8. Supóngase, para ser específicos, que el problema es minimizar alguna función, como el costo de un recorrido en el PAV. Supóngase, también, que se tiene un método para obtener una cota inferior en el costo de cualquier solución entre las del conjunto de soluciones representado por algún nodo n . Si la mejor solución encontrada hasta ahora cuesta menos que la cota inferior para el nodo n , no es necesario explorar ninguno de los nodos por debajo de n .

Ejemplo 10.9. Se revisará una forma de alcanzar una cota inferior en ciertos conjuntos de soluciones para el PAV, los representados por nodos en un árbol de solu-

† Obsérvese que no es necesario considerar todas las $n!$ permutaciones, ya que el punto de partida de un recorrido no importa. Por tanto, se consideran sólo aquellas permutaciones que empiecen con 1.

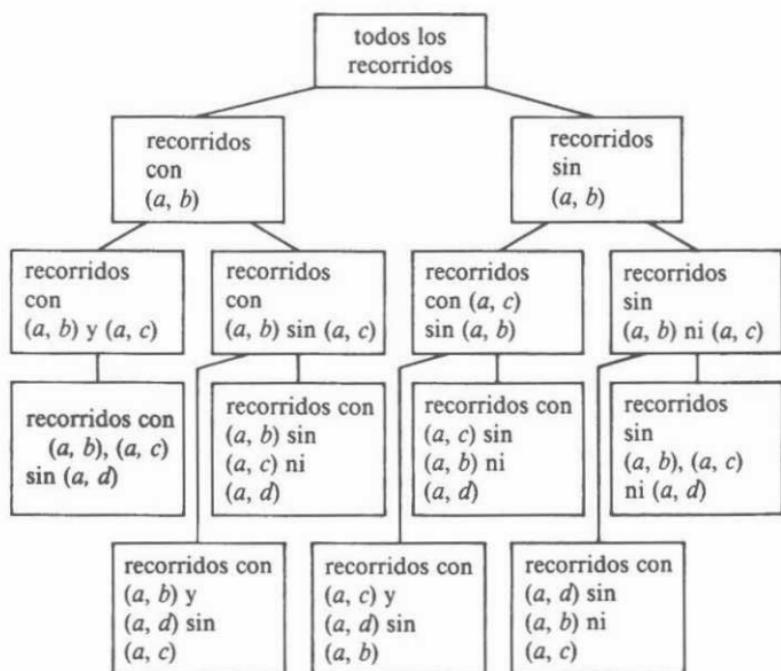


Fig. 10.20. Principio de un árbol solución para un caso del PAV.

ciones, como se sugirió en la figura 10.20. Primero, supóngase que se desea una cota inferior para todas las soluciones de un caso dado del PAV. Obsérvese que el costo de cualquier recorrido puede expresarse como la semisuma, sobre todos los nodos n , del costo de las dos aristas del recorrido adyacentes a n . Esta observación da lugar a la siguiente regla: la suma de las dos aristas de recorrido adyacentes al nodo n no es menor que la suma de las dos aristas de costo menor adyacentes a n . Así, ningún recorrido puede costar menos que la semisuma sobre todos los nodos n de las dos aristas de menor costo que inciden sobre n .

Por ejemplo, considérese el caso del PAV que se muestra en la figura 10.21. A diferencia del caso de la figura 10.15, la medida de la distancia de las aristas no es euclíadiana, es decir, no tiene relación con la distancia en el plano entre las ciudades que conecta. Tal medida del costo puede ser el tiempo o la tarifa del viaje, por ejemplo. En este caso, las aristas de menor costo adyacentes al nodo a son (a, d) y (a, b) , con un costo total de 5. Para el nodo b , se encuentran (a, b) y (b, e) con un costo total de 6. De igual forma, las dos aristas de menor costo adyacentes a c , d y e , totalizan 8, 7 y 9, respectivamente. Así pues, la cota inferior del costo de un recorrido es $(5+6+8+7+9)/2 = 17.5$.

Ahora, supóngase que se pide una cota inferior en el costo de un subconjunto de recorridos definidos por algún nodo en el árbol de búsqueda. Si el árbol de búsqueda se construye como en el ejemplo 10.8, cada nodo representa recorridos definidos

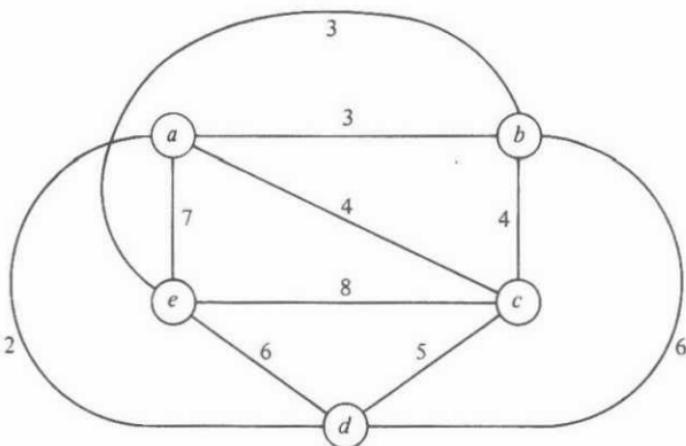


Fig. 10.21. Un caso del PAV.

por un conjunto de aristas que deben estar en el recorrido y un conjunto de aristas que tal vez no estén. Esas restricciones alteran las opciones para escoger las dos aristas de menor costo en cada nodo. Es evidente que una arista restringida a permanecer en cualquier recorrido debe incluirse entre las dos aristas seleccionadas, sin importar si son del menor o segundo menor costo o no †. Asimismo, una arista restringida a estar fuera no puede seleccionarse, aunque su costo sea menor.

Así, si existe la restricción de incluir la arista (a, e) y excluir (b, c) , las dos aristas para el nodo a son (a, d) y (a, e) , con un costo total de 9. Para b , se seleccionan (a, b) y (b, e) , igual que antes, con un costo total de 6. Para c , no es posible elegir (b, c) , y se eligen (a, c) y (c, d) , con un costo total de 10. Para d , se seleccionan (a, d) y (c, d) , como antes, mientras que para e , se deben escoger (a, e) y (b, e) . La cota inferior para esas restricciones es $(9+6+9+7+10)/2 = 20.5$. □

Ahora se construye el árbol de búsqueda a partir de lo sugerido en el ejemplo 10.8. Se consideran las aristas en orden lexicográfico, como en ese ejemplo. Por cada ramificación, al considerar los dos hijos de un nodo, se intenta inferir decisiones adicionales sobre las aristas que deben incluirse o excluirse de los recorridos representados por esos nodos. Las reglas utilizadas para esas inferencias son:

1. Si la exclusión de una arista (x, y) hiciera imposible para x o y tener dos aristas adyacentes en el recorrido, entonces debe incluirse (x, y) .
2. Si incluir (x, y) hace que x o y tengan más de dos aristas adyacentes en el recorrido, o que se complete un ciclo que no sea recorrido con las aristas ya incluidas, entonces debe excluirse (x, y) .

† Se verá que las reglas para la construcción del árbol de búsqueda eliminan cualquier conjunto de restricciones que no puedan producir un recorrido, por ejemplo, porque se requieran tres aristas adyacentes a un nodo para formar parte del recorrido.

Al ramificar, después de hacer todas las inferencias posibles, se calculan las cotas inferiores de ambos hijos. Si la cota inferior de un hijo es tan grande o mayor que el recorrido de costo menor encontrado hasta ese momento, se «poda» ese hijo y no se construyen ni consideran sus descendientes. Curiosamente, hay situaciones donde la cota inferior de un nodo n es menor que la mejor solución existente encontrada, y aún así ambos hijos de n se pueden podar, porque sus cotas inferiores exceden del costo de la mejor solución encontrada hasta el momento.

Si no es posible podar ningún hijo, como regla heurística se considerará primero el hijo con la cota inferior más pequeña, con la esperanza de alcanzar más rápido una solución más económica que la mejor encontrada hasta el momento †. Después de considerar un hijo, se debe evaluar otra vez si puede podarse su hermano, ya que pudo haberse encontrado una mejor solución nueva. Para el caso de la figura 10.21, se obtiene árbol de la figura 10.22. Para interpretar los nodos de ese árbol, es útil entender que las letras mayúsculas son los nombres de los nodos del árbol de búsqueda; los números son las cotas inferiores, y se listan las restricciones aplicadas al nodo, pero no sus antecesores, escribiendo xy si la arista (x, y) debe incluirse, y \bar{xy} , si (x, y) debe excluirse. Obsérvese también que las restricciones introducidas en un nodo se aplican a todos sus descendientes. Así, para obtener todas las restricciones aplicables en un nodo debe seguirse el camino desde ese nodo hasta la raíz.

Por último, se hace hincapié en que, como para la búsqueda de retroceso general, se construye el árbol nodo por nodo, reteniendo sólo un camino, como en el algoritmo recursivo de la figura 10.17, o su contraparte no recursiva. Quizá se prefiera la versión no recursiva, para mantener dentro de una pila la lista de restricciones.

Ejemplo 10.10. La figura 10.22 muestra el árbol de búsqueda para el caso del PAV de la figura 10.22. Con el fin de ver cómo se construye, se parte de la raíz A de la figura 10.22. La primera arista en orden lexicográfico es (a, b) , así que se consideran los hijos B y C , correspondientes a las restricciones ab y \bar{ab} , respectivamente. De momento no hay una «mejor solución hasta aquí», por lo que se considerarán B y C ‡. Forzar la inclusión de (a, b) no sube la cota inferior, pero excluir tal arista produce la cota 18.5, ya que las dos aristas válidas más económicas para los nodos a y b totalizan 6 y 7, respectivamente, comparadas con 5 y 6 sin restricciones. Siguiendo el procedimiento heurístico, se considerarán primero los descendientes del nodo B .

La siguiente arista en orden lexicográfico es (a, c) . Se introducen los hijos D y E correspondientes a los recorridos en que (a, c) está incluida y excluida, respectivamente. En el nodo D , se infiere que ni (a, d) ni (a, e) pueden estar en un recorrido; de otra forma, a puede tener demasiadas aristas incidentes. De acuerdo con el procedimiento heurístico, se analiza E antes que D , para ramificar sobre la arista (a, d) . Los hijos F y G se introducen con cotas inferiores 18 y 23, respectivamente. Para cada uno de esos hijos se conocen unas 3 aristas incidentes sobre a , por lo que se puede deducir algo acerca de la arista restante (a, e) .

† Una alternativa es usar un procedimiento heurístico para obtener una buena solución con las restricciones requeridas para cada hijo. Por ejemplo, se deberá poder modificar el algoritmo ávido del PAV para respetar las restricciones.

‡ Se puede empezar con alguna solución encontrada heurísticamente, como la ávida, aunque eso no afecte a nuestro ejemplo. La solución ávida para la figura 10.21 tiene un costo de 21.

Considérense primero los hijos de F . La primera arista restante en orden lexicográfico es (b, c) , y si se incluye, al haber incluido ya (a, b) , no se pueden incluir (b, d) ni (b, e) . Al haber eliminado (a, e) y (b, e) , se deben tener (c, e) y (d, e) . No se puede tener (c, d) , porque entonces c y d tendrían 3 aristas incidentes. Queda, por último un recorrido (a, b, c, e, d, a) , cuyo costo es 23. Igualmente, el nodo I , donde (b, c) está excluida, puede probarse que representa sólo el recorrido (a, b, e, c, d, a) , de costo 21. Este recorrido tiene el menor costo encontrado hasta el momento.

Ahora se retrocede a E y se considera su segundo hijo, G . Pero G tiene una cota inferior de 23, la cual excede al mejor costo actual, 21. Así, se poda G . Ahora, se retrocede hasta B y se considera su otro hijo, D . La cota inferior en D es 20.5, pero debido a que los costos son enteros, se sabe que ningún recorrido representado por D tiene costo menor que 21. Dado que ya existe un recorrido tan económico, no es necesario explorar los descendientes de D y, por tanto, se poda. Ahora, se retrocede hasta A y se considera su segundo hijo, C .

En el nivel del nodo C , sólo se ha considerado la arista (a, b) . Los nodos J y K se introducen como hijos de C . J corresponde a aquellos recorridos que tienen a (a, c) , pero no a (a, b) , y su cota inferior es 18.5. K corresponde a los recorridos que no contienen (a, b) ni (a, c) , y se infiere que esos recorridos tienen a (a, d) y (a, e) . La cota inferior de K es 21 y se puede podar de inmediato, dado que ya se conoce un recorrido de menor costo.

A continuación se consideran los hijos en J , L y M , y se poda M porque su cota inferior excede del costo del mejor recorrido actual. Los hijos de L son N y P , correspondientes a recorridos que tienen (b, c) , y que excluyen a (b, c) . Considerando el grado de los nodos b y c , y recordando que las aristas seleccionadas no pueden formar un ciclo con menos de cinco ciudades, se infiere que los nodos N y P representan recorridos individuales. Uno de esos, (a, c, b, e, d, a) , tiene menor costo que cualquier otro, 19. Ya se ha explorado o podado todo el árbol y, por tanto, se ha terminado. □

10.5 Algoritmos de búsqueda local

Algunas veces, la siguiente estrategia producirá una solución óptima para un problema.

1. Empezar con una solución aleatoria.
2. Aplicar a la solución actual una transformación de un conjunto dado de transformaciones para mejorar la solución; la mejora es la nueva solución «actual».
3. Repetir hasta que ninguna transformación del conjunto pueda mejorar la solución actual.

La solución resultante puede ser óptima o no. En principio, si «el conjunto de transformaciones dado» incluye todas las que toman una solución y la reemplazan por otra, entonces no se parará hasta alcanzar la solución óptima. Pero, en ese caso, el tiempo para aplicar (2) es el mismo que el necesario para examinar todas las soluciones, y todo el enfoque no tiene mucho sentido.

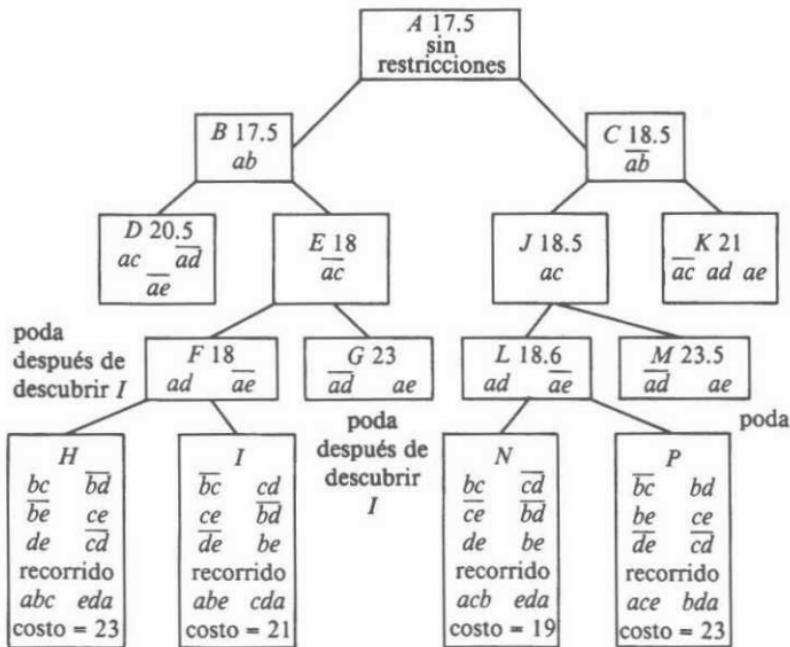


Fig. 10.22. Árbol de búsqueda para la solución del PAV.

El método tiene sentido cuando se puede restringir el conjunto de transformaciones a un conjunto pequeño, de modo que en poco tiempo se puedan considerar todas las transformaciones posibles; tal vez deben permitirse las $O(n^2)$ u $O(n^3)$ transformaciones cuando el problema es de «tamaños n . Si el conjunto de transformaciones es pequeño, es natural considerar las soluciones, que pueden ser transformadas entre sí en un paso como «cercanas». Las transformaciones se llaman «transformaciones locales», y el método se conoce como *búsqueda local*.

Ejemplo 10.11. Un problema que se puede resolver con exactitud mediante búsqueda local es el de los árboles abarcadores minimales. Las transformaciones locales son aquellas en las que se toma alguna arista que no se encuentra en el árbol abarcador actual, se agrega al árbol, que debe producir un ciclo único, y después eliminar exactamente una arista del ciclo (probablemente la de costo mayor) para formar un árbol nuevo.

Por ejemplo, considérese el grafo de la figura 10.21. Se empieza con el árbol mostrado en la figura 10.23(a). Una transformación que se puede realizar es agregar la arista (d, e) y quitar otra del ciclo formado, el cual es (e, a, c, d, e) . Si se elimina la arista (a, e) , se reduce el costo del árbol de 20 a 19. Esta transformación puede hacerse dejando el árbol de la figura 10.23(b), al cual se intenta aplicar de nuevo una transformación de mejora, como insertar la arista (a, d) y eliminar la (c, d) del ciclo

formado. El resultado se muestra en la figura 10.23(c). Después es posible introducir (a, b) y eliminar (b, c) , como en la figura 10.23(d), y más tarde introducir (b, e) en favor de (d, e) . El árbol resultante de la figura 10.23(e) es minimal. Se puede comprobar si cada arista ausente en ese árbol tiene costo mayor que cualquier otra arista dentro del ciclo que forma. Así, ninguna transformación es aplicable a la figura 10.23(c). \square

El tiempo requerido por el algoritmo del ejemplo 10.11 en un grafo de n nodos y a aristas depende del número de veces que se necesite mejorar la solución. La sola prueba de que ninguna transformación es aplicable puede llevar un tiempo $O(na)$, ya que deben probarse a aristas, y cada una puede formar un ciclo de longitud cercana a n . Así, el algoritmo no es tan bueno como los algoritmos de Prim o de Kruskal, pero sirve como ejemplo de que puede obtenerse una solución óptima por búsquedas locales.

Algoritmos de aproximación por búsqueda local

Los algoritmos de búsqueda local han tenido gran efectividad como métodos heurísticos para la solución de problemas cuyas soluciones exactas requieren tiempo exponencial. Un método común de búsqueda es empezar con un número de soluciones aleatorias y aplicar las transformaciones locales a cada una, hasta alcanzar una solución *localmente óptima*, que ninguna transformación pueda mejorar. Con frecuencia se alcanzarán diferentes soluciones localmente óptimas, a partir de la mayor parte o de todas las soluciones iniciales aleatorias, como se sugiere en la figura

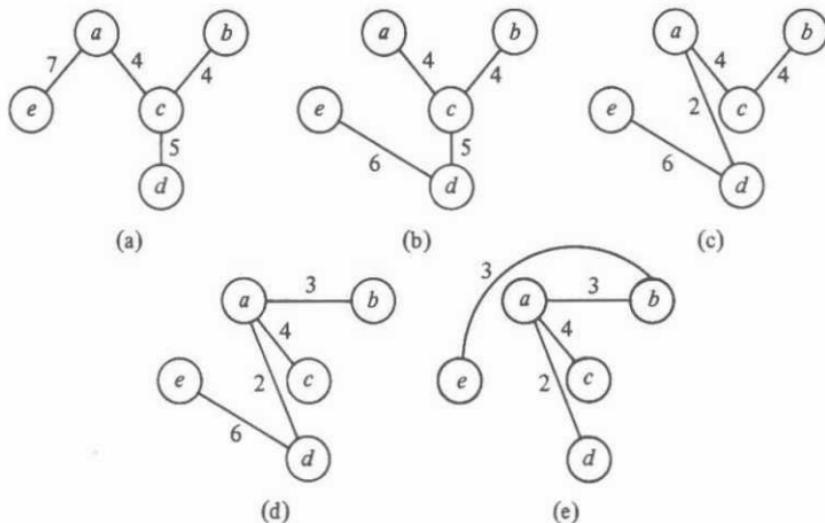


Fig. 10.23. Búsqueda local para un árbol abarcador minimal.

10.24. Si se tiene suerte, una de ellas será *globalmente optimal*, esto es, tan buena como cualquier otra solución.

En la práctica, tal vez no se pueda encontrar una solución globalmente óptima como la que se sugiere en la figura 10.24, ya que el número de soluciones localmente óptimas puede ser enorme. Sin embargo, es posible escoger por lo menos aquella solución localmente óptima que tenga el menor costo entre todas las que se encontraron. Como el número de clases de transformaciones locales utilizadas para resolver varios problemas es muy grande, se cerrará la sección con dos ejemplos: el PAV y un problema simple de «colocación de paquetes».

Problema del agente viajero

El PAV es uno de los problemas en que las técnicas de búsqueda local han sido muy satisfactorias. La transformación más simple que se ha usado se conoce como «opción doble». Consiste en tomar dos aristas cualesquiera, tales como (A, B) y (C, D) de la figura 10.25, quitándolas y reconectando sus extremos para formar un recorrido nuevo. En esta figura, el recorrido nuevo va desde B hasta C en el sentido de las manecillas del reloj, después sigue por la arista (C, A) , desde A hasta D en sentido contrario al de las manecillas del reloj y finalmente por la arista (D, B) . Si la suma de las longitudes de (A, C) y (B, D) es menor que la suma de las longitudes de (A, B) y (C, D) , se tendrá un recorrido mejorado †. Obsérvese que no se pueden conectar

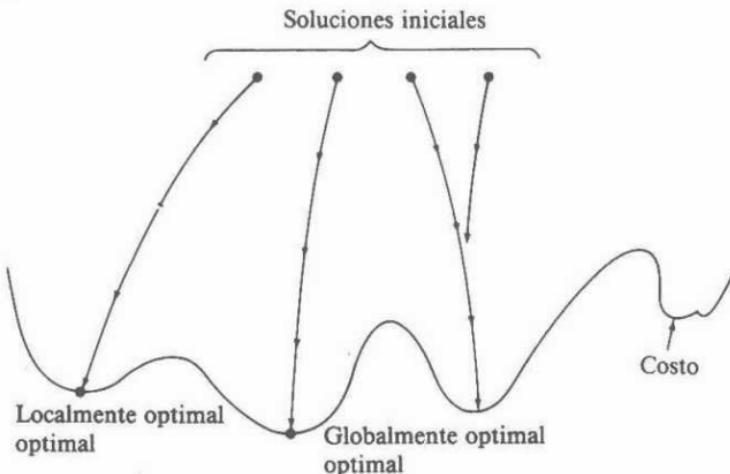


Fig. 10.24. Búsqueda local en el espacio de soluciones.

† No hay que dejarse engañar por el dibujo de la figura 10.25. Es cierto que si las longitudes de las aristas son distancias en el plano, las aristas con puntos deben ser mayores que las que reemplazan. De hecho, las opciones dobles en el plano mejoran el costo exactamente cuando las aristas eliminadas, (A, B) y (C, D) , se cruzan, mientras que las reemplazadas no lo hacen. Sin embargo, no existe motivo alguno para suponer que las distancias de la figura 10.25 sean distancias en un plano, o si lo son, puede suceder que se hayan cruzado (A, B) y (C, D) y no (A, C) y (B, D) .

A con *D* y *B* con *C*, pues, el resultado no sería un recorrido, sino dos ciclos disjuntos. Para encontrar un recorrido localmente óptimal, se comienza con un recorrido aleatorio, y se consideran todos los pares de aristas no adyacentes, como (*A*, *B*) y (*C*, *D*) de la figura 10.25. Si el recorrido puede mejorarse reemplazando esas aristas por (*A*, *C*) y (*B*, *D*), se hace así, y se prosigue considerando los pares de aristas que no se consideraron anteriormente. Obsérvese que las aristas introducidas (*A*, *C*) y (*B*, *D*) deben emparejarse cada una con todas las demás del recorrido, ya que pueden resultar mejoras adicionales.

Ejemplo 10.12. Reconsidérese la figura 10.21, para comenzar con el recorrido de la figura 10.26(a). Se reemplazan (*a*, *e*) y (*c*, *d*), cuyo costo total es 12, por (*a*, *d*) y (*c*, *e*), con un costo total de 1, como se muestra en la figura 10.26(b). Despues se sustituyen (*a*, *b*) y (*c*, *e*) por (*a*, *c*) y (*b*, *e*), dando el recorrido óptimal mostrado en la figura 10.26(c). Se puede comprobar que no haya ningún par de aristas que se pueda eliminar de esta figura y ser reemplazado con ventaja por aristas cruzadas con los mismos extremos. Como un caso adicional, (*b*, *c*) y (*d*, *e*) juntas tienen el costo relativamente alto de 10. Pero (*c*, *e*) y (*b*, *e*) son peores, al costar juntas 14. □

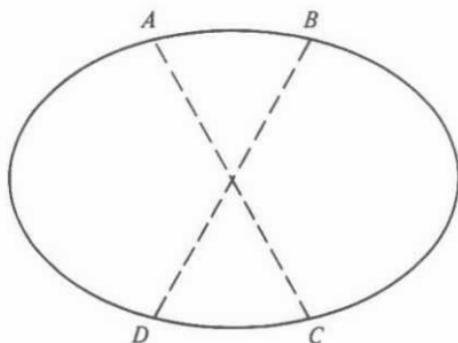


Fig. 10.25. Opción doble.

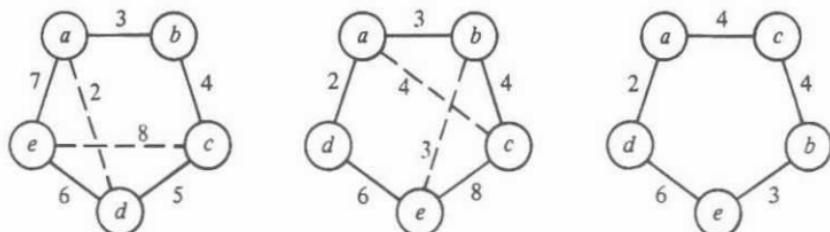


Fig. 10.26. Optimación de un caso del PAV con opción doble.

Se puede generalizar la opción doble a opción k para alguna k constante, donde se eliminan hasta k aristas y se reconectan los segmentos restantes en cualquier orden hasta producir un recorrido. Obsérvese que no se requiere que las aristas suprimidas no sean adyacentes en general, si bien para el caso de la opción doble no tenía sentido considerar la supresión de dos aristas adyacentes. Obsérvese también que para $k > 2$, existe más de una forma de conectar los segmentos. Por ejemplo, la figura 10.27 muestra el proceso general de opción triple utilizando cualquiera de los ocho conjuntos de aristas siguientes.

1. $(A, F), (D, E), (B, C)$ (como fue el recorrido)
2. $(A, F), (C, E), (D, B)$ (una opción doble)
3. $(A, E), (F, D), (B, C)$ (otra opción doble)
4. $(A, E), (F, C), (B, D)$ (una opción triple real)
5. $(A, D), (C, E), (B, F)$ (otra opción triple real)
6. $(A, D), (C, F), (B, E)$ (otra opción triple real)
7. $(A, C), (D, E), (B, F)$ (una opción doble)
8. $(A, C), (D, F), (B, E)$ (una opción triple)

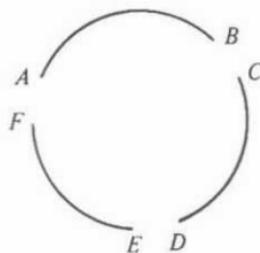


Fig. 10.27. Segmentos de un recorrido después de la supresión de tres aristas.

Es fácil verificar que, para una k fija, el número de diferentes transformaciones con opción k que se necesita considerar, si existen n vértices, es $O(n^k)$. Por ejemplo, el número exacto es $n(n-3)/2$ para $k = 2$. Sin embargo, el tiempo requerido para obtener un recorrido localmente óptimo puede ser mucho mayor que esto, ya que se pudieron haber realizado muchas transformaciones locales antes de alcanzar un recorrido localmente óptimo, y cada transformación de mejora introduce aristas nuevas que pueden participar en transformaciones posteriores que mejoran aún más el recorrido fijado. Lin y Kernighan [1973] han encontrado que la opción de profundidad variable es un método muy poderoso en la práctica, y tiene una buena oportunidad de obtener el recorrido óptimo en problemas de ciudades entre 40 y 100.

Colocación de paquetes

El problema de la colocación unidimensional de paquetes puede plantearse como sigue. Se tiene un grafo no dirigido, cuyos vértices se denominan «paquetes». Las aristas están etiquetadas por sus «pesos», y el peso $w(a, b)$ de la arista (a, b) es el nú-

mero de hilos existente entre los paquetes a y b . El problema es ordenar los vértices p_1, p_2, \dots, p_n de manera que la suma de $|i - j| w(p_i, p_j)$ en todos los pares i y j se minimice. Esto es, se desea minimizar la suma de las longitudes de los hilos necesarios para conectar todos los paquetes con el número requerido de hilos.

Al problema de la colocación de paquetes se le han dado muchas aplicaciones. Por ejemplo, los «paquetes» pueden ser tarjetas lógicas en un fichero, y el peso de una interconexión entre tarjetas es el número de hilos que las conectan. Un problema similar surge con el diseño de circuitos integrados a partir de disposiciones de módulos estándar e interconexiones entre ellos. Una generalización del problema de la colocación unidimensional de paquetes permite la colocación de los «paquetes», que tienen alto y ancho, en una región bidimensional, al tiempo que se minimiza la suma de las longitudes de los hilos entre los paquetes. Este problema también es aplicable en el diseño de circuitos integrados, entre otras áreas.

Hay ciertas transformaciones locales que se podrían utilizar para encontrar los óptimos locales para diversos casos del problema de la colocación unidimensional de paquetes. He aquí algunas:

1. Intercambiar los paquetes adyacentes p_i y p_{i+1} si el orden resultante es de costo menor. Sea $L(j)$ la suma de pesos de las aristas que se extienden hacia la izquierda de p_j , esto es, $\sum_{k=j+1}^n w(p_k, p_j)$. Igualmente, sea $R(j) \sum_{k=j+1}^n w(p_k, p_j)$. Se obtienen mejoras si $L(i) - R(i) + R(i+1) - L(i+1) + 2w(p_i, p_{i+1})$ es negativa. Sería útil verificar esta fórmula calculando los costos antes y después del intercambio y tomar la diferencia.
2. Tomar un paquete p_i e insertarlo entre p_i y p_{i+1} para algunas i y j .
3. Intercambiar dos paquetes cualesquiera p_i y p_j .

Ejemplo 10.13. Supóngase que se tiene el grafo de la figura 10.21 para representar un caso de la colocación de paquetes. Se considerará sólo conjunto de transformaciones simples (1). Una colocación inicial, a, b, c, d, e , se muestra en la figura 10.28(a); tiene un costo de 97. Obsérvese que la función de costo pesa las aristas por su distancia, de modo que (a, e) contribuye $4 \times 7 = 28$ al costo de 97. Considérese el intercambio de d y e . Se tiene que $L(d) = 13$, $R(d) = 6$, $L(e) = 24$, y $R(e) = 0$. Así, $L(d) - R(d) + R(e) - L(e) + 2w(d, e) = -5$, y se puede intercambiar d y e para mejorar la colocación a (a, b, c, e, d) , con un costo de 92, como se muestra en la figura 10.28(b).

En la figura 10.28(b), se intercambian c y e con cierta ventaja, produciendo la figura 10.28(c), cuya colocación (a, b, e, c, d) tiene un costo 91. La figura 10.28(c) es localmente óptima para el conjunto de transformaciones (1). No es globalmente óptima; (a, c, e, d, b) tiene un costo de 84. □

Como en el PAV, es difícil estimar exactamente el tiempo que lleva alcanzar un óptimo local. Sólo se observa que para el conjunto de transformaciones (1), hay sólo $n-1$ transformaciones a considerar. Más aún, si se calculan $L(i)$ y $R(i)$ una vez, tan sólo es necesario cambiarlas cuando p_i se intercambia con p_{i-1} o p_{i+1} . Además, calcular de nuevo es fácil. Si p_i y p_{i+1} se intercambian, por ejemplo, los nuevos $L(i)$ y

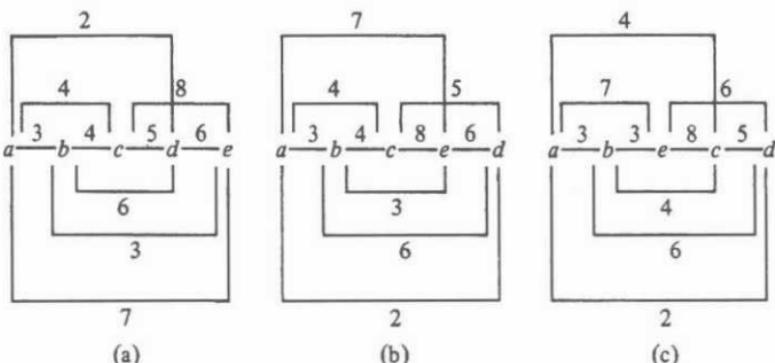


Fig. 10.28. Optimaciones locales.

$R(i)$ son, respectivamente, $L(i+1) - w(p_i, p_{i+1})$ y $R(i+1) + w(p_i, p_{i+1})$. Así, un tiempo $O(n)$ es suficiente para probar una transformación de mejora y calcular de nuevo los $L(i)$ y $R(i)$. También se requiere un tiempo $O(n)$ para asignar valores iniciales a $L(i)$ y $R(i)$, si se emplea la recurrencia

$$L(1) = 0$$

$$L(i) = L(i-1) + w(p_{i-1}, p_i)$$

y una recurrencia similar para R .

En comparación, los conjuntos de transformaciones (2) y (3) tienen, cada uno, $O(n^2)$ miembros. Por tanto, llevará un tiempo $O(n^2)$ tan sólo confirmar que se tiene una solución localmente óptima. Sin embargo, como sucedió con el conjunto (1), no es posible acotar con exactitud el tiempo total requerido al efectuar una secuencia de mejoras, ya que cada mejora puede crear oportunidades adicionales de perfeccionamiento.

Ejercicios

- 10.1 ¿Cuántos movimientos realizan los algoritmos para desplazar n discos en el problema de las torres de Hanoi?
- *10.2 Pruébese que el algoritmo recursivo (dividir para vencer) de las torres de Hanoi y el algoritmo simple no recursivo descrito al principio de la sección 10.1 efectúan los mismos pasos.
- 10.3 Muéstrense las acciones del algoritmo de la multiplicación de enteros de dividir para vencer de la figura 10.3 al multiplicar 1011 por 1101.
- *10.4 Generalícese la programación del torneo de tenis de la sección 10.1 para torneos en los que el número de jugadores no sea una potencia de dos. Suge-

rencia. Si el número de jugadores n es impar, un jugador debe quedar sin jugar algún día, y el torneo necesitará n días para terminar, en vez de $n - 1$. No obstante, si hay dos grupos con un número impar de jugadores, los que obtienen el día sin jugar de cada grupo pueden jugar entre sí.

- 10.5** El número de combinaciones de m objetos de n , denotado por $\binom{n}{m}$, para $n \geq 1$ y $0 \leq m \leq n$, se puede definir recursivamente por

$$\binom{n}{m} = 1 \text{ si } m = 0 \text{ o } m = n$$

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \text{ si } 0 < m < n$$

- a) Proporcione una función recursiva para calcular $\binom{n}{m}$.
- b) ¿Cuál es el tiempo de ejecución en el peor caso, como una función de n ?
- c) Obténgase un algoritmo de programación dinámica para calcular $\binom{n}{m}$. *Sugerencia:* El algoritmo construye una tabla conocida como triángulo de Pascal.
- d) ¿Cuál es el tiempo de ejecución de la respuesta a c) como una función de n ?

- 10.6** Otra forma de calcular el número de combinaciones de m objetos de n es mediante $(n)(n-1)(n-2)\dots(n-m+1)/(1)(2)\dots(m)$.

- a) ¿Cuál es el tiempo de ejecución en el peor caso de este algoritmo como una función de n ?
- *b) ¿Es posible calcular la función de las «apuestas de la Serie Mundial de Béisbol» $P(i, j)$ de la sección 10.2 de una forma similar? ¿Con qué rapidez puede realizarse este cálculo?

- 10.7** a) Escríbase otra vez el cálculo de las apuestas de la figura 10.7 para tener en cuenta el hecho de que el primer equipo tiene una probabilidad p de ganar cualquier partido dado.
 b) Si los Dodgers han ganado un partido, y los Yankees, dos, pero los Dodgers tienen una probabilidad 0.6 de ganar cualquier juego, ¿quién es el ganador más probable de la Serie Mundial?

- 10.8** El cálculo de las apuestas de la figura 10.7 requiere un espacio $O(n^2)$. Reescríbase el algoritmo para usar sólo un espacio $O(n)$.

- ***10.9** Pruébese que de la ecuación (10.4) resultan exactamente $\binom{i+j}{i}$ llamadas a P .

- 10.10** Encuéntrese una triangulación minimal para un octágono regular, suponiendo que las distancias son euclidianas.

- 10.11** El problema de división en párrafos, en una forma muy simple, puede formularse como sigue: se da una secuencia de palabras p_1, p_2, \dots, p_k de longitudes l_1, l_2, \dots, l_k , que se desea agrupar en líneas de longitud L . Las palabras están separadas por espacios cuya amplitud ideal es b , pero los espa-

cios pueden reducirse o ampliarse si es necesario (pero sin solapamiento de palabras), de tal forma que una línea $p_i, p_{i+1} \dots p_j$ tenga exactamente longitud L . Sin embargo, la multa por reducción o ampliación es la magnitud de la cantidad total que los espacios se comprimen o amplían. Esto es, el costo de fijar la línea $p_i, p_{i+1} \dots p_j$ es $(j-i) |b' - b|$, donde b' , el ancho real de los espacios, es $(L - l_i - l_{i+1} - \dots - l_j) / (j-i)$. No obstante, si $j = k$ (la última línea), el costo es cero, a menos que $b' < b$, ya que no es necesario ampliar la última línea. Plantéese un algoritmo de programación dinámica para encontrar la separación de menor costo de p_1, p_2, \dots, p_k en líneas de longitud L . *Sugerencia:* Para $i = k, k-1, \dots, 1$, calcúlese el menor costo de fijar p_i, p_{i+1}, \dots, p_k .

- 10.12** Supóngase que se dan n elementos x_1, x_2, \dots, x_n relacionados por un orden lineal $x_1 < x_2 < \dots < x_n$, y que se desea disponer los m elementos dentro de un árbol binario de búsqueda. Supóngase que p_i es la probabilidad que se requiere de que una solicitud para encontrar un elemento se refiera a x_i . Entonces, para cualquier árbol binario de búsqueda, el costo promedio de una búsqueda es $\sum_{i=1}^n p_i(d_i + 1)$, donde d_i es la profundidad del nodo que contiene a x_i . Dados los p_i y en el supuesto de que los x_i nunca cambian, es posible encontrar un árbol binario de búsqueda que reduzca el costo. Dése un algoritmo de programación dinámica para hacerlo. ¿Cuál es el tiempo de ejecución de ese algoritmo? *Sugerencia:* Calcúlese para todas las i y j el costo de búsqueda óptimal entre todos los árboles que contienen sólo $x_i, x_{i+1}, \dots, x_{i+j-1}$, esto es, los j elementos comenzando desde x_i .
- **10.13** ¿Para qué valores de monedas produce una solución óptima el algoritmo ávido que calcula cambios de la sección 10.3?
- 10.14**
- Escríbase el algoritmo de triangulación recursivo analizado en la sección 10.2.
 - Muéstrese que el algoritmo recursivo produce exactamente 3^{s-4} llamadas a problemas no triviales cuando se inicia en un problema de tamaño $s \geq 4$.
- 10.15** Describáse un algoritmo ávido para
- el problema de la colocación unidimensional de paquetes, y
 - el problema de la separación en párrafos (Ejercicio 10.11). Proporciójense un ejemplo donde el algoritmo no produzca una respuesta óptima, o demuéstrese que tal ejemplo no existe.
- 10.16** Plantéese una versión no recursiva del algoritmo del árbol de búsqueda de la figura 10.17.
- 10.17** Considérese un árbol de juego en el cual hay seis fichas y los jugadores 1 y 2 se turnan de una a tres fichas. El jugador que tome la última ficha pierde el juego.

- a) Dibújese el árbol de juego completo.
 - b) Si el árbol de juego completo se explorara mediante la técnica de poda alfa-beta y se analizaran primero los nodos que representan las configuraciones con el menor número de fichas, ¿qué nodos se podarían?
 - c) ¿Quién resulta vencedor si ambos jugadores hacen sus mejores jugadas?
- *10.18 Desarróllese un algoritmo de ramificación y acotamiento para el PAV, basado en la idea de comenzar un recorrido en el vértice 1 y de que en cada nivel la ramificación se basa en el nodo que sigue en el recorrido (en vez de en la elección de una arista en particular, como en la Fig. 10.22). ¿Qué estimación es correcta para la cota inferior de las configuraciones, que son listas de vértices $1, v_1, v_2, \dots$ que empiezan un recorrido? ¿Cómo se comporta el algoritmo en la figura 10.21, suponiendo que a es el vértice 1?
- *10.19 Un posible algoritmo de búsqueda local para el problema de la separación en párrafos es permitir transformaciones locales que pasen la primera palabra de una línea a la línea anterior o la última palabra de una línea a la línea siguiente. ¿Este algoritmo será localmente óptimal, en el sentido de que toda solución localmente óptima es una solución globalmente óptima?
- 10.20 Si las transformaciones locales sólo consisten en opciones dobles, ¿hay algún recorrido localmente óptimo en la figura 10.21 que no sea globalmente óptimo?

Notas bibliográficas

Hay muchos ejemplos importantes de los algoritmos de clasificación del resultado por división, incluyendo la transformada rápida de Fourier $O(n\log n)$ de Cooley y Tukey [1965], el algoritmo de la multiplicación de enteros $O(n\log n \log \log n)$ de Schonhage y Strassen [1971], y el algoritmo de multiplicación de matrices $O(n^{2.81})$ de Strassen [1969]. El algoritmo de multiplicación de enteros $O(n^{1.59})$ es de Karatsuba y Ofman [1962]. Moenck y Borodin [1972] desarrollaron diferentes algoritmos eficientes del de dividir para vencer para aritmética modular y evaluación e interpolación polinomiales.

La programación dinámica fue popularizada por Bellman [1957]. La aplicación de programación dinámica a la triangulación se debe a Fuchs, Kedem y Uzelton [1977]. El ejercicio 10.11 es de Knuth [1981]. Knuth [1971] da una solución al problema de los árboles de búsqueda binaria óptimales del ejercicio 10.12.

En Lin y Kernighan [1973] se describe una técnica heurística eficiente para el problema del agente viajero.

Véanse Aho, Hopcroft y Ullman [1974], y Garey y Johnson [1979] sobre un análisis de problemas NP-completos y otros problemas computacionalmente difíciles.

11

Estructuras de datos y algoritmos para almacenamiento externo

Este capítulo comienza considerando las diferencias en las formas de acceso entre la memoria principal y los dispositivos de almacenamiento externo como los discos. Despues se presentan varios algoritmos para clasificación de archivos de datos almacenados en forma externa. Se concluye el capítulo con un análisis de estructuras de datos y algoritmos, como los archivos indizados y los árboles B, que son muy adecuados para el almacenamiento y recuperación de información en dispositivos de almacenamiento secundario.

11.1 Un modelo para cómputos con almacenamiento externo

En los algoritmos estudiados en capítulos anteriores, se ha supuesto que la cantidad de datos de entrada es lo bastante pequeña como para que quepan en la memoria al mismo tiempo. Pero, ¿qué sucede si se desea clasificar a todos los empleados de gobierno de acuerdo con su antigüedad, o almacenar toda la información de los impuestos de la nación? En problemas como éstos, la cantidad de datos por procesar supera la capacidad de la memoria principal. La mayor parte de los grandes sistemas de cómputo tienen dispositivos de almacenamiento externo conectados en línea, como discos o dispositivos de almacenamiento masivo, en los cuales se pueden almacenar cantidades muy grandes de datos. Sin embargo, esos dispositivos tienen características de acceso que difieren mucho de las de la memoria principal. Se han desarrollado diversas estructuras de datos y algoritmos para utilizar con más eficiencia esos dispositivos. Este capítulo comprende las estructuras de datos y algoritmos para clasificar y recuperar la información almacenada en memoria secundaria.

El lenguaje Pascal, y algunos otros, tienen el tipo de datos archivo, destinado a representar datos almacenados en memoria secundaria. Aunque el lenguaje utilizado no tenga este tipo de datos, es indudable que el sistema operativo debe manejar la noción de archivos en memoria secundaria. Tanto si se habla de los archivos de Pascal como de los manipulados directamente por el sistema operativo, se encuentran limitaciones a la forma en que puede accederse a los archivos. El sistema operativo divide la memoria secundaria en *bloques* de igual tamaño. El tamaño de los bloques varía entre los distintos sistemas operativos, pero de 512 a 4096 bytes es lo típico.

Se considera un archivo como si estuviera almacenado en una lista enlazada de bloques, aunque lo más común es que el sistema operativo utilice una disposición

tipo árbol, donde los bloques que contienen el archivo son hojas, y cada uno de los nodos interiores apunta a varios bloques del archivo. Si, por ejemplo, cuatro bytes son suficientes para contener la dirección de un bloque y los bloques tienen 4096 bytes de longitud, entonces un bloque raíz puede tener apuntadores hasta para 1024 bloques. Así, los archivos de hasta 1024 bloques, es decir, cerca de cuatro millones de bytes, pueden representarse con un bloque raíz y los bloques que contienen el archivo. Los archivos de hasta 2^{20} bloques, o 2^{32} bytes, pueden representarse con un bloque raíz que apunte a 1024 bloques en un nivel intermedio, cada uno de los cuales apunta a 1024 bloques...hoja que contienen una parte del archivo, y así sucesivamente.

La operación básica en archivos consiste en llevar un solo bloque a un *buffer* (almacenamiento temporal) de la memoria principal; un buffer no es más que un área reservada de memoria principal cuyo tamaño es idéntico al de un bloque. Un sistema operativo típico facilita la lectura de los bloques de acuerdo con el orden en que aparecen en la lista de bloques que conforman el archivo. Esto es, al principio se lee el primer bloque y se guarda en el buffer de ese archivo, después se reemplaza por el segundo, que queda escrito dentro del mismo buffer, y así sucesivamente.

Ahora se puede ver la razón de las reglas para la lectura de archivos en Pascal. Cada archivo está almacenado en una secuencia de bloques, con un número entero de registros en cada bloque. (Se puede desperdiciar espacio, puesto que se evita dividir un registro entre bloques.) El cursor de lectura siempre apunta a uno de los registros del bloque que se encuentra en el buffer en ese instante. Cuando ese cursor se deba mover a un registro que no esté en el buffer, es el momento de leer el siguiente bloque del archivo.

Igualmente, se puede considerar el proceso de escritura de archivos de Pascal como la creación de un archivo en un buffer. Cuando se «escriben» los registros en archivo, se colocan en el buffer de ese archivo, en la posición inmediata siguiente a los registros colocados previamente. Cuando el buffer no puede contener otro registro completo, se copia el buffer en un bloque disponible de memoria secundaria y ese bloque se agrega al final de la lista de bloques del archivo. Se considera ahora que el buffer está vacío, y que se pueden escribir más registros en él.

Costo de las operaciones con almacenamiento secundario

Dada la naturaleza de los dispositivos de almacenamiento secundario, como los discos, el tiempo para encontrar un bloque y leerlo a la memoria principal es muy grande, comparado con el tiempo que lleva procesar el dato. Por ejemplo, supóngase que se tiene un bloque de 1000 enteros dentro de un disco que gira a 1000 rpm. El tiempo que lleva colocar la cabeza sobre la pista que contenga este bloque (*tiempo de búsqueda*), más el tiempo consumido en esperar que el bloque quede bajo la cabeza (*tiempo de latencia*), puede ser de 100 milisegundos en promedio. El proceso de escritura de un bloque en un lugar particular dentro del almacenamiento secundario lleva una cantidad similar de tiempo. Sin embargo, la máquina puede efectuar 100 000 instrucciones en esos 100 milisegundos. Este tiempo es más que suficiente

para hacer un procesamiento simple a los mil enteros una vez que están en la memoria principal, como la suma de ellos o la ubicación del máximo; incluso puede ser suficiente para clasificación rápida de los enteros.

Cuando se evalúa el tiempo de ejecución de los algoritmos que operan sobre datos almacenados en forma de archivos, es de primordial importancia considerar el número de veces que se lee un bloque a la memoria principal o se escribe un bloque en la memoria secundaria; esa operación se denomina *acceso a bloques*. Se supone que el tamaño de los bloques lo fija el sistema operativo, así que no se puede hacer que un algoritmo se ejecute con mayor rapidez incrementando el tamaño del bloque, para reducir el número de accesos a los bloques. Como consecuencia, lo que se debe considerar para los algoritmos que emplean almacenamiento externo será el número de accesos a los bloques. Se inicia el estudio de algoritmos para almacenamiento externo con la clasificación externa.

11.2 Clasificación externa

La clasificación de datos organizados como archivos o de forma más general, la clasificación de datos almacenados en memoria secundaria, se conoce como clasificación «externa». Este estudio de clasificación externa parte del supuesto de que los datos están almacenados en un archivo de Pascal. Se presenta la forma en que un algoritmo de «clasificación por intercalación» puede ordenar un archivo de n registros en sólo $O(\log n)$ pasadas por el archivo; esta cifra es mucho mejor que los $O(n)$ pasos requeridos por los algoritmos estudiados en el capítulo 8. Después se estudia cómo la utilización de ciertas características del sistema operativo para controlar la lectura y escritura de bloques en tiempos adecuados puede acelerar la clasificación al reducir el tiempo en que el computador está ocioso, esperando la lectura o escritura de un bloque hacia o desde la memoria principal.

Clasificación por intercalación

La idea esencial en la clasificación por intercalación es organizar un archivo en *fragmentos* cada vez más grandes, esto es, secuencias de registros r_1, \dots, r_k , donde la clave de r_i no es mayor que la clave de r_{i+1} para $1 \leq i < k$. Se dice que un archivo r_1, \dots, r_m de registros está *organizado en fragmentos de longitud k* si para toda $i \geq 0$ tal que $ki \leq m$, $r_{k(i-1)+1}, r_{k(i-1)+2}, \dots, r_{ki}$ es un fragmento de longitud k , y además si m no es divisible entre k , y $m = pk + q$, donde $q < k$, por lo que la secuencia de registros $r_{m-q+1}, r_{m-q+2}, \dots, r_m$, llamada *cola*, es un fragmento de longitud q . Por ejemplo, la secuencia de enteros mostrada en la figura 11.1 está organizada en fragmentos de longitud 3. Obsérvese que la cola es de longitud menor que 3, pero consta de registros ordenados, a saber, 5 y 12.

7	15	29	8	11	13	16	22	31	5	12
---	----	----	---	----	----	----	----	----	---	----

Fig. 11.1. Archivo con fragmentos de longitud tres.

El paso básico en una clasificación por intercalación en archivos es empezar con dos archivos, como f_1 y f_2 , organizados en fragmentos de longitud k . Suponiendo que

1. el número de fragmentos, incluyendo las colas, en f_1 y f_2 difiere a lo sumo en 1,
2. como máximo uno de f_1 y f_2 tiene cola, y
3. el único con cola tiene por lo menos tantos fragmentos como el otro.

Es un proceso sencillo leer un fragmento de f_1 y f_2 , combinar ambos y unir el fragmento resultante, de longitud $2k$, en alguno de los archivos g_1 y g_2 , que se han organizado en fragmentos de longitud $2k$. Alternando entre g_1 y g_2 , se hace que esos archivos no sólo se organicen en fragmentos de longitud $2k$, sino que satisfagan (1), (2) y (3). Para ver que (2) y (3) se satisfacen, es útil observar que la cola entre los fragmentos de f_1 y f_2 queda combinada en (o quizás ya está) el último fragmento creado.

Se empieza por dividir los n registros en dos archivos f_1 y f_2 , lo más uniformemente posible. Cualquier archivo se puede considerar organizado en fragmentos de longitud 1. Después, se combinan en fragmentos de longitud 1 y se distribuyen en los archivos g_1 y g_2 , organizados en fragmentos de longitud 2. Se vacían f_1 y f_2 , y se combinan g_1 y g_2 en f_1 y f_2 , ahora estarán organizados en fragmentos de longitud 4. Después, se combinan f_1 y f_2 para crear g_1 y g_2 organizados en fragmentos de longitud 8, y así sucesivamente.

Después de i pasadas de esta naturaleza, se tendrán dos archivos constituidos por fragmentos de longitud 2^i . Si $2^i \geq n$, uno de los dos archivos estará vacío y el otro tendrá un solo fragmento de longitud n , esto es, estará clasificado. Como $2^i \geq n$ cuando $i \geq \log n$, $\lceil \log n \rceil$ pasadas son suficientes. Cada pasada requiere la lectura de dos archivos y la escritura de otros dos, longitud aproximada $n/2$. El número total de bloques leídos o escritos en cada paso es aproximadamente $2n/b$, donde b es el número de registros que caben en un bloque. De este modo, el número de bloques leídos y escritos por el proceso de clasificación completo es $O((n \log n)/b)$ o, de otra forma, la cantidad de lecturas y escrituras es casi la misma que la requerida para hacer $O(\log n)$ pasadas por los datos almacenados en un solo archivo. Esta cifra supone una importante mejora de las $O(n)$ pasadas requeridas por muchos de los algoritmos de clasificación tratados en el capítulo 8.

La figura 11.2 muestra el proceso de combinación en Pascal. Se leen dos archivos organizados en fragmentos de longitud k y se escriben dos archivos organizados en fragmentos de longitud $2k$. Como ejercicio, se deja la especificación de un algoritmo que siga las ideas anteriores y que utilice el procedimiento *combina* de la figura 11.2 $\log n$ veces para clasificar un archivo de n registros.

```

procedure combina ( k: integer; { longitud del fragmento de entrada }
  f1, f2, g1, g2: file of tipo_registro );
var
  commuta_salida: boolean; { indica si la escritura es en g1 (verdadero) o en
                            g2 (falso) }
  ganador: integer; { elige el archivo con la menor clave del registro actual }
  usado: array [1..2] of integer; { usado[j] indica cuántos registros se han
                                leído hasta ahora en el fragmento actual del archivo f_j }
  fin: array [1..2] of boolean; { fin[j] es verdadero si ha terminado el
                                archivo f_j }

```

```

fragmento de  $f_1$  – se han leído  $k$  registros o se ha alcanzado el fin del
archivo de  $f_1\}$ 
actual: array [1..2] of tipo_registro; { los registros actuales de ambos
archivos }
procedure toma_registro ( i: integer ); { avanza el archivo  $f_i$ , pero no más allá
del final del archivo o del fragmento. Fija  $fin[i]$  si se encuentra el fin del
archivo o del fragmento }
begin
  usado[i] := usado[i] + 1;
  if (usado[i] = k) or
    ( $i = 1$ ) and eof(1) or
    ( $i = 2$ ) and eof(2) then fin[i] := true
  else if  $i = 1$  then read(f1, actual[1])
  else read(f2, actual[2])
end; { toma_registro }

begin { combina }
  conmuta_salida := true; { los primeros fragmentos intercalados van hacia g1 }
  rewrite(g1); rewrite(g2);
  reset(1); reset(2);
  while not eof(1) or not eof(2) do begin { intercala dos archivos }
    { asigna valor inicial }
    usado[1] := 0; usado[2] := 0;
    fin[1] := false; fin[2] := false;
    toma_registro(1); toma_registro(2);
    while not fin[1] or not fin[2] do begin { intercala dos fragmentos }
      { elige ganador }
      if fin[1] then ganador := 2
        {  $f_2$  gana por "omisión"; el fragmento de  $f_1$  se terminó }
      else if fin[2] then ganador := 1
        {  $f_1$  gana por omisión }
      else { ningún fragmento se ha terminado }
        if actual[1].clave < actual[2].clave then ganador := 1
        else ganador := 2;
      { escribe el registro del ganador }
      if conmuta_salida then write(g1, actual[ganador])
      else write(g2, actual[ganador]);
      { avanza el archivo ganador }
      toma_registro(ganador)
    end;
    { se ha terminado la intercalación de dos fragmentos; se conmuta el
    archivo de salida y se repite }
    conmuta_salida := not conmuta_salida
  end
end; { combina }

```

Fig. 11.2. Procedimiento *combina*.

Obsérvese que el procedimiento *combina* de la figura 11.2 nunca requiere un fragmento completo en memoria; lee y escribe un registro de cada vez. No se desea almacenar los fragmentos completos que se encuentran en memoria principal que obliguen a tener dos archivos de entrada. En otro caso, se podrían leer dos fragmentos al mismo tiempo desde un solo archivo.

Ejemplo 11.1. Considérese la lista de 23 números que se muestra dividida en dos archivos en la figura 11.3(a). Se empieza por combinar fragmentos de longitud 1 para crear los dos archivos de la figura 11.3(b). Por ejemplo, los primeros fragmentos de longitud uno son 28 y 31, y se combinan tomando primero 28 y luego 31. Los dos fragmentos siguientes de longitud uno, 3 y 5, se combinan para formar uno de longitud dos, que se coloca en el segundo archivo de la figura 11.3(b). Los fragmentos se separan en la figura 11.3(b) por medio de líneas verticales que no son parte del archivo. Obsérvese que el segundo archivo de la figura 11.3(b) tiene una cola de longitud uno, el registro 22, mientras que el primer archivo no tiene cola.

Se pasa, de la figura 11.3(b) a la figura 11.3(c) combinando los fragmentos de longitud dos. Por ejemplo, los fragmentos 28, 31 y 3, 5, se combinan para formar 3, 5, 28, 31 de la figura 11.3(c). Al alcanzar los fragmentos de longitud 16 de la figura 11.3(e), un archivo tiene un fragmento completo y el otro sólo tiene una cola, de longitud 7. En la última etapa, donde los archivos están en apariencia organizados como fragmentos de longitud 32, sucede que se tiene un archivo formado sólo por una cola, de longitud 23, y el segundo se encuentra vacío. El fragmento sencillo de longitud 23 es, por supuesto, la clasificación que se buscaba. □

Aceleración de la clasificación por intercalación

Por medio de un ejemplo simple se ha mostrado el proceso de la clasificación por intercalación partiendo de fragmentos de longitud 1. Se aprovechará más el tiempo si se empieza con un paso que, para una k apropiada,lea grupos de k registros dentro de la memoria principal, los ordene, por ejemplo, con clasificación rápida, y los devuelva a la memoria secundaria como un fragmento de longitud k .

Por ejemplo, un millón de registros, necesitarían 20 pasadas por los datos para hacer la clasificación empezando con fragmentos de longitud 1. Sin embargo, si se pueden tener 10 000 registros a la vez en la memoria principal, también se pueden leer 100 grupos de 10 000 registros en una pasada, clasificar cada grupo, y dejar 100 fragmentos de longitud 10 000 distribuidos uniformemente entre dos archivos. Siete pasadas de intercalación más culminarán con un archivo organizado como un fragmento de longitud $10\,000 \times 2^7 = 1\,280\,000$, lo que es mayor que un millón y significa que los datos están clasificados.

Minimización del tiempo transcurrido

28	3	93	10	54	65	30	90	10	69	8	22
31	5	96	40	85	9	39	13	8	77	10	

(a) archivos iniciales

28	31	93	96	54	85	30	39	8	10	8	10
3	5	10	40	9	65	13	90	69	77	22	

(b) organizados en fragmentos de longitud 2

3	5	28	31	9	54	65	85	8	10	69	77
10	40	93	06	13	30	39	90	8	10	22	

(c) organizados en fragmentos de longitud 4

3	5	10	28	31	40	93	96	8	8	10	10	22	69	77
9	13	30	39	54	65	85	90							

(d) organizados en fragmentos de longitud 8

3	5	9	10	13	28	30	31	39	40	54	65	85	90	93	96
8	8	10	10	22	69	77									

(e) organizados en fragmentos de longitud 16

3 5 8 8 9 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

(f) organizados en fragmentos de longitud 32

Fig. 11.3. Clasificación por intercalación de una lista.

chivo que se está leyendo, tal como sucede durante el proceso de clasificación por intercalación. Sin embargo, el hecho es que el tiempo transcurrido por dicho proceso es mayor que el tiempo empleado en el cálculo efectuado con los datos presentes en la memoria principal. Si se clasifican archivos realmente grandes, donde la operación lleva horas, el tiempo transcurrido resulta un factor importante, aun si no se paga, y debe buscarse la forma de que el proceso de la clasificación por intercalación lleve un tiempo total mínimo.

Como se ha mencionado, es común que el tiempo para leer datos de disco o cin-

ta sea mayor que el tiempo consumido en hacer cálculos simples con esos datos, como la intercalación. Por tanto, es de esperar que si sólo hay un canal dedicado a la transferencia de datos entre la memoria principal y la memoria auxiliar, este canal formará un embotellamiento, el canal de datos estará ocupado todo el tiempo, y el tiempo total transcurrido se igualará con el tiempo utilizado en el movimiento de los datos. Esto es, todos los cálculos se efectuarán en cuanto los datos se encuentren disponibles, mientras se leen o escriben datos adicionales.

Aun en este ambiente simple, debe tenerse cuidado para asegurar la terminación en una cantidad mínima de tiempo. Para ver qué puede fallar, supóngase que se lee, cada vez un bloque, de dos archivos de entrada f_1 y f_2 , en forma alterna. Los archivos están organizados en fragmentos de longitud bastante mayor que el tamaño de un bloque, así que para intercalar dos fragmentos es necesario leer muchos bloques de cada archivo. Sin embargo, supóngase que todos los registros contenidos en el fragmento del archivo f_1 preceden a todos los registros del archivo f_2 . Entonces, como se leen bloques en forma alterna, todos los bloques de f_2 tienen que permanecer en memoria. Es posible que no haya espacio para conservar todos esos bloques en la memoria principal, y aun si lo hubiera, es necesario, después de leer todos los bloques del fragmento, esperar mientras se copia y escribe todo el fragmento que proviene de f_2 .

Para evitar esos problemas, se consideran las claves de los últimos registros de los últimos bloques leídos de f_1 y f_2 , por ejemplo k_1 y k_2 , respectivamente. Si algún fragmento se agota, es obvio que se lee la siguiente de otro archivo. Sin embargo, si un fragmento no se agota, se continúa leyendo un bloque de f_1 si $k_1 < k_2$, y de f_2 , en caso contrario. Esto es, debe determinarse cuál de los dos fragmentos tendrá primero todos sus registros seleccionados en ese momento en la memoria principal, y se llenan primero registros de ese fragmento. Si la selección de registros se efectúa más rápido que la lectura, se sabe que al haber leído el último bloque de los dos fragmentos, no puede haber más que dos bloques llenos de registros sin intercalar; los registros pueden estar distribuidos en tres bloques, pero no en más.

Intercalación múltiple

Si la lectura y escritura entre las memorias principal y secundaria es el «embottellamiento», puede ahorrarse tiempo si se tiene más de un canal de datos. Supóngase que se tienen $2m$ unidades de disco, cada una con su propio canal de comunicación. Se podrían colocar m archivos, f_1, f_2, \dots, f_m en m de las unidades de disco, organizados como fragmentos de longitud k . Entonces, se pueden leer m fragmentos, uno de cada archivo, y combinarlos en un fragmento de longitud mk , el cual se coloca en alguno de los m archivos de salida, g_1, g_2, \dots, g_m , cada uno tomando un fragmento a la vez.

El proceso de intercalación en memoria principal puede llevarse a cabo en $O(\log m)$ pasos por registro si se organizan los m registros candidatos, esto es, los registros más pequeños de cada archivo que hasta ese momento no se habían seleccionado, en un árbol parcialmente ordenado u otra estructura de datos que maneje

las operaciones en colas de prioridad INSERTA y SUPRIME-MIN en tiempo logarítmico. Para seleccionar el registro con la clave más pequeña de la cola de prioridad, se realiza SUPRIME-MIN, y después INSERTA, en la cola de prioridad del siguiente registro del archivo del ganador, como reemplazo del registro seleccionado.

Si hay n registros, y la longitud de los fragmentos se multiplica por m en cada paso, después de i pasadas los fragmentos serán de longitud m^i . Si $m^i \geq n$, esto es, después de $i = \log_m n$ pasadas, la lista completa estará clasificada. Como $\log_m n = -\log_2 n / \log_2 m$, se ahorra en un factor de $\log_2 m$ el número de veces que se lee cada registro. Más aún, si m es el número de unidades de disco utilizadas para los archivos de entrada, y m son usadas para la salida, es posible procesar datos m veces tan rápido como si existiera sólo una unidad de disco para la entrada y una para la salida, o $2m$ veces tan rápido como si los archivos de entrada y salida estuvieran almacenados en una unidad de disco. Lamentablemente, incrementar m en forma indefinida no acelera el procesamiento en un factor de $\log m$. La razón de esto es que para una m suficientemente grande, el tiempo requerido por la intercalación en memoria principal, que en realidad se incrementa como $\log m$, será superior al de lectura o escritura de datos. En este punto, si hay incrementos posteriores en m , aumentarán el tiempo transcurrido, ya que el cálculo en memoria principal se convertirá en el embotellamiento.

Clasificación en varias fases

Es posible realizar una clasificación por intercalación de m -caminos con sólo $m + 1$ archivos, como alternativa a la estrategia descrita antes, que emplea $2m$ archivos. Se efectúa una secuencia de pasadas al intercalar fragmentos de m de los archivos en otros más largos en el archivo restante. Es necesario tener en cuenta los siguientes:

1. En una pasada, cuando los fragmentos de cada uno de los m archivos se intercalan en fragmentos del $(m + 1)$ -ésimo archivo, no es necesario usar todos los fragmentos en cada uno de los m archivos de entrada. Antes bien, cada archivo, cuando es de salida, se ocupa con fragmentos de cierta longitud. Se usan algunos de esos fragmentos para ayudar a llenar cada uno de los otros m archivos cuando les llegue el turno de ser archivos de salida.
2. Cada paso produce archivos de longitud diferente. Puesto que cada uno de los archivos cargados con fragmentos en las m pasadas previas contribuye a los fragmentos del paso actual, la longitud en una pasada es la suma de las longitudes de los fragmentos producidos en las m pasadas previas. (Si se han dado menos de m pasadas, habrá que considerar todos los anteriores como si produjeran fragmentos de longitud 1.)

Este proceso de clasificación por intercalación se conoce como *clasificación en varias fases (polyphase sorting)*. El cálculo exacto de los números de pasadas necesarias como una función de m y n (el número de registros), y el cálculo de la distribución inicial de los fragmentos en m archivos se dejan como ejercicio. Sin embargo, aquí se dará un ejemplo para dar idea del caso general.

Ejemplo 11.2. Si $m = 2$, se comienza con dos archivos f_1 y f_2 , organizados en fragmentos de longitud 1. Los registros de f_1 y f_2 se intercalan para hacer fragmentos de longitud 2 en un tercer archivo, f_3 . Sólo se intercalan suficientes fragmentos para vaciar f_1 . Despues, se intercalan los restantes de longitud 1 de f_2 con un número igual de fragmentos de longitud 2 de f_3 . Los resultantes, de longitud 3, quedarán colocados en f_1 . Despues, deben intercalarse los de longitud 2 de f_3 con los de longitud 3 de f_1 . Esos fragmentos, de longitud 5, se colocan en f_2 , que quedó vacío en la pasada anterior.

La secuencia de longitudes de los fragmentos 1, 1, 2, 3, 5, 8, 13, 21, ..., es la sucesión de Fibonacci. Esta secuencia se genera por medio de la relación de recurrencia $F_0 = F_1 = 1$, y $F_i = F_{i-1} + F_{i-2}$, para $i \geq 2$. Obsérvese que la razón entre los números de Fibonacci consecutivos F_{i+1}/F_i se acerca a la «razón dorada» $(\sqrt{5} + 1)/2 = 1.618\dots$ conforme i crece.

De aquí se deduce que para que el proceso continúe hasta que la lista quede clasificada, los números iniciales de registros en f_1 y f_2 deben ser dos números consecutivos de Fibonacci. Por ejemplo, la figura 11.4 muestra qué sucede al empezar con $n = 34$ registros (34 es el número de Fibonacci F_8), distribuidos 13 en f_1 y 21 en f_2 . (13 y 21 son el sexto y el séptimo números de Fibonacci, así que la razón F_7/F_6 es muy cercana a 1.618; de hecho, es 1.615.) El estado de un archivo se representa en la figura 11.4 como $a(b)$, lo cual significa que tiene a fragmentos de longitud b . □

después de la pasada	f_1	f_2	f_3
inicial	13(1)	21(1)	vacío
1	vacío	8(1)	13(2)
2	8(3)	vacío	5(2)
3	3(3)	5(5)	vacío
4	vacío	2(5)	3(8)
5	2(13)	vacío	1(8)
6	1(13)	1(21)	vacío
7	vacío	vacío	1(34)

Fig. 11.4. Ejemplo de clasificación en varias fases.

Un caso en el que la velocidad de entrada/salida no es un cuello de botella

Cuando la lectura de archivos es el cuello de botella, el siguiente bloque debe escogerse con sumo cuidado. Como ya se dijo, la situación a evitar es aquella en la que se tienen que almacenar varios bloques de un fragmento, debido a que ese fragmento tenía registros con claves grandes, los cuales serán escogidos después de la mayor parte o de todos los registros del otro fragmento. El truco para evitar este problema consiste en determinar con rapidez qué fragmento agotará primero sus registros en memoria principal, comparando los últimos de esos registros de cada archivo.

Cuando el tiempo requerido para leer datos en la memoria principal es comparable o menor que el tiempo requerido para procesar los datos, se vuelve aún más crítica la selección del archivo de entrada desde el cual leer un bloque con cuidado, ya que no hay esperanza de construir una reserva de registros dentro de la memoria principal en caso de que el proceso de intercalación empiece repentinamente al tomar más registros de un fragmento que del otro. El «truco» mencionado antes resulta útil en diversas situaciones, como se verá a continuación.

Considérese el caso donde la intercalación es el cuello de botella, y no la lectura o la escritura, por dos razones.

1. Como se ha visto, al tener disponibles varias unidades de disco o cinta, es posible acelerar la entrada/salida lo suficiente para que el tiempo requerido por la intercalación supere al tiempo de entrada o al de salida.
2. Los canales de alta velocidad pueden estar pronto disponibles en el mercado.

Por tanto, se considerará un modelo simple del problema que se puede presentar cuando la intercalación *«es el cuello de botella en una clasificación realizada con datos almacenados en la memoria secundaria.* Específicamente, se supone que

- a) Se intercalan fragmentos mucho más grandes que los bloques.
- b) Hay dos archivos de entrada y dos de salida. Los archivos de entrada están almacenados en un disco (o algún otro dispositivo conectado a la memoria principal a través de un solo canal) y los archivos de salida están en otra unidad similar con un solo canal.
- c) Los tiempos para
 - I) leer un bloque
 - II) escribir un bloque, y
 - III) seleccionar suficientes registros con las claves más pequeñas entre los dos fragmentos que se encuentran en ese momento en memoria principal, para llenar un bloque.

son todos iguales.

En dichos supuestos, se considera una clase de estrategias de intercalación donde varios buffers de entrada (espacios para contener un bloque) se ubican en la memoria principal. En todo momento, alguno de esos buffers contendrá los registros no seleccionados de los dos fragmentos de entrada, y uno de ellos estará en el proceso de ser leído de uno de los archivos de entrada. Los otros dos contendrán la salida, es decir, los registros seleccionados en el orden de intercalación adecuado. En todo momento, uno de esos buffers se encontrará en el proceso de escritura en alguno de los archivos de salida y el otro se estará llenando con registros seleccionados de los buffers de entrada.

Una *transferencia* consiste en hacer lo siguiente (quizá todo al mismo tiempo):

1. la lectura de un bloque de entrada en un buffer de entrada.
2. el llenado de uno de los buffers de salida con los registros seleccionados, es decir, los registros con las claves más pequeñas, entre todos los que se tengan en ese momento en el buffer de entrada.

3. la escritura del otro buffer de salida en uno de los dos archivos de salida que se esté generando.

Por las suposiciones, (1), (2) y (3) requieren el mismo tiempo. Para obtener el máximo de eficiencia, se deben efectuar en paralelo. Se puede hacer esto, a menos que (2), la selección de registros con las claves más pequeñas, incluya algunos de los registros que se están leyendo en ese momento †. Así, se debe buscar una estrategia para seleccionar los buffers que se van a leer, de modo que al principio de cada transferencia los b registros no seleccionados con las claves menores ya se encuentren listos en los buffers de entrada, donde b es el número de registros que llenan un bloque o buffer.

Las condiciones en las cuales la intercalación puede efectuarse en paralelo con la lectura son simples. Sean k_1 y k_2 las claves más grandes de todos los registros no seleccionados en memoria principal del primero y segundo fragmentos, respectivamente. Entonces, en la memoria principal deben encontrarse por lo menos b registros no seleccionados cuyas claves no excedan de $\min(k_1, k_2)$. En primer lugar se mostrará cómo hacer la intercalación con seis buffers, tres para cada archivo, y después se mostrará que es suficiente con cuatro buffers si se reparten entre los dos archivos.

Esquema de seis buffers de entrada

Este primer esquema se representa en el dibujo de la figura 11.5. Los dos buffers de salida no se muestran; existen tres buffers para cada archivo; cada uno tiene capacidad para b registros. El área sombreada representa los registros disponibles, y las claves están en orden ascendente en el sentido de las manecillas del reloj. Siempre, el número total de registros no seleccionados es $4b$ (a menos que el número de registros que permanecen en los fragmentos que se encuentran intercalando sea menor). Inicialmente, se leen los dos primeros bloques de cada fragmento en los buffers ††. Como siempre hay $4b$ registros disponibles, y a lo sumo $3b$ pueden proceder de un archivo, se sabe que hay por lo menos b registros que vienen de cada archivo. Si k_1 y k_2 son las claves más grandes disponibles de los dos fragmentos, debe haber b registros con las claves iguales o menores que k_1 y b registros con las claves menores o iguales que k_2 . Así, hay b registros con claves iguales o menores que $\min(k_1, k_2)$.

† Es tentador suponer que si (1) y (2) requieren el mismo tiempo, entonces la selección no podría nunca coincidir con la lectura; si el bloque completo no se hubiera leído aún, podrían seleccionarse de entre los primeros registros del bloque aquellos que tuvieran las claves menores. Sin embargo, por su naturaleza, en las lecturas de disco transcurre un período largo antes de que se encuentre el bloque y pueda realizarse la lectura. Por tanto, la única suposición segura es que nada del bloque que se está leyendo en una transferencia está disponible en ese momento para la selección.

†† Si éstos no son los primeros fragmentos tomados de cada archivo, entonces este paso inicial únicamente puede hacerse después de leer los anteriores fragmentos y sus últimos $4b$ registros se estén intercalando.

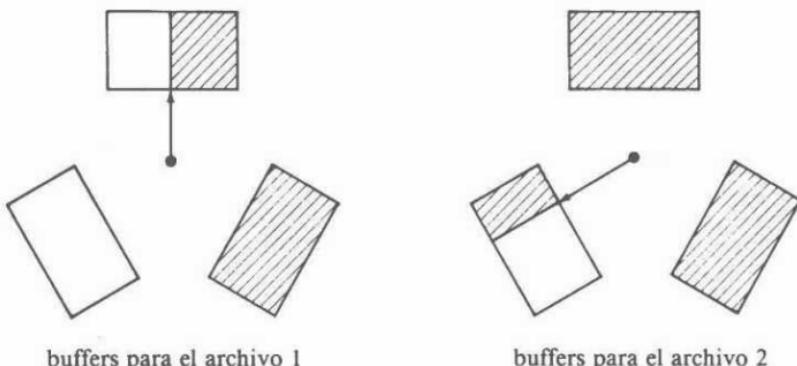


Fig. 11.5. Intercalación con seis buffers de entrada.

La pregunta sobre qué archivo leer a continuación es trivial. Por lo general, dado que dos buffers estarán llenos parcialmente, como en la figura 11.5, habrá sólo un buffer vacío y deberá llenarse. Si sucede, como cuando se está empezando, que cada fragmento tiene dos buffers completamente llenos y uno vacío, se toma cualquiera de los que se encuentran vacíos. Obsérvese que la demostración de que no es posible agotar un fragmento [existen b registros con claves iguales o menores que $\min(k_1, k_2)$] dependía sólo del hecho de que estuvieron presentes $4b$ registros.

Además, las flechas de la figura 11.5 representan apuntadores a los primeros registros disponibles (cuyas claves son menores) en ambos fragmentos. En Pascal, se representaría ese apuntador con dos enteros. El primero, en el intervalo $1..3$, representa el buffer apuntado, y el segundo, en el intervalo $1..b$, representa el registro dentro del buffer. En forma alternativa, es posible dejar que los buffers sean el primero, el medio y el último tercio de un arreglo y usar un entero en el intervalo $1..3b$. En otros lenguajes, donde los apuntadores pueden apuntar hacia elementos de arreglos, puede preferirse un apuntador de tipo \uparrow tipo_registro.

Esquema de cuatro buffers

La figura 11.6 sugiere un esquema con cuatro buffers. En el principio de cada transferencia, están disponibles $2b$ registros. Dos de los buffers de entrada están asignados a uno de los archivos; B_1 y B_2 de la figura 11.6 están asignados al archivo uno. Uno de estos buffers estará parcialmente lleno (vacío en el caso extremo), y el otro, lleno. El cuarto buffer no está comprometido, y se llenará desde uno de los archivos durante la transferencia.

Se mantendrá, por supuesto, la propiedad que permite intercalar en paralelo con la lectura; al menos b registros de la figura 11.6 deben tener claves menores o iguales que $\min(k_1, k_2)$, donde k_1 y k_2 son las claves de los últimos registros disponibles en los dos archivos, como se indica en la figura 11.6. Se denomina *segura* a la configuración que cumple esa propiedad. Al principio, se lee un bloque de cada frag-

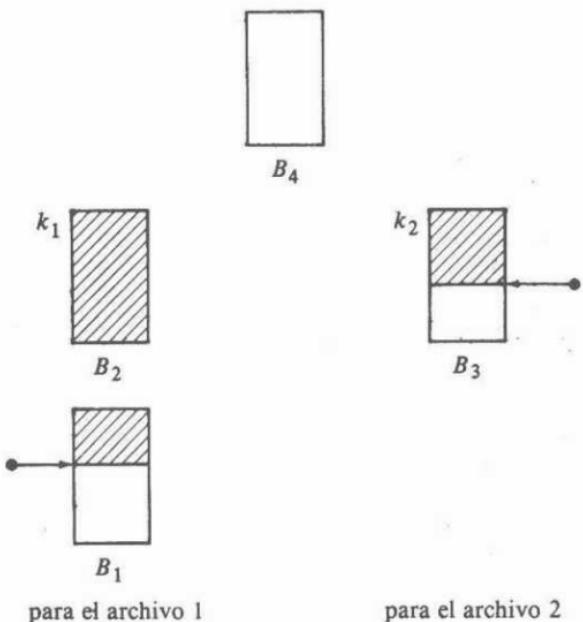


Fig. 11.6. Intercalación con cuatro buffers de entrada.

mento (este es el caso extremo, donde \$B_1\$ está vacío y \$B_3\$ está lleno en la Fig. 11.6), de modo que la configuración inicial esté segura. En el supuesto de que la figura 11.6 es segura, es necesario mostrar que la configuración lo será después de completar la siguiente transferencia.

Si \$k_1 < k_2\$, se escoge \$B_4\$ para llenarlo con el siguiente bloque del archivo uno y, en otro caso, llenarlo con el del archivo dos. Supóngase primero que \$k_1 < k_2\$. Ya que \$B_1\$ y \$B_3\$ de la figura 11.6 tienen exactamente \$b\$ registros, se debe, durante la siguiente transferencia, agotar \$B_1\$; de otra forma, es necesario agotar \$B_3\$ y contradecir la seguridad de la figura 11.6. Así, después de una transferencia, la configuración se ve como en la figura 11.7(a).

Para comprobar que la figura 11.7(a) es segura, considérense dos casos. Primero, si \$k_3\$, la última clave del bloque recién leído \$B_4\$, es menor que \$k_2\$, entonces, como \$B_4\$ está lleno, es muy probable que haya \$b\$ registros iguales o menores que \$\min(k_3, k_2)\$, y la configuración es segura. Si \$k_2 \leq k_3\$, y dado que se supuso que \$k_1 < k_2\$ (de otro modo se hubiera llenado \$B_4\$ del archivo dos), los \$b\$ registros de \$B_2\$ y \$B_3\$ tienen claves menores o iguales que \$\min(k_2, k_3) = k_2\$.

Ahora se estudiará el caso donde \$k_1 \geq k_2\$ en la figura 11.6. Se escoge leer el siguiente bloque del archivo dos. La figura 11.7(b) muestra la situación resultante. Como en el caso \$k_1 < k_2\$, se argumenta que \$B_1\$ debe agotarse y, por eso, se muestra que el archivo uno tiene asignado sólo el buffer \$B_2\$ de la figura 11.7(b). La demostración de que esta figura es segura es igual que la de la figura 11.7(a).

Obsérvese que, como en el esquema con seis buffers, no se lee un archivo más allá del fin de un fragmento. Sin embargo, si no es necesario leer un bloque desde uno de los fragmentos actuales, puede leerse un bloque del siguiente fragmento de ese archivo. Así, existe la oportunidad de leer un bloque de cada uno de los fragmentos siguientes, y entonces será posible iniciar la intercalación de los fragmentos tan pronto como se hayan seleccionado los últimos registros del fragmento anterior.

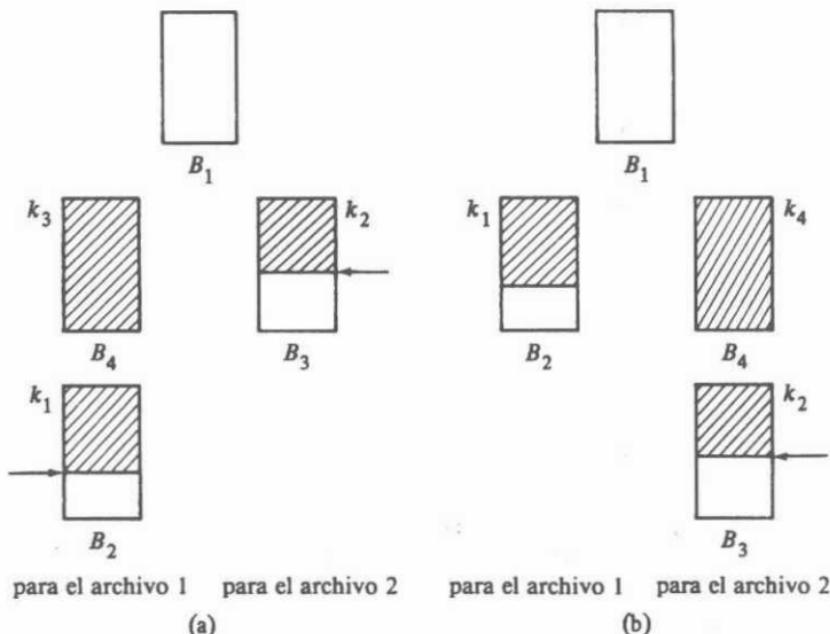


Fig. 11.7. Configuración después de una transferencia.

11.3 Almacenamiento de información en archivos

En esta sección, se analizan las estructuras de datos y los algoritmos para el almacenamiento y recuperación de información en archivos almacenados en forma externa. Se considerará un archivo como una secuencia de registros, donde cada registro consiste en la misma secuencia de campos. Los campos pueden ser de *longitud fija*, con un número predeterminado de bytes, o de *longitud variable*, con un tamaño arbitrario. Los archivos con registros de longitud fija se suelen utilizar en los sistemas de administración de bases de datos para almacenar datos muy estructurados. Los archivos con registros de longitud variable se usan típicamente para almacenar información de textos; no están disponibles en Pascal. En esta sección, se supondrá

que sólo hay campos de longitud fija; las técnicas empleadas en los registros de longitud fija pueden modificarse con facilidad para trabajar con registros de longitud variable.

Las operaciones con archivos que se considerarán son las siguientes:

1. **INSERTA** un registro determinado en un archivo en particular.
2. **SUPRIME** de un archivo en particular todos los registros que tengan un valor asignado en cada campo de conjunto de campos designado.
3. **MODIFICA** todos los registros de un archivo en particular asignando valores asignados a ciertos campos en los registros que tengan un valor asignado en otro conjunto de campos.
4. **RECUPERA** todos los registros que tengan valores asignados en cada uno de los campos de un conjunto asignado.

Ejemplo 11.3. Por ejemplo, si se tiene un archivo cuyos registros constan de tres campos: *nombre*, *dirección* y *teléfono*. Se podrá preguntar por todos los registros con *teléfono* = 555-1234, insertar el registro (Juan Pérez, calle Manzana 12, 555-1234) o eliminar todos los registros con *nombre* = «Juan Pérez» y *dirección* = «calle Manzana 12». Como otro ejemplo, se puede desear la modificación de todos los registros con *nombre* = «Juan Pérez» para fijar el campo *teléfono* a 555-1234. □

En buena medida, se pueden considerar las operaciones con archivos como si los archivos fueran conjuntos de registros y las operaciones fueran las que se analizaron en los capítulos 4 y 5. Sin embargo, existen dos diferencias importantes. Primero, cuando se habla de archivos en dispositivos de almacenamiento externo, es forzoso utilizar la medición del costo comentada en la sección 11.1, en la evaluación de las estrategias de organización de archivos. Esto es, se supone que los archivos están almacenados en cierta cantidad de bloques físicos, y que el costo de una operación es el número de bloques que se van a leer en memoria principal o escribir desde memoria principal en el almacenamiento externo.

La segunda diferencia es que los registros, siendo tipos de datos concretos en la mayoría de los lenguajes de programación, puede esperarse que tengan apunadores a ellos, mientras que los elementos abstractos de un conjunto, normalmente no tendrán «apunadores» hacia ellos. En particular, los sistemas de bases de datos con frecuencia hacen uso de apunadores a registros cuando organizan datos. La consecuencia de dichos apunadores es que los registros suelen considerarse *adheridos*; no pueden moverse por el almacenamiento, debido a la posibilidad de que un apuntador de algún lugar desconocido no consiga apuntar al registro si éste se ha movido.

Una forma simple de representar apunadores a registros es la siguiente. Cada bloque tiene una *dirección física*, que es la localización del inicio del bloque en el dispositivo de almacenamiento externo; es función del sistema de archivos cuidar las direcciones físicas. Una forma de representar las direcciones de los registros es usar la dirección física del bloque que contiene el registro junto con un *desplazamiento*, que da el número de bytes que preceden en el bloque al principio del registro. Esos pares físicos dirección-desplazamiento pueden almacenarse en campos de tipo «apuntador a registro».

Organización simple

La forma más simple, y también menos eficiente, de realizar las operaciones de archivos anteriores es usar primitivas de lectura y escritura de archivos como las de Pascal. En esta «organización» (que en realidad es una «falta de organización»), los registros pueden almacenarse en cualquier orden. La recuperación de un registro con valores especificados en ciertos campos se efectúa rastreando el archivo y viendo en cada registro si se encuentran los valores especificados. Una inserción en un archivo puede realizarse agregando el registro al final del archivo.

Para la modificación de los registros, se rastrea el archivo y se prueba cada registro para ver si corresponde a los campos designados, y de ser así, se hacen los cambios necesarios en el registro. Una operación de eliminación trabaja casi siempre de la misma forma, pero al encontrar un registro cuyos campos corresponden a los valores requeridos para que la eliminación se lleve a cabo, se debe encontrar la forma de eliminar el registro. Una posibilidad es correr todos los registros siguientes, una posición hacia adelante en sus respectivos bloques, y pasar el primer registro de cada bloque siguiente a la última posición del bloque anterior del archivo. Sin embargo, este enfoque no funciona si los registros están adheridos, porque un apuntador al i -ésimo registro del archivo apuntaría entonces al $(i + 1)$ -ésimo registro.

Si los registros están adheridos, es necesario usar un enfoque distinto. Se marcan de alguna manera los registros eliminados, pero sin mover registros para llenar el hueco, ni insertar un registro nuevo en ese espacio. Así, lógicamente, el registro se elimina, pero su espacio continúa ocupado en el archivo. Esto es necesario para que, si se sigue un apuntador a un registro eliminado, se descubra que el registro apuntado fue eliminado y se tome la acción apropiada, como hacer que el apuntador sea NIL y no se le vuelva a seguir. Dos formas de marcar los registros cuando se eliminan son:

1. Sustituir el registro por algún valor que nunca pueda ser el valor de un registro «real», y cuando se siga un apuntador, suponer que el registro está eliminado si tiene ese valor.
2. Que cada registro tenga un *bit de eliminación*, un solo bit que es 1 en registros que se han eliminado, y 0 en caso contrario.

Aceleración de operaciones con archivos

La desventaja obvia de los archivos secuenciales es que las operaciones son lentas. Cada operación requiere la lectura del archivo completo, y algunos bloques pueden requerir ser reescritos también. Por fortuna, existen organizaciones de archivos que permiten acceder a un registro, leyendo en la memoria principal sólo una pequeña fracción del archivo completo.

Para hacer posible dichas organizaciones, se supone que cada registro de archivo tiene una *clave*, un conjunto de campos que identifica de manera única cada registro. Por ejemplo, el campo *nombre* del archivo *nombre-dirección-teléfono* puede considerarse una clave. Esto es, se puede suponer que no existen simultáneamente

dos registros en el archivo con el mismo valor en el campo *nombre*. La recuperación de un registro dando valores a sus campos clave, es una operación habitual que se realiza con gran facilidad en muchas organizaciones de archivo comunes.

Otro elemento necesario para lograr operaciones rápidas con archivos es la capacidad de acceder a bloques en forma directa, en vez de recorrer en secuencia los bloques que contienen el archivo. Muchas de las estructuras de datos utilizadas para operaciones rápidas con archivos usarán apuntadores a los propios bloques, los cuales son direcciones físicas de los bloques, como se describió antes. Desafortunadamente, no es posible escribir programas en Pascal, ni en muchos otros lenguajes, que se ocupen de los datos en el nivel de bloques físicos y sus direcciones; dichas operaciones se efectúan de ordinario con mandatos del sistema operativo. Sin embargo, se hará una breve descripción informal de la forma de trabajo de las organizaciones que utilizan el acceso directo a bloques.

Archivos con función de dispersión

La dispersión (*hashing*) es una técnica muy utilizada para tener acceso rápido a información almacenada en archivos secundarios. La idea básica es similar a la dispersión abierta estudiada en la sección 4.7. Los registros de un archivo se dividen entre varias *cubetas*, y cada una consiste en una lista enlazada de uno o más bloques de almacenamiento externo. La organización es similar a la presentada en la figura 4.10. Existe una tabla de cubetas que contiene B apuntadores, uno para cada cubeta. En la tabla de cubetas, cada apuntador es la dirección física del primer bloque de la lista enlazada de bloques para esa cubeta.

Las cubetas están numeradas 0, 1, ..., $B - 1$. Una función de dispersión h hace corresponder cada valor de clave con uno de los enteros 0 a $B - 1$. Si x es una clave, $h(x)$ es el número de la cubeta que contiene el registro con la clave x , si tal registro está presente en alguna. Los bloques que forman cada cubeta se encuentran encadenados y forman una lista enlazada. Así, el encabezado del i -ésimo bloque de una cubeta contiene un apuntador a las direcciones físicas del $(i + 1)$ -ésimo bloque. El último bloque de una cubeta contiene un apuntador NIL en su encabezado.

Esta organización se ilustra en la figura 11.8. La principal diferencia entre las figuras 11.8 y 4.10 es que aquí, los elementos almacenados en un bloque de una cubeta no tienen que estar encadenados con apuntadores; sólo los bloques deben estar encadenados.

Si el tamaño de la tabla de cubetas es pequeño, puede quedar almacenada en memoria principal. De otra forma, puede almacenarse en forma secuencial, en tantos bloques como sea necesario. Para buscar el registro con clave x , se calcula $h(x)$, para encontrar el bloque de la tabla de cubetas que contengan el apuntador al primer bloque de la cubeta $h(x)$. Después, se leen los bloques de la cubeta $h(x)$ de manera sucesiva, hasta encontrar el que contenga el registro con la clave x . Si se agotan todos los bloques de la lista enlazada para la cubeta $h(x)$, se concluye que x no es la clave de ningún registro.

Esta estructura es eficiente en operaciones donde se especifican los valores de

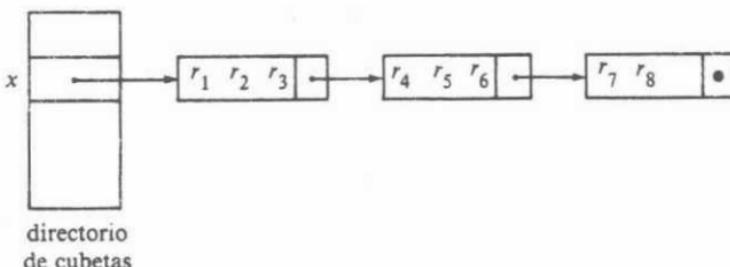


Fig. 11.8. Función de dispersión con cubetas que contienen bloques encadenados.

los campos de la clave, como la recuperación de un registro con un valor especificado en la clave o una inserción de un registro (lo que, como es obvio, especifica el valor de la clave para ese registro). El número promedio de accesos a bloques requerido para una operación que especifica la clave de un registro es aproximadamente el número promedio de bloques en una cubeta, que es n/bk si n es el número de registros, un bloque contiene b registros, y k es el número de cubetas. Así, en promedio, las operaciones basadas en claves son k veces más rápidas con esta organización que con la de archivos no organizados. Lamentablemente, las operaciones no basadas en claves no se aceleran, puesto que se deben examinar esencialmente todas las cubetas durante esas otras operaciones. La única forma general de incrementar la velocidad en operaciones que no se basan en claves es usar índices secundarios, que se analizan en el final de esta sección.

Para insertar un registro con la clave x , primero se verifica si ya existe un registro con la misma clave. De ser así, se informa de un error, dado que se supone que la clave identifica de manera unívoca cada registro. Si no existe el registro con la clave x , se inserta el registro nuevo en el primer bloque de la cadena para la cubeta $h(x)$ en el que pueda caber. Si el registro no cabe en ningún bloque de los existentes para la cubeta $h(x)$, se llama al sistema de archivos para encontrar un bloque nuevo en el cual se pueda colocar el registro. Este bloque nuevo se agrega al final de la cadena de la cubeta $h(x)$.

Fara eliminar un registro con clave x , primero debe localizarse, y después se le asigna su bit de eliminación. Otra estrategia posible (que no puede usarse cuando los registros están adheridos) es sustituir el registro eliminado por el último de la cadena de $h(x)$. Si la eliminación del último registro causa el vaciado del último bloque de la cadena se puede devolver el bloque vacío al sistema de archivos para usarlo de nuevo más tarde.

Una organización de archivos de acceso con función de dispersión bien diseñada, requiere sólo unos cuantos accesos a bloques para cada operación de archivo. Si la función de dispersión es buena y el número de cubetas es más o menos igual al número de registros en el archivo, dividido entre el número de registros que pueden caber en un bloque, entonces la cubeta promedio consta de un bloque. Excluyendo el número de accesos a bloques para buscar la tabla de cubetas, una recuperación típica basada en claves hará un solo acceso a un bloque, y una inserción, eliminación o modificación típicas efectuarán dos accesos a bloques. Si el número prome-

dio de registros por cubeta excede en mucho de la cantidad que cabe en un bloque, se puede reorganizar periódicamente la tabla de dispersión incrementando al doble la cantidad de cubetas y dividiendo cada cubeta en dos. Esas ideas se estudiaron al final de la sección 4.8.

Archivos indizados

Otra forma común de organizar un archivo de registros es mantener el archivo clasificado de acuerdo con los valores de las claves. Entonces es posible buscar en el archivo como se haría en un diccionario o en un directorio telefónico, revisando sólo el primer nombre o palabra de cada página. Para facilitar la búsqueda, se crea un segundo archivo, llamado *índice disperso*, que consta de pares (x, b) , donde x es un valor de una clave y b es la dirección física del bloque en el cual el primer registro tiene la clave con valor x . El índice disperso se mantiene clasificado de acuerdo con los valores de las claves.

Ejemplo 11.4. En la figura 11.9, se observa un archivo y su archivo de índice disperso. Se supone que tres registros del archivo principal, o tres pares del archivo de índices, caben en un bloque. Sólo se muestran los valores de las claves de los registros del archivo principal, que se suponen enteros. □

Para recuperar un registro con una clave dada x , primero se busca el par (x, b) en el archivo índice. Lo que en realidad se busca es la z más grande tal que $z \leq x$ y exista un par (z, b) en el archivo índice. Después, la clave x aparece en el bloque b , si está presente en el archivo principal.

Hay varias estrategias para buscar en el archivo de índices; la más simple es la *búsqueda lineal*. Se lee el archivo índice desde el principio hasta encontrar el par (x, b) o el primer par (y, b) , donde $y > x$. En el último caso, el par precedente (z, b) debe tener $z < x$, y si el registro con la clave x está en algún sitio, es en el bloque b' .

La búsqueda lineal sólo es posible para archivos de índices pequeños. Un método más rápido es la *búsqueda binaria*. Supóngase que el archivo de índices está almacenado en los bloques b_1, b_2, \dots, b_n . Para buscar la clave x , se toma el bloque medio $b_{\lceil n/2 \rceil}$ y se compara x con la clave y del primer par de ese bloque. Si $x < y$, se repite la búsqueda en los bloques $b_1, b_2, \dots, b_{\lceil n/2 \rceil - 1}$. Si $x \geq y$, pero x es menor que la clave del bloque $b_{\lceil n/2 \rceil + 1}$, (o si $n = 1$ de modo que no hay tal bloque), se utiliza la búsqueda lineal para ver si x corresponde al primer componente de un par de índi-

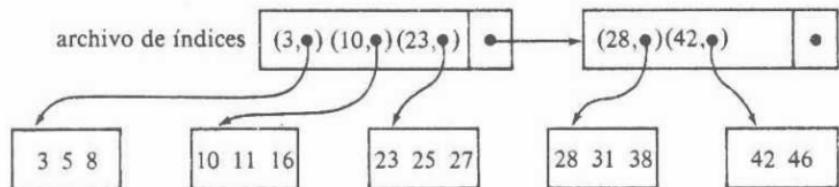


Fig. 11.9. Un archivo principal y su índice disperso.

ces en el bloque $b_{\lfloor n/2 \rfloor}$. De otra forma, se repite la búsqueda en los bloques $b_{\lfloor n/2 \rfloor + 1}, b_{\lfloor n/2 \rfloor + 2}, \dots, b_n$. Con la búsqueda binaria es necesario examinar sólo $\lceil \log_2(n+1) \rceil$ bloques del archivo índice.

Para asignar valor inicial a un archivo indizado, se clasifican los registros de acuerdo con los valores de sus claves, y después se distribuyen los registros entre los bloques, en ese orden; se puede decidir empaquetar tantos como quepan en cada bloque. Otra posibilidad sería preferir dejar espacio para unos registros adicionales que puedan insertarse más tarde. La ventaja es que después es menos probable que las inserciones sobrecarguen el bloque en el cual la inserción se lleva a cabo, con el consecuente requerimiento de tener acceso a los bloques adyacentes. Después de la división de los registros en los bloques de alguna de esas formas, se crea el archivo de índices examinando cada bloque y encontrando la primera clave de cada bloque. Como en el caso del archivo principal, puede dejarse algún espacio para crecimiento posterior en los bloques que contienen el archivo de índices.

Supóngase que se tiene un archivo clasificado con registros almacenados en los bloques b_1, b_2, \dots, b_m . Para insertar un nuevo registro en este archivo, se emplea el archivo de índices para determinar qué bloque b_i debe contener el registro nuevo. Si el registro nuevo debe quedar en b_i , se coloca en el orden adecuado. Después, si el registro nuevo queda como primer registro del bloque b_i , se ajusta el archivo índice.

Si el registro nuevo no cabe en b_i , hay varias estrategias posibles. Tal vez la más simple sea ir al bloque b_{i+1} , el cual puede encontrarse por medio del archivo de índices para ver si el último registro de b_i puede pasarse al principio de b_{i+1} . De ser así, este último registro se pasa a b_{i+1} , y el nuevo puede insertarse en la posición adecuada en b_i . El elemento del archivo de índices correspondiente a b_{i+1} , y posiblemente el de b_i , debe ajustarse en forma apropiada.

Si b_{i+1} también está lleno, o si b_i es el último bloque ($i = m$), entonces se obtiene un bloque nuevo del sistema de archivos. El registro nuevo se inserta en el bloque nuevo a continuación de b_i . Ahora se emplea este mismo procedimiento para insertar un registro que corresponda al nuevo bloque en el archivo de índices.

Archivos no clasificados con índice denso

Otra forma de organizar un archivo de registros es mantener el archivo en orden aleatorio y tener otro, llamado *índice denso*, para ayudar a ubicar los registros. El índice denso consta de pares (x, p) , donde p es un apuntador al registro con la clave x en el archivo principal. Esos pares están clasificados por el valor de la clave, de modo que una estructura como el índice disperso mencionado antes, o el árbol B mencionado en la siguiente sección, puede usarse para ayudar a encontrar las claves en el índice denso.

Con esta organización se emplean los índices densos para encontrar la localización en el archivo principal de un registro con una clave dada. Para insertar un nuevo registro, se sigue la pista del último bloque del archivo principal y ahí se insertan los registros nuevos, tomando un nuevo bloque del sistema de archivos si el último bloque está lleno. También se inserta un apuntador a ese registro en el archivo del

índice denso. Para eliminar un registro, tan sólo se ajusta el bit de eliminación en el registro, y se elimina el elemento correspondiente en el índice denso (tal vez, ajustando también un bit de eliminación).

Índices secundarios

Mientras que las estructuras dispersas e indizadas incrementan bastante la velocidad de las operaciones basadas en claves, ninguna de ellas sirven de ayuda cuando las operaciones implican una búsqueda de registros a partir de valores en campos que no sean campos clave. Si se desea encontrar los registros con valores dados en algún conjunto de campos F_1, \dots, F_k , es necesario un *índice secundario* de aquellos campos. Un índice secundario es un archivo formado de pares (v, p) , donde v es una lista de valores, uno para cada uno de los campos F_1, \dots, F_k , y p es un apuntador a un registro. Puede haber más de un par con una v dada, y cada apuntador asociado indica un registro del archivo principal que tiene a v como lista de valores de los campos F_1, \dots, F_k .

Para recuperar registros dados los valores de los campos F_1, \dots, F_k , en el índice secundario se buscan el o los registros con esa lista de valores. El índice secundario mismo puede organizarse en cualquiera de las formas comentadas para la organización de archivos por valores de claves. Esto es, se presume que v sea la clave de (v, p) .

Por ejemplo, una organización con función de dispersión en realidad no depende de que las claves sean únicas, aunque si hubiera muchos registros con la misma «clave», éstos podrían distribuirse en cubetas de manera no uniforme, con el efecto de que la función de dispersión no incrementaría mucho la velocidad de acceso. En el caso extremo, como cuando hay sólo dos valores para los campos de un índice secundario, todas las cubetas excepto dos estarían vacías, y la tabla de dispersión sólo podría incrementar la velocidad de las operaciones en un factor de dos a lo sumo, sin importar cuántas cubetas haya. De modo semejante, un índice disperso no requiere que las claves sean únicas, pero si no lo son, puede haber dos o más bloques del archivo principal que tengan la misma «clave» menor, y todos esos bloques se recorren al buscar registros con ese valor.

Con cualquier organización, de índice disperso o de índice con función de dispersión para el archivo de índices secundarios, puede desearse ahorrar espacio agrupando todos los registros con el mismo valor. Esto es, los pares $(v, p_1), (v, p_2), \dots, (v, p_m)$ pueden reemplazarse por v seguida de la lista p_1, p_2, \dots, p_m .

Cabe preguntarse si el mejor tiempo de respuesta a las operaciones aleatorias no puede obtenerse al crear un índice secundario para cada campo, o aun para todos los subconjuntos de los campos. Lamentablemente, hay un precio por cada índice secundario que se desee crear. Primero, está el espacio necesario para almacenar el índice secundario, y eso puede ser o no un problema, dependiendo de si el espacio es una prioridad.

Además, cada índice secundario creado disminuye la velocidad de todas las inserciones y eliminaciones. La razón es que al insertar un registro se debe insertar también un elemento para ese registro en cada índice secundario, para que los índices

secundarios sigan representando exactamente el archivo. La actualización de un índice secundario requiere por lo menos dos accesos a bloques, ya que se debe leer y escribir un bloque. Sin embargo, puede requerir mucho más de dos accesos a bloques, ya que es necesario encontrar ese bloque, y cualquier organización empleada para el archivo del índice secundario requerirá unos cuantos accesos adicionales, en promedio, para encontrar cualquier bloque. Algo parecido sucede en cada eliminación. La conclusión es que la selección de los índices secundarios requiere seguir un criterio, puesto que debe determinarse qué conjuntos de campos se especificarán en operaciones muy frecuentes de modo que el tiempo ahorrado teniendo esos índices secundarios disponibles compense los costos de la actualización de los índices en cada inserción y eliminación.

11.4 Árboles de búsqueda externa

La estructura de datos tipo árbol presentada en el capítulo 5 para representar conjuntos puede usarse también para representar archivos externos. El árbol B, una generalización de los árboles 2-3 analizados en el capítulo 5, es especialmente adecuada para almacenamiento externo, y se ha convertido en la organización estándar para índices en sistemas de bases de datos. Esta sección presenta las técnicas básicas de recuperación, inserción y eliminación de información en árboles B.

Árboles de búsqueda múltiple

Un árbol de búsqueda m -ario es una generalización del árbol binario de búsqueda en el cual cada nodo tiene como máximo m hijos. Generalizando la propiedad del árbol binario de búsqueda, se requiere que si n_1 y n_2 son dos hijos de algún nodo, y n_1 está a la izquierda de n_2 , los elementos descendientes de n_1 sean todos menores que los de n_2 . Las operaciones MIEMBRO, INSERTA y SUPRIME de un árbol de búsqueda m -ario se realizan con una generalización de las operaciones en árboles binarios de búsqueda, estudiadas en la sección 5.1.

Sin embargo, aquí interesa el almacenamiento de registros en archivos, donde los archivos se depositan en bloques de almacenamiento externo. La adaptación correcta de la idea de árboles múltiples consiste en pensar en los nodos como bloques físicos. Un nodo interior contiene apuntadores a sus m hijos y también contiene $m-1$ claves que separan los descendientes del hijo. Los nodos hojas son bloques también, y contienen los registros del archivo principal.

Si se emplea un árbol binario de búsqueda de n nodos para representar un archivo almacenado en forma externa, pueden requerirse, en promedio, $\log_2 n$ accesos a bloques para recuperar un registro del archivo. En cambio, si se utiliza un árbol de búsqueda m -ario para representar el archivo, requerirá, en promedio, sólo $\log_m n$ accesos a bloques para recuperar un registro. Para $n = 10^6$, el árbol binario de búsqueda puede requerir cerca de 20 accesos a bloques, mientras que un árbol de 128 caminos requerirá sólo 3.

No se puede hacer m arbitrariamente grande, pues cuanto mayor sea m , mayor deberá ser el bloque. Más aún, es más lento leer y procesar un bloque más grande, así que hay un valor óptimo de m que reduce la cantidad de tiempo necesario para recorrer un árbol m -ario de búsqueda externa. En la práctica, se obtiene un valor cercano al mínimo para un amplio intervalo de m . (Véase Ejercicio 11.18.)

Arboles B

Un árbol B es una clase especial de árbol m -ario balanceado que permite recuperar, eliminar e insertar registros de un archivo externo con buen rendimiento en el peor caso. Es una generalización de los árboles 2-3 de la sección 5.4. Formalmente, un *árbol B de orden m* es un árbol de búsqueda m -ario con las siguientes propiedades:

1. La raíz es una hoja o tiene al menos dos hijos.
2. Cada nodo, excepto la raíz y las hojas, tiene entre $[m/2]$ y m hijos.
3. Cada camino desde la raíz hasta una hoja tiene la misma longitud.

Obsérvese que todo árbol 2-3 es un árbol B de orden 3. La figura 11.10 muestra un árbol B de orden 5, en el cual se supone que caben como máximo tres registros en un bloque hoja.

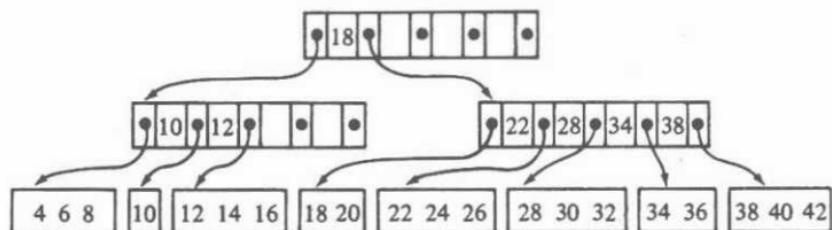


Fig. 11.10. Árbol B de orden 5.

Se puede considerar un árbol B como un índice jerárquico en el cual cada nodo ocupa un bloque en el almacenamiento externo. La raíz del árbol B es el índice de primer nivel. Cada nodo que no es hoja en el árbol B tiene la forma

$$(p_0, k_1, p_1, k_2, p_2, \dots, k_n, p_n)$$

donde p_i es un apuntador al i -ésimo hijo del nodo, $0 \leq i \leq n$, y k_i es una clave, $1 \leq i \leq n$. Las claves dentro de un nodo están clasificadas en orden tal que $k_1 < k_2 < \dots < k_n$. Todas las claves del subárbol al que apunta p_0 son menores que k_1 . Para $1 \leq i < n$, todas las claves del subárbol apuntado por p_i tienen valores mayores o iguales que k_i y menores que k_{i+1} . Todas las claves en el subárbol apuntado por p_n son mayores que k_n .

Hay varias formas de organizar las hojas. Aquí se supondrá que el archivo principal está almacenado sólo en las hojas, y que cada hoja ocupa un bloque.

Recuperación

Para recuperar un registro r con clave x , se sigue el camino desde la raíz hasta la hoja que contiene a r , si es que existe en el archivo. Se sigue este camino pasando sucesivamente nodos interiores ($p_0, k_1, p_1, \dots, k_n, p_n$) del almacenamiento externo a la memoria principal y buscando la posición de x relativa a las claves k_1, k_2, \dots, k_n . Si $k_i \leq x < k_{i+1}$, a continuación se busca el nodo apuntado por p_i y se repite el proceso. Si $x < k_1$, se aplica p_0 para alcanzar el siguiente nodo; si $x \geq k_n$, se utiliza p_n . Cuando este proceso llega a una hoja, debe buscarse el registro con la clave x . Si el número de elementos de un nodo es pequeño, es posible utilizar la búsqueda lineal dentro del nodo, de otra forma, conviene utilizar una búsqueda binaria.

Inserción

La inserción en un árbol B es similar a la inserción en árboles 2-3. Para insertar un registro r con clave x en un árbol B, primero se aplica el procedimiento de búsqueda para localizar la hoja L a la cual corresponde r . Si existe espacio suficiente para r en L , se inserta r en el orden correspondiente. En este caso no se requieren modificaciones a los antecesores de L .

Si no hay espacio para r en L , es necesario pedir al sistema de archivos un bloque nuevo L' para pasar la segunda mitad de los registros de L a L' , insertando r en el lugar adecuado en L o L' .[†] Sea el nodo P el padre de L ; P es conocido, ya que el procedimiento de búsqueda siguió un camino desde la raíz hasta L , pasando por P . Se aplica ahora el procedimiento de inserción recursivamente para colocar en P una clave k' y un apuntador l' a L' ; k' y l' se insertan inmediatamente después de la clave y y del apuntador de L . El valor de k' es el de la clave más pequeña de L' .

Si P ya tiene m apuntadores, la inserción de k' y l' dentro de P hará que P se dividiera y requiera la inserción de una clave y un apuntador en el padre de P . Los efectos de esta inserción pueden transmitirse a los antecesores del nodo L en dirección a la raíz, por el camino que se siguió con el procedimiento de búsqueda original. Incluso puede ser necesario dividir la raíz, en cuyo caso se creará una nueva raíz con las dos mitades de la raíz anterior como sus dos hijos. Esta es la única situación en la que un nodo puede tener menos de $m/2$ hijos.

Eliminación

Para eliminar un registro r con una clave x , primero se encuentra la hoja L que contiene a r . Despues, se elimina r de L , si existe. Si r es el primer registro en L , es necesario ir a P , el padre de L , para ajustar el valor de la clave en la entrada de P para que L sea el nuevo valor de la primera clave de L . Sin embargo, si L es el primer hijo de P , la primera clave de L no se registra en P , sino que aparece en alguno de

[†] Esta estrategia es la más sencilla de las varias respuestas aplicables a la situación en que ha de dividirse un bloque. En los ejercicios se mencionan algunas otras opciones que proporcionan una mayor ocupación media de los bloques con una inserción más laboriosa.

los antecesores de P , de forma específica, en el menor antecesor A tal que L no sea el descendiente más a la izquierda de A . Así pues, se debe propagar el cambio en la menor clave de L de vuelta por la trayectoria de la raíz a L .

Si L queda vacío después de la eliminación, se devuelve al sistema de archivos †; después se ajustan las claves y los apuntadores de P para reflejar la eliminación de L . Si el número de hijos de P ahora es menor que $m/2$, se examina el nodo P' situado inmediatamente a la izquierda (o a la derecha) de P en el mismo nivel del árbol. Si P' tiene por lo menos $[m/2] + 1$ hijos, se distribuyen las claves y los apuntadores de P y P' en forma equitativa entre ambos, cuidando, por supuesto, el orden de clasificación, de manera que ambos nodos tengan por lo menos $[m/2]$ hijos. Después, se modifican los valores de las claves para P y P' en el padre de P y, si fuera necesario, se propagan recursivamente los efectos de este cambio a todos los antecesores de P que se vean afectados.

Si P' tiene exactamente $[m/2]$ hijos, es bueno combinar P y P' en un solo nodo con $2[m/2] - 1$ hijos (esto es, m hijos a lo sumo). Despues, se deben quitar en el padre la clave y el apuntador correspondientes a P' . Esta eliminación puede hacerse con una aplicación recursiva del procedimiento de eliminación.

Si los efectos de la eliminación se propagan hasta la raíz, puede requerirse la combinación de los dos únicos hijos de ésta. En ese caso, el nodo combinado resultante quedará como nueva raíz, y la raíz anterior puede devolverse al sistema de archivos. La altura del árbol B se verá reducida en uno.

Ejemplo 11.5. Considérese el árbol B de orden 5 de la figura 11.10. Insertar el registro con clave 23 en este árbol produce el árbol B de la figura 11.11. Para insertar 23, se debe partir el bloque que contiene 22, 23, 24 y 26, ya que se ha supuesto que un bloqué se llena con tres registros. Los dos registros más pequeños permanecen en el mismo bloqué y los otros dos se colocan en un bloqué nuevo. Un par apuntador-clave del nodo nuevo debe insertarse en el padre, que entonces se divide, porque no puede contener seis apuntadores. La raíz recibe el par apuntador-clave para el nodo nuevo, pero no se divide, porque tiene capacidad en exceso.

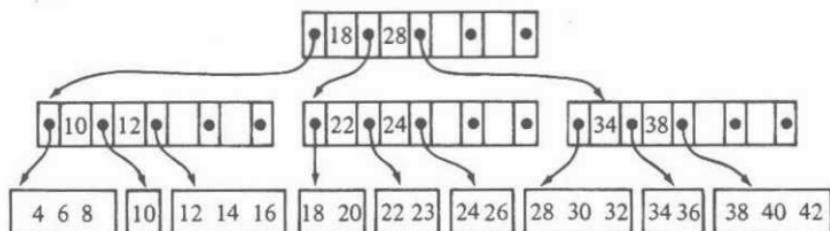


Fig. 11.11. Árbol B después de la inserción.

† Se pueden utilizar estrategias para evitar que los bloques hoja queden totalmente vacíos. Como ejemplo, a continuación se describe un esquema que impide que los nodos interiores queden ocupados en menos de la mitad de su capacidad, y esta técnica se puede aplicar también a las hojas, con un valor de m igual al mayor número de registros que caben en un bloqué.

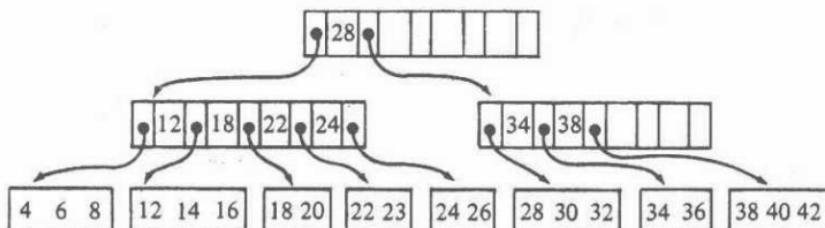


Fig. 11.12. Árbol B después de la eliminación.

La eliminación del registro 10 del árbol B de la figura 11.11 produce el árbol B de la figura 11.12. Aquí, el bloque que contiene a 10 se descarta; su padre queda con sólo dos hijos, y el hermano derecho tiene el número mínimo, tres; así, se combina el padre con su hermano, haciendo un nodo con cinco hijos. □

Análisis del tiempo de las operaciones con árboles B

Supóngase que se tiene un archivo con n registros organizado en un árbol B de orden m . Si cada hoja contiene en promedio b registros, el árbol tiene cerca de $[n/b]$ hojas. Los caminos más largos posibles se producen si cada nodo interior tiene la menor cantidad posible de hijos, esto es, $m/2$. En este caso, habrá unos $2[n/b]/m$ padres de hojas, $4[n/b]/m^2$ padres de padres de hojas, y así sucesivamente.

Si hay j nodos en el camino que va de la raíz a una hoja, entonces $2^{j-1}[n/b]/m^{j-1} \geq 1$, pues si no habría menos de un nodo en el nivel de la raíz. Por tanto, $[n/b] \geq (m/2)^{j-1}$, y $j \leq 1 + \log_{m/2} [n/b]$. Por ejemplo, si $n = 10^6$, $b = 10$ y $m = 100$, entonces $j \leq 3.5$. Obsérvese que b no es el número máximo de registros que se pueden poner en un bloque, sino un promedio, o número esperado. Sin embargo, redistribuyendo los registros entre bloques vecinos cuando uno se vacía hasta menos de la mitad, se puede asegurar que b es por lo menos la mitad del valor máximo. Obsérvese también que se ha supuesto que cada nodo interior tiene el mínimo número posible de hijos; en la práctica, el nodo interior promedio tendrá más que el mínimo, por lo que el análisis anterior resulta conservador.

Para una inserción o eliminación, se necesitan j accesos a bloques para localizar la hoja apropiada. El número exacto de accesos adicionales a bloques necesario para culminar la inserción o la eliminación, y distribuir sus efectos por el árbol, es difícil de calcular. La mayor parte de las veces sólo es necesario escribir de nuevo un bloque, la hoja que contiene al bloque de interés. Así, $2 + \log_{m/2} [n/b]$ puede tomarse como el número aproximado de accesos a bloques para inserción o eliminación.

Comparación de métodos

Se ha estudiado la función de dispersión, los índices dispersos y los árboles B como métodos posibles de organización de archivos externos. Es interesante comparar en

cada método el número de accesos a bloques relacionados con una operación con archivos.

La función de dispersión suele ser el método más rápido de los tres, y requiere en promedio dos accesos a bloques por cada operación (excluyendo los accesos requeridos para buscar la tabla de cubetas), si el número de cubetas es lo bastante grande para que la cubeta típica use sólo un bloque. Sin embargo, con la función de dispersión no es fácil acceder a los registros en el orden de clasificación.

Un índice disperso en un archivo de n registros permite que las operaciones sobre archivos se realicen en unos $2 + \log(n/bb')$ accesos a bloques mediante la búsqueda binaria; aquí, b es el número de registros que caben en un bloque y b' es el número de pares apuntador-clave que caben en un bloque del archivo índice. Los árboles B permiten operaciones con archivos en $2 + \log_{m/2} [n/b]$ accesos a bloques, donde m , el grado máximo de los nodos interiores, es aproximadamente b' . Los índices dispersos y los árboles B permiten el acceso a los registros en el orden de clasificación.

Todos estos métodos son muy buenos comparados con la búsqueda secuencial obvia en un archivo. Las diferencias de tiempo entre ellos, no obstante, son pequeñas y difíciles de determinar analíticamente, especialmente considerando que los parámetros importantes, como la longitud esperada del archivo y la razón de ocupación de bloques, son difíciles de predecir.

Parece ser que los árboles B se están popularizando con rapidez como medio de acceso a archivos en sistemas de bases de datos. En parte, se debe a su capacidad para manejar consultas de registros con claves en un intervalo determinado (lo que aprovecha el hecho de que los registros aparecen ordenados en el archivo principal). El índice disperso también maneja con eficiencia dichas consultas, pero es casi seguro que es menos eficiente que los árboles B. Intuitivamente, la razón de que los árboles B sean superiores a los índices dispersos es que se puede considerar un árbol B como un índice disperso sobre un índice disperso sobre un índice disperso, y así sucesivamente (aunque rara vez se necesitan más de tres niveles de índices).

Los árboles B también funcionan relativamente bien cuando se usan como índices secundarios, donde las «claves» no definen únicamente un registro. Aunque los registros con un valor dado en los campos designados de un índice secundario se extienden sobre muchos bloques, se pueden leer todos con un número de accesos a bloque igual al número de bloques que contienen esos registros, más el número de sus antecesores en el árbol B. En comparación, si esos registros, más otro grupo de tamaño similar tienen la misma función de dispersión y llegan a la misma cubeta, entonces la recuperación de cualquier grupo de una tabla de dispersión puede requerir un número de accesos a bloques cercano al doble del número de bloques en los cuales puede caber cada grupo. Es factible que haya otras razones en favor de los árboles B, como su rendimiento cuando varios procesos tienen acceso simultáneamente a la estructura, pero eso está fuera del alcance de este libro.

Ejercicios

- 11.1** Escribase un programa *concatena* que tome una secuencia de nombres de archivos como argumentos y escriba el contenido de los mismos en la salida estándar, concatenando así los archivos.
- 11.2** Escribase un programa *incluye* que copie su entrada en su salida, excepto cuando encuentre una línea de la forma `#incluye archivo`, en cuyo caso reemplaza esta línea por el contenido del archivo mencionado. Obsérvese que los archivos incluidos también pueden contener proposiciones `#incluye`.
- 11.3** ¿Cómo se comporta el programa del ejercicio 11.2 cuando un archivo se incluye a sí mismo?
- 11.4** Escribase un programa *compara* que compare dos archivos, registro por registro, para determinar si son iguales.
- *11.5** Reescribase el programa de comparación de archivos del ejercicio 11.4 con el algoritmo SCL de la sección 5.6 para encontrar la subsecuencia común más larga de registros en ambos archivos.
- 11.6** Escribase un programa *encuentra* que tome dos argumentos que consten de una cadena y un nombre de archivo, e imprima todas las líneas del archivo que contengan la cadena como una subcadena. Por ejemplo, si la cadena es «*ado*» y el archivo es una lista de palabras, entonces *encuentra* imprime todas las palabras que contienen el trígrama «*ado*».
- 11.7** Escribase un programa que lea un archivo y escriba en su salida estándar los registros del archivo clasificados.
- 11.8** ¿Cuáles son las primitivas que Pascal ofrece para tratar con archivos externos? ¿Cómo pueden mejorarse?
- *11.9** Supóngase que se maneja una clasificación en varias fases con tres archivos, y en la i -ésima fase se crea un archivo con r_i fragmentos de longitud l_i . En la n -ésima fase, se desea un fragmento en uno de los archivos y ninguno en los otros dos. Explíquese por qué cada uno de los siguientes enunciados debe ser cierto.
- $l_i = l_{i-1} + l_{i-2}$ para $i \geq 1$, donde l_0 y l_{-1} se consideran las longitudes de los fragmentos en los dos archivos ocupados inicialmente.
 - $r_i = r_{i-2} - r_{i-1}$ (o, en forma equivalente, $r_{i-2} = r_{i-1} + r_i$ para $i \geq 1$), donde r_0 y r_{-1} son el número de fragmentos en los dos archivos iniciales.
 - $r_n = r_{n-1} = 1$ y, por tanto, r_n, r_{n-1}, \dots, r_1 , forma una sucesión de Fibonacci.
- *11.10** ¿Qué condición debe agregarse a las del ejercicio 11.9 para hacer posible una clasificación en varias fases,
- con fragmentos iniciales de longitud uno (esto es, $l_0 = l_{-1} = 1$)?
 - la ejecución para k fases, pero con fragmentos iniciales diferentes a uno permitido?

Sugerencia. Considérense unos cuantos ejemplos, como $l_n = 50$, $l_{n-1} = 31$ o $l_n = 50$, $l_{n-1} = 32$.

****11.11** Generalíicense los ejercicios 11.9 y 11.10 a clasificaciones de varias fases con más de tres archivos.

****11.12** Muéstrese que:

- a) Cualquier algoritmo de clasificación externa que utilice sólo una cinta como almacenamiento externo debe requerir un tiempo $\Omega(n^2)$ para clasificar n registros.
- b) Un tiempo $O(n \log n)$ es suficiente si hay dos cintas para almacenamiento externo.

11.13 Supóngase que se tiene un archivo externo de arcos dirigidos $x \rightarrow y$ que forma un grafo acíclico, y que no hay suficiente espacio en la memoria interna para contener el conjunto completo de vértices o aristas al mismo tiempo.

- a) Escribábase un programa de clasificación topológica externa que escriba un ordenamiento lineal de vértices, de modo que si $x \rightarrow y$ es un arco dirigido, el vértice x aparezca antes de y en el ordenamiento lineal.
- b) ¿Cuál es la complejidad de tiempo y espacio de este programa como una función del número de accesos a bloques?
- c) ¿Qué haría este programa si el grafo dirigido fuera cíclico?
- **d) ¿Cuál es el número mínimo de accesos a bloque necesarios para clasificar topológicamente un grafo dirigido acíclico almacenado externamente?

11.14 Supóngase que se tiene un archivo de un millón de registros, donde cada registro ocupa 100 bytes. Los bloques son de 1000 bytes de longitud, y un apuntador a un bloque ocupa 4 bytes. Diséñese una organización con función de dispersión para este archivo. ¿Cuántos bloques se necesitan para la tabla de cubetas y las cubetas?

11.15 Diséñese una organización con árboles B para el archivo del ejercicio 11.14.

11.16 Escribanse programas para implantar las operaciones RECUPERA, INSERTA, SUPRIME y MODIFICA en

- a) archivos con función de dispersión,
- b) archivos indizados,
- c) archivos con árboles B.

11.17 Escribábase un programa para encontrar el k -ésimo elemento más grande en

- a) un archivo con índice disperso
- b) un archivo con árbol B

- *11.18 Supóngase que se necesitan $a + bm$ milisegundos para leer un bloque que contiene un nodo de un árbol m -ario de búsqueda, y $c + d \log_2 m$ milisegundos para procesar cada nodo en memoria interna. Si hay n nodos en el árbol, es necesario leer cerca de $\log_m n$ nodos para localizar un registro dado. Por tanto, el tiempo total requerido para encontrar un registro dado en el árbol es

$$(\log_m n)(a + bm + c + d \log_2 m) = (\log_2 n)((a + c + bm)/\log_2 m) + d$$

milisegundos. Háganse estimaciones razonables para los valores de a , b , c y d , y represéntese gráficamente esta cantidad como una función de m . ¿Para qué valor de m se obtiene el mínimo?

- *11.19 Un árbol B^* es un árbol B en el que cada nodo interno está lleno en dos terceras partes por lo menos (en vez de la mitad). Diséñese un esquema de inserción para árboles B^* que retrase la división de nodos internos hasta que dos nodos hermanos estén llenos. Los dos nodos hermanos pueden entonces dividirse en tres, cada uno lleno en dos terceras partes. ¿Qué ventajas y desventajas tienen los árboles B^* en relación con los árboles B ?
- *11.20 Cuando la clave de un registro es una cadena de caracteres, se puede ahorrar espacio almacenando sólo un prefijo de la clave como separador clave de cada nodo interno de un árbol B . Por ejemplo, «gato» y «perro», pueden separarse por el prefijo «g» o «ga» de «gato». Diséñese un algoritmo de inserción en árboles B que utilice prefijos de claves como separadores que sean siempre lo más cortos posible.
- *11.21 Supóngase que en cierto archivo las operaciones de inserción y eliminación se efectúan en una fracción p del tiempo, y en el tiempo $1-p$ restante se realizan recuperaciones donde se especifica exactamente un campo. Hay k campos en los registros, y una recuperación especifica el i -ésimo campo con probabilidad q_i . Supóngase también que una recuperación requiere a milisegundos si no hay índice secundario para el campo especificado, y b milisegundos, si lo hay, y que una inserción o eliminación requiere $c + sd$ milisegundos, donde s es el número de índices secundarios. Determínese, como una función de a , b , c , d , p y las q_i , qué índices secundarios deben crearse para el archivo, de manera que el tiempo promedio por operación sea minimice.
- **11.22 Supóngase que las claves son de un tipo que puede ordenarse linealmente, como los números reales, y que se conoce la distribución de probabilidades con que aparecerán las claves de valores dados en el archivo. Se puede aprovechar este conocimiento para mejorar una búsqueda binaria cuando se busca una clave en un índice disperso. Un esquema, llamado *búsqueda por interpolación*, usa esta información estadística para predecir dónde es más probable que se encuentre una clave x , en el intervalo de bloques de índices B_1, \dots, B_p , a los cuales se ha limitado la búsqueda. Proporcíonese

- a) un algoritmo para aprovechar el conocimiento estadístico en esta forma, y
 b) una demostración de que $O(\log \log n)$ accesos a bloques son suficientes, en promedio, para encontrar una clave.
- 11.23** Supóngase que se tiene un archivo externo de registros, y que cada registro consta de una arista de un grafo G y un costo asociado a esa arista.
- Escríbase un programa para construir un árbol abarcador de costo mínimo para G , suponiendo que existe suficiente memoria para almacenar todos los vértices de G , pero no todas las aristas.
 - ¿Cuál es la complejidad de tiempo de ese programa como una función del número de vértices y aristas?
- Sugerencia.* Un enfoque posible de este problema es mantener en memoria un bosque con los componentes conexos actuales. Cada arista es leída y procesada como sigue: si la siguiente arista tiene extremos en dos componentes distintos, se agrega y se mezclan los componentes. Si la arista crea un ciclo en un componente ya existente, se agrega y se elimina la arista de mayor costo del ciclo (que puede ser la arista actual). Este enfoque es similar al algoritmo de Kruskal, pero no requiere clasificar las aristas, lo cual es un aspecto importante de este problema.
- 11.24** Supóngase que se tiene un archivo que contiene una secuencia de números positivos y negativos a_1, a_2, \dots, a_n . Escríbase un programa $O(n)$ para encontrar una subsecuencia contigua a_i, a_{i+1}, \dots, a_j que tenga la suma mayor $a_i + a_{i+1} + \dots + a_j$ de cualquier subsecuencia.

Notas bibliográficas

Como material adicional sobre clasificación externa, véase Knuth [1973]. Material posterior sobre estructuras de datos externas y su uso en sistemas de bases de datos puede encontrarse en esa obra y en Ullman [1982] y Wiederhold [1982]. La clasificación en varias fases se estudia en Shell [1971]. El esquema de intercalación con seis buffers de la sección 11.2 es de Friend [1956], y el de cuatro, de Knuth [1973].

La selección de índices secundarios, de la cual el ejercicio 11.21 es una simplificación, se analiza en Lum y Ling [1970] y Schkolnick [1975]. Los árboles B se presentaron originalmente en Bayer y McCreight [1972]. En Comer [1979], se examinan muchas variantes, y en Gudes y Tsur [1980] se evalúa el rendimiento en la práctica.

La información acerca del ejercicio 11.12, clasificación con una y dos cintas, puede encontrarse en Floyd y Smith [1973]. El ejercicio 11.22 sobre búsqueda por in-

terpolación se analiza con detalle en Yao y Yao [1976], y Perl, Itai y Avni [1978].

Una implantación excelente del enfoque sugerido en el ejercicio 11.23 para el problema del árbol abarcador externo de costo mínimo fue estudiada por V. A. Vyssotsky, alrededor de 1960 (sin publicar). El ejercicio 11.24 se debe a M. I. Shamos.

12

Administración de memoria

En este capítulo se analizan las estrategias básicas para reutilizar el espacio en memoria o compartirlo entre objetos distintos que crecen y se contraen de manera arbitraria. Por ejemplo, se estudiarán los métodos que mantienen listas enlazadas de espacio disponible, y técnicas de «recolección de basura», que sirven para conocer la disponibilidad de espacio sólo cuando parece que se ha terminado el espacio disponible.

12.1 Aspectos de la administración de memoria

En la operación de sistemas de cómputo existen muchas situaciones en las que se administra un recurso limitado de memoria, es decir, se comparte entre varios «competidores». Un programador que no se ocupe de la realización de programas del sistema (compiladores, sistemas operativos, etcétera) tal vez no perciba dichas actividades, debido a que suelen realizarse «entre bastidores». Como ejemplo, los programadores de Pascal saben que el procedimiento *new(p)* hará que el apuntador *p* señale hacia un objeto nuevo del tipo correcto; pero, ¿de dónde procede el espacio para el objeto? El procedimiento *new* tiene acceso a una región grande de memoria, conocida como «estructura dinámica» (*heap*), que las variables del programa no usan. De esa región, se selecciona un bloque no utilizado de bytes consecutivos suficiente para contener un objeto del tipo al que apunta *p*, y se hace que *p* contenga la dirección del primer byte de ese bloque. Pero ¿cómo sabe el procedimiento *new* qué bytes de la memoria están «desocupados»? La sección 12.4 sugiere la respuesta.

Aún más misterioso es lo que sucede si se modifica el valor de *p*, bien por una asignación o por otra llamada a *new(p)*. El bloque de memoria al que apunta *p* puede ser ahora *inaccesible*, en el sentido de que no hay forma de llegar a él mediante las estructuras de datos del programa, y se puede reutilizar su espacio. Por otro lado, antes de cambiar *p*, su valor pudo haberse copiado en otra variable. En ese caso, el bloque de memoria será parte todavía de las estructuras de datos del programa. ¿Cómo se puede saber si un bloque de la región de memoria utilizada por el procedimiento *new* ya no es requerido por el programa?

El tipo de administración de memoria que se efectúa en Pascal sólo es uno más. Por ejemplo, en algunas situaciones, como Pascal, objetos de tamaños distintos comparten el mismo espacio de memoria; en otras situaciones, todos los objetos que comparten el espacio son del mismo tamaño. Esta distinción con respecto a los tamaños

de los objetos es una forma de clasificar las clases de problemas de administración de memoria a que uno debe enfrentarse. A continuación se presentan algunos ejemplos más.

1. En el lenguaje de programación LISP, el espacio de memoria se divide en *celdas* que, en esencia, son registros que constan de dos campos; cada campo puede contener un *átomo* (un objeto del tipo elemental, como puede ser un entero) o un apuntador a una celda. Los átomos y apuntadores son del mismo tamaño, así que todas las celdas requieren el mismo número de bytes. Todas las estructuras de datos conocidas pueden construirse a partir de esas celdas. Por ejemplo, las listas enlazadas de átomos pueden usar los primeros campos de las celdas para contener átomos, y los segundos campos, para contener apuntadores a las siguientes celdas de la lista. Los árboles binarios pueden representarse utilizando el primer campo de cada celda para apuntar al hijo izquierdo, y el segundo, para apuntar al hijo derecho. Al ejecutar un programa en LISP, el espacio de memoria empleado para contener una celda puede ser a la vez parte de estructuras distintas en momentos diferentes, ya sea porque una celda se mueve entre estructuras, o porque se separa de todas las estructuras y su espacio se utiliza de nuevo.
2. Un sistema de archivos, en general, divide los dispositivos de almacenamiento secundario, como los discos, en bloques de longitud fija. Por ejemplo, UNIX siempre usa bloques de 512 bytes. Los archivos se almacenan en una secuencia de bloques (que no son necesariamente consecutivos). Conforme se crean y destruyen archivos, los bloques del almacenamiento secundario quedan disponibles para ser utilizados de nuevo.
3. Un sistema operativo de multiprogramación típico permite que varios programas comparten la memoria principal al mismo tiempo. Cada programa requiere una cantidad determinada de memoria, la cual es conocida por el sistema operativo, y este requisito es parte de la solicitud de servicio emitida cuando se desea ejecutar el programa. Mientras en los ejemplos (1) y (2) los objetos que comparten memoria (celdas y bloques, respectivamente) eran todos del mismo tamaño, distintos programas requieren cantidades diferentes de memoria. Así, cuando termina un programa que usa 100K bytes, puede reemplazarse por otros dos que manejen 50K cada uno, o uno 20K y otro 70K (con 10K no utilizados). Como solución distinta, los 100K bytes liberados a la terminación del programa pueden combinarse con los 50K adyacentes que estén sin utilizar, y entonces puede ejecutarse un programa que requiera hasta 150K. Otra posibilidad es que ningún programa nuevo quepa en el espacio liberado, y que 100K bytes quede momentáneamente sin utilizar.
4. Hay muchos lenguajes de programación, como SNOBOL, APL o SETL, que asignan espacio a objetos de tamaño arbitrario. A esos objetos, que son valores asignados a variables, se les asigna un bloque de espacio de un gran bloque de memoria, que suele denominarse *estructura dinámica (heap)*. Cuando cambia el valor de una variable, al valor nuevo se le asigna un espacio en la estructura dinámica, y un apuntador para que la variable apunte al nuevo valor. Es posible que el valor anterior de la variable quede ahora sin utilizar, y su espacio se pueda volver a utilizar. Sin embargo, los lenguajes como SNOBOL o SETL realizan asig-

naciones como $A = B$ haciendo que el apuntador de A apunte al mismo objeto que el apuntador de B ; si A o B fueran reasignados, el objeto anterior no se liberaría y su espacio no podría solicitarse.

Ejemplo 12.1. En la figura 12.1(a) se observa la estructura dinámica que puede manejar en un programa en SNOBOL con las variables A , B y C . El valor de cualquier variable en SNOBOL es una cadena de caracteres y, en este caso, el valor de A y B es «BUENOS DIAS» y el valor de C es «PASA LA SAL».

Se ha elegido la representación de cadenas de caracteres por medio de apunadores a bloques de memoria en la estructura dinámica. Esos bloques tienen sus dos primeros bytes (el número 2 es un valor típico que podría cambiarse) dedicados a un entero que da la longitud de la cadena; por ejemplo, «BUENOS DIAS» tiene longitud 11, contando el espacio entre palabras, así que el valor de A (y de B) ocupa 13 bytes.

Si el valor de B se cambia por «BUENAS NOCHES», se puede encontrar un bloque vacío en el montón con 15 bytes para almacenar el nuevo valor de B , incluyendo los dos bytes de la longitud. Se hace que el apuntador de B apunte al nuevo valor, como se muestra en la figura 12.1(b). El bloque que contiene el entero 11 y «BUENOS DIAS» aún es útil, pues A continúa apuntando a él. Si el valor de A cambia, ese bloque quedará sin uso y puede volver a utilizarse. La forma de saber que no hay apunadores a tales bloques es un tema importante en este capítulo □

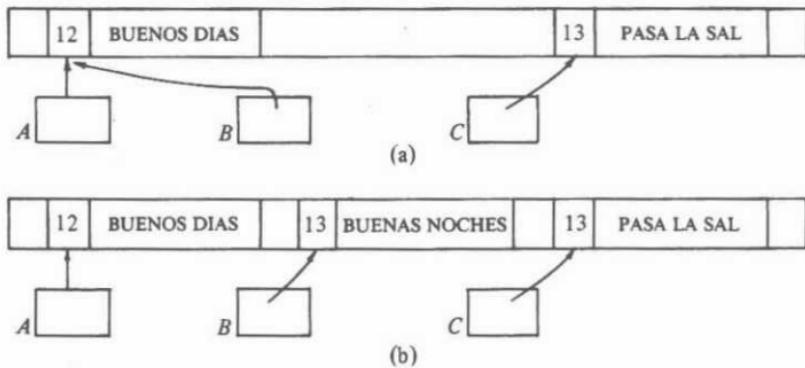


Fig. 12.1. Variables de cadenas en una estructura dinámica.

En los cuatro ejemplos anteriores, se pueden observar diferencias a lo largo, por lo menos, de dos «dimensiones» ortogonales. El primer aspecto es si los objetos que se encuentran compartiendo el espacio son de la misma longitud o no. En los dos primeros ejemplos, los programas en LISP y el almacenamiento de archivos, los objetos, celdas tipo LISP en un caso, y bloques con partes de archivos, en el otro, son del mismo tamaño. Este hecho permite ciertas simplificaciones del problema de la administración de memoria. Por ejemplo, en la realización en LISP, una región de memoria se divide en espacios, cada uno de los cuales puede contener exactamente una celda. El problema de la administración es encontrar espacios vacíos para ubi-

car las celdas recién creadas y nunca se necesita almacenar una celda en una posición en que se translapen dos espacios. Igualmente, en el segundo ejemplo, un disco se divide en bloques de tamaño idéntico, y a cada bloque se le asigna parte de un archivo; nunca se emplea un bloque para almacenar partes de dos o más archivos, aunque un archivo termine en mitad de un bloque.

En contraste, el tercero y cuarto ejemplos cubren la asignación de memoria para un sistema de multiprogramación y administración de una estructura dinámica para los lenguajes que tratan con variables cuyos valores son objetos «grandes», y aquí se habla de asignación de espacio en bloques de tamaño diferente. Este requisito presenta ciertos problemas que no se presentan en el caso de longitudes fijas. Por ejemplo, se teme la *fragmentación*, una situación en la cual hay mucho espacio sin utilizar, pero está distribuido en fragmentos tan pequeños que no puede encontrarse espacio para un objeto grande. Se profundizará más acerca de la administración de estructuras dinámicas en las secciones 12.4 y 12.5.

El segundo aspecto importante es si la *recolección de basura*, un término muy descriptivo para la recuperación de espacio no utilizado, se efectúa explícita o implícitamente, esto es, mediante un mandato del programa o sólo como respuesta a una petición de espacio que no puede satisfacerse de otra manera. En el caso de la administración de archivos, cuando se elimina un archivo, los bloques que se utilizaron para contenerlo son conocidos por el sistema de archivos. Por ejemplo, el sistema de archivos puede grabar la dirección de uno o más «bloques maestros» para cada archivo existente; los bloques maestros listan las direcciones de todos los bloques utilizados por el archivo. Así, cuando se elimina un archivo, el sistema de archivos puede hacerlo explícitamente disponible para reutilizar todos los bloques eliminados por ese archivo.

En contraste, las celdas de LISP continúan ocupando su espacio en memoria cuando se apartan de las estructuras de datos del programa. Debido a la posibilidad de que haya varios apuntadores a una celda, no se puede decir cuándo una celda está separada por completo y, por tanto, tampoco recoger en forma explícita las celdas, como se hizo con los bloques de un archivo eliminado. Tarde o temprano, todos los espacios en memoria corresponderán a celdas útiles o inútiles, y la siguiente solicitud de espacio para otra celda activará implícitamente una «recolección de basura»; en ese momento, el intérprete de LISP marca todas las celdas útiles, por medio de un algoritmo similar al que se presentará en la sección 12.3, y después enlazará todos los bloques que contengan celdas inútiles en una lista de espacio disponible, para poderlas reutilizar.

La figura 12.2 ilustra las cuatro clases de administración de memoria y da un ejemplo de cada una. De la figura 12.2 ya se han comentado los ejemplos de bloques de tamaño fijo. La administración de la memoria principal en un sistema de programación múltiple es un ejemplo de petición explícita de bloques con longitud variable. Esto es, cuando un programa termina, el sistema operativo, sabiendo qué área de memoria se otorgó al programa y sabiendo que ningún otro programa puede usar ese espacio, hace que el espacio quede inmediatamente disponible para cualquier otro programa.

La administración de una estructura dinámica en SNOBOL o en muchos otros lenguajes es un ejemplo de bloques con longitud variable y recolección de basura.

		recuperación del espacio no utilizado	
		explícita	recolección de basura
tamaño del bloque	fijo	sistema de archivos	LISP
	variable	sistema de multiprogramación	SNOBOL

Fig. 12.2. Ejemplos de las cuatro estrategias de administración de memoria.

Como en LISP, un intérprete típico de SNOBOL no intenta recuperar bloques de memoria hasta que se agota el espacio. En ese momento, el intérprete realiza una recolección de basura igual que lo hace un intérprete de LISP, pero con la posibilidad adicional de que las cadenas se muevan en torno a la estructura dinámica para reducir la fragmentación, y que los bloques libres adyacentes se combinen para formar bloques más grandes. Obsérvese que los dos últimos pasos no tienen sentido en un ambiente LISP.

12.2 Administración de bloques de igual tamaño

Supóngase que se tiene un programa que maneja celdas con un par de campos cada una; cada campo puede ser un apuntador a una celda o puede contener un «átomo». Por supuesto, la situación es igual a la de un programa escrito en LISP, pero el programa puede escribirse casi en cualquier lenguaje, incluso en Pascal, si se definen las celdas del tipo registro variante. Las celdas vacías que se encuentran disponibles para su incorporación a una estructura de datos se colocan en una lista de espacio disponible, y cada variable del programa se representa por un apuntador a una celda; la celda apuntada puede pertenecer a una gran estructura de datos.

Ejemplo 12.2. En la figura 12.3, se observa una estructura de datos posible. *A*, *B* y *C* son variables, y las letras minúsculas representan átomos. Obsérvense algunos fenómenos interesantes: la celda que contiene el átomo *a* está apuntada por la variable *A* y por otra celda; la celda que contiene el átomo *c* está apuntada por dos celdas distintas; las celdas que contienen *q* y *h* son un caso especial, porque aunque se apuntan mutuamente, no son accesibles desde las variables *A*, *B* o *C*, ni están en la lista de espacio disponible. □

Supóngase que cuando se ejecuta el programa, se pueden quitar celdas nuevas de la lista de espacio disponible; por ejemplo, puede ser conveniente sustituir el apuntador nulo de la celda con el átomo *c* de la figura 12.3 por un apuntador a una celda nueva que contenga el átomo *i* y un apuntador nulo. Esta celda se eliminará de la

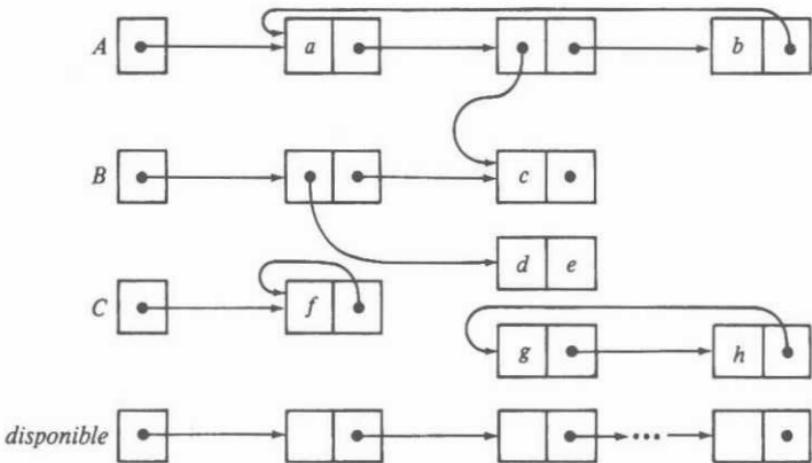


Fig. 12.3. Red de celdas.

cabeza de la lista de espacio disponible. También es posible que de vez en cuando los apuntadores cambien, de forma que las celdas se separen de las variables del programa, como sucedió con las celdas *g* y *h* de la figura 12.3. Por ejemplo, la celda en que se encuentra el átomo *c*, pudo haber apuntado alguna vez a la celda que contiene a *g*. Como otro ejemplo, el valor de la variable *B* puede cambiar en un momento dado, lo cual si nada más ha cambiado, aparta la celda apuntada por *B* en la figura 12.3, al igual que la celda que contiene a *d* y *e* (pero no la celda que contiene a *c*, pues ésta es accesible desde *A*). Las celdas que no pueden alcanzarse desde ninguna variable y que no están en la lista de espacio disponible son conocidas como celdas *inaccesibles*.

Cuando se separan las celdas, y no las vuelve a necesitar el programa, sería deseable que volvieran a la lista de espacio disponible, para poderlas usar de nuevo. Si no se reclaman dichas celdas, con el tiempo se llegará a la situación inaceptable de que el programa no esté empleando todas las celdas, aunque requiera una celda nueva, y la lista de espacio disponible esté vacía. Es entonces cuando debe realizarse una recolección de basura, lo que consume tiempo. Este paso de recolección de basura es «implícito», en el sentido de que no lo llamó explícitamente la petición de espacio.

Cuentas de referencia

Un enfoque atractivo para la detección de celdas inaccesibles consiste en incluir en cada celda una *cuenta de referencia*, esto es, un campo de tipo entero cuyo valor sea igual al número de apuntadores a la celda. Es fácil mantener las cuentas de referencia; cuando se hace que un apuntador apunte a una celda, se agrega un uno a la cuenta de referencia de esa celda, y cuando se reasigna un apuntador no nulo, primero

disminuye en uno la cuenta de referencia de la celda apuntada. Si una cuenta de referencia llega a valer cero, la celda será inaccesible y se podrá devolver a la lista de disponibles.

Lamentablemente, las cuentas de referencia no siempre funcionan. Las celdas con g y h de la figura 12.3 son inaccesibles y enlazadas en un ciclo. Sus cuentas de referencia valen 1, de manera que no se pueden devolver a la lista de disponibles. Se puede intentar detectar ciclos de celdas inaccesibles de distintas formas, pero tal vez no valga la pena hacerlo. Las cuentas de referencia son útiles para estructuras que no tienen ciclos de apuntadores. Un ejemplo de estructura sin posibilidades de ciclos es una colección de variables apuntando a bloques que contienen datos, como en la figura 12.1. De esta forma, se puede hacer la recolección de basura de forma explícita, recogiendo un bloque cuando su cuenta de referencia ha alcanzado el valor cero. Sin embargo, cuando las estructuras de datos permiten ciclos de apuntadores, la estrategia basada en cuentas de referencia por lo general es inferior a otro enfoque que se analizará en la siguiente sección, tanto en función del espacio requerido por las celdas como del tiempo relacionado con el caso de las celdas inaccesibles.

12.3 Algoritmos de recolección de basura para bloques de igual tamaño

Ahora se presenta un algoritmo para detectar qué celdas de una colección de los tipos sugeridos en la figura 12.3 son accesibles desde las variables del programa. Se definirá con precisión el problema definiendo un tipo de celda en Pascal que es una variante de tipo registro; las cuatro variantes, que se llamarán PP , PA , AP , AA , se determinan según cuáles de los dos campos de datos son apuntadores y cuáles son átomos. Por ejemplo, PA significa que el campo izquierdo es un apuntador y el campo derecho es un átomo. Un campo booleano adicional en las celdas, llamado *marca*, indica si la celda es accesible. Es decir, poniendo *marca* en verdadero al hacer la recolección de basura, se «marca» la celda, indicando que es accesible. En la figura 12.4 se muestran las definiciones de los tipos.

```

type
  tipo_átomo = { algún tipo apropiado, de preferencia,
    del mismo tamaño que los apuntadores }
  patrones = (PP, PA, AP, AA);
  tipo_celda = record
    marca: boolean;
    case patrón: patrones of
      PP: (izquierda: ↑ tipo_celda; derecha: ↑ tipo_celda);
      PA: (izquierda: ↑ tipo_celda; derecha: tipo_átomo);
      AP: (izquierda: tipo_átomo; derecha: ↑ tipo_celda);
      AA: (izquierda: tipo_átomo; derecha: tipo_átomo);
  end;

```

Fig. 12.4. Definición del tipo de las celdas.

Se supone que hay un arreglo de celdas, que ocupa casi toda la memoria, y una colección de variables, que son apuntadores a las celdas. Por simplicidad, se da por hecho que hay sólo una variable, llamada *fuente*, apuntando a una celda, pero la extensión a muchas variables no es difícil †. Esto es, se declara

```
var
    fuente: ↑ tipo_celda;
    memoria: array [1..tamaño_memoria] of tipo_celda;
```

Para marcar las celdas accesibles desde *fuente*, primero se «desmarcan» todas las celdas, sean o no accesibles, recorriendo todo el arreglo *memoria* y poniendo el campo *marca* en falso. Después, se realiza una búsqueda primera de profundidad en el grafo que emana de *fuente*, marcando todas las celdas visitadas. Las celdas visitadas son exactamente aquellas que son accesibles. Después, se recorre el arreglo *memoria* para agregar a la lista de espacio disponible todas las celdas no marcadas. La figura 12.5 muestra un procedimiento *bpf* para realizar la búsqueda en profundidad; *bpf* se llama con el procedimiento *recoge* que desmarca todas las celdas, y marca las celdas accesibles llamando a *bpf*. No se presenta el código para enlazar la lista de espacio disponible debido a las peculiaridades de Pascal. Por ejemplo, aunque se pueden enlazar las celdas disponibles con todas las celdas izquierdas o todas las derechas, ya que se ha supuesto que los apuntadores y los átomos tienen el mismo tamaño, no está permitido reemplazar átomos por apuntadores en las celdas que sean del tipo variante *AA*.

- (1) procedure *bpf*(*celda_actual*: ↑ tipo_celda);
 { Si se marcó la celda actual no se hace nada; en caso contrario,
 debe marcarse y llamar a *bpf* en cualquier celda apuntada por
 la celda actual }
- begin
- (2) with *celda_actual* ↑ do
 if *marca* = false then begin
 marca := true;
 if (*patrón* = *PP*) or (*patrón* = *PA*) then
 if *izquierda* <> nil then
 bpf(*izquierda*);
 if (*patrón* = *PP*) or (*patrón* = *AP*) then
 if *derecha* <> nil then
 bpf(*derecha*);
 end
 end; { *bpf* }
- (11) procedure *recoge*;
 var
 i: integer;

† Cada lenguaje de programación debe proporcionar un método propio de representación del conjunto de variables actual, y cualquiera de los métodos estudiados en los capítulos 4 y 5 es adecuado. Por ejemplo, la mayor parte de las aplicaciones usan una tabla de dispersión para guardar las variables.

```

begin
(12)    for i := 1 to tamaño_memoria do { "desmarcar" todas las celdas }
(13)        memoria[i].marca := false;
(14)        bpf(fuente); { marcar las celdas accesibles }
(15)        { aquí va el código para la recogida }
end; { recoge }

```

Fig. 12.5. Algoritmo para marcar celdas accesibles.

Recolección en el mismo sitio

El algoritmo de la figura 12.5 tiene un defecto sutil; en un ambiente de programación donde la memoria es limitada, puede suceder que no haya espacio disponible para almacenar la pila requerida para las llamadas recursivas a *bpf*. Como se ilustró en la sección 2.6, cada vez que *bpf* se llama a sí mismo, Pascal (o cualquier otro lenguaje que permita la recursión) crea un «registro de activación» para esa llamada particular a *bpf*. En general, un registro de activación contiene espacio para parámetros y variables locales al procedimiento, del cual cada llamada necesita su propia copia. Algo que también es necesario en cada registro de activación es una «dirección de retorno», el lugar al cual debe regresar el control cuando termina esta llamada recursiva al procedimiento.

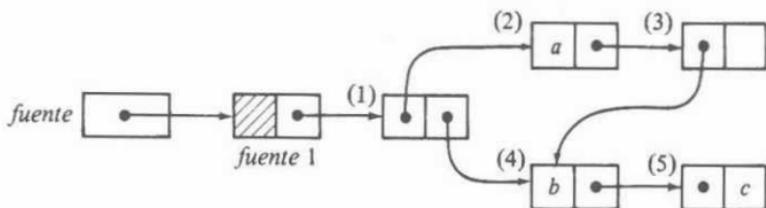
En el caso particular de *bpf*, sólo se necesita espacio para el parámetro y la dirección de retorno [esto es, fue llamado desde la línea (14) de *recoge*, desde la línea (7) o la (10) de otra invocación de *bpf*]. No obstante, esta solicitud de espacio es suficiente para que, si toda la memoria se enlaza en una sola cadena extendida desde *fuente* (por lo que el número de llamadas activas a *bpf* puede ser en algún momento igual a la longitud de esta cadena), se requerirá bastante más espacio para la pila de registros de activación que el asignado para la memoria. Si ese espacio quizás no estuviera disponible, sería imposible realizar la marcación.

Por fortuna, hay un ingenioso algoritmo, conocido como *algoritmo de Deutsch-Schorr-Waite*, para hacer la marcación en el mismo sitio. Es necesario convencerse de que la secuencia de celdas sobre la cual se ha realizado una llamada a *bpf*, sin haber terminado, en realidad forma un camino desde *fuente* hasta la celda en la cual se hizo la llamada actual a *bpf*. Así, se puede usar una versión no recursiva de *bpf*, y en vez de una pila de registros de activación para registrar el camino de celdas desde *fuente* hasta la celda que se está examinando en ese momento, se pueden usar los campos apuntadores del camino para contener el camino mismo. Esto es, cada celda del camino, excepto la última, contiene en el campo *derecha* o *izquierda* un apuntador a su *predecesor*, la celda más cercana a *fuente*. Se describirá el algoritmo de Deutsch-Schorr-Waite con un campo extra de un solo bit, llamado *atrás*, que es de un tipo enumerado (*I,D*), e informa si el campo izquierdo o el derecho apunta al predecesor. Más tarde, se evaluará la forma de almacenar la información contenida en *atrás* en el campo *patrón*, sin necesidad de espacio adicional en las celdas.

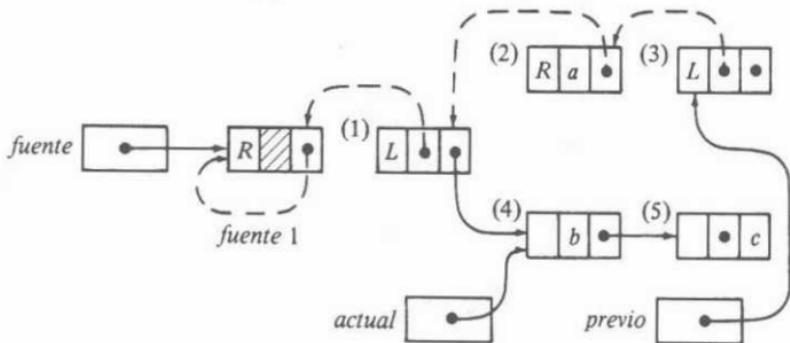
El nuevo procedimiento para búsqueda en profundidad no recursiva, que se llama *bpfnr*, se vale de un apuntador, *actual*, a la celda actual, y un apuntador, *previo*, al predecesor de la celda actual. La variable *fuente* apunta a una celda *fuentel* que

tiene un apuntador sólo en su campo derecho \dagger . Antes de marcar se asigna valor inicial a *fuentel* para tener $atrás = D$, y su campo derecho apuntando a sí mismo. A la celda que suele apuntar *fuentel*, apunta ahora *actual*, y a *fuentel* ahora *previo*. Se detiene la operación de marcado ocurre cuando $actual = previo$, lo cual sólo ocurre cuando ambos apuntan a *fuentel*, y se ha revisado por completo la estructura.

Ejemplo 12.3. La figura 12.6(b) muestra una posible estructura que emana de *fuente*. Si hace una búsqueda primera de profundidad en la estructura, se visitan (1), (2), (3) y (4), en ese orden. La figura 12.6(b) muestra las modificaciones hechas a los apuntadores cuando la celda actual es (4). Se muestra el valor del campo *atrás*, aunque no sucede lo mismo con los campos *marca* y *patrón*. El camino actual va de (4)



(a) Estructura de una celda



(b) Situación en la que la celda (4) es la actual

Fig. 12.6. Uso de apuntadores para representar el camino de regreso a *fuente*.

a (3) a (2) y a (1), de vuelta a *fuentel*; está representado por líneas (apuntadores) de puntos. Por ejemplo, la celda (1) tiene $atrás = I$, ya que el campo *izquierda* de (1), que en la figura 12.6 contiene un apuntador a (2), está apuntando hacia atrás, en vez de hacia adelante a lo largo del camino; sin embargo, se restaurará ese apuntador cuando la búsqueda en profundidad por fin regrese de la celda (2) a la (1). Análogamente, en la celda (2), $atrás = D$, y el campo derecho de (2) apunta hacia atrás en el camino hacia (1), en vez de dirigirse a (3), como lo hacía en la figura 12.6(a). □

\dagger Esta incomodidad es necesaria debido a las peculiaridades de Pascal.

Son tres los pasos básicos para realizar la búsqueda en profundidad:

1. *Avanzar.* Si se determina que la celda actual tiene uno o más apuntadores no nulos, se avanza al primero de ellos, esto es, se sigue el apuntador en *izquierda*, pero si no lo hay, se sigue el apuntador en *derecha*. Por «avanzar» se entiende hacer que la celda apuntada se convierta en la celda actual y, la celda actual, en la anterior. Para ayudar a encontrar el camino de regreso, se hace que el apuntador recién seguido apunte a la celda anterior. Esos cambios se muestran en la figura 12.7(a), en el supuesto de que se siga al apuntador izquierdo. En esa figura, los apuntadores anteriores se representan con líneas de trazo continuo, y los nuevos, con líneas de puntos.
2. *Comutador.* Si se determina que las celdas siguientes a la actual ya se han revisado (por ejemplo, la celda actual puede tener sólo átomos, puede estar marcada, o se pudo haber «replegado» a la celda actual desde la apuntada por el campo *derecha* de la actual), se consulta el campo *atrás* de la celda anterior. Si ese valor es *I*, y el campo *derecha* de la celda previa contiene un apuntador no nulo a alguna celda *C*, se hace que *C* se convierta en la celda actual, y la identidad de la celda anterior no se cambia. Sin embargo, el valor de *atrás* en la celda anterior se hace *D*, y el apuntador izquierdo en esa celda recibe su valor correcto; esto es, se hace que apunte a la celda actual anterior. Para mantener el camino de regreso a *fuente* desde la celda anterior, se hace que el apuntador a *C* en el campo *derecha* apunte hacia donde apuntaba izquierdo. La figura 12.7(b) muestra esos cambios.
3. *Replegar.* Si se determina, como en (2), que las celdas que parten de la actual se han recorrido, pero el campo *atrás* de la celda anterior es *D*, o es *I*, pero el campo derecho contiene un átomo o un apuntador nulo, entonces se han revisado todas las celdas que parten de la anterior. Se efectúa el repliegue haciendo que la celda anterior sea la actual y que la celda siguiente en el camino de la celda anterior a *fuente* sea la nueva celda anterior. Esos cambios se muestran en la figura 12.7(c), en el supuesto de que *atrás = D* en la celda anterior.

Una coincidencia fortuita es que cada paso de la figura 12.7 se puede considerar como la rotación simultánea de tres apuntadores. Por ejemplo, en la figura 12.7(a), se reemplazan simultáneamente (*previo*, *actual*, *actual izquierda*) por (*actual*, *actual izquierda*, *previo*), respectivamente. Debe subrayarse la simultaneidad: la ubicación de *actual izquierda* no cambia al asignar un valor nuevo a *actual*. Para realizar esas modificaciones a los apuntadores es útil contar con un procedimiento *rotar*, que se muestra en la figura 12.8. Obsérvese en especial que el uso de parámetros por referencia asegura que las ubicaciones de los apuntadores se establezcan antes de cambiar ningún valor.

Ahora se considera el diseño del procedimiento no recursivo *bpfnr* para hacer el marcado. Este procedimiento es uno de esos raros procesos que son más fáciles de entender cuando se escriben con etiquetas y proposiciones *goto*. En especial, existen dos «estados» del procedimiento, «avance» representado por la etiqueta 1, y «repliegue», representado por la etiqueta 2. Se introduce inicialmente el primer estado, y también siempre que se pasa a una celda nueva, por un paso de avance o uno de

conmutación. En este estado, se intenta otro paso de avance, y sólo se efectúa un repliegue o una conmutación si existe un bloqueo, que se puede deber a dos razones: 1) la celda recién alcanzada ya está marcada, o 2) no hay apuntadores no nulos en la celda. Cuando hay un bloqueo, se cambia al segundo estado o «replegado».

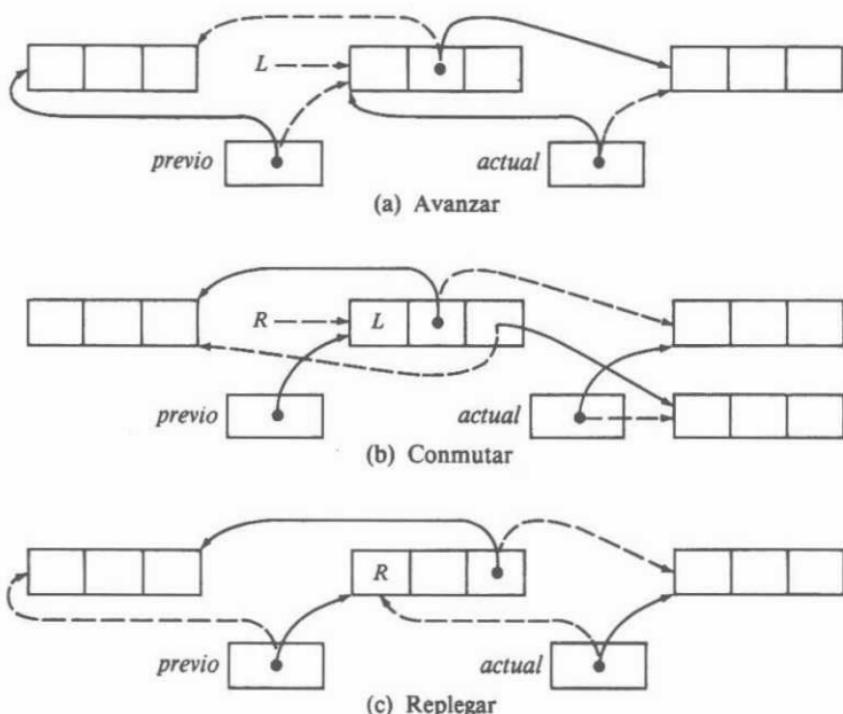


Fig. 12.7. Tres pasos básicos.

```
procedure rota ( var p1, p2, p3: ↑ tipo_celda );
  var
    temp: ↑ tipo_celda;
  begin
    temp := p1;
    p1 := p2;
    p2 := p3;
    p3 := temp
  end; { rota }
```

Fig. 12.8. Procedimiento de modificación de apuntadores.

El segundo estado se alcanzará cuando ocurra un repliegue o cuando no sea posible permanecer en estado de avance porque hay un bloqueo. En el estado de repliegue, se prueba si ya se ha replegado hasta la celda ficticia *fuentel*. Como se analizó antes, se reconocerá esta situación porque *previo = actual*, en cuyo caso se pasa al estado 3. De otra forma, se decide un repliegue para permanecer en ese estado o se pasa al estado de avance. El código de *bpfnr* se muestra en la figura 12.9; este código utiliza las funciones *bloque_izq*, *bloque_der* y *bloque*, que prueban si los campos izquierdo o derecho de una celda, o ambos, tienen un átomo o un apuntador nulo, *bloque* también prueba si hay una celda marcada.

```

function bloque_izquierdo ( celda: tipo_celda): boolean;
{ prueba si el campo izquierda es un átomo o un apuntador nulo }
begin
  with celda do
    if (patrón = PP) or (patrón = PA) then
      if izquierda <> nil then return (false);
    return (true)
  end; { bloque_izquierdo }

function bloque_derecho ( celda : tipo_celda ): boolean;
{ prueba si el campo derecha es un átomo o un apuntador nulo }
begin
  with celda do
    if (patrón = PP) or (patrón = AP) then
      if derecha <> nil then
        return (false);
    return (true)
  end; { bloque_derecho }

function bloque ( celda : tipo_celda ): boolean;
{ prueba si la celda está marcada o no contiene un apuntador no nulo s }
begin
  if (celda.marca = true) or bloque_izquierdo(celda) and bloque_derecho
    (celda) then
    return (true)
  else
    return (false)
end; { bloque }

procedure bpfnr; { marca las celdas accesibles desde fuente }
var
  actual, previo: †tipo_celda;
begin { asignación de valor inicial }
  actual := fuente1.derecha; { celda apuntada por fuente 1 }
  previo := fuente1; { previo apunta a fuente1 }
  fuente1.atrás := D;
  fuente1.derecha := fuente1; { fuente1 se apunta a sí mismo }
  fuente1.marca := true;

```

```

estado 1: { intenta avanzar }
    if bloque(actual↑) then begin { prepara el repliegue }
        actual↑.marca := true;
        goto estado2
    end
    else begin { marca y avanza }
        actual↑.marca := true;
        if bloque_izquierdo(actual↑) then begin { sigue al apuntador derecho }
            actual↑.atrás := D;
            rota(previo, actual, actual↑.derecha); {realiza los cambios
                de la figura 12.7(a), pero siguiendo al apuntador derecho }
            goto estado1
        end
        else begin { sigue al apuntador izquierdo }
            actual↑.atrás := I;
            rota(previo, actual, actual↑.izquierda);
            { realiza los cambios de la figura 12.7(a) }
            goto estado1
        end
    end
end;

estado2: { termina, repliega o conmuta }
    if previo = actual then { termina }
        goto estado3
    else if (previo↑.atrás = I) and
        not bloque_derecho(previo↑) then begin { conmuta }
        previo↑.atrás := D;
        rota(previo↑.izquierda, actual, previo↑.derecha);
        { realiza los cambios de la figura 12.7(b) }
        goto estado1
    end
    else if previo↑.atrás = D then { repliega }
        rota (previo, previo↑.derecha, actual)
        { realiza los cambios de la figura 12.7(c) }
    else { previo↑.atrás = I }
        rota(previo, previo↑.izquierda, actual);
        { realiza los cambios de la figura 12.7(c), pero con el campo
            izquierda de la celda previa comprendido en el camino }
    goto estado2
end;

estado3: { poner aquí el código para enlazar celdas no marcadas en la
            lista de espacio disponible }
end; { bpfnr }

```

Fig. 12.9. Algoritmo no recursivo para marcar celdas.

Algoritmo Deutsch-Schorr-Waite sin un bit extra para el campo *atrás*

Es posible, aunque poco probable, que el bit extra utilizado en las celdas por el campo *atrás* haga que las celdas necesiten un byte extra, o incluso de una palabra adicional. En tal caso, es bueno saber que en realidad no se necesita el bit extra, al menos si se programa en un lenguaje que, a diferencia de Pascal, permita utilizar los bits del campo *patrón* para propósitos distintos de los declarados: designadores del formato de registro variante. El «truco» consiste en observar que si se usa el campo *atrás*, como su celda está en el camino de vuelta a *fuente1*, los valores posibles del campo *patrón* están restringidos. Por ejemplo, si *atrás* = *I*, entonces se sabe que el *patrón* debe ser *PP* o *PA*, pues es obvio que el campo *izquierda* tiene un apuntador. Los mismo sucede cuando *atrás* = *D*. Así, si se dispone dos bits para representar *patrón* y (cuando sea necesario) *atrás*, se puede codificar la información necesaria como en la figura 12.10, por ejemplo.

Debe observarse que en el programa de la figura 12.9, siempre se sabe si se está usando *atrás*, para saber qué interpretación de la figura 12.10 es aplicable. Tan sólo cuando *actual* apunta a un registro, el campo *atrás* en ese registro no se usa; cuando *previo* lo apunta, entonces sí. Por supuesto, cuando se mueven esos apuntadores, es necesario ajustar los códigos; por ejemplo, si *actual* apunta a una celda con los bits 10, que se interpretan de acuerdo con la figura 12.10 como *patrón* = *AP*, y se decide avanzar, de modo que *previo* apuntará ahora a esta celda, se fija *atrás* = *D*, pues sólo el campo derecho contiene un apuntador, y los bits apropiados son 11. Obsérvese que si el *patrón* fuera *AA*, que no tiene representación en la columna central de la figura 12.10, no se querrá que *previo* apunte a la celda, pues no hay apuntadores que seguir en un movimiento de avance.

Código	En el camino hacia <i>fuente1</i>	Fuera del camino
00	<i>atrás</i> = <i>I</i> , <i>patrón</i> = <i>PP</i>	<i>patrón</i> = <i>PP</i>
01	<i>atrás</i> = <i>I</i> , <i>patrón</i> = <i>PA</i>	<i>patrón</i> = <i>PA</i>
10	<i>atrás</i> = <i>D</i> , <i>patrón</i> = <i>PP</i>	<i>patrón</i> = <i>AP</i>
11	<i>atrás</i> = <i>D</i> , <i>patrón</i> = <i>AP</i>	<i>patrón</i> = <i>AA</i>

Fig. 12.10. Interpretación de dos bits como *patrón* y *atrás*.

12.4 Asignación de almacenamiento para objetos de diferentes tamaños

Considérese ahora el manejo de una estructura dinámica, como lo presenta la figura 12.1, donde hay una colección de apuntadores a bloques asignados. Los bloques contienen datos de algún tipo. En la figura 12.1, por ejemplo, los datos son cadenas de caracteres. Aunque el tipo de los datos almacenados en la estructura dinámica no tiene por qué ser cadenas de caracteres, se supone que los datos no tienen apuntadores a localidades de la estructura dinámica.

El problema de la administración de estructuras dinámicas tiene aspectos que facilitan y también dificultan su mantenimiento en comparación con las estructuras

de listas de celdas de igual tamaño, tratadas en la sección anterior. El factor principal que facilita el problema es que el marcado de bloques usados no es un proceso recursivo; sólo se tienen que seguir los apuntadores externos de la estructura dinámica y marcar los bloques apuntados. No es necesaria una búsqueda en profundidad de una estructura enlazada ni de un algoritmo como el Deutsch-Schorr-Waite.

Por otro lado, la administración de la lista de espacio disponible no es tan simple como en la sección 12.3. Podría imaginarse que las regiones vacías [por ejemplo, en la Fig. 12.1(a) hay tres regiones vacías] están enlazadas como se sugiere en la figura 12.11. Ahí se observa una estructura dinámica de 3000 palabras dividida en cinco bloques. Dos bloques de 200 y 600 palabras, respectivamente, contienen los valores de X e Y . Los tres bloques restantes están vacíos, y están enlazados en una cadena que parte de *dispo*, el encabezado para el espacio disponible.

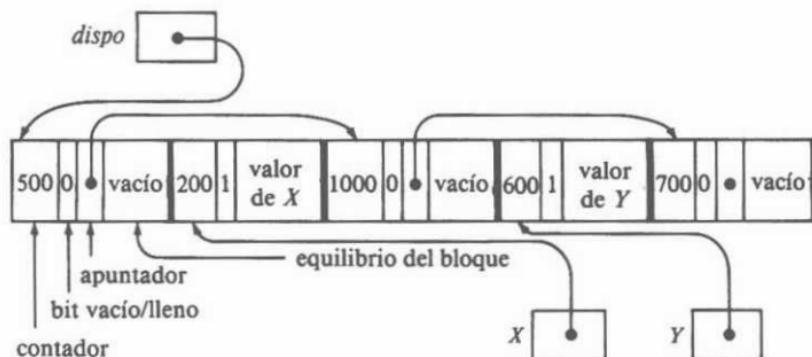


Fig. 12.11. Una estructura dinámica con lista de espacio disponible.

Para que los bloques vacíos se puedan encontrar siempre que se requiera almacenar datos nuevos, y pueda disponerse de los que contienen datos inútiles, en esta sección se hacen las siguientes suposiciones.

1. Cada bloque es bastante grande para contener
 - a) un *contador* que indique el tamaño (en bytes o palabras, de acuerdo con el computador) del bloque,
 - b) un apuntador (para enlazar el bloque al espacio disponible), más
 - c) un bit para indicar si el bloque está vacío o no; este bit se conoce como *bit vacío / lleno* o *usado / no usado*.
2. Un bloque vacío tiene, desde la izquierda (dirección más baja), un contador que indica su longitud, un bit vacío/lleno con un valor de cero, que indica que el bloque está vacío, un apuntador al siguiente bloque disponible, y el espacio libre.
3. Un bloque que contiene datos tiene, desde la izquierda, un contador, un bit vacío / lleno con valor 1 indicando que el bloque está en uso, y los datos †.

† Obsérvese que en la figura 12.1, en vez de un contador que indica la longitud del bloque, se emplea la longitud de los datos.

Una consecuencia interesante de las suposiciones anteriores es que los bloques deben ser capaces de almacenar datos, algunas veces (cuando están en uso), y apun-tadores, en otras (cuando están en desuso), exactamente en el mismo lugar, con lo que es imposible o muy incómodo escribir programas que manipulen bloques de esta clase en Pascal o cualquier otro lenguaje fuertemente orientado a tipos. Así, esta sección debe ser discursiva por necesidad; sólo pueden escribirse programas en seu-do-Pascal, nunca programas reales en Pascal. Sin embargo, no hay ningún problema para escribir programas que hagan las cosas descritas en lenguaje ensamblador o en la mayoría de los lenguajes de programación de sistemas, como C.

Fragmentación y compactación de bloques vacíos

Para ver uno de los problemas especiales que se presentan en el manejo de estruc-turas dinámicas, supóngase que la variable Y de la figura 12.11 cambia, de forma que el bloque que representa a Y debe devolverse al espacio disponible. Es más fácil insertar el bloque nuevo al principio de la lista de disponibles, como se sugiere en la figura 12.12. En esa figura se observa un caso de *fragmentación*, la tendencia a representar grandes áreas vacías en la lista de espacio disponible por medio de «frag-mentos», esto es, varios bloques pequeños formando el total. En el caso en cuestión, los últimos 2300 bytes de la estructura dinámica de la figura 12.12 están vacíos, pero el espacio está dividido en tres bloques de 1000, 600 y 700 bytes, y esos bloques no están siquiera, en orden consecutivo en la lista de disponibles. Sin alguna forma de recolección de basura, sería imposible satisfacer una petición de, por ejemplo, un blo-que de 2000 bytes.

Es obvio que cuando se devuelve un bloque a la lista de disponibles es conve-niente observar los bloques inmediatamente a la izquierda y a la derecha del bloque que se está haciendo disponible. Encontrar el bloque de la derecha es sencillo; si el blo-que que se está recuperando empieza en la posición p y el contador vale c , el blo-que de la derecha empieza en la posición $p + c$. Si se conoce p (por ejemplo, el apun-tador Y de la Fig. 12.11 contiene el valor p del bloque hecho disponible en la Fig. 12.12), basta con leer los bytes que empiezan en la posición p , tantos como se re-quieran para contener c , y obtener el valor c . Del byte $p + c$, se salta el campo del

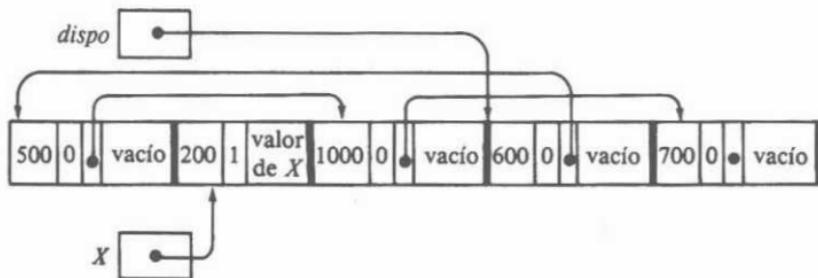


Fig. 12.12. Después de devolver el bloque de Y .

contador para encontrar el bit que indica si el bloque está vacío o no. Si está vacío, los bloques que empiezan en p y en $p + c$ pueden combinarse.

Ejemplo 12.4. Supóngase que la estructura dinámica de la figura 12.11 empieza en la posición 0. Entonces, el bloque de Y que se está devolviendo empieza en el byte 1700, así que $p = 1700$ y $c = 600$. El bloque que empieza en $p + c = 2300$ también está vacío, por lo que se pueden combinar en un solo bloque que empieza en 1700 con un contador de 1300, la suma de los contadores de los dos bloques. \square

Sin embargo, no es tan fácil fijar la lista de disponibles después de combinar los bloques. Se puede crear el bloque combinado agregando tan sólo el contador del segundo bloque a c . Sin embargo, el segundo bloque permanecerá enlazado en la lista de disponibles y deberá separarse, para lo cual es necesario encontrar el apuntador a ese bloque desde su predecesor en la lista de disponibles. Se presentan varias estrategias, pero ninguna se recomienda en especial.

1. Recorrer la lista hasta encontrar un apuntador con valor $p + c$. Este apuntador debe estar en el bloque anterior al que se ha combinado con su vecino. Reemplazar el apuntador encontrado por el apuntador del bloque de $p + c$. Esto, en efecto, elimina el bloque que empieza en la posición $p + c$ de la lista de disponibles. Su espacio queda disponible, por supuesto; es parte del bloque que empieza en p . En promedio, habrá que revisar la mitad de la lista de disponibles, así que el tiempo requerido es proporcional a esa longitud.
2. Usar una lista doblemente enlazada para el espacio disponible. Entonces, el bloque predecesor puede encontrarse con rapidez y el bloque en $p + c$ se puede eliminar de la lista. Este enfoque requiere un tiempo constante, independiente de la longitud de la lista de disponibles, pero requiere espacio adicional para otro apuntador en cada bloque vacío, incrementando así el tamaño mínimo de un bloque para contener un contador, un bit vacío/lleto, y dos apuntadores.
3. Conservar clasificada toda la lista de espacio disponible de acuerdo con la posición. Entonces, se sabe que el bloque de la posición p es el predecesor en la lista del bloque en $p + c$, y la manipulación del apuntador necesaria para eliminar el segundo bloque puede efectuarse en un tiempo constante. Sin embargo, la inserción de un bloque disponible nuevo requiere revisar en promedio la mitad de la lista, por lo que no es más eficiente que el método (1).

De los tres métodos, el primero y el tercero requieren un tiempo proporcional a la longitud de la lista de disponibles para devolver un bloque al espacio disponible y combinarlo con su vecino derecho, si está vacío. Este tiempo puede ser prohibitivo o no, dependiendo de lo larga que sea la lista y del tiempo total del programa se consume en la manipulación de la estructura dinámica. El segundo método —doble enlace para la lista de disponibles— tiene sólo el defecto de incrementar el tamaño mínimo de los bloques. Lamentablemente, cuando se considera cómo combinar un bloque devuelto con su vecino de la izquierda, como en los otros métodos, se observa que el doble enlace no ayuda a encontrar vecinos izquierdos en menos tiempo que el requerido para recorrer la lista de disponibles.

Encontrar un bloque inmediatamente a la izquierda del actual no es fácil. La po-

sición p de un bloque y de su contador c determinan la posición del bloque de la derecha, pero esto no sirve como guía para encontrar el principio del bloque izquierdo. Es necesario encontrar un bloque vacío que empiece en alguna posición p_1 y tenga un contador c_1 tal que $p_1 + c_1 = p$. Para esto hay tres posibles estrategias.

1. Revisar la lista de disponibles buscando un bloque en la posición p_1 y con contador c_1 donde $p_1 + c_1 = p$. Esta operación lleva un tiempo proporcional a la longitud de la lista de disponibles.
2. Conservar un apuntador en cada bloque (usado o no) indicando la posición del bloque de la izquierda. Este enfoque permite encontrar el bloque izquierdo en un tiempo constante; se prueba si está vacío, en cuyo caso se combina con el bloque en cuestión. Se puede encontrar el bloque en la posición $p + c$ y hacer que apunte al principio del bloque nuevo, para poder mantener esos apuntadores a la izquierda †.
3. Mantener clasificada la lista de disponibles de acuerdo con la posición. Entonces el bloque vacío de la izquierda se encuentra al insertar en la lista el bloque vaciado recientemente, y sólo es necesario revisar, usando la posición y el contador del bloque vacío anterior, que no se interponga ningún bloque no vacío.

Al igual que con la intercalación de bloques vacíos recientes con el bloque de la derecha, el primero y tercer enfoques para la búsqueda e intercalación con el bloque de la izquierda requieren un tiempo proporcional a la longitud de la lista de disponibles. El método (2) también requiere tiempo constante, pero tiene una desventaja sobre los problemas relativos a la lista de disponibles doblemente enlazada (ya sugerida en relación con la búsqueda de los bloques vecinos de la derecha). Mientras que el doble enlace de los bloques vacíos aumenta el tamaño mínimo de los bloques, no puede decirse que este enfoque desperdicie espacio, ya que sólo los bloques que no se usan para almacenar datos son los que se enlazan. Sin embargo, apuntar a los vecinos izquierdos requiere un apuntador en los bloques utilizados y en los no utilizados, y con justicia puede acusársele de consumir espacio. Si el tamaño promedio de los bloques es de cientos de bytes, el espacio extra para un apuntador puede ser despreciable. Por otro lado, el espacio adicional puede ser prohibitivo si el bloque típico tiene sólo 10 bytes de longitud.

Para resumir las implicaciones de estas exploraciones en cuanto a cómo combinar bloques vacíos recientes con vecinos vacíos, hay tres enfoques para tratar la fragmentación.

1. Usar uno de varios enfoques, como puede ser conservar clasificada la lista de disponibles, que requiere un tiempo proporcional a la longitud de la lista cada vez que un bloque queda sin utilizar, pero permite encontrar y combinar vecinos vacíos.
2. Usar una lista de espacio disponible doblemente enlazada cada vez que un bloque quede sin utilizar, y también utilizar apuntadores a los vecinos izquierdos de todos los bloques, estén o no disponibles, para combinar vecinos vacíos en un tiempo constante.

† Como ejercicio, se debe descubrir cómo mantener los apuntadores cuando un bloque se divide en dos; se toma una parte para un elemento de datos nuevo, mientras que la otra permanece vacía.

3. No hacer nada explícito para combinar vecinos vacíos. Cuando no sea posible encontrar un bloque tan grande como para contener datos nuevos, revisar los bloques de izquierda a derecha, combinando vecinos vacíos y después crear una lista nueva de disponibles. Un bosquejo del programa que hace esto se muestra en la figura 12.13.

```

(1)  procedure combina;
var
(2)    p, q: apuntadores a bloques;
        { p indica el extremo izquierdo del bloque vacío que se está
          acumulando; q indica un bloque a la derecha de p que se
          incorporará en el bloque p si está vacío }
begin
(3)    p:= bloque más a la izquierda de la estructura dinámica;
(4)    vaciar la lista de disponibles;
(5)    while p < extremo derecho de la estructura dinámica do
(6)      if p apunta a un bloque lleno con contador c then
(7)        p := p + c; { saltar los bloques llenos }
(8)      else begin { p apunta al principio de una secuencia de
        bloques vacíos; deben combinarse }
(9)        q := p + c; { asignar el valor inicial q al siguiente bloque }
(10)       while q apunta a un bloque vacío con un contador, d, y q <
            extremo derecho de la estructura dinámica do begin
            agregar d al contador del bloque apuntado por p;
            q := q + d
        end
(13)       inserta el bloque apuntado por p en la lista de disponibles;
(14)       p := q;
    end
end; { combina }

```

Fig. 12.13. Combinación de bloques vacíos adyacentes.

Ejemplo 12.5. Como ejemplo, considérese el programa de la figura 12.13, aplicado a la estructura dinámica de la figura 12.12. Supóngase que el byte de más a la izquierda de la estructura dinámica es cero, así que inicialmente, $p = 0$. Como $c = 500$ para el primer bloque, a q se le asigna un valor inicial $p + c = 500$. Cuando el bloque que empieza en 500 está lleno, el ciclo de las líneas (10) a (12) no se ejecuta y el bloque que consta de los bytes 0 a 499 se coloca en la lista de disponibles, haciendo que $dispo$ apunte al byte 0 y poniendo un apuntador nil en el lugar designado en ese bloque (después del contador y el bit vacío/lleno). Después, se asigna a p el valor 500 en la línea (14), y se incrementa a 700 en la línea (7). El apuntador q toma el valor 1700 en la línea (9), y después, 2300 y 3000 en la línea (12), al tiempo que 600 y 700 se agregan al contador 1000 en el bloque que empieza en 700. Cuando q excede del byte de más a la derecha, 2999, el bloque que empieza en 700, que ahora tiene

el contador 2300, se inserta en la lista de disponibles. Entonces, en la línea (14), p se ajusta en 3000 y el ciclo externo finaliza en la línea (5). \square

El número total de bloques y el número de bloques disponibles puede no ser muy diferente, y probablemente la frecuencia con que se puede encontrar que no hay bloques vacíos bastante grandes sea baja, se cree que el método (3), combinar bloques vacíos adyacentes sólo cuando se termine el espacio adecuado, es superior a (1) en una situación real. El método (2) es un posible competidor, pero considera los requisitos de espacio adicional y el hecho de que se requiere un tiempo extra cada vez que se inserta o elimina un bloque de la lista de disponibles, se cree que (2) será preferible a (3) en circunstancias muy raras, y tal vez se pueda olvidar.

Selección de bloques disponibles

Se ha tratado con detalle qué debe suceder cuando deja de necesitarse un bloque y se puede devolver al espacio disponible. Existe también el proceso inverso de proporcionar bloques para contener nuevos datos. Es evidente que se debe seleccionar algún bloque disponible y usarlo parcial o totalmente para contener los datos nuevos. Hay dos temas a tener en cuenta. Primero, ¿qué bloque vacío se selecciona? Segundo, si es necesario usar sólo parte de un bloque seleccionado, ¿qué parte se usa?

La segunda pregunta es fácil de aclarar. Si se utiliza un bloque con contador c y se requieren $d < c$ bytes de ese bloque, se escogen los últimos d bytes. De esta forma, sólo es necesario reemplazar c por $c-d$, y el bloque vacío restante puede permanecer en la lista de disponibles \dagger . \square

Ejemplo 12.6. Supóngase que se requieren 400 bytes para la variable W en la situación representada en la figura 12.12. Podría decidirse tomar los 400 bytes finales de los 600 que existen en el primer bloque de la lista de disponibles. Tal situación se muestra en la figura 12.14.

La selección de un bloque para colocar datos nuevos no es fácil, debido a que existen conflictos entre las metas de tales estrategias. Se desea, por un lado, poder tomar con rapidez un bloque vacío en el cual quepan los datos y, por otro, hacer una selección de un bloque vacío que reduzca la fragmentación. Hay dos estrategias que representan extremos en el espectro conocidas como «primer ajuste» y «mejor ajuste», y se describen a continuación.

1. *Primer ajuste.* Para seguir la estrategia de *primer ajuste*, cuando se necesita un bloque de tamaño d , se revisa la lista de disponibles desde el principio hasta llegar a un bloque de tamaño $c \geq d$. Utilíicense las últimas d palabras en ese bloque, como se describió antes.
2. *Mejor ajuste.* Para seguir la estrategia de *mejor ajuste*, cuando se necesita un bloque de tamaño d , se examina toda la lista de disponibles para encontrar el bloque de tamaño mínimo d que sea lo más cercano posible a d . Se toman las últimas d palabras de ese bloque.

\dagger Si $c - d$ es tan pequeño que un contador y un apuntador no caben, hay que utilizar el bloque completo y eliminarlo de la lista de disponibles.

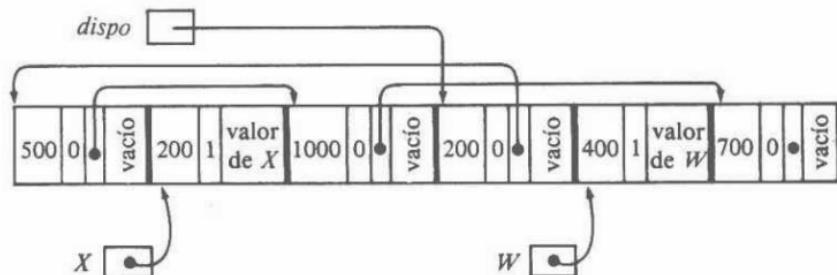


Fig. 12.14. Configuración de la memoria.

Pueden hacerse algunas observaciones acerca de estas estrategias. La estrategia de mejor ajuste es bastante más lenta que la de primer ajuste, ya que con esta última se puede esperar encontrar, en promedio, un bloque suficiente con rapidez, mientras que con la de mejor ajuste es necesario recorrer toda la lista de disponibles. La estrategia de mejor ajuste puede acelerarse si se mantienen listas de bloques disponibles de acuerdo con varias gamas de tamaños. Por ejemplo, se puede mantener una lista de bloques disponibles con longitud entre 1 y 16 bytes †, entre 17 y 32, entre 33 y 64, y así sucesivamente. Esta «mejora» no beneficia en forma apreciable la estrategia de primer ajuste, y, de hecho, puede alestarla si las estadísticas de los tamaños de bloques son malas. (Compárese la búsqueda del primer bloque de tamaño mayor que 32 en la lista de disponibles completa y en la lista de bloques de tamaño 17 a 32, por ejemplo.) Una última observación es que un espectro de estrategias se puede definir entre las dos planteadas aquí, buscando el mejor ajuste entre los primeros k bloques disponibles para algún tamaño fijo k .

La estrategia de mejor ajuste parece reducir la fragmentación en comparación con la de primer ajuste, en el sentido de que la primera tiende a producir «fragmentos» muy pequeños, es decir, bloques abandonados. Y aunque el número de esos fragmentos es casi el mismo que para el primer ajuste, tienden a ocupar un área menor. Sin embargo, el mejor ajuste no tiende a producir «fragmentos» de tamaño medio. En cambio, los bloques disponibles tienden a ser fragmentos muy pequeños o bloques devueltos al espacio disponible. Como consecuencia, hay secuencias de peticiones que la estrategia de primer ajuste puede satisfacer y la de mejor ajuste no, y viceversa.

Ejemplo 12.7. Supóngase, como en la figura 12.12, que la lista de disponibles consta de los bloques de tamaño 600, 500, 1000 y 700, en ese orden. Si se emplea la estrategia de primer ajuste y se hace una petición de un bloque de tamaño 400, se obtiene el bloque de tamaño 600 que es el primero de la lista en el cual cabe un bloque de tamaño 400. La lista de disponibles tiene ahora bloques de tamaño 200, 500, 1000 y 700. Así que es imposible satisfacer de inmediato tres peticiones de bloques de tamaño 600 (aunque se puede hacer después de combinar bloques vacíos adyacentes o al recorrer bloques utilizados sobre la estructura dinámica).

† En realidad, hay un tamaño de bloque mínimo mayor que 1, puesto que los bloques deben contener un apuntador, un contador y un bit vacío/lleno si se van a encadenar a una lista de disponibles.

No obstante, si se aplicara la estrategia de mejor ajuste con la lista de disponibles 600, 500, 1000 y 700, y llegara la petición de 400, podría colocarse donde ajusta mejor, esto es, en el bloque de 500, dejando una lista de bloques disponibles de 600, 100, 1000 y 700. En este caso, se pueden satisfacer tres peticiones de bloques de tamaño 600 sin necesidad de recurrir a la reorganización del almacenamiento.

Por otro lado, hay situaciones donde, al empezar con la lista 600, 500, 1000, 700 de nuevo, la estrategia de mejor ajuste puede fallar, mientras que la de primer ajuste puede funcionar sin necesidad de reorganización del almacenamiento. Dada la petición de 400 bytes, el mejor ajuste puede, igual que antes, dejar la lista en 600, 100, 1000 y 700, mientras que el primer ajuste la deja en 200, 500, 1000 y 700. Supóngase que las dos peticiones siguientes son de 1000 y 700, de manera que cualquier estrategia asignaría completamente los dos últimos bloques vacíos, dejando 600 y 100 en el caso del mejor ajuste, y 200 y 500, en el caso del primer ajuste. Ahora bien, el primer ajuste puede satisfacer peticiones de bloques de tamaño 200 y 500, mientras que el mejor ajuste obviamente no podrá. □

12.5 Sistemas de manejo de memoria por afinidades (*buddy systems*)

Hay una familia de estrategias para mantener una estructura dinámica que evite parcialmente los problemas de fragmentación y distribución difícil de bloques vacíos. Esas estrategias, llamadas «sistemas de manejo de memoria por afinidades», consumen poco tiempo al combinar bloques vacíos adyacentes. El inconveniente es que los bloques llegan en un surtido limitado de tamaños, así que se puede desperdiciar algún espacio colocando datos en un bloque más grande de lo necesario.

La idea central de todos los sistemas por afinidades es que los bloques son sólo de ciertos tamaños; por ejemplo, $s_1 < s_2 < s_3 < \dots < s_k$ son todos los tamaños entre los cuales pueden encontrarse los bloques. Algunas selecciones comunes para la secuencia s_1, s_2, \dots son 1, 2, 4, 8, ... (el sistema de afinidades exponencial) y 1, 2, 3, 5, 8, 13, ... (el sistema de afinidades de Fibonacci, donde $s_{i+1} = s_i + s_{i-1}$). Todos los bloques vacíos de tamaño s_i están enlazados en una lista, y hay un arreglo de encabezados de lista de disponibles, una para cada tamaño s_i permitido †. Si se requiere un bloque de tamaño d para un grupo de datos nuevo, se escoge un bloque disponible de tamaño s_i tal que $s_i \geq d$, pero $s_{i-1} < d$, esto es, el tamaño más pequeño permitido en el cual caben los datos nuevos.

Surgen algunas dificultades cuando no existen bloques vacíos del tamaño deseado s_i . En ese caso, es necesario encontrar un bloque de tamaño s_{i+1} y dividirlo en dos, uno de tamaño s_i y el otro de tamaño $s_{i+1} - s_i$ ††. El sistema de afinidades impone la restricción de que $s_{i+1} - s_i$ sea alguna s_j , para $j \leq i$. Ahora se ve la forma en

† Puesto que los bloques vacíos deben contener apuntadores (y, como se verá, otra información también), en realidad no se inicia la secuencia de tamaños permitidos en 1, sino en algún número apropiado más grande en la secuencia, como 8 bytes.

†† Por supuesto, si no hay bloques vacíos de tamaño s_{i+1} , se crea uno dividiendo un bloque de tamaño s_{i+2} , y así sucesivamente. Si no existen bloques de ningún tamaño mayor, en realidad no hay espacio y es necesario reorganizar la estructura dinámica como en la sección siguiente.

que se puede restringir la selección de valores para las s_i . Si se hace que $j = i - k$, para alguna $k \geq 0$, dado que $s_{i+1} - s_i = s_{i-k}$ se sigue que

$$s_{i+1} = s_i + s_{i-k} \quad (12.1)$$

La ecuación (12.1) es aplicable cuando $i > k$, y junto con los valores para s_1, s_2, \dots, s_k , determina completamente a s_{k+1}, s_{k+2}, \dots . Por ejemplo, si $k = 0$, (12.1) queda como

$$s_{i+1} = 2s_i \quad (12.2)$$

empezando con $s_1 = 1$ en (12.2), se obtiene la secuencia exponencial 1, 2, 4, 8, ... Por supuesto, no importa el valor inicial de s_1 ; las s crecerán exponencialmente en (12.2). Otro ejemplo, si $k = 1$, $s_1 = 1$ y $s_2 = 2$, (12.1) se transforma en

$$s_{i+1} = s_i + s_{i-1} \quad (12.3)$$

esta ecuación define la sucesión de Fibonacci: 1, 2, 3, 5, 8, 13, ...

Para cualquier valor de k escogido en (12.1), habrá un *sistema de afinidades de orden k-ésimo*. Para cualquier k , la secuencia de tamaños permitidos crece exponencialmente; esto es, la razón s_{i+1}/s_i se aproxima a alguna constante mayor que uno. Por ejemplo, para $k = 0$, s_{i+1}/s_i es exactamente 2. Para $k = 1$ la razón se aproxima a la «razón dorada» $(\sqrt{5}+1)/2 = 1.618$, y la razón decrece conforme k crece, pero nunca es menor que 1.

Distribución de bloques

En el sistema de afinidades de orden k -ésimo, cada bloque de tamaño s_{i+1} se puede considerar formado por un bloque de tamaño s_i y otro de tamaño s_{i-k} . Para ser más específicos, supóngase que el bloque de tamaño s_i está a la izquierda (en las posiciones numeradas menores) del bloque de tamaño s_{i-k} †. Si se considera la estructura dinámica como un solo bloque de tamaño s_n , para alguna n grande, entonces las posiciones en las que los bloques de tamaño s_i pueden empezar están completamente determinadas.

Las posiciones en el sistema exponencial, o de orden 0-ésimo, son fáciles de determinar. Si se supone que las posiciones en la estructura están numeradas a partir de 0, un bloque de tamaño s_i empieza en cualquier posición que comience con un múltiplo de 2^i , esto es, 0, $2^i, \dots$. Más aún, cada bloque de tamaño 2^{i+1} , que empieza, por ejemplo, en $j2^{i+1}$ está compuesto de dos «grupos afines» de tamaño 2^i , que empiezan en las posiciones $(2j)2^i$, que son $j2^{i+1}$, y $(2j+1)2^i$. Así, es fácil encontrar el grupo afín de un bloque de tamaño 2^i . Si empieza en algún múltiplo par de 2^i , como $(2j)2^i$, su afín está a la derecha, en la posición $(2j+1)2^i$. Si empieza en un múltiplo impar de 2^i , por ejemplo, en $(2j+1)2^i$, su afín está a la izquierda, en $(2j)2^i$.

Ejemplo 12.8. No es simple el comportamiento de los sistemas de afinidades de orden mayor que cero. La figura 12.15 muestra el sistema de afinidades de Fibonacci

† A veces es conveniente considerar que los bloques de tamaño s_i, s_{i-k} forman un bloque de tamaño s_{i+1} como «afines»; de ahí el término «sistema de afinidades».

utilizado en una estructura dinámica de tamaño 55, con bloques de tamaños s_i , s_2, \dots , $s_8 = 2, 3, 5, 8, 13, 21, 34$ y 55. Por ejemplo, el bloque de tamaño 3 que empieza en 26 es afín al bloque de tamaño 5 que empieza en 21; juntos, forman el bloque de tamaño 8 que empieza en 21 y que es afín al bloque de tamaño 5 que empieza en 29. Juntos, estos conforman el bloque de tamaño 13 que empieza en 21, y así sucesivamente. □

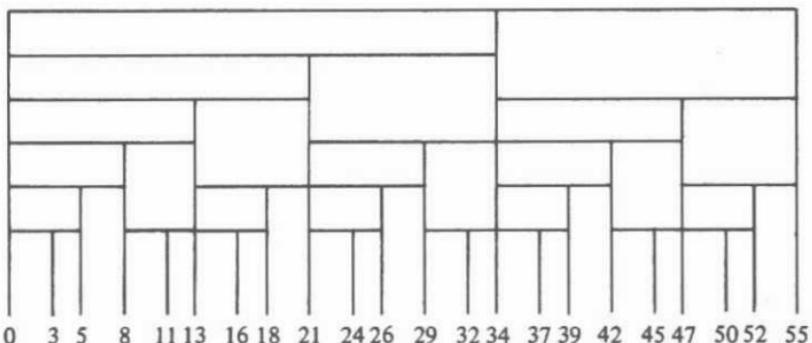


Fig. 12.15. División de una estructura dinámica según el sistema de afinidades de Fibonacci.

Asignación de bloques

Si se requiere un bloque de tamaño n , se escoge cualquiera de la lista de disponibles de tamaño s_i , donde $s_i \geq n$ e $i = 1$ o $s_{i-1} < n$, esto es, el bloque de mejor ajuste. En un sistema de afinidades de orden k -ésimo, si no hay bloques de tamaño s_i , es posible elegir un bloque de tamaño s_{i+1} o s_{i+k+1} para dividirlo, obteniendo en cualquier caso algún bloque de tamaño s_i . Si no existen bloques de esos tamaños, se crea uno aplicando la estrategia de división recursivamente para el tamaño s_{i+1} .

Sin embargo, hay una pequeña trampa. En un sistema de orden k -ésimo, puede ser que no se dividan los bloques de tamaño s_1, s_2, \dots, s_k , ya que de eso puede resultar un bloque de tamaño menor que s_1 . Se debe usar dicho bloque completo si no se encuentra disponible otro menor. Este problema no surge si $k = 0$, esto es, en el sistema exponencial. Puede reducirse en el sistema de Fibonacci si se empieza con $s = 1$, pero esa selección quizás no sea aceptable, puesto que los bloques de tamaño uno (byte o palabra, tal vez), pueden ser muy pequeños para contener un apuntador y un bit vacío/lleno.

Devolución de bloques al espacio disponible

Cuando un bloque queda listo para reutilizarse, se puede ver una de las ventajas de los sistemas de afinidades. Algunas veces se puede reducir la fragmentación combinando el bloque que quedó disponible recientemente con su afín, si éste está dispo-

nible también †. De hecho, si fuera el caso, se puede combinar el bloque resultante con su afín, si está vacío, y así sucesivamente. La combinación de grupos afines vacíos requiere sólo una cantidad constante de tiempo, lo que la convierte en una alternativa atractiva cuando se presentan combinaciones periódicas de bloques vacíos adyacentes, como se sugirió en la sección anterior, que requiere un tiempo proporcional al número de bloques vacíos.

El sistema exponencial hace que la localización de grupos afines sea especialmente fácil. Si se acaba de devolver el bloque de tamaño 2^i que empieza en $p2^i$, su afín está en $(p+1)2^i$ si p es par, y en $(p-1)2^i$ si p es impar.

Para un sistema de orden $k \geq 1$, la localización de afines no es tan simple. Para hacerla más fácil es necesario almacenar cierta información en cada bloque.

1. Un bit vacío/lleno, como tienen todos los bloques.
2. El *índice de tamaño*, un entero i tal que el bloque sea de tamaño s_i .
3. El *contador de afinidad izquierdo*, descrito a continuación.

En cada par de afines, uno (el *afín izquierdo*) está a la izquierda del otro (el *afín derecho*). Intuitivamente, el contador de afinidad izquierdo de un bloque dice cuántas veces consecutivas éste es todo o parte de un afín izquierdo. En un aspecto formal, toda la estructura dinámica, tratada como un bloque de tamaño s_n , tiene un contador de afinidad izquierdo igual a 0. Cuando se divide cualquier bloque de tamaño s_{i+1} , con el contador de afinidad izquierdo b , en bloques de tamaño s_i y s_{i-k} , que son los afines izquierdo y derecho, respectivamente, el afín izquierdo tiene un contador $b+1$, mientras que el derecho tiene un contador igual a 0, independiente de b . Por ejemplo, en la figura 12.15, el bloque de tamaño 3 que empieza en 0 tiene un contador de afinidad izquierdo igual a 6, y el bloque de tamaño 3 que empieza en 13 tiene un contador izquierdo igual a 2.

Además de la información anterior, los bloques vacíos, pero no los utilizados, tienen apunadores de avance y de retroceso para la lista de disponibles del tamaño adecuado. Los apunadores bidireccionales facilitan las combinaciones de afines, que deben eliminarse de la lista de disponibles.

La forma en que se maneja esta información es como sigue. Supóngase que k es el orden del sistema de afinidades. Cualquier bloque que empiece en la posición p con un contador de afinidad izquierdo igual a 0 es un afín derecho. Así, si tiene índice de tamaño j , su afín izquierdo es de tamaño s_{j+k} y empieza en la posición $p-s_{j+k}$. Si el contador de afinidad izquierdo es mayor que 0, entonces el bloque es un afín izquierdo de un bloque de tamaño s_{j-k} , el cual está localizado en la posición $p+s_j$.

Si se combina un afín izquierdo de tamaño s_i , teniendo un contador de afinidad izquierdo igual a b , con un afín derecho de tamaño s_{i-k} , el bloque resultante tendrá un índice de tamaño $i+1$, que empieza en la misma posición que el bloque de tamaño s_i , y tiene un contador de afinidad izquierdo $b-1$. Así, la información necesaria puede mantenerse con facilidad al combinar dos afines vacíos. Como ejercicio, se puede comprobar que la información se mantiene al dividir un bloque vacío de tamaño s_{i+1} en un bloque ocupado de tamaño s_i y uno vacío de tamaño s_{i-k} .

† Igual que en la sección anterior, se supone que un bit de cada bloque está reservado para indicar si el bloque está en uso o vacío.

Si se mantiene toda esta información, y se ligan las listas de disponibles en ambas direcciones, sólo se emplea una cantidad constante de tiempo en cada división de un bloque o se combinan los afines en un bloque más grande. Como el número de combinaciones nunca puede exceder del número de divisiones, la cantidad total de trabajo es proporcional a este número. No es difícil reconocer que la mayor parte de las solicitudes de un bloque no requieren divisiones, ya que hay disponible un bloque del tamaño correcto. Sin embargo, hay situaciones extremas en las que cada asignación requiere bastantes divisiones. El ejemplo extremo es donde se solicita repetidamente un bloque del tamaño más pequeño y después se devuelve. Si hay n tamaños diferentes, se requieren por lo menos n/k divisiones en un sistema de orden k -ésimo, las cuales estarán seguidas de n/k combinaciones cuando se devuelve el bloque.

12.6 Compactación del almacenamiento

Hay ocasiones en las que, aun después de combinar todos los bloques adyacentes, no es posible satisfacer una solicitud de un bloque nuevo. Esto puede deberse simplemente a que en el montón no hay espacio para formar un bloque del tamaño deseado. Pero es más típica una situación como la que se muestra en la figura 12.11, donde si bien hay 2200 bytes disponibles, no se puede satisfacer una solicitud de un bloque mayor de 1000. El problema es que el espacio disponible está dividido entre varios bloques no contiguos. Hay dos enfoques generales a este problema.

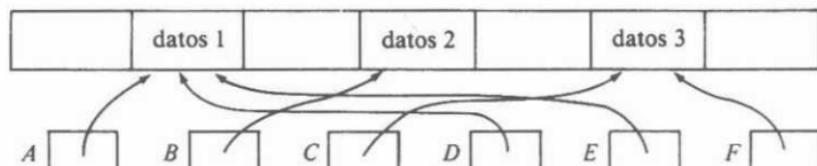
1. Lograr que el espacio disponible para el conjunto de datos pueda estar compuesto de varios bloques vacíos. Si ocurre así, también se puede requerir que todos los bloques sean del mismo tamaño y consistan en espacio para un apuntador y para los datos. En un bloque utilizado, el apuntador indica el siguiente bloque usado para los datos, y es nulo en el último bloque. Por ejemplo, si se estuvieran almacenando datos cuyo tamaño fuera casi siempre pequeño, se escogerían bloques de tamaño igual a 16 bytes, con 4 utilizados para un apuntador y 12 para los datos. Si los conjuntos de datos fueran en general más grandes, podrían escogerse bloques con varios cientos de bytes, asignando de nuevo cuatro para un apuntador y el resto para los datos.
2. Cuando falla la combinación de bloques vacíos adyacentes, y no se es capaz de proveer un bloque bastante grande, se mueven los datos por la estructura dinámica de manera que todos los bloques llenos queden en el extremo izquierdo (posición más baja), y se forme un bloque grande disponible a la derecha.

El método (1), que usa cadenas de bloques para un conjunto de datos, tiende a consumir mucho espacio. Si se escoge un tamaño de bloque pequeño, se emplea una fracción grande de espacio para «las cabeceras», los apuntadores necesarios para mantener las cadenas. Al utilizar bloques grandes, se ocupa menos espacio en las cabeceras, pero muchos bloques estarán casi desperdiciados, almacenando pocos datos. La única situación en que esta clase de enfoque es preferible, es cuando los conjuntos de datos típicos sean muy grandes. Por ejemplo, muchos sistemas de archi-

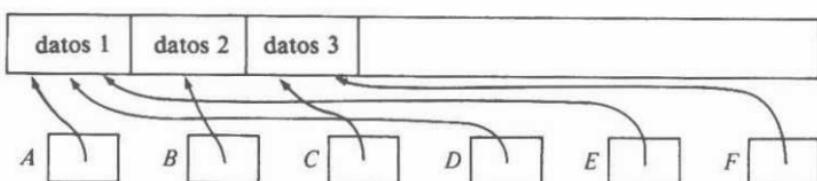
vos trabajan de esta forma, dividiendo la estructura dinámica, que por lo común es una unidad de disco, en bloques del mismo tamaño, por ejemplo 512 a 4096 bytes, dependiendo del sistema. Como muchos archivos son mucho más grandes que esos números, no se desperdicia demasiado el espacio, y los apuntadores a los bloques que conforman un archivo requieren relativamente poco espacio. La asignación de espacio bajo esta disciplina es casi directa, debido a lo que se ha aprendido en las secciones anteriores, por lo que no se analizará aquí la manera de realizarla.

Problema de la compactación

Un problema que debe enfrentarse con frecuencia es el de tomar una colección de bloques en uso, como la sugerida en la figura 12.16(a), donde cada bloque puede ser de tamaño diferente y estar apuntado por más de un apuntador, y correrlos hacia la izquierda hasta que todo el espacio disponible quede en el extremo derecho de la estructura dinámica, como se muestra en la figura 12.16(b). Como es natural, los apuntadores deben continuar apuntando a los mismos datos.



(a) Antes de la compactación



(b) Despues de la compactación

Fig. 12.16. Proceso de compactación del almacenamiento.

Existen soluciones simples a este problema asignando un poco de espacio adicional en cada bloque, y se analizará otro método más complicado, pero eficiente, que no requiere espacio extra en los bloques utilizados, sólo el que se ha usado hasta ahora en cualquiera de los esquemas de manejo del espacio estudiados, es decir, un bit vacío/lleido y un contador que indique el tamaño del bloque.

Un esquema simple para la compactación consiste en revisar primero todos los bloques desde la izquierda, sean llenos o vacíos, y calcular una «dirección de avance» para cada bloque lleno. La dirección de avance de un bloque es su posición actual menos la suma de todo el espacio vacío que se encuentra a su izquierda, esto es, la posición a la cual habrá de pasar finalmente. Es fácil calcular la dirección de avance. Al revisar los bloques desde la izquierda, se acumula la cantidad de espacio vacío que se encuentra para sustraer esta cantidad a la posición de cada bloque visto. El algoritmo se plantea en la figura 12.17.

```

(1)    var
        p: integer; { posición del bloque actual }
        hueco: integer; { cantidad total de espacio vacío hallado
                          hasta ahora }
    begin
(2)      p := extremo izquierdo de la estructura dinámica;
(4)      hueco := 0;
(5)      while p ≤ extremo derecho de la estructura dinámica do begin
            { hace que p apunte al bloque B }
(6)          if B está vacío then
                hueco := hueco + contador en el bloque B
            else { B está lleno }
                dirección de avance de B := p - hueco;
(8)                p := p + contador en el bloque B
    end
end;

```

Fig. 12.17. Cálculo de las direcciones de avance.

Una vez calculada la dirección de avance, se revisan todos los apuntadores al montón †. Se sigue cada apuntador hacia algún bloque *B* para reemplazar el apuntador por la dirección de avance encontrada en ese bloque. Por último, se mueven todos los bloques llenos hacia su dirección de avance. Este proceso es similar al de la figura 12.17, con la línea (8) reemplazada por

```

for i:= p to p-1 + contador en B do
    estructura dinámica [i - hueco] := estructura dinámica[i];

```

para pasar el bloque *B* a la izquierda en una cantidad igual a *hueco*. Obsérvese que el movimiento de bloques llenos, que requiere un tiempo proporcional a la cantidad de bloques en uso dentro de la estructura dinámica, probablemente dominará los demás costos de compactación.

† En el resto del texto se supone que la colección de dichos apuntadores está disponible. Por ejemplo, una realización normal de SNOBOL almacena pares que constan de un nombre de variable y un apuntador al valor de ese nombre en una tabla de dispersión, con la función de dispersión calculada a partir del nombre. Examinar la tabla completa permite visitar todos los apuntadores.

Algoritmo de Morris

F. L. Morris descubrió un método para la compactación de la estructura dinámica sin usar espacio en los bloques para la dirección de avance. No obstante, requiere un bit de marca asociado con cada apuntador y con cada bloque para indicar el final de una cadena de apuntadores. La idea esencial es crear una cadena de apuntadores que salen de una posición fija en cada bloque lleno y enlazados todos a ese bloque. Por ejemplo, en la figura 12.16(a) se observan tres apuntadores, A , D y E , apuntando al bloque lleno del extremo izquierdo. En la figura 12.18, se encuentra la cadena deseada de apuntadores. Una porción de los datos de tamaño igual a la de un apuntador se ha extraído del bloque y colocado al final de la cadena, donde se encontraba el apuntador A .

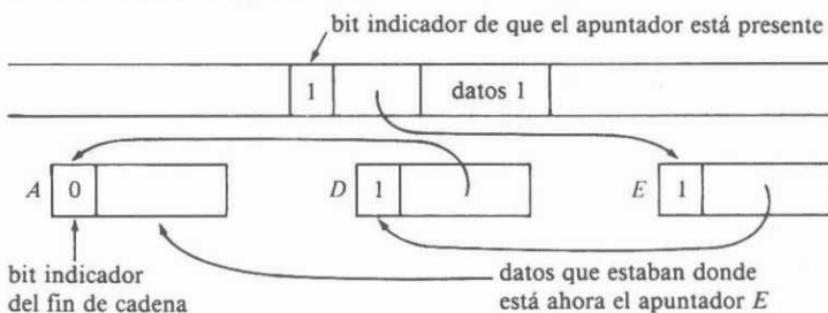


Fig. 12.18. Encadenamiento de apuntadores.

El método para crear dichas cadenas de apuntadores es como sigue. Se examinan todos los apuntadores en cualquier orden conveniente. Supóngase un apuntador p al bloque B . Si el bit de marca en el bloque B es cero, entonces p es el primer apuntador encontrado que apunta a B . Se colocan en p los contenidos de las posiciones de B utilizadas para la cadena de apuntadores, y se hace que esas posiciones de B apunten a p . Despues, se hace el bit de marca en B igual a 1, indicando que ahora tiene un apuntador, y el bit de marca en p igual a 0, indicando el fin de la cadena de apuntadores y la presencia de los datos desplazados.

Ahora bien, al considerar el apuntador p al bloque B , el bit de marca en B está en uno; entonces, B ya tiene la cabeza de la cadena de apuntadores. Se copia el apuntador de B en p , para que B apunte a p , y se hace el bit de marca en p igual a 1; de este modo, p queda, efectivamente, en la cabeza de la cadena.

Una vez que se tienen todos los apuntadores a cada bloque enlazados en una cadena que parte desde ese bloque, se pasan los bloques llenos tan a la izquierda como sea posible, igual que en el sencillo algoritmo analizado con anterioridad. Por último, se examina cada bloque en su nueva posición y se recorre su cadena de apuntadores. Cada apuntador localizado se actualiza para que apunte al bloque en su nueva posición. Al encontrar el final de la cadena se reinstalan los datos de B , contenidos en el último apuntador, a su lugar correcto en el bloque B y se fija el bit de marca del bloque igual a 0.

Ejercicios

- 12.1** Considérese la siguiente estructura dinámica de 1000 bytes, donde los bloques en blanco están en uso, y los bloques etiquetados están enlazados en una lista libre en orden alfabético. Los números indican la posición del primer byte de cada bloque.

0	100	200	400	500	575	700	850	900	999
<i>a</i>		<i>b</i>		<i>c</i>	<i>d</i>		<i>e</i>	<i>f</i>	

Supóngase que se realizan las siguientes peticiones:

- 1) asignar un bloque de 120 bytes,
- 2) asignar un bloque de 70 bytes,
- 3) devolver al frente de la lista de disponibles el bloque que se encuentra entre los bytes 700 y 849, y
- 4) asignar un bloque de 130 bytes.

Proporciónese la lista de espacio libre, en orden, después de ejecutar la secuencia de pasos anterior, suponiendo que los bloques libres se seleccionan de acuerdo con la estrategia de

- a) primer ajuste
- b) mejor ajuste.

- 12.2** Obsérvese la siguiente estructura dinámica, donde las regiones en blanco se encuentran en uso y las regiones etiquetadas están vacías.

0	100	200	300	500
	<i>a</i>		<i>b</i>	

Establézcase una secuencia de solicitudes que puedan satisfacerse al emplear

- a) primer ajuste, pero no mejor ajuste
- b) mejor ajuste, pero no primer ajuste

- *12.3** Supóngase que se utiliza un sistema de afinidades exponencial con tamaños 1, 2, 4, 8 y 16 en una estructura dinámica de tamaño 16. Si se solicita un bloque de tamaño n , para $1 \leq n \leq 16$, es necesario asignar un bloque de tamaño 2^i , donde $2^{i-1} < n \leq 2^i$. La porción no utilizada del bloque, si existe, no puede emplearse para satisfacer ninguna otra solicitud. Si se necesita un bloque de tamaño 2^i , $i < 4$, y no existe dicho bloque libre, primero se busca un bloque de tamaño 2^{i+1} y se divide en dos partes del mismo tamaño. Si no existe un bloque de tamaño 2^{i+1} , se busca y se parte un bloque de tamaño 2^{i+2} , y así sucesivamente. Si se llega a buscar un bloque libre de tamaño 32, el proceso falla y no puede satisfacerse la solicitud. A

efectos de este ejercicio, no se combinan bloques libres adyacentes en la estructura dinámica.

Existen secuencias de solicitud a_1, a_2, \dots, a_n cuya suma es menor que 16, tal que la última petición no puede satisfacerse. Por ejemplo, considérese la secuencia 5, 5, 5. La primera petición hace que el bloque inicial de tamaño 16 se parte en dos bloques de tamaño 8, uno de los cuales se usa para satisfacer la solicitud. El bloque libre restante de tamaño 8 satisface la segunda, y no queda espacio libre para satisfacer la tercera.

Encuéntrese una secuencia a_1, a_2, \dots, a_n de enteros entre 1 y 16 (no es necesario que sean idénticos), cuya suma sea lo más pequeña posible, tal que, tratada como una secuencia de solicitudes de bloques de tamaño a_1, a_2, \dots, a_n , la última petición no se satisface. Explíquese por qué esta secuencia de peticiones no se satisface, pero cualquier otra secuencia cuya suma sea más pequeña sí puede satisfacerse.

- 12.4** Considérese la compactación de memoria en la administración de bloques de igual tamaño. Supóngase que cada bloque consta de un campo para datos y otro para apuntador, y que se han marcado todos los bloques que se encuentran en uso actualmente. Los bloques están ubicados entre las localidades de memoria a y b . Se desea relocalizar todos los bloques activos de manera que ocupen memoria contigua partiendo de a . Recuérdese que, para relocalizar un bloque, el campo del apuntador de cualquier bloque que apunta al bloque relocalizado debe actualizarse. Diséñese un algoritmo para la compactación de bloques.
- 12.5** Se da un arreglo de tamaño n . Proporcionese un algoritmo para correr todos los elementos del arreglo k lugares en forma cíclica, en sentido contrario al de las manecillas del reloj, sólo con una cantidad constante de memoria adicional, independiente de k y n . *Sugerencia.* Téngase en cuenta qué sucede si se invierten los k primeros elementos, los $n-k$ últimos elementos y, por último, el arreglo completo.
- 12.6** Diséñese un algoritmo para sustituir una subcadena y de una cadena xyz por otra subcadena y' , con la menor cantidad posible de memoria adicional. ¿Cuál es la complejidad de tiempo y espacio de este algoritmo?
- 12.7** Escribase un programa para hacer una copia de una lista dada. ¿Cuál es la complejidad de tiempo y espacio del programa?
- 12.8** Escribase un programa para determinar si dos listas son idénticas. ¿Cuál es la complejidad de tiempo y espacio de este programa?
- 12.9** Obténgase el algoritmo de compactación de las estructuras dinámicas de Morris, mostrado en la sección 12.6.
- *12.10** Diséñese un esquema de asignación de almacenamiento para una situación en la que la memoria se asigna y libera en bloques de longitud 1 y 2. Proporcionense cotas sobre la eficiencia del algoritmo.

Notas bibliográficas

La administración eficiente almacenamiento es un tema central en muchos lenguajes de programación, incluyendo SNOBOL [Farber, Griswold y Polonsky (1964)], LISP [McCarthy (1965)], APL [Iverson (1962)] y SETL [Schwartz (1973)]. Nicholls [1975] y Pratt [1975] comentan las técnicas de administración de almacenamiento en el contexto de la compilación de lenguajes de programación.

El sistema de afinidades para asignación de almacenamiento fue publicado por primera vez por Knowlton [1965]. Los sistemas de afinidades de Fibonacci fueron estudiados por Hirschberg [1973].

El elegante algoritmo de marcado utilizado en la recolección de basura fue descubierto por Peter Deutsch (Deutsch y Bobrow [1966]) y por Schorr y Waite [1965]. El esquema de compactación de estructuras dinámicas de la sección 12.6 se debe a Morris [1978].

Robson [1971] y Robson [1974] analizan la cantidad de memoria requerida para algoritmos dinámicos de asignación de almacenamiento. Robson [1977] presenta un algoritmo acotado en espacio de trabajo para copiar estructuras cíclicas. Fletcher y Silver [1966] contiene otra solución al ejercicio 12.5 que utiliza poca memoria adicional.