

# 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

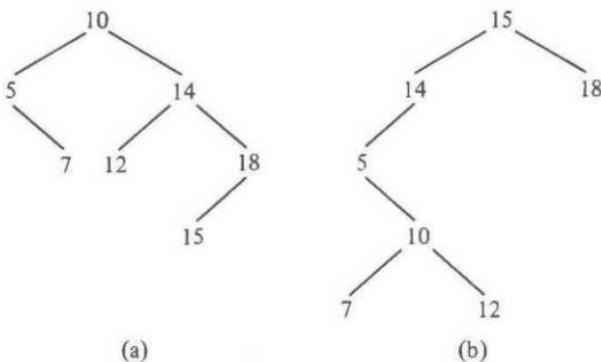
Se comenzará con los árboles binarios de búsqueda, una estructura de datos básica para la representación de conjuntos cuyos elementos están clasificados de acuerdo con algún orden lineal. Como de costumbre, se designará ese orden por medio del signo  $<$ . Esta estructura de datos es útil cuando se tiene 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 †. 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:  $\uparrow$  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 =  $\uparrow$  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.

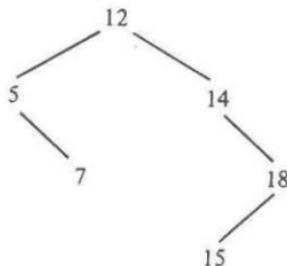
```

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-

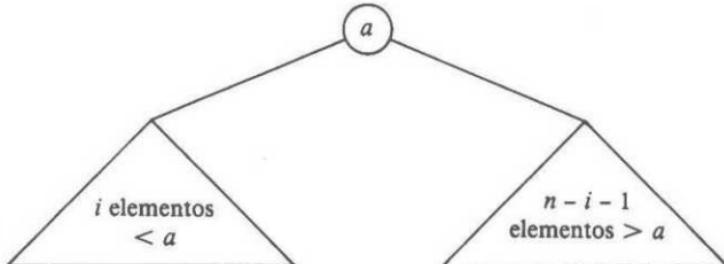


Fig. 5.7. Árbol binario de búsqueda.

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  $i \leq \lceil n/2 \rceil - 1$ , lo cual no excede de  $i\log(n/2)$ , y aquellos para los que  $i > \lceil n/2 \rceil - 1$ , que no excede de  $i\log n$ . Así, (5.3) puede escribirse otra vez

$$P(n) \leq 2 + \frac{8}{n^2} \left[ \sum_{i=1}^{\lceil n/2 \rceil - 1} ilog(n/2) + \sum_{i=\lceil n/2 \rceil}^{n-1} ilogn \right] \quad (5.4)$$

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.

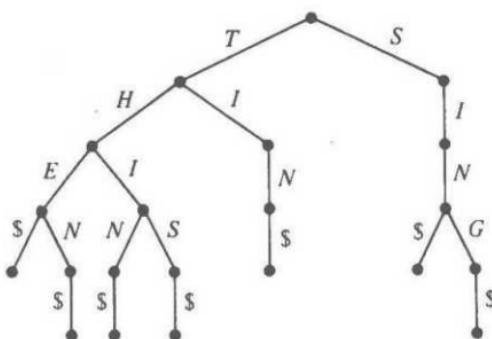


Fig. 5.9. Un trie.

**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  $O(\log n)$ .

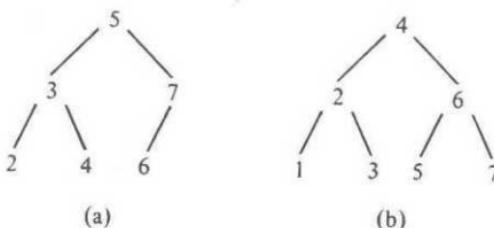
Existen otros enfoques que dan en el peor caso un tiempo  $O(\log n)$  por operación para diccionarios y colas de prioridad, como el llamado «árbol 2-3». Un árbol 2-3 tiene las siguientes propiedades.

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

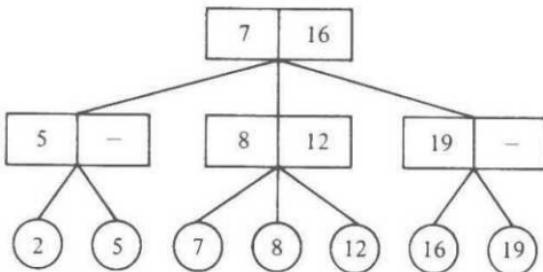


**Fig. 5.12.** Árboles completos.

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.

† 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.

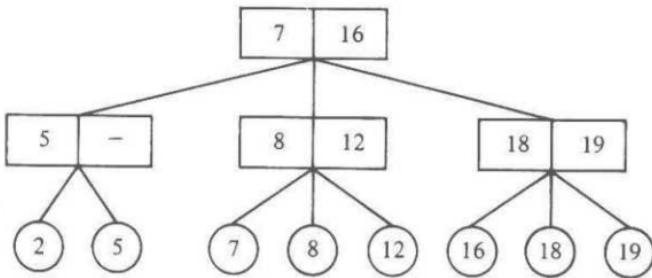


Fig. 5.14. Árbol 2-3 con el 18 insertado.

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. □

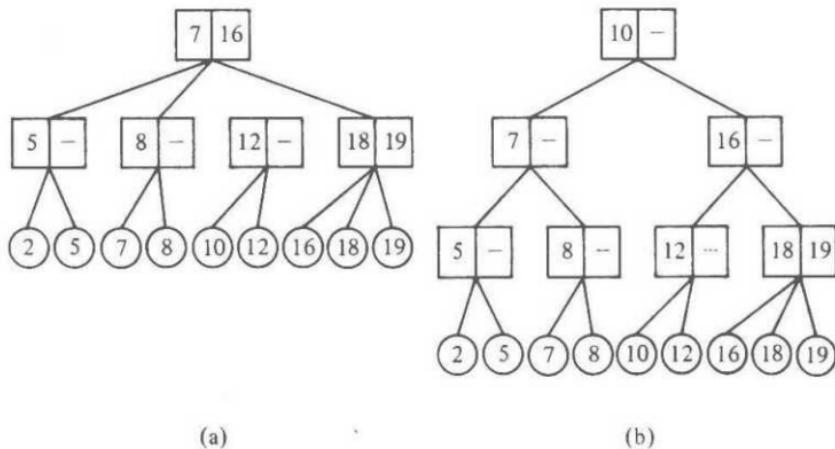


Fig. 5.15. Inserción de 10 en el árbol de la figura 5.14.

### 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).

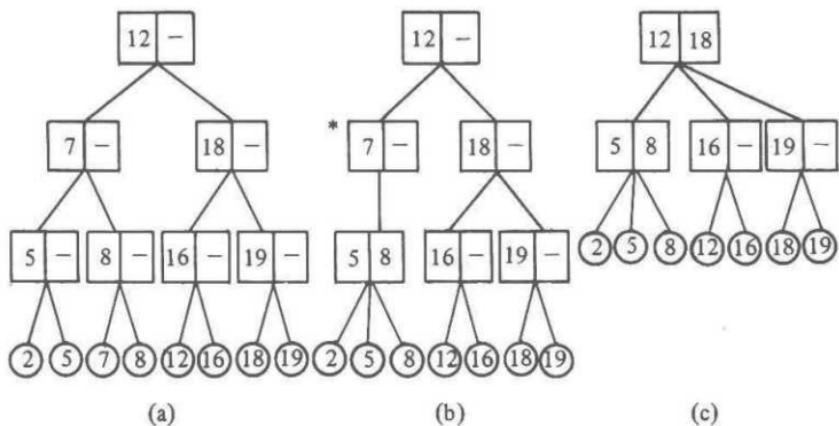


Fig. 5.16. Supresión en un árbol 2-3.

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;
        insertal(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);  $\text{COMBINA}$  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,  $\text{ENCUENTRA}$  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  $n$  <sup>f</sup> 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-

<sup>f</sup> 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 }

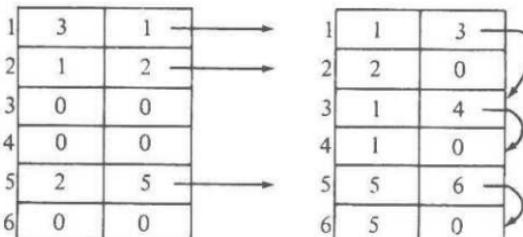
end
else { B es al menos tan grande como A }
  { código similar al del caso anterior, pero con A y B intercambiados }
end; { COMBINA }

function 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.

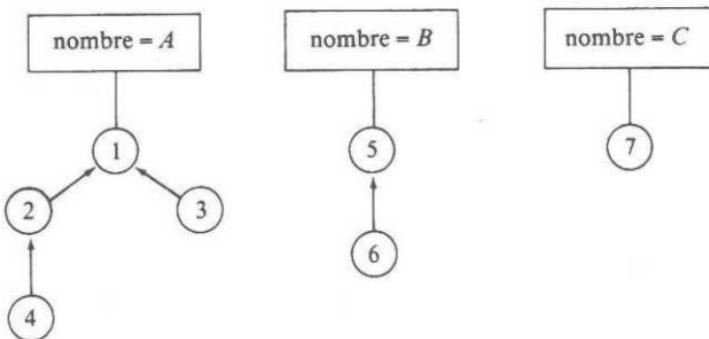


Fig. 5.25. CONJUNTO\_CE representado por una colección de árboles.

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

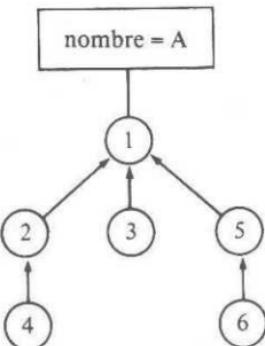


Fig. 5.26. Combinación de  $B$  dentro de  $A$ .

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.

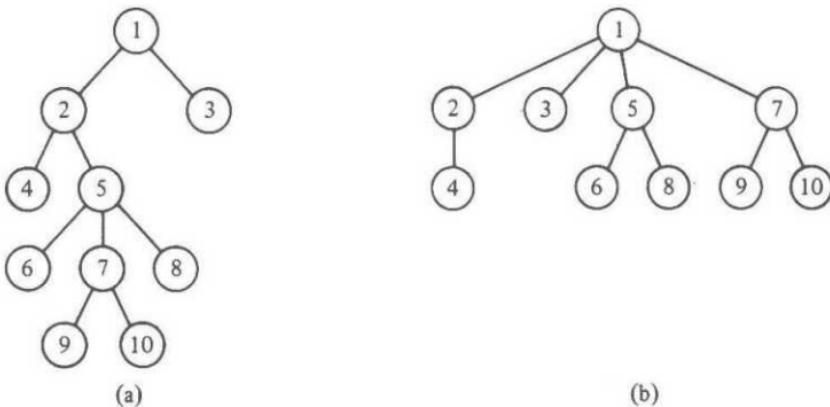


Fig. 5.27. Ejemplo de compresión de caminos.

### 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  $O(nlogn)$ .

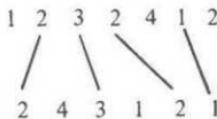


Fig. 5.28. Subsecuencia común más larga.

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.

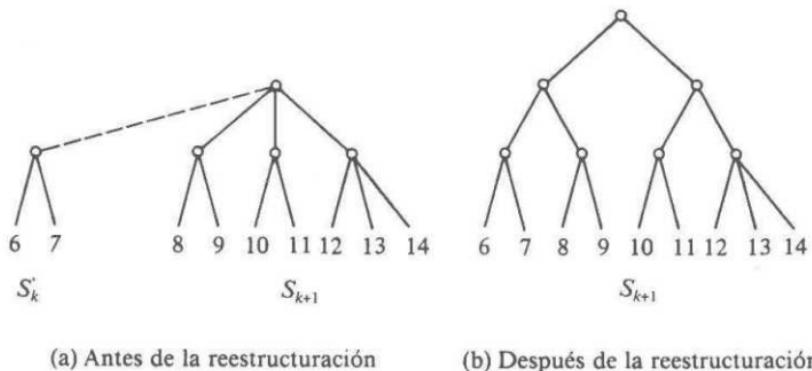


Fig. 5.30. Ejemplo de COMBINA.

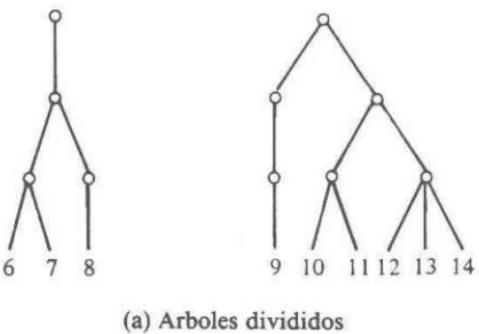


Fig. 5.31. Ejemplo de DIVIDE.

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.

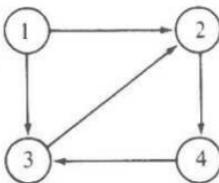


Fig. 6.1. Grafo dirigido.

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.

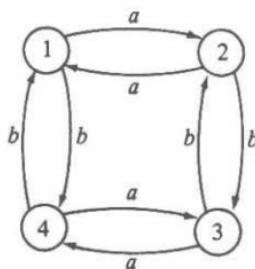


Fig. 6.2. Grafo dirigido de transiciones.

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.

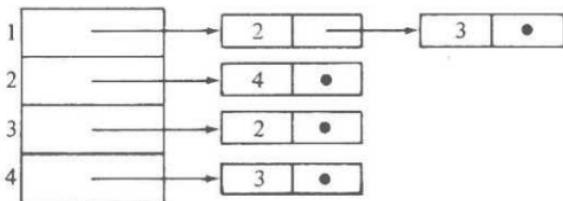


Fig. 6.4. Representación con lista de adyacencia para el grafo dirigido de la figura 6.1

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. □

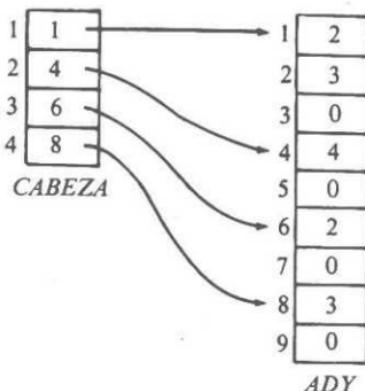
### TDA grafo dirigido

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  $\Lambda$  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  $\Lambda$  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  $\Lambda$ . 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  $w$ <sup>†</sup>. 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

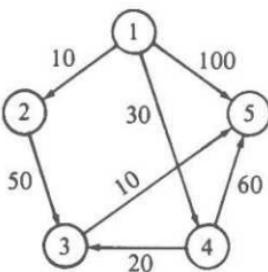
```

<sup>†</sup> 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. □

Iteración	$S$	$w$	$D[2]$	$D[3]$	$D[4]$	$D[5]$
inicial	{1}	—	10	$\infty$	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.

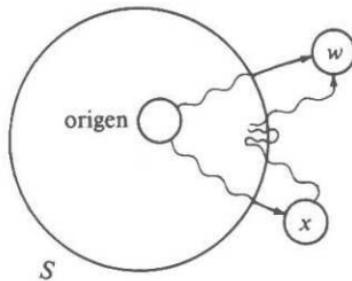


Fig. 6.11. Camino hipotético más corto a  $w$ .

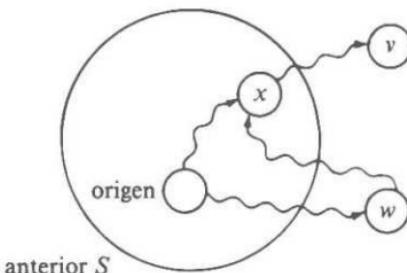


Fig. 6.12. Camino especial más corto imposible.

### 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  $A_k[i, j]$ .

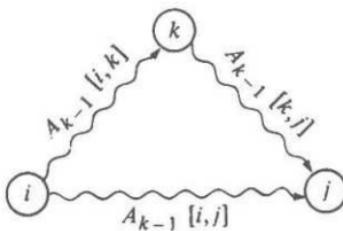


Fig. 6.13. Inclusión de  $k$  entre los vértices que van de  $i$  a  $j$ .

**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-

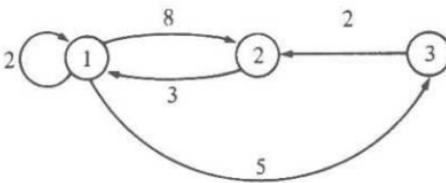


Fig. 6.14. Grafo dirigido ponderado.

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		0	8	5
2	3	0	$\infty$		3	0	8
3	$\infty$	2	0		$\infty$	2	0
$A_0[i, j]$				$A_1[i, j]$			
	1	2	3		1	2	3
1	0	8	5		0	7	5
2	3	0	8		3	0	8
3	5	2	0		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

$$\max_{w \in V} \{ \text{longitud mínima de un camino de } w \text{ a } v \}$$

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.

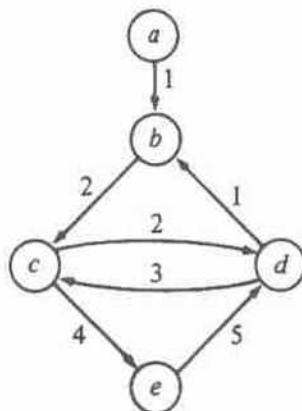


Fig. 6.22. Grafo dirigido ponderado.

Las excentricidades de los vértices son

vértice	excentricidad
$a$	$\infty$
$b$	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>	$\infty$	0	2	4	6
<i>c</i>	$\infty$	3	0	2	4
<i>d</i>	$\infty$	1	3	0	7
<i>e</i>	$\infty$	6	8	5	0
máx	$\infty$	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-

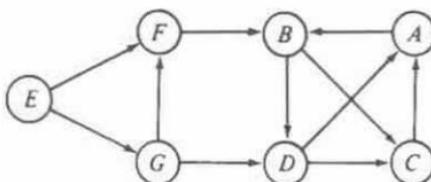


Fig. 6.25. Grafo dirigido.

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.

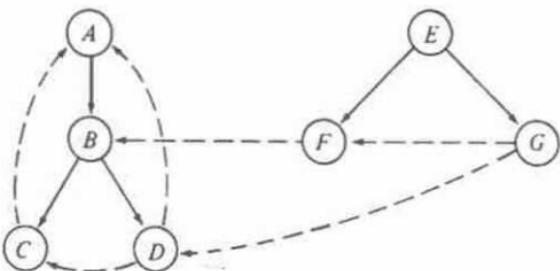


Fig. 6.26. Bosque abarcador en profundidad para la figura 6.25.

¿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.

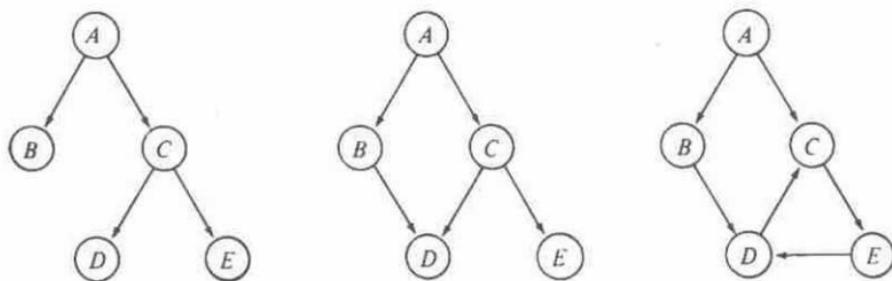


Fig. 6.27. Tres grafos dirigidos.

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  $S$ .

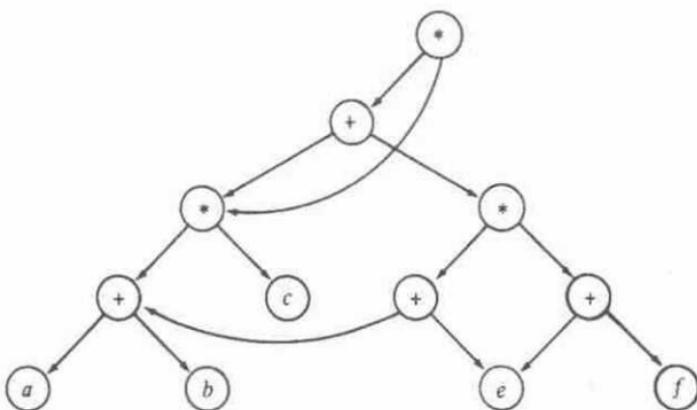


Fig. 6.28. Gda para expresiones aritméticas.

**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  $P(S)$ .  $\square$

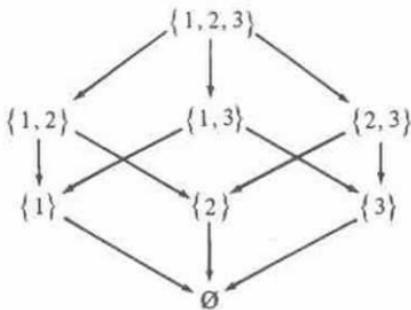


Fig. 6.29. Gda para inclusiones propias.

### 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. □

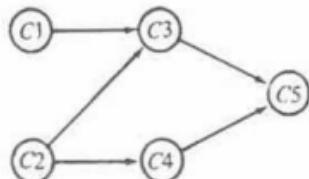


Fig. 6.31. Gda de prerrequisitos.

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.  $\square$

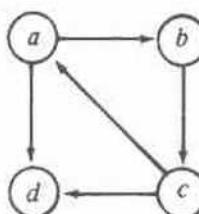


Fig. 6.32. Grafo dirigido.

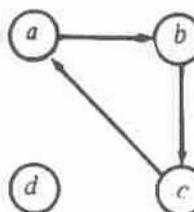


Fig. 6.33. Componentes fuertes del grafo de la figura 6.32.

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  $x$ .

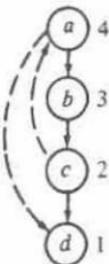
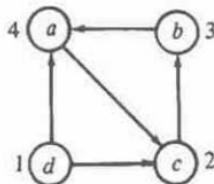
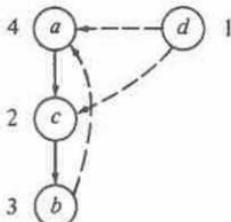


Fig. 6.35. Despues del paso 1.

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.

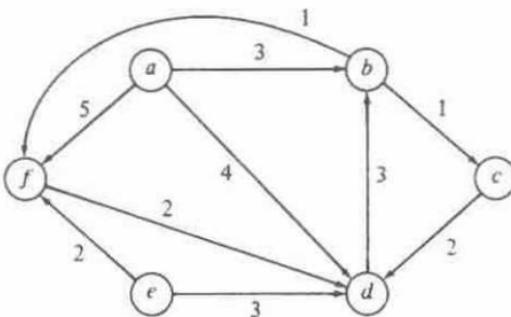


Fig. 6.38. Grafo dirigido con costos de los arcos.

- 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  $d$ .  $\square$

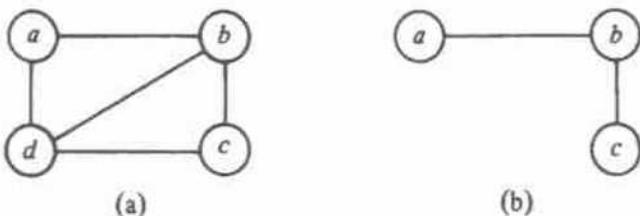


Fig. 7.1. Grafo con uno de sus subgrafos.

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.  $\square$

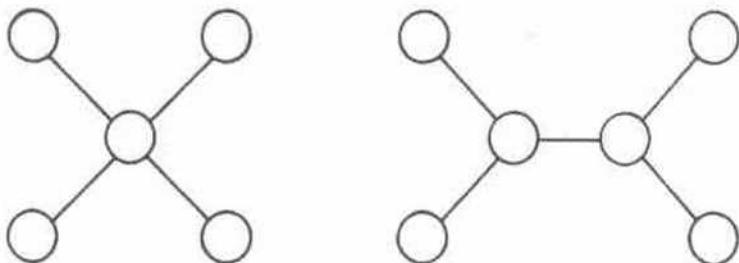


Fig. 7.2. Grafo no conexo.

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

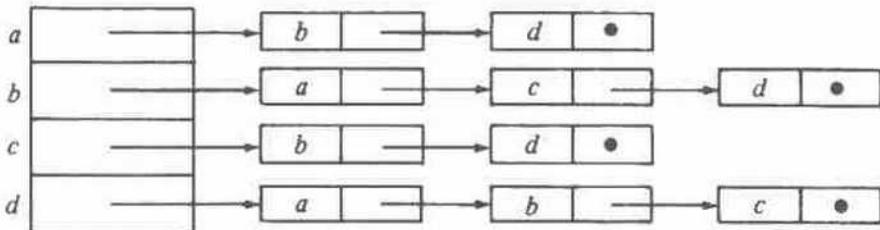
Los métodos de representación de grafos dirigidos se pueden emplear también para representar los no dirigidos. Una arista no dirigida entre  $v$  y  $w$  se representa simplemente con dos aristas dirigidas, una de  $v$  a  $w$ , y otra de  $w$  a  $v$ .

**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.

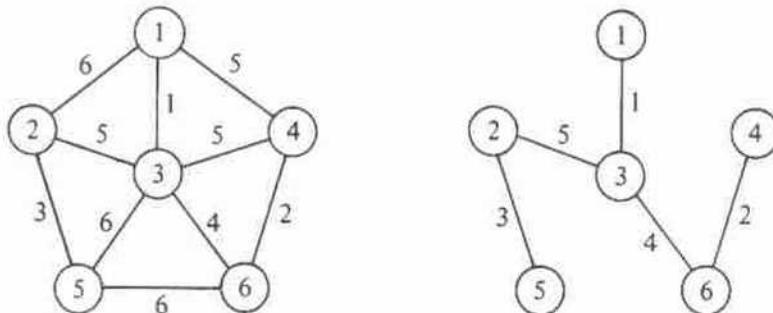


Fig. 7.4. Grafo y árbol abarcador.

### 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  $(u, v)$ .

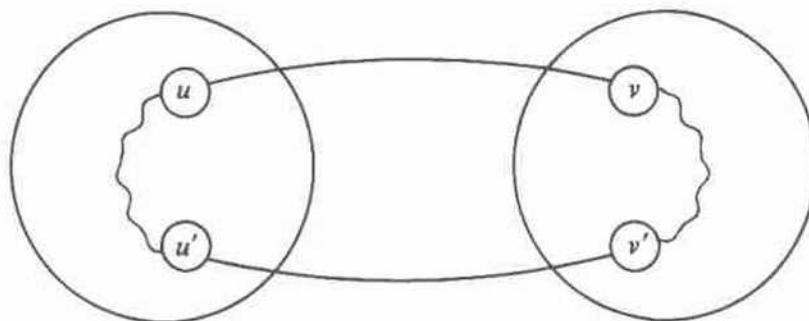


Fig. 7.5. Ciclo resultante.

### 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  $(i, \text{MAS\_CERCANO}[i])$ .

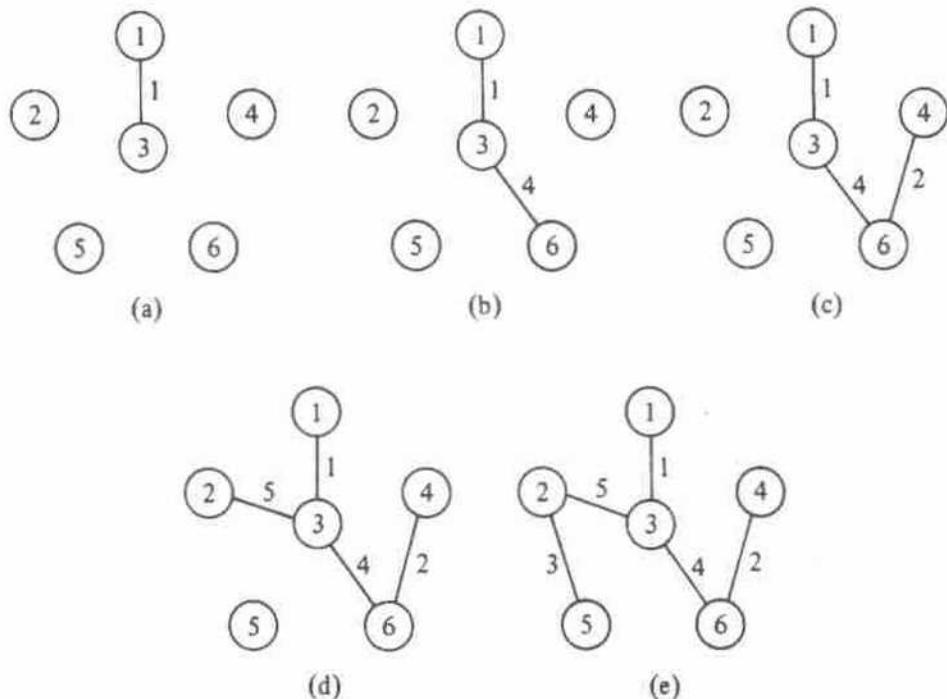


Fig. 7.7. Secuencias de aristas añadidas por el algoritmo de Prim.

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

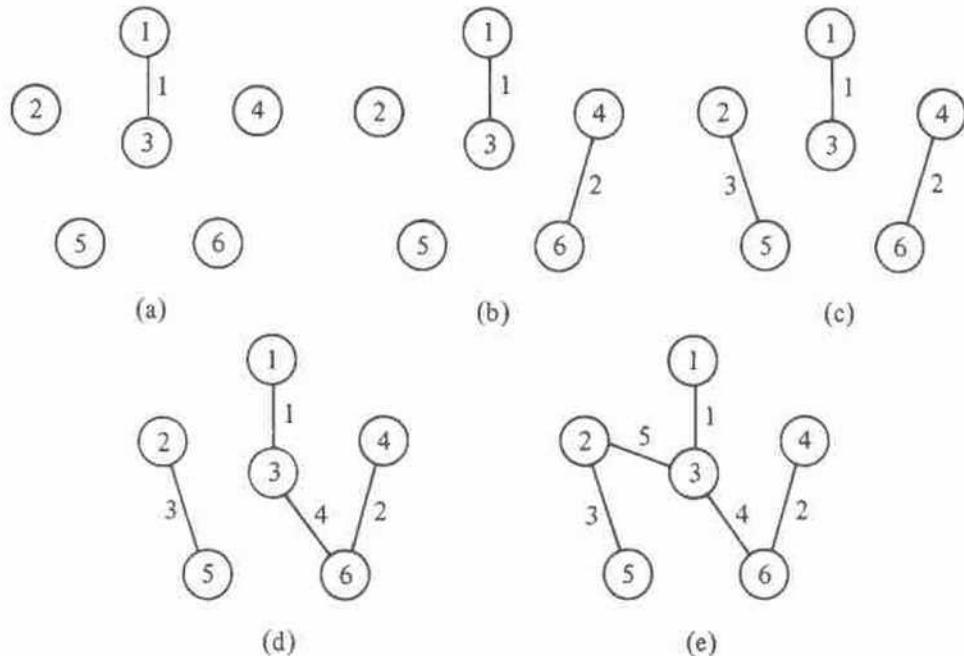


Fig. 7.9. Secuencia de aristas añadidas por el algoritmo de Kruskal.

† 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};
            dec(comp_n);
    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  $g$ .  $\square$

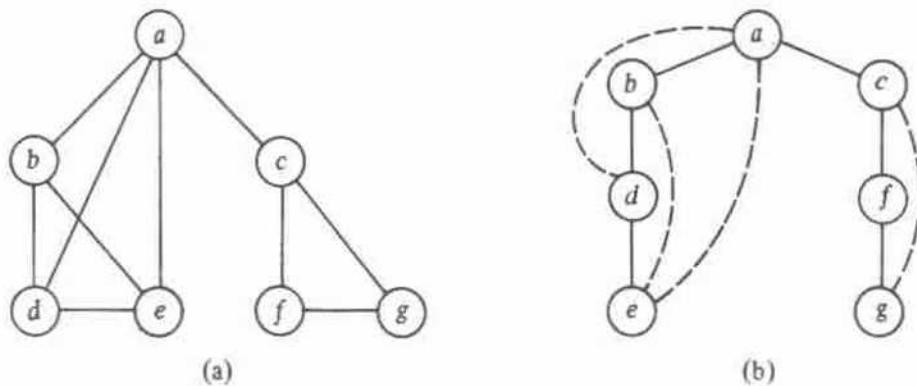


Fig. 7.11. Un grafo y su búsqueda en profundidad.

### 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.

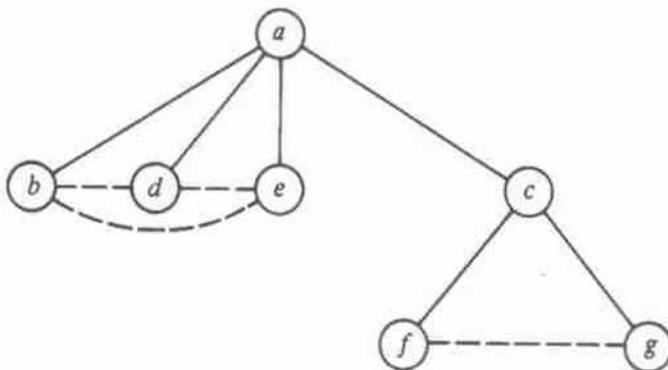


Fig. 7.13. Búsqueda en amplitud para  $G$ .

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.

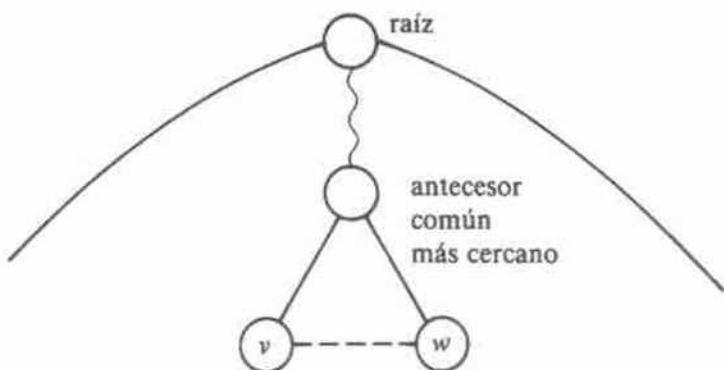


Fig. 7.14. Ciclo encontrado por la búsqueda en amplitud.

## 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*).

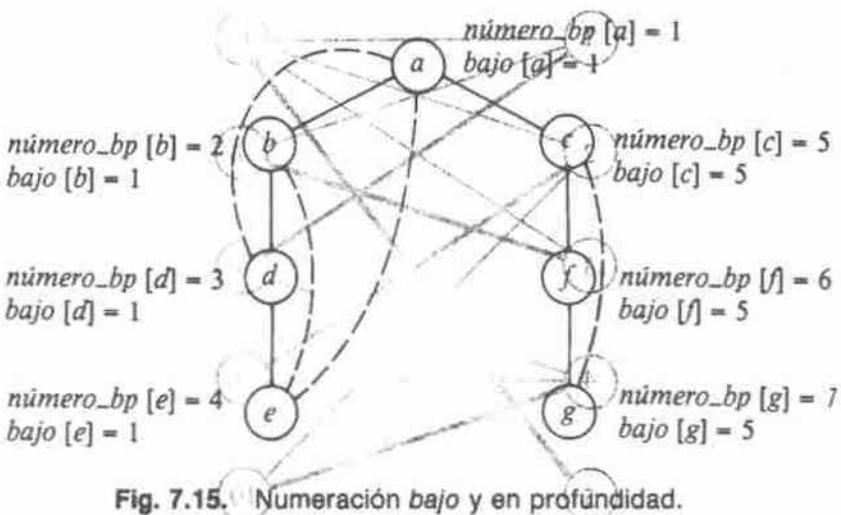


Fig. 7.15. Numeración *bajo* y en profundidad.

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.

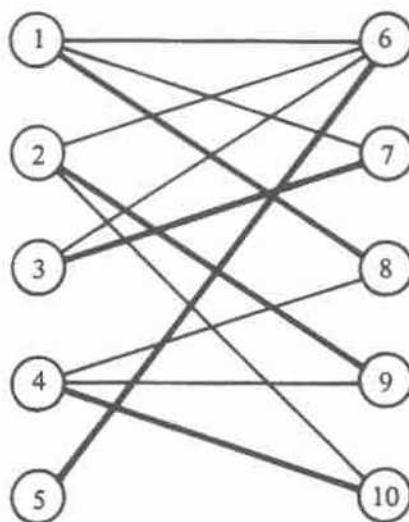


Fig. 7.16. Grafo bipartido.

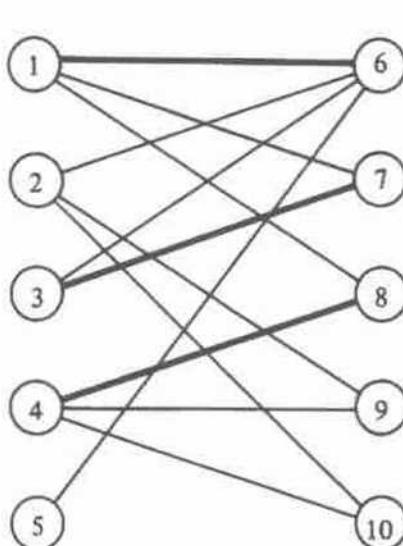
El problema del pareamiento se puede formular en términos generales como sigue. Dado un grafo  $G = (V, A)$ , el subconjunto de las aristas de  $A$  en el que ningún par de aristas es incidente sobre el mismo vértice de  $V$ , se conoce como *pareamiento*. La tarea de la selección de subconjuntos máximos 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

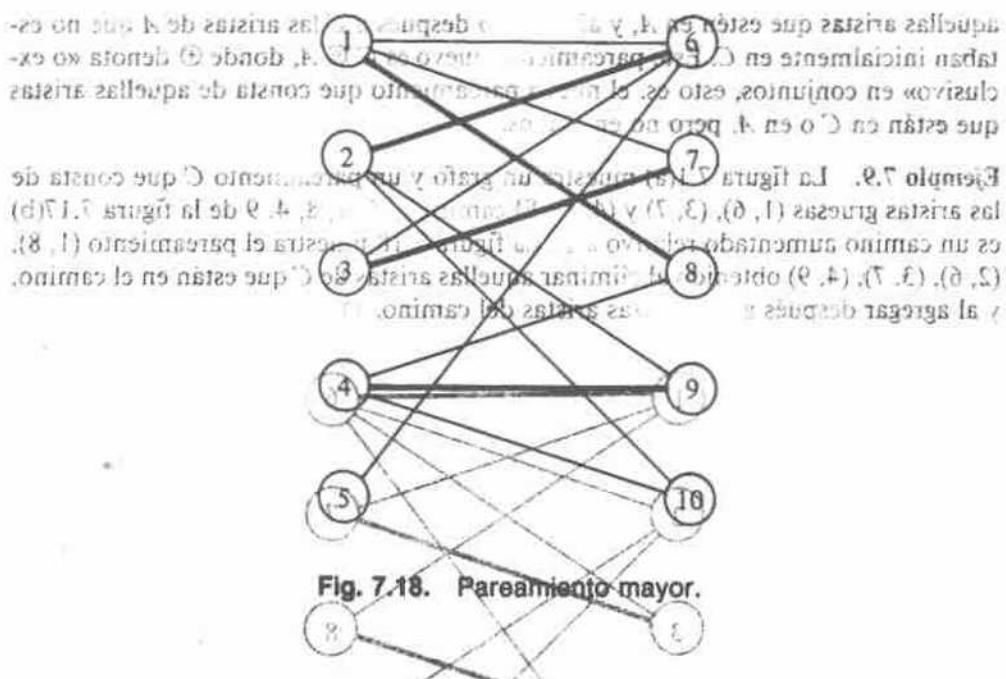


Fig. 7.18. Pareamiento mayor.

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.  $\square$

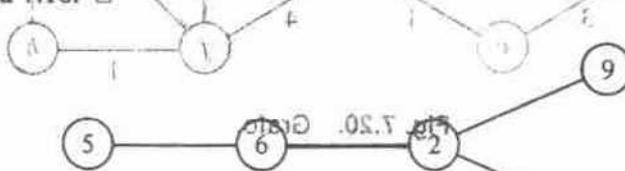


Figura 7.19. El grafo de caminos aumentados para el problema de pareamiento de la figura 7.17(a).

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  $d$ .

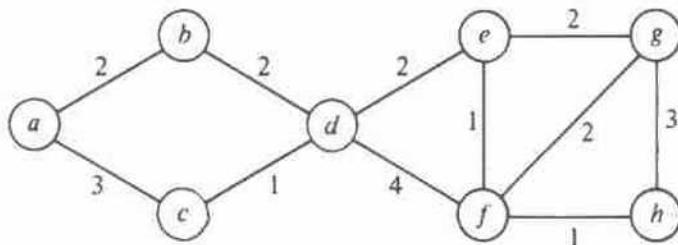


Fig. 7.20. Grafo.

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 tipo\_registro.

```

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  $i = 2, 3, 4, 5$ . Despues de cada recorrido, está garantizado que los elementos por fuera de la línea estarán ordenados entre sí, aunque su orden no tenga relación con los registros encontrados bajo la línea, los cuales se insertarán después. □

	initial	después de $i=2$	después de $i=3$	después de $i=4$	después de $i=5$	después de $i=6$
Pelée	Pelée	Pelée	Pelée	Pelée	Pelée	Pelée
Etna	Etna	Etna	Etna	Etna	Etna	Etna
Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa
Agung	Agung	Agung	Agung	Agung	Agung	Agung
Vesubio	Vesubio	Vesubio	Vesubio	Vesubio	Vesubio	Vesubio
Santa Elena	Santa Elena	Santa Elena	Santa Elena	Santa Elena	Santa Elena	Santa Elena

Fig. 8.6. Recorridos de la clasificación por inserción.

El segundo método de clasificación se conoce como «clasificación por inserción». La idea en que se sustenta la «clasificación 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

seleccionar el más pequeño entre  $A[1], \dots, A[i]$  e intercambiarlo con  $A[i]$ .

Un programa más completo se muestra en la figura 8.7. Se puede utilizar como se muestra en la figura 8.7 para clasificar una lista de  $n$  elementos. El algoritmo se basa en la idea de que el menor de los  $n$  elementos se encuentra en la posición  $A[1]$ . Se intercambia el elemento en  $A[1]$  con el elemento en  $A[i]$  si  $A[1] > A[i]$ . Si  $A[1] \leq A[i]$ , se pasa al paso siguiente.

**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 Vesubio 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  $A[n]$ , Vesubio 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  $\Omega(n^2)$  en buena parte de las secuencias de entrada de  $n$  elemen-

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 acuerdo a la  
**descripción** dada en la figura 8.7.  
**(5)**      **if**  $A[j].clave < clave_menor$  **then begin**  
               **clave\_menor** :=  $A[j].clave$ ;  
**(6)**      **end;**  
**(7)**      **indice\_menor** :=  $j$ ;  $i$  :=  $i + 1$   
**end;**

Este algoritmo lleva hasta  $O(n^2)$  pasos para clasificar  $n$  elementos. Se basa en el principio de **intercambio** ( $A[i], A[i+1], \dots, A[n-1]$ ) considerado en la figura 8.6. Se basa en el principio de que el ciclo **for** intercambia los elementos de  $i$  a  $n-1$  y el ciclo **for** **do** intercambia los elementos de  $i+1$  a  $n-1$ . Así, se lleva tanto tiempo para mover los elementos de  $i$  a  $n-1$  como para moverlos de  $i+1$  a  $n-1$ .

Fig. 8.7. Clasificación por selección.

	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
Pelé	<u>Agung</u>	Agung	Agung	Agung	Agung	Agung
Etna	<u>Etna</u>	Etna	Etna	Etna	Etna	Etna
Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa	Krakatoa
Agung	<u>Pelé</u>	Pelé	Pelé	Pelé	Pelé	Pelé
Vesubio	Vesubio	Vesubio	Vesubio	Vesubio	Vesubio	Santa Elena
Santa Elena	Santa Elena	Santa Elena	Santa Elena	Santa Elena	Santa Elena	<u>Vesubio</u>
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

$$\Omega(1 + n) = \Omega(n)$$

que ( $i$  es fijo) es  $c_3n + c_4$ .  $\sum_{j=i}^{n-1} c_2(n-j) = c_2(n-i) = c_2(n-i)^2/2 + (c_2 - 1)c_2(n-i)/2$ , donde el término  $c_2(n-i)^2/2$  es el costo de obtener los  $n-i$  pasos, donde el término  $c_2(n-i)$  tiene en cuenta los incrementos y pruebas de  $j$ . Cómo la última fórmula no excede  $(c_2/2 + c_3)n^2$ , para  $n \geq 1$ , se observa que la complejidad de tiempo de la clasificación de burbuja es  $\Omega(n^2)$ . 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.

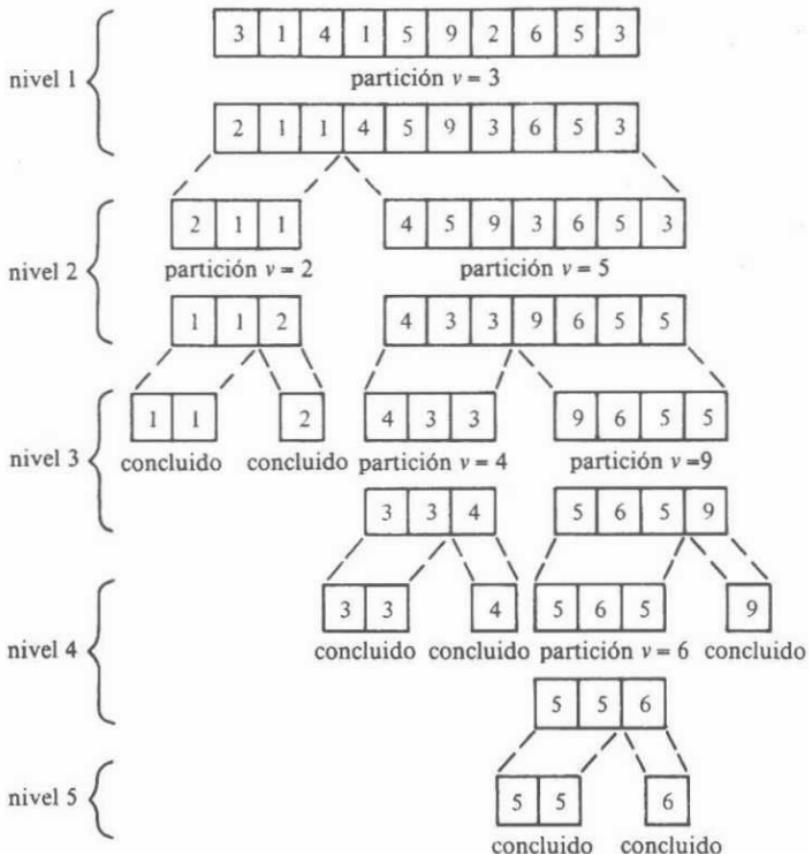


Fig. 8.9. Operación de la clasificación rápida.

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

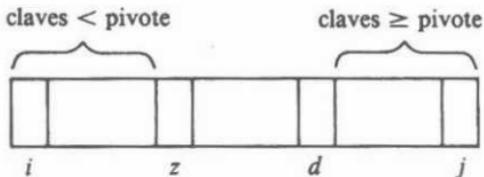
```

**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.

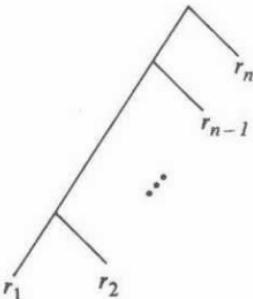


Fig. 8.15. Peor secuencia posible de selecciones de pivotes.

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  $A[i+1] \geq \dots \geq A[i+2] \geq \dots \geq A[n]$ .<sup>†</sup> 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-

<sup>†</sup> 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 parente 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
        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

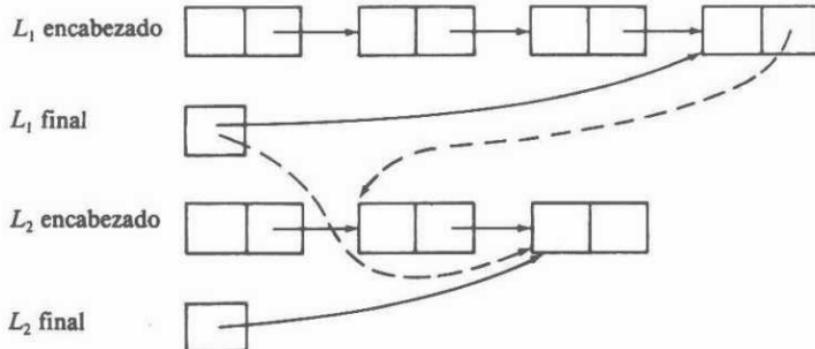


Fig. 8.19. Concatenación de listas enlazadas.

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  $\text{clave\_menor}$  y  $\text{clave\_mayor}$ .

```

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  $\text{INSERTA}$  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	

**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. □

### 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  $k_1 > k_2$ .<sup>†</sup> 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

<sup>†</sup> 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.

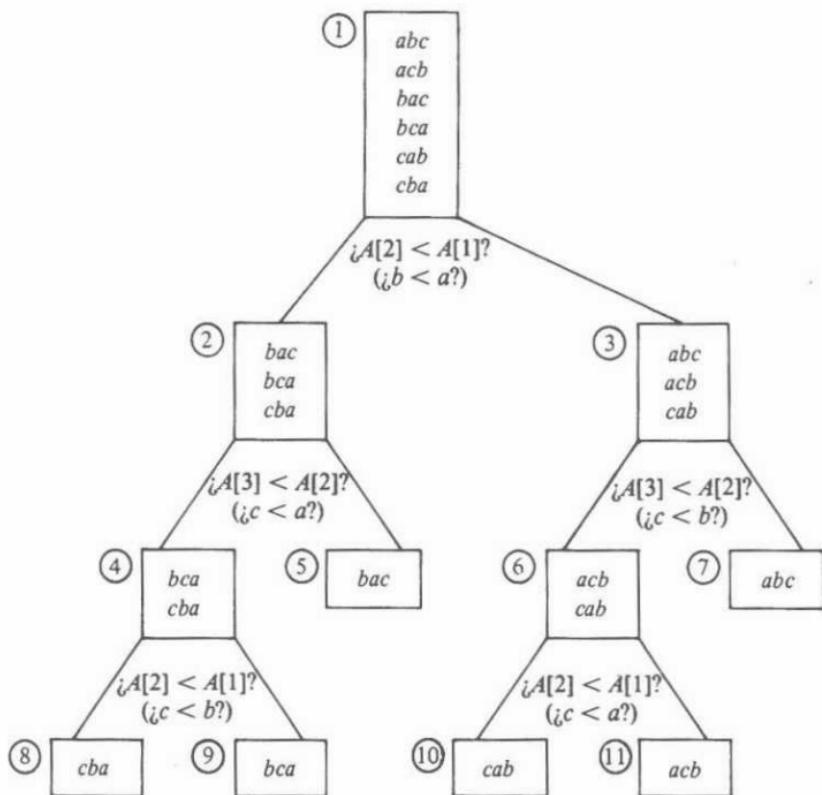


Fig. 8.23. Árbol de decisión para la clasificación por inserción con  $n = 3$ .

(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).

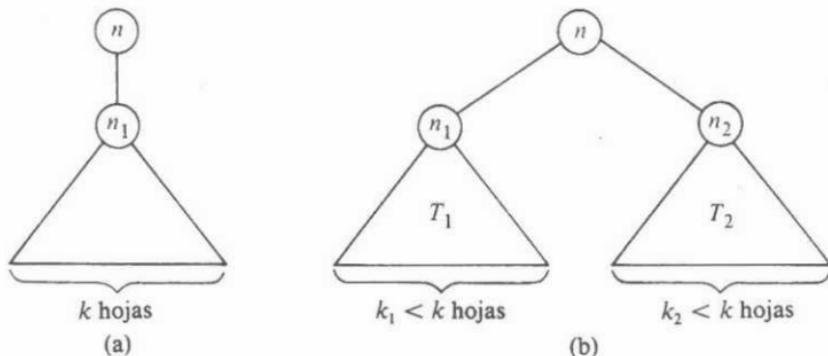


Fig. 8.24. Formas posibles del árbol binario  $T$ .

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.