CAPÍTULO

22 Colas de prioridad con mezcla

Este capítulo examina las colas de prioridad que soportan una operación adicional, `mezclar`. La operación `mezclar`, muy importante en el diseño avanzado de algoritmos, combina dos colas de prioridad en una (y destruye las iniciales). Representaremos las colas de prioridad como árboles generales. Esto hace que la operación `reducirClave` sea algo más simple, lo que es importante en algunas aplicaciones.

En este capítulo veremos:

- Una discusión acerca de los *montículos sesgados*, que no son más que colas de prioridad combinables implementadas mediante árboles binarios.
- Una discusión acerca de los *montículos de emparejamientos*, que son colas de prioridad combinables basadas en los árboles M-arios. Los montículos de emparejamientos son una alternativa práctica a los montículos binarios, aun cuando la operación `mezclar` no sea necesaria.

22.1 Los montículos sesgados

Un *montículo sesgado* es un tipo particular (en el sentido que precisaremos más adelante) de árbol binario con ordenación de montículos. En dichos árboles no se impone en principio ninguna restricción estructural, por lo que, a diferencia de lo que sucede en los montículos o en los árboles binarios de búsqueda equilibrados, no existe ninguna garantía de que la profundidad del árbol sea logarítmica. De esta forma, los montículos sesgados resultan similares a los árboles de ensanchamiento.

Los *montículos sesgados* son árboles binarios con ordenación de montículos. No se impone ninguna condición de equilibrio, pero aun así, todas las operaciones pueden realizarse en tiempo amortizado logarítmico.

22.1.1 La mezcla es importante

Si para representar una cola de prioridad empleamos un árbol binario con orden y no restringido estructuralmente, entonces la operación de mezclar se convierte en una rutina imprescindible. Esto es debido a que a partir de ella podemos realizar otras operaciones como las siguientes:

- `h.insertar( x )`: crea un árbol con un solo nodo que contiene a `x` y lo combina con la cola de prioridad.
- `h.buscarMin( )`: devuelve el elemento que se encuentra en la raíz.
- `h.eliminarMin( )`: elimina la raíz y mezcla sus hijos izquierdo y derecho.
- `h.reducirClave( p, nuevoValor )`: asumiendo que `p` es una referencia a un nodo de la cola de prioridad, podemos disminuir apropiadamente la clave de `p` y desligarlo de su padre. Esto nos devuelve dos colas de prioridad que pueden ser combinadas. Observe que `p` (que significa posición) no cambia (al contrario que en la operación equivalente de los montículos binarios).

`reducirClave` se implementa desligando un subárbol de su padre y empleando después la rutina `mezclar`.

En consecuencia, sólo es necesario mostrar cómo implementar la mezcla, ya que las otras operaciones resultan triviales. La operación `reducirClave` es muy importante en algunas aplicaciones avanzadas. Mostramos una de ellas en la Sección 14.3, en la que se discutió el algoritmo de Dijkstra para calcular el camino mínimo en un grafo. La operación `reducirClave` no se emplea en nuestra implementación ya que es complicado mantener la posición de cada elemento dentro del montículo binario. En un montículo de emparejamientos, la posición puede mantenerse como una referencia al correspondiente nodo del árbol, y a diferencia de lo que sucedía en los montículos binarios, la posición nunca cambia.

Esta sección discute una implementación de las colas de prioridad con mezcla que emplea árboles binarios: los montículos sesgados. En primer lugar, se muestra que si no nos importa demasiado la eficiencia, la mezcla de dos árboles con ordenación de montículos es sencilla. A continuación, se realiza una pequeña modificación (el montículo sesgado) que evita las ineficiencias obvias del algoritmo original. Por último, se realiza una demostración de que la operación `mezclar` para los montículos sesgados es logarítmica en coste amortizado y se comenta la relevancia práctica de este resultado.

22.1.2 Mezcla simple de árboles con ordenación de montículos

Dos árboles pueden mezclarse fácilmente de forma recursiva.

Supongamos que tenemos dos árboles con orden, H_1 y H_2, que debemos mezclar. Claramente, si alguno de los dos árboles es vacío, el otro es el resultado de la mezcla. En caso contrario, para mezclar los dos árboles comparamos sus raíces. La mezcla se realiza de forma recursiva combinando el árbol de mayor raíz con el subárbol derecho[1] del árbol de menor raíz.

El resultado es que los caminos derechos se mezclan. Debemos ser cuidadosos para no generar caminos derechos excesivamente largos.

La Figura 22.1 muestra que el efecto de esta estrategia recursiva es que los caminos derechos de las dos colas de prioridad se combinan para formar la nueva cola de prioridad. Todos los nodos del camino derecho mantienen su subárbol izquierdo inicial, y sólo cambian los nodos del camino derecho. Observe que en realidad los árboles que aparecen como argumento en el ejemplo mostrado en la Figura 22.1 no podrían obtenerse empleando únicamente inserciones y mezclas, ya que, tal y como hemos comentado, durante una mezcla no pueden añadirse hijos izquierdos. El efecto consiguiente es que, lo que aparenta ser un árbol binario con ordenación de montículos es en realidad una secuencia ordenada que se corresponde con el camino derecho del árbol. De este modo, todas las operaciones tienen coste lineal.

[1] Evidentemente, podemos emplear cualquiera de los subárboles. Aquí se emplea el derecho por simplicidad.

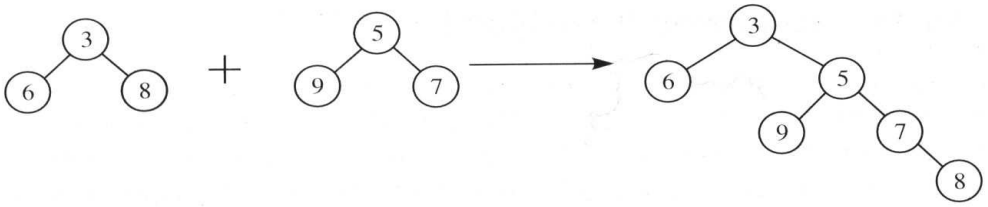


Figura 22.1 Mezcla simplista de dos árboles con ordenación de montículos: se mezclan los caminos derechos.

Afortunadamente, una sencilla modificación asegura que el camino derecho no va a crecer más.

22.1.3 El montículo sesgado: una modificación sencilla

La mezcla en la Figura 22.1 crea un árbol de mezcla temporal. Una modificación simple es la siguiente: antes de finalizar la mezcla, intercambiamos el hijo derecho e izquierdo de cada nodo del camino derecho del árbol temporal. En principio, sólo aquellos nodos que estaban en los caminos derechos iniciales estarán en el camino derecho del árbol temporal. Como resultado del intercambio, mostrado en la Figura 22.2, estos nodos formarán el camino izquierdo del árbol resultante. Cuando se realizan las mezclas de este modo, el tipo de árboles con ordenación de montículos resultante se conoce como *montículo sesgado*.

Para evitar el problema de generar caminos derechos excesivamente largos, convertimos el camino derecho obtenido tras la mezcla, en el camino izquierdo. Este tipo de mezclas nos devuelve *montículos sesgados*.

Una descripción recursiva es como sigue: sea L el árbol con la raíz más pequeña y R el otro árbol. Se realiza entonces el siguiente proceso:

1. Si uno de los árboles es vacío, el otro es el resultado de la mezcla.
2. En caso contrario, tomamos como *Temp* el subárbol derecho de L.
3. El subárbol izquierdo de L se convierte en su nuevo subárbol derecho.
4. Se calcula el resultado de la mezcla recursiva de *Temp* y R, que se convierte en el nuevo subárbol izquierdo de L.

Por medio del intercambio de hijos confiamos en que la longitud del camino derecho no crecerá excesivamente. Por ejemplo, si mezclamos un par de árboles con un único camino derecho prolongado, los nodos que intervienen en dichos caminos no reaparecerán en ningún camino derecho, al menos durante un tiempo. Aunque es posible obtener árboles en los que cada nodo aparece en un camino derecho, esto sólo puede obtenerse como resultado de un gran número de mezclas relativamente eficientes. En la siguiente sección, demostramos rigurosamente este hecho estableciendo que el coste amortizado de la operación de mezcla es logarítmico.

Aún es posible tener un camino derecho largo. Sin embargo, ésta es una rara posibilidad que debe estar precedida por varias mezclas en las que intervengan caminos derechos cortos.

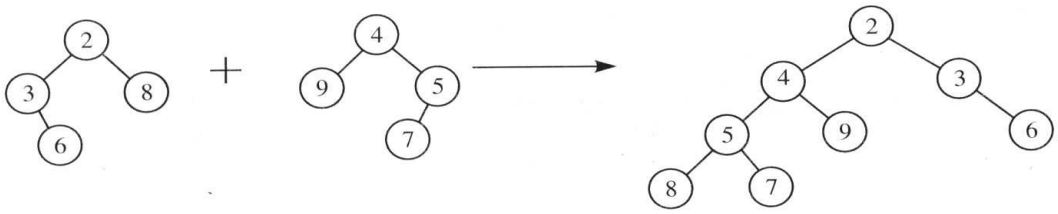


Figura 22.2 Mezcla de montículos sesgados; se mezclan los caminos derechos y el resultado se convierte en el camino izquierdo.

22.1.4 Análisis del montículo sesgado

El coste de una mezcla es el total de nodos en los caminos derechos de los dos árboles combinados.

Supongamos que tenemos dos montículos, H_1 y H_2, que tienen r_1 y r_2 nodos en sus respectivos caminos derechos. Entonces el tiempo empleado en realizar su mezcla es proporcional a $r_1 + r_2$. Cuando pagamos una unidad por cada nodo en los caminos derechos, el coste de una mezcla es proporcional a lo abonado. Como los árboles no tienen ninguna restricción estructural, todos los nodos en ambos árboles pueden caer en sus caminos derechos. Esto genera una cota $\Theta(N)$ para el caso peor de la mezcla (el Ejercicio 22.4 pide la construcción de un ejemplo). En contraposición, como mostraremos muy pronto, el tiempo amortizado de la mezcla de montículos sesgados es $O(\log N)$.

Al igual que en los árboles de ensanchamiento, la demostración se hace introduciendo una función de potencial que cancela el coste variable de las operaciones de los montículos sesgados. Queremos que la función de potencial aumente en $O(\log N) - (r_1 + r_2)$ de modo que el coste total de una mezcla y el cambio de potencial sólo sea $O(\log N)$. Si el potencial es mínimo antes de la primera operación, aplicando la técnica de sumas solapadas tenemos garantizado que el gasto total de M operaciones es $O(M \log N)$, igual que en los árboles de ensanchamiento.

Lo que se necesita entonces es alguna clase de función de potencial que capture el efecto de las operaciones de los montículos sesgados. Encontrar una función tal es bastante complicado. Sin embargo, una vez encontrada, la demostración es relativamente corta.

DEFINICIÓN: Un nodo p es *pesado* si el tamaño de su subárbol derecho es mayor que el tamaño del subárbol izquierdo; en caso contrario, se dice que es *ligero*. En particular, cuando los dos subárboles de un nodo tienen el mismo tamaño tenemos que es ligero.

La función de potencial es el número de nodos pesados. Sólo los nodos en los caminos derechos pueden ver modificado su estado pesado/ligero. El número de nodos ligeros en un camino derecho es logarítmico.

En la Figura 22.3, antes de la mezcla, los nodos 3 y 4 son pesados. Después de la mezcla, sólo el nodo 3 es pesado. Se pueden comprobar fácilmente tres hechos. En primer lugar, como resultado de la mezcla sólo los nodos del camino derecho pueden ver modificado su carácter pesado/ligero, ya que ningún otro nodo ve alterados sus subárboles. En segundo lugar, toda hoja es ligera. En tercer lugar, el número de nodos ligeros en el camino derecho de un nodo N es, a lo sumo, $\lfloor \log N \rfloor + 1$. Esto es debido a que el hijo derecho de un nodo ligero tiene menos de la mitad del tamaño que su padre, por lo que puede aplicarse el principio de la división por la mitad. El $+1$ adicional viene como resultado de que las hojas sean ligeras. A partir de estos preliminares, podemos demostrar los Teoremas 22.1 y 22.2.

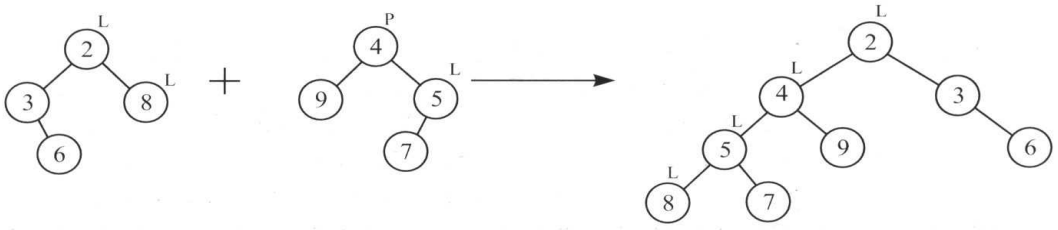


Figura 22.3 Cambio en el estado pesado/ligero después de la mezcla.

Teorema 22.1

Sean H_1 y H_2 dos montículos sesgados, con N_1 y N_2 nodos, respectivamente, y sea N su tamaño combinado (esto es, $N_1 + N_2$). Supongamos que el camino derecho de H_1 tiene l_1 nodos ligeros y h_1 nodos pesados, es decir $l_1 + h_1$ nodos en total. Por su parte, el camino derecho de H_2 tiene l_2 nodos ligeros y h_2 nodos pesados, es decir $l_2 + h_2$ nodos en total. Si se define el potencial como el número total de nodos pesados en la colección de montículos sesgados que estamos manejando, entonces el coste de la mezcla es, a lo sumo, $2 \log N + (h_1 + h_2)$, pero el cambio en el potencial es, como mucho, $2 \log N - (h_1 + h_2)$.

Demostración

El coste de la mezcla es el número de nodos en los caminos derechos, o sea, $l_1 + l_2 + h_1 + h_2$. Como el número de nodos ligeros es logarítmico, $l_1 \leqslant \lfloor \log N_1 \rfloor + 1$ y $l_2 \leqslant \lfloor \log N_2 \rfloor + 1$. De modo que $l_1 + l_2 \leqslant \log N_1 + \log N_2 + 2 \leqslant 2 \log N$, donde la última desigualdad se obtiene a partir del Teorema 21.4. Con todo ello, el coste de la mezcla es, a lo sumo, $2 \log N + (h_1 + h_2)$. La cota del cambio de potencial se deduce del hecho de que sólo los nodos que intervienen en la mezcla pueden cambiar su estado pesado/ligero, y de que cualquier nodo pesado del camino debe convertirse en ligero, ya que sus hijos se intercambian. Incluso si todos los nodos ligeros se convirtieran en pesados, el cambio de potencial estaría aún limitado por $l_1 + l_2 - (h_1 + h_2)$. Empleando el argumento anterior, esto es, a lo sumo, $2 \log N - (h_1 + h_2)$.

Teorema 22.2

El coste amortizado de los montículos sesgados para las operaciones `mezclar`, `insertar` *y* `eliminarMin` *es, a lo sumo,* $4 \log N$.

Demostración

Sea Φ_i el potencial de la colección de montículos sesgados inmediatamente después de la i*-ésima operación. Observe que $\Phi_0 = 0$ y $\Phi_i \geqslant 0$. Una inserción comienza por crear un árbol con un único nodo, cuya raíz es, por definición, ligera. De este modo, no se altera el potencial anterior a la mezcla subsiguiente. Una operación* `eliminarMin` *comienza por desechar la raíz de un árbol, y como consecuencia de lo cual no puede aumentar el potencial (podría, de hecho, reducirse). Tras ello se produce la mezcla de sus hijos. En definitiva, todo queda reducido a considerar los costes de la mezcla. Sea c_i el coste de la mezcla que se produce como resultado de la i-ésima operación. Entonces $c_i + \Phi_i - \Phi_{i-1} \leqslant 4 \log N$. Aplicando la técnica de sumas solapadas sobre M operaciones, obtenemos $\Sigma_{i=1}^{M} c_i \leqslant 4 M \log N$, ya que la diferencia $\Phi_M - \Phi_0$ no es negativa.*

El montículo sesgado es un destacado ejemplo de una estructura relativamente simple, cuyo análisis no lo es en absoluto. Sin embargo, es cierto que dicho análisis es sencillo de concluir una vez que la función de potencial adecuada ha sido identificada. Desgraciadamente, aún no disponemos de ningún tipo de teoría general que nos permita elegir las funciones de potencial adecuadas para cada problema. Normalmente, es necesario probar con muchas funciones antes de dar con la adecuada.

Una de las partes más difíciles del análisis es encontrar una función de potencial adecuada.

Debe emplearse un algoritmo no recursivo, ya que existe la posibilidad de un desbordamiento de la pila.

Un comentario final resulta oportuno. Aunque la descripción inicial del algoritmo emplea recursión y la recursión permite obtener un código más sencillo, ésta no puede emplearse en la práctica. Esto es debido a que el coste lineal en el caso peor puede provocar un desbordamiento de la pila, si implementáramos el algoritmo con recursión. Como consecuencia, debe implementarse la correspondiente versión iterativa del algoritmo. En lugar de alargarnos en el estudio de todos estos detalles técnicos un tanto enojosos, preferimos presentar una estructura de datos alternativa un poco más complicada: el *montículo de emparejamientos*. Esta estructura de datos no ha sido todavía completamente analizada, pero parece comportarse muy bien en la práctica.

22.2 Los montículos de emparejamientos

El *montículo de emparejamientos* es un árbol *M*-ario con ordenación de montículos, no restringido estructuralmente. Cumple la propiedad de que el coste de todas las operaciones, salvo la eliminación, es constante en el caso peor. Aunque `eliminarMin` es lineal en el caso peor, puede comprobarse que cualquier *secuencia* de operaciones de los montículos de emparejamientos tiene coste amortizado logarítmico. Se ha conjeturado, pero no demostrado, que pueden conseguirse mejores resultados. Más exactamente, que todas las operaciones, excepto `eliminarMin`, tienen coste amortizado constante, mientras que `eliminarMin` tiene coste amortizado logarítmico.

El *montículo de emparejamientos* es un árbol *M*-ario con ordenación de montículos, no restringido estructuralmente. Su análisis está incompleto, pero parece tener un buen comportamiento en la práctica. El montículo de emparejamientos se almacena empleando la representación primer hijo/siguiente hermano. Para implementar `reducirClave` se emplea una tercera referencia.

La Figura 22.4 muestra un montículo de emparejamientos abstracto. Su implementación usa la representación primer hijo/siguiente hermano, presentada en el Capítulo 17. La operación `reducirClave`, tal y como mostraremos más tarde, necesita que cada nodo contenga una referencia adicional. Cada nodo que sea el hijo más izquierdo de otro contiene una referencia a su padre; en otro caso, el nodo es un hermano derecho y contiene una referencia a su hermano izquierdo. En la Figura 22.5 se muestra esta representación, en la que la línea más oscura indica que dos referencias (una en cada dirección) conectan los pares de nodos.

22.2.1 Operaciones del montículo de emparejamientos y teoría

La mezcla es simple: el árbol con mayor raíz se convierte en el hijo izquierdo del árbol con menor raíz. La inserción y la reducción de claves también son sencillas.

En principio, las operaciones básicas de los montículos de emparejamientos son sencillas. Ésta es la razón por la que se comportan bien en la práctica. Para combinar dos montículos de emparejamientos, convertimos el montículo con la raíz mayor en el primer hijo del montículo con la raíz menor. La inserción es un caso

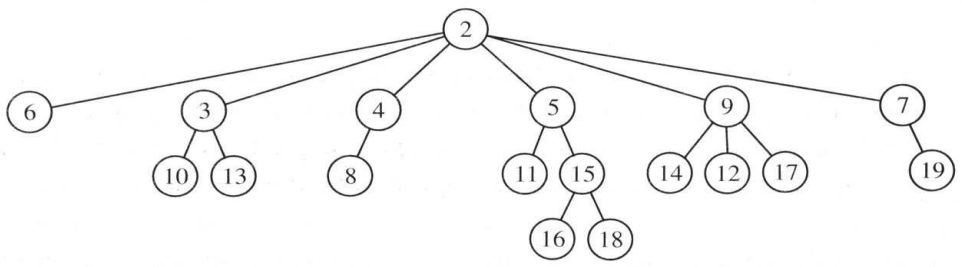


Figura 22.4 Representación abstracta de un montículo de emparejamientos simple.

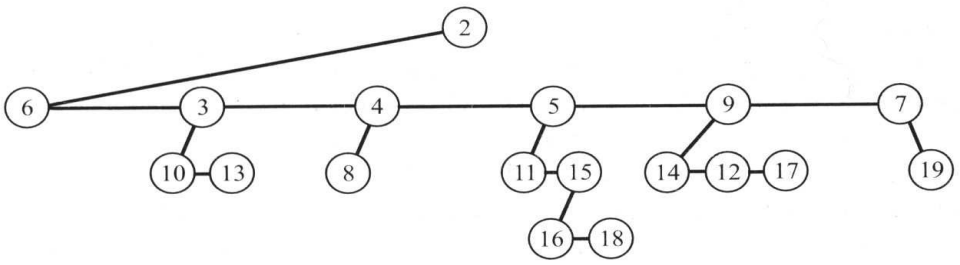


Figura 22.5 Representación del montículo de emparejamientos de la Figura 22.4; la línea oscura representa un par de referencias que conecta los nodos en ambas direcciones.

particular de mezcla. Para realizar una operación `reducirClave`, reducimos el valor del nodo correspondiente. Debido a que no mantenemos en todos los nodos referencias a sus padres, no sabemos si esto incumple el orden del montículo. Por este motivo, desligamos de su padre el nodo ajustado y completamos `reducirClave` combinando los dos montículos de emparejamientos obtenidos. La Figura 22.5 muestra cómo desligar un nodo de su padre, eliminándolo de lo que es, básicamente, una lista enlazada de hijos. Además se verifica un hecho importante: todas las operaciones descritas tienen un coste constante. Sin embargo, cuando consideramos `eliminarMin` no tenemos tanta suerte.

`eliminarMin` es una operación costosa, ya que la nueva raíz podría ser cualquiera de los c hijos de la raíz original. Son necesarias $c - 1$ mezclas.

Para realizar `eliminarMin`, debemos eliminar la raíz del árbol, creando una colección de montículos. Si la raíz tiene c hijos, la combinación de todos ellos en un único montículo requiere $c - 1$ mezclas. Como consecuencia, si hay muchos hijos, el proceso requerirá bastante tiempo. En particular, si la secuencia de inserción fuera 1, 2, ..., N, entonces es sencillo comprobar que el primero de ellos estará en la raíz y que el resto serán sus hijos. Consecuentemente, `eliminarMin` es $O(N)$. Lo mejor que podemos esperar es preparar las mezclas de modo que no se repitan instancias costosas de la operación `eliminarMin`.

El orden en el que se combinan los subárboles del montículo es importante. El algoritmo más sencillo es la *mezcla en dos pasadas*.

La variante más simple y práctica de todas las propuestas es la *mezcla en dos pasadas*. En primer lugar mezclamos pares de hijos yendo de izquierda a derecha[2]. Después de este primer recorrido, se ha reducido a la mitad el número de árboles a combinar. El segundo recorrido se hace de derecha a izquierda. En cada uno de los pasos combinamos el árbol situado más a la derecha que nos queda de la primera pasada con el obtenido después de la mezcla actual. Por ejemplo, si tenemos hijos desde c_1 hasta c_8, el primer recorrido realiza las mezclas c_1 y c_2, c_3 y c_4, c_5 y c_6, y c_7 y c_8. Los resultados son d_1, d_2, d_3 y d_4. Realizamos la segunda pasada mezclando d_3 y d_4; d_2 se combina con este resultado, y d_1 se mezcla con el resultado de dicha combinación, completándose así la rutina `eliminarMin`. La Figura 22.6 muestra el resultado de `eliminarMin` sobre el montículo de emparejamientos de la Figura 22.5.

Se han propuesto una gran variedad de alternativas. Muchas de ellas son indistinguibles, pero usar un único recorrido de izquierda a derecha es una mala idea.

Son posibles también otras estrategias de mezcla. Por ejemplo, podemos insertar en una cola cada subárbol (correspondiente a un hijo), sacar repetidamente de la cola dos árboles e insertar de nuevo en la cola el resultado de su mezcla. Después de $c - 1$ mezclas, sólo permanece un árbol en la cola y ése es el resultado de `eliminarMin`. Sin embargo, usar una pila en lugar de una cola puede estropear el

[2] Debemos tener cuidado si el número de hijos es impar. Cuando esto sucede, completamos el primer paso combinando el último hijo con el árbol obtenido de la mezcla de más a la derecha.

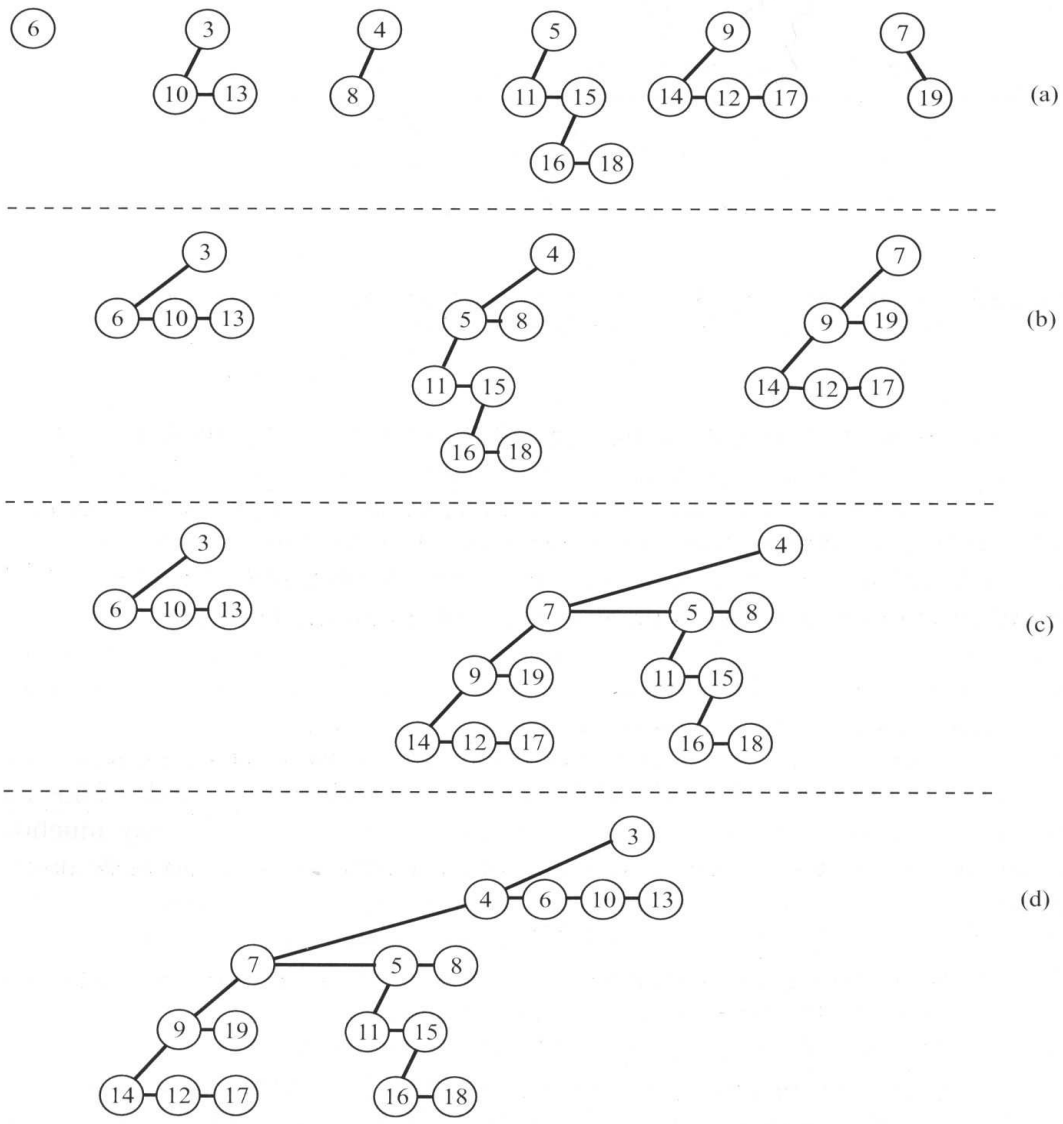


Figura 22.6 Recombinación de hermanos después de `eliminarMin`; en cada mezcla el árbol de mayor raíz se convierte en el hijo izquierdo del árbol de menor raíz: (a) los árboles obtenidos; (b) tras el primer paso; (c) tras la primera mezcla del segundo paso; (d) tras la segunda mezcla del segundo paso.

proceso, ya que puede suceder que la raíz del árbol que resulte tenga $c - 1$ hijos. Si esto sucede repetidas veces en una misma secuencia de operaciones, la rutina `eliminarMin` tendrá, por cada operación, un coste amortizado lineal, en lugar de logarítmico. El Ejercicio 22.8 pide al lector que construya una secuencia con estas características.

22.2.2 Implementación del montículo de emparejamientos

El atributo `prev` apunta o al padre del nodo o a un hermano izquierdo.

La funcionalidad de `MonticuloEmparejamientos` se describe en la Figura 22.7 y el esqueleto de dicha clase se muestra en la Figura 22.8. El nodo básico de un montículo de emparejamientos, `NodoEmparejamientos`, se puede ver en la

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase MonticuloEmparejamientos
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ***********************OPERACIONES PÚBLICAS************************
11 // void insertar( x )         --> Inserta x
12 // NodoEmparejamientos anyadirElem( x )
13                               --> Inserta x y devuelve su posición
14 // Comparable eliminarMin( ) --> Devuelve y elimina el menor elemento
15 // Comparable buscarMin( )   --> Devuelve el menor elemento
16 // boolean esVacio( )        --> Devuelve true si vacío; si no, false
17 // void vaciar( )            --> Elimina todos los elementos
18 // void reducirClave( NodoEmparejamientos p, nuevoValor )
19 //                           --> Disminuye el valor en el nodo p
20 // ***************************ERRORES********************************
21 // buscarMin y eliminarMin lanzan DesbordamientoInferior sobre el vacío
```

Figura 22.7 Descripción de la clase `MonticuloEmparejamientos`.

```
1  /**
2   * Implementa un montículo de emparejamientos.
3   * Permite la operación reducirClave, pero necesita del uso
4   * de anyadirElem en lugar de insertar. Siempre se mantiene
5   * el orden del montículo; no se permiten operaciones perezosas.
6   */
7  public class MonticuloEmparejamientos implements ColaPrioridad
8  {
9      public MonticuloEmparejamientos( )
10       { vaciar( ); }
11     public Comparable buscarMin( ) throws DesbordamientoInferior
12       { /* Figura 22.10 */ }
13     public void insertar( Comparable x )
14       { /* Figura 22.11 */ }
15     public NodoEmparejamientos anyadirElem( Comparable x )
16       { /* Figura 22.11 */ }
17     public Comparable eliminarMin( ) throws DesbordamientoInferior
18       { /* Figura 22.12 */ }
19     public void reducirClave( NodoEmparejamientos p, Comparable nuevoValor )
20                                                 throws ValorIlegal
21       { /* Figura 22.13 */ }
22     public boolean esVacio( )
23       { return tamanyoActual == 0; }
24     public void vaciar( )
25       { tamanyoActual = 0; raiz = null; }
26
27     private int   tamanyoActual;   // Número de elementos del montículo
28     private NodoEmparejamientos raiz;
29
30     private NodoEmparejamientos
31      comparaYEnlaza( NodoEmparejamientos primero, NodoEmparejamientos segundo )
32       { /* Figura 22.15 */ }
33     private NodoEmparejamientos
34      combinaHermanos( NodoEmparejamientos primerHermano )
35       { /* Figura 22.16 */ }
36 }
```

Figura 22.8 Esqueleto de la clase de los montículos de emparejamientos.

Figura 22.9, y consiste en un elemento y tres referencias. Dos de ellas son el hijo izquierdo y el hermano siguiente. La tercera referencia es `prev`, que apunta a su padre, si el nodo es el primer hijo, o a un hermano izquierdo, en caso contrario. La clase `NodoEmparejamientos` es pública, por lo que se pueden emplear referencias hacia sus objetos, pero sus atributos y el constructor son de acceso amistoso y no son accesibles a los usuarios de la clase.

`anyadirElem` devuelve una referencia que apunta al nuevo nodo que empleará `reducirClave`.

Los atributos de `NodoEmparejamientos` son, una referencia al nodo raíz (`raiz`) y un entero que representa el número de elementos almacenados en el montículo. Este último atributo se utiliza para simplificar la operación `reducirClave`. `MonticuloEmparejamientos` debe implementar el interfaz `ColaPrioridad`. Como resultado, el comportamiento de `insertar` no puede cambiar. Sin embargo, deseamos devolver una referencia al nuevo `NodoEmparejamientos` para utilizarla al invocar a `reducirClave`. Por este motivo implementamos `anyadirElem`. Esta rutina invoca a `insertar` y devuelve el nuevo `NodoEmparejamientos` recién creado. Para hacerlo, comparte el atributo `nuevoNodo` (que se describe en la implementación de `insertar`). No se incluye ningún método `mezclar`; su implementación es sencilla, y se deja como ejercicio al lector en el Ejercicio 22.10.

`buscarMin` se implementa en la Figura 22.10. Como el valor mínimo se encuentra en la raíz, esta rutina se codifica fácilmente. El método `insertar`, mostrado en la Figura 22.11, crea un árbol con un único nodo y lo combina con `raiz`

```
1  package EstructurasDatos;
2  import Soporte.*; import Soporte.Comparable;
3
4  /**
5   * Clase pública para utilizarla con MonticuloEmparejamientos. Sólo es
6   * pública para permitir devolver referencias a reducirClave.
7   * No tiene métodos públicos ni atributos públicos.
8   * @see MonticuloEmparejamientos
9   */
10 public class NodoEmparejamientos
11 {
12     /**
13      * Construye el NodoEmparejamientos.
14      * @param elElemento el valor almacenado en el nodo.
15      */
16     NodoEmparejamientos( Comparable elElemento )
17     {
18         dato              = elElemento;
19         izquierdo         = null;
20         siguienteHermano  = null;
21         prev              = null;
22     }
23
24         // Acceso amistoso; accesible por otras rutinas del paquete
25     Comparable dato;
26     NodoEmparejamientos izquierdo;
27     NodoEmparejamientos siguienteHermano;
28     NodoEmparejamientos prev;
29 }
```

Figura 22.9 Clase `NodoEmparejamientos`.

```
1   /**
2    * Busca el menor elemento de la cola de prioridad.
3    * @return el menor elemento.
4    * @exception DesbordamientoInferior si la cola de prioridad está vacía.
5    */
6   public Comparable buscarMin( ) throws DesbordamientoInferior
7   {
8       if( esVacio( ) )
9           throw new DesbordamientoInferior( "Montículo vacío" );
10      return raiz.dato;
11  }
```

Figura 22.10 Método `buscarMin` de la clase de los montículos de emparejamientos.

```
1   private NodoEmparejamientos nuevoNodo = null; // Último nodo insertado
2
3   /**
4    * Inserción en la cola de prioridad.
5    * Se permiten duplicados.
6    * @param x el elemento a insertar.
7    */
8   public void insertar( Comparable x )
9   {
10      nuevoNodo = new NodoEmparejamientos( x );
11
12      tamanyoActual++;
13      if( raiz == null )
14          raiz = nuevoNodo;
15      else
16          raiz = comparaYEnlaza( raiz, nuevoNodo );
17  }
18
19  /**
20   * Inserción en la cola de prioridad y devolución de un NodoEmparejamientos
21   * que puede emplear reducirClave.
22   * Se permiten duplicados.
23   * @param x el elemento a insertar.
24   * @return el nodo que contiene el elemento recién insertado.
25   */
26  public NodoEmparejamientos anyadirElem( Comparable x )
27  {
28      insertar( x );
29      return nuevoNodo;
30  }
```

Figura 22.11 Métodos `insertar` y `anyadirElem` de la clase de los montículos de emparejamientos.

para obtener un nuevo árbol. Como hemos mencionado anteriormente en esta misma sección, `anyadirElem` invoca a `insertar` y devuelve una referencia al nuevo nodo recién creado. Observe que debemos considerar el caso especial de insertar un elemento en un árbol vacío.

`eliminarMin` se implementa como una llamada a `combinaHermanos`.

En la Figura 22.12 se implementa el método `eliminarMin`. Si el montículo de emparejamientos está vacío, tenemos un error. En caso contrario, después de guardar el valor de la raíz, en la línea 12 llamamos a `combinaHermanos` para

```
1  /**
2   * Elimina el menor elemento de la cola de prioridad.
3   * @exception DesbordamientoInferior si la cola de prioridad está vacía.
4   */
5  public Comparable eliminarMin( ) throws DesbordamientoInferior
6  {
7      Comparable x = buscarMin( );
8
9      if( raiz.izquierdo == null )
10         raiz = null;
11     else
12         raiz = combinaHermanos( raiz.izquierdo );
13
14     tamanyoActual--;
15
16     return x;
17 }
```

Figura 22.12 Método `eliminarMin` de la clase de los montículos de emparejamientos.

combinar los subárboles de las raíces y enlazar el resultado a la nueva raíz. Si no hay subárboles que mezclar, en la línea 10 inicializamos `raiz` a `null`. Las rutinas `insertar` y `eliminarMin` ajustan adecuadamente `tamanyoActual`.

```
1  /**
2   * Cambia el valor del elemento guardado en el montículo
      de emparejamientos.
3   * @param p cualquier nodo devuelto por anyadirElem.
4   * @param nuevoValor el nuevo valor, que debe ser menor que
5   *     el valor almacenado ahora.
6   * @exception ValorIlegal si nuevoValor es mayor que
7   *     el valor almacenado ahora.
8   */
9  public void reducirClave( NodoEmparejamientos p, Comparable nuevoValor )
10                                                  throws ValorIlegal
11 {
12     if( p.dato.menorQue( nuevoValor ) )
13         throw new ValorIlegal( "ReducirClave incorrecto" );
14     p.dato = nuevoValor;
15     if( p != raiz )
16     {
17         if( p.siguienteHermano != null )
18             p.siguienteHermano.prev = p.prev;
19         if( p.prev.izquierdo == p )
20             p.prev.izquierdo = p.siguienteHermano;
21         else
22             p.prev.siguienteHermano = p.siguienteHermano;
23         p.siguienteHermano = null;
24
25         raiz = comparaYEnlaza( raiz, p );
26     }
27 }
```

Figura 22.13 Método `reducirClave` de la clase de los montículos de emparejamientos.

La rutina `reducirClave` se implementa en la Figura 22.13. Nótese que si el nuevo valor es mayor que el inicial, entonces podríamos destruir el orden del montículo. Pero no hay ningún modo de saberlo sin examinar antes los hijos. Como puede haber bastantes, hacerlo sería muy ineficiente. Como consecuencia, supondremos que es un error intentar aumentar la clave empleando `reducirClave`. (El Ejercicio 22.9 pide al lector que describa un algoritmo para `aumentarClave`.) Después de realizar este test, disminuimos el valor del nodo. Si dicho nodo es la raíz, ya hemos terminado. En caso contrario, eliminamos el nodo de la lista de hijos en la que se encuentra, por medio del código en las líneas 17 a 23. Después de hacer esto, combinamos el árbol resultante con la raíz.

Las dos rutinas restantes son `comparaYEnlaza`, que combina dos árboles, y `combinaHermanos`, que combina todos los hermanos dado el primero de ellos. La Figura 22.14 muestra cómo se combinan dos submontículos. El procedimiento se generaliza para permitir que el segundo de los submontículos pueda tener hermanos (esto es necesario para el segundo recorrido de la mezcla en dos pasadas). Como ya hemos mencionado en este capítulo, el submontículo con la raíz más grande se convierte en el hijo situado más a la izquierda del otro submontículo. El código se muestra en la Figura 22.15. Nótese que existen muchos puntos en los que se comprueba que una referencia no es `null` antes de acceder a su atributo `prev`. Esto sugiere que quizás sería útil tener un centinela `nodoNulo`, tal y como es costumbre en las implementaciones avanzadas de los árboles de búsqueda. Esto se deja como Ejercicio 22.13.

Por último, en la Figura 22.16 se implementa `combinaHermanos`. Empleamos el vector `vectorArboles` para almacenar los subárboles. En el peor de los casos, tendremos $N - 1$ hermanos, por lo que el vector emplea `tamanyoActual` para determinar su capacidad. Comenzamos por separar los subárboles y guardarlos en `vectorArboles`, empleando para ello el bucle de las líneas 18 a 24. Suponiendo que tenemos más de un hermano para combinar, en las líneas 28 a 30 realizamos el recorrido de izquierda a derecha. Terminamos el proceso de mezcla con el recorrido de derecha a izquierda en las líneas 41 a 43. Una vez que hemos terminado, el resultado se encuentra en la posición 0 del vector, y puede ser devuelto.

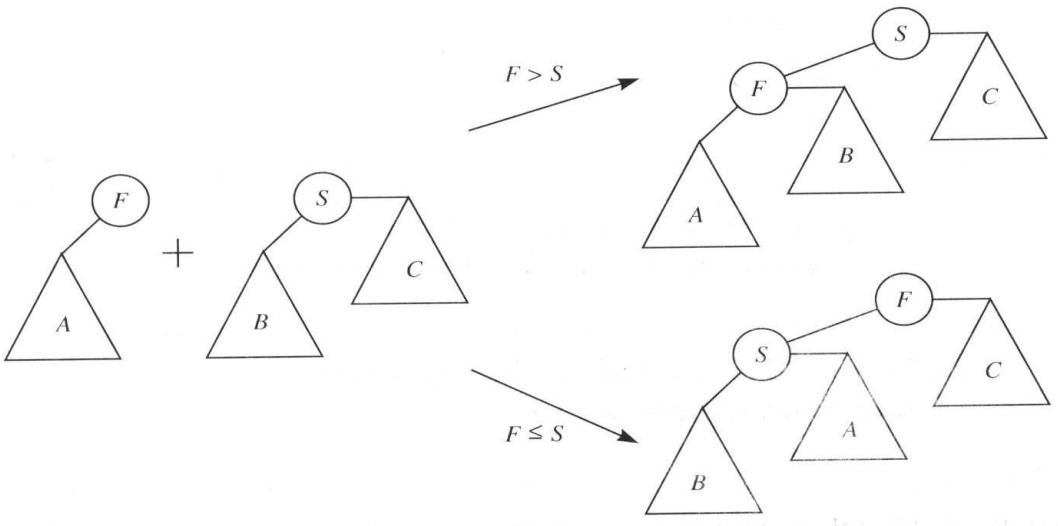


Figura 22.14 `comparaYEnlaza` combina dos árboles.

```
1  /**
2   * Método interno básico para mantener el
3   * orden en el montículo.
4   * Enlaza primero y segundo para mantener el orden.
5   * @param primero la raíz del árbol 1, no debe ser null.
6   *    primero.siguienteHermano DEBE ser null en la entrada.
7   * @param segundo la raíz del árbol 2, puede ser null.
8   * @return el resultado de la mezcla.
9   */
10 private NodoEmparejamientos
11 comparaYEnlaza( NodoEmparejamientos primero, NodoEmparejamientos segundo )
12 {
13     if( segundo == null )
14         return primero;
15
16     if( segundo.dato.menorQue( primero.dato ) )
17     {
18         // Enlaza primero como el hijo más izquierdo de segundo
19         segundo.prev = primero.prev;
20         primero.prev = segundo;
21         primero.siguienteHermano = segundo.izquierdo;
22         if( primero.siguienteHermano != null )
23             primero.siguienteHermano.prev = primero;
24         segundo.izquierdo = primero;
25         return segundo;
26     }
27     else
28     {
29         // Enlaza segundo como el hijo más izquierdo de primero
30         segundo.prev = primero;
31         primero.siguienteHermano = segundo.siguienteHermano;
32         if( primero.siguienteHermano != null )
33             primero.siguienteHermano.prev = primero;
34         segundo.siguienteHermano = primero.izquierdo;
35         if( segundo.siguienteHermano != null )
36             segundo.siguienteHermano.prev = segundo;
37         primero.izquierdo = segundo;
38         return primero;
39     }
40 }
```

Figura 22.15 Rutina `comparaYEnlaza`.

En la práctica, el almacenamiento dinámico (y la recogida de basura) del vector puede ser extremadamente costoso y en muchas ocasiones no será necesario. Puede ser mejor usar un objeto estático con un determinado tamaño inicial, expandiéndolo cuando sea necesario. Esto se propone al lector en el Ejercicio 22.11.

22.2.3 Aplicación: el algoritmo de Dijkstra para la obtención de caminos mínimos

`reducirClave` es una mejora en el algoritmo de Dijkstra cuando debe invocarse muchas veces.

Para mostrar cómo puede emplearse la operación `reducirClave`, rescribimos el algoritmo de Dijkstra estudiado en la Sección 14.3. Recordemos que manejamos una cola de prioridad de objetos de tipo `Camino`, ordenados según el atributo

```
1  /**
2   * Método interno que implementa la mezcla en dos pasadas.
3   * @param primerHermano la raiz;
4   *      se supone que no es null.
5   */
6  private NodoEmparejamientos
7   combinaHermanos( NodoEmparejamientos primerHermano )
8  {
9      if( primerHermano.siguienteHermano == null )
10         return primerHermano;
11
12         // Creación del vector
13     NodoEmparejamientos [ ] vectorArboles =
14                         new NodoEmparejamientos[ tamanyoActual ];
15
16         // Almacenamiento de los árboles en el vector
17     int numeroHermanos = 0;
18     for( ; primerHermano != null; numeroHermanos++ )
19     {
20         vectorArboles[ numeroHermanos ] = primerHermano;
21         primerHermano.prev.siguienteHermano = null;  // rompe los enlaces
22         primerHermano = primerHermano.siguienteHermano;
23     }
24     vectorArboles[ numeroHermanos ] = null;
25
26     // Combinación de pares de subárboles, de izquierda a derecha
27     int i = 0;
28     for( ; i+1 < numeroHermanos; i += 2 )
29         vectorArboles[ i ] = comparaYEnlaza( vectorArboles[ i ],
30                                              vectorArboles[ i + 1 ] );
31
32         // j es el resultado del último comparaYEnlaza.
33         // Si el número de árboles es impar, coge el último de ellos.
34     int j = i - 2;
35     if( j = = numeroHermanos - 3 )
36         vectorArboles[ j ] = comparaYEnlaza( vectorArboles[ j ],
37                                              vectorArboles[ j + 2 ] );
38
39         // Recorrido de derecha a izquierda, combinando el último
40         // con el siguiente al último. El resultado es el nuevo último.
41     for( ; j >= 2; j -= 2 )
42         vectorArboles[ j - 2 ] = comparaYEnlaza( vectorArboles[ j - 2 ],
43                                                  vectorArboles[ j ] );
44
45     return vectorArboles[ 0 ];
46 }
```

Figura 22.16 El núcleo del algoritmo de mezcla; implementa una mezcla en dos pasadas para combinar todos los hermanos, a partir del primero de ellos.

`dist`. En cada momento, por cada vértice del grafo, necesitamos un solo objeto de tipo `Camino` en la cola de prioridad, pero por comodidad mantenemos varios. En esta sección, retocaremos el código de modo que, cuando un vértice *w* vea reducida su distancia, se buscará su posición en la cola de prioridad, realizándose una operación `reducirClave` sobre su correspondiente objeto `Camino`.

El código modificado se muestra en la Figura 22.17. Los cambios son relativamente pequeños. En primer lugar, en la línea 5 declaramos que `mm` es un montículo de emparejamientos en lugar de un montículo binario. Manejamos también un vector de referencias, `posicionesMonticulo`, que apuntan a nodos de

```
1  // Algoritmo de Dijkstra empleando montículos de emparejamientos
2  private boolean dijkstraMezcla( int nodoInicio )
3  {
4    int v, w;
5    MonticuloEmparejamientos mm = new MonticuloEmparejamientos( );
6    Camino vrec;
7    NodoEmparejamientos [ ] posicionesMonticulo;
8
9    limpiarDatos( );
10   posicionesMonticulo = new NodoEmparejamientos[ numVertices ];
11   for ( int i = 0; i < numVertices; i++ )
12        posicionesMonticulo[ i ] = null;
13   tabla[ nodoInicio ].dist = 0;
14   mm.insertar( new Camino( nodoInicio, 0 ) );
15
16   try
17   {
18      while( !mm.esVacio( ) )
19      {
20         vrec = (Camino) mm.eliminarMin( );
21         v = vrec.dest;
22
23         ListaIter p = new ListaEnlazadaIter( tabla[v].ady );
24         for( ; p.estaDentro( ); p.avanzar( ) )
25         {
26             w       = ( (Arista)p.recuperar( ) ).dest;
27             int cvw = ( (Arista)p.recuperar( ) ).coste;
28
29             if( cvw < 0 )
30                 return false;
31             if( tabla[ w ].dist > tabla[ v ].dist+cvw )
32             {
33                 tabla[ w ].dist = tabla[ v ].dist+cvw;
34                 tabla[ w ].ant = v;
35
36                 Camino nuevoValor = new Camino( w, tabla[ w ].dist );
37                 if( posicionesMonticulo[ w ] == null )
38                     posicionesMonticulo[ w ] =
39                                     mm.anyadirElem( nuevoValor );
40                 else
41                     mm.reducirClave( posicionesMonticulo[ w ],
42                                      nuevoValor );
43             }
44         }
45      }
46   }
47   catch( Exception e ) { }   // Esto no puede suceder
48   return true;
49 }
```

Figura 22.17 Algoritmo de Dijkstra que emplea montículos de emparejamientos y la rutina `reducirClave`.

montículos de emparejamientos. Inicialmente todas las referencias son `null` (líneas 11 y 12). Cada vez que se inserta un elemento en el montículo de emparejamientos, modificamos el vector `posicionesMonticulo`. Esto se hace en la línea 38[3]. El algoritmo se ha simplificado. Ahora llamamos a `eliminarMin` en tanto y cuando el montículo no esté vacío, en lugar de hacerlo repetidas veces hasta que aparece un nodo no consultado. Como consecuencia, ya no es necesario el atributo `eliminado`. Compare las líneas 18 a 21 con el correspondiente código de la Figura 14.29. Todo lo que queda por comentar son las actualizaciones tras la línea 31, que indican que se ha realizado un cambio. Si el vértice no ha sido insertado nunca en la cola de prioridad, lo insertamos por primera vez, actualizando el vector `posicionesMonticulo`. En caso contrario, en la línea 41 invocamos la rutina `reducirClave`.

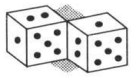
Que la implementación del algoritmo de Dijkstra que emplea montículos binarios sea más rápida que la que emplea montículos de emparejamientos depende de varios factores. Un estudio, detallado en las referencias, indica que el montículo de emparejamientos es ligeramente mejor que el montículo binario cuando ambos se implementan cuidadosamente. Los resultados dependen en gran medida de los detalles de codificación y del número de llamadas a `reducirClave`. Son necesarios estudios más profundos para decidir cuándo son más adecuados en la práctica los montículos de emparejamientos.

Resumen

Este capítulo describe dos estructuras de datos que permiten la mezcla de sus elementos, y son eficientes en términos de coste amortizado: el montículo sesgado y el montículo de emparejamientos. Ambas son de sencilla implementación, ya que no tienen ninguna restricción estructural. El montículo de emparejamientos parece tener un cierto interés en la práctica, pero su análisis completo continúa siendo un intrincado problema abierto.

El siguiente capítulo, que es ya el último de este texto, describe una estructura de datos que se emplea para manipular conjuntos disjuntos y cuyo análisis amortizado también es interesante.

Elementos del juego



mezcla en dos pasadas El orden en el que se combinan los montículos de emparejamientos es importante. El algoritmo más sencillo para hacerlo es la mezcla en dos pasadas, en la que los montículos se mezclan en pares, de izquierda a derecha. Después se realiza un nuevo recorrido de derecha a izquierda para concluir la mezcla.

montículo de emparejamientos Árbol *M*-ario con ordenación de montículos y sin restricciones estructurales. Su análisis no está completo, pero parece comportarse bastante bien en la práctica.

[3] El `nodoInicio` no estará incluido en el montículo de emparejamientos, por lo que en la línea 14 no empleamos `anyadirElem`.

montículo sesgado Árbol binario con ordenación de montículos. No tiene ninguna condición de equilibrio, pero todas sus operaciones tienen un coste amortizado logarítmico.

Errores comunes

1. Una implementación recursiva de los montículos sesgados no puede emplearse en la práctica, ya que la profundidad de la recursión puede ser lineal.
2. En la clase de los montículos sesgados debemos ser cuidadosos para actualizar adecuadamente las referencias `prev`.
3. Deben realizarse a lo largo del código de los montículos de emparejamientos ciertos tests para comprobar que las referencias no son `null`.
4. Tras la realización de una mezcla, dos montículos de emparejamientos no deben compartir ningún nodo.

En Internet

La clase de los montículos de emparejamientos está disponible en el directorio **DataStructures**. La Figura 22.17 forma parte de la clase `Grafo` explicada en el Capítulo 14 (**Graph.java** está en el directorio **Part3**).

PairHeap.java	Contiene la implementación de los montículos de emparejamientos. Es la versión inglesa de la clase `MonticuloEmparejamientos`.
PairNode.java	Contiene la definición de los nodos de los montículos de emparejamientos. Es la versión inglesa de `NodoEmparejamientos`.

Ejercicios

Cuestiones breves

22.1. Muestre el montículo sesgado resultante a partir de las siguientes secuencias de inserción:

a) 1, 2, 3, 4, 5, 6, 7
b) 4, 3, 5, 2, 6, 7, 1

22.2. Muestre el montículo de emparejamientos resultante a partir de las siguientes secuencias de inserción:

a) 1, 2, 3, 4, 5, 6, 7
b) 4, 3, 5, 2, 6, 7, 1

22.3. Muestre el resultado de dos operaciones `eliminarMin` sobre cada uno de los montículos de los Ejercicios 22.1 y 22.2.

Problemas teóricos

22.4. Dé una secuencia de operaciones que conduzca a una operación `mezclar` que requiera tiempo lineal para demostrar que la cota amortizada logarít-

mica de las operaciones del montículo sesgado no es una cota para el caso peor.

22.5. Muestre que las operaciones `reducirClave` y `aumentarClave` pueden implementarse en los montículos sesgados con un coste amortizado logarítmico.

22.6. Describa un algoritmo `arreglarMonticulo` de coste lineal en la clase de los montículos sesgados.

22.7. Muestre que guardando el tamaño del camino derecho de cada nodo del árbol podemos imponer una condición de equilibrio que permite que todas las operaciones tengan coste logarítmico en el caso peor. Esta estructura de datos se conoce como *montículo izquierdista*.

22.8. Muestre que no es adecuado emplear una pila en la implementación de la operación `combinaHermanos` de la clase de los montículos de emparejamientos; para ello, construya una secuencia de coste amortizado lineal por operación.

22.9. Describa cómo implementar `aumentarClave` en la clase de los montículos de emparejamientos.

Problemas prácticos

22.10. Incluya un método público `mezclar` en la clase de los montículos de emparejamientos. Asegúrese de que cada nodo aparece en un único árbol.

22.11. Use un vector que se expanda siempre que sea necesario, para almacenar `vectorArboles` en `combinaHermanos`.

Prácticas de programación

22.12. Implemente una versión no recursiva del montículo sesgado.

22.13. Implemente los montículos de emparejamientos empleando un centinela `nodoNulo`.

22.14. Implemente un algoritmo que utilice colas para `combinaHermanos` y compare su eficiencia con el algoritmo de dos pasadas codificado en la Figura 22.16.

22.15. Si la operación `reducirClave` no está permitida, las referencias a los padres no son necesarias. Implemente el montículo de emparejamientos sin dichas referencias y compare su eficiencia con la del montículo binario y/o el montículo sesgado y/o el árbol de ensanchamiento.

22.16. Diseñe un applet que muestre el comportamiento de los montículos sesgados o de los montículos de emparejamientos.

Bibliografía

El *montículo izquierdista* [1] fue la primera cola de prioridad combinable eficientemente. Es la variación sugerida en el Ejercicio 22.7, pensada para el caso peor de los montículos sesgados. Los montículos sesgados se describen en [5]. Dicho artículo incluye las soluciones de los Ejercicios 22.4 y 22.5.

[2] describe el montículo de emparejamientos y demuestra que cuando se emplea la mezcla en dos pasadas, el coste amortizado de todas las operaciones es a lo

sumo logarítmico, aunque aún no se sabe si esta cota es lo suficientemente ajustada. En particular, en [6] puede encontrarse alguna evidencia en el sentido de que el coste amortizado de todas las operaciones, excepto `eliminarMin`, podría ser constante, mientras que el coste de `eliminarMin` sí parece ser logarítmico. De este modo, cualquier secuencia de D `eliminarMin` y otras I operaciones es $O(I + D \log N)$. Sin embargo, en [4] aparecen otras evidencias que parecen sugerir que esto no es así. Una estructura de datos que sí consigue con seguridad las cotas deseadas, aunque es demasiado complicada para ser usada en la práctica, es el *montículo de Fibonacci* [3]. Parece que el montículo de emparejamientos podría ser la alternativa práctica al buen comportamiento teórico del montículo de Fibonacci. Los montículos izquierdistas y de Fibonacci se discuten en [7].

En [4] puede encontrarse una comparación entre distintas colas de prioridad en el contexto del problema del árbol de cobertura mínimo (discutido en la Sección 23.2.1), empleándose un método muy similar al algoritmo de Dijkstra.

1. C. A. Crane, «Linear Lists and Priority Queues as Balanced Binary Trees», *Technical Report STAN-CS-72-259,* Computer Science Department, Stanford University, Palo Alto, Calif. (1972).

2. M. L. Fredman, R. Sedgewick, D. D. Sleator, y R. E. Tarjan, «The Pairing Heap: A New Form of Self-adjusting Heap», *Algorithmica* **1** (1986), 111-129.

3. M. L. Fredman y R. E. Tarjan, «Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms», *Journal of the ACM* **34** (1987), 596-615.

4. B. M. E. Moret y H. D. Shapiro, «An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree», *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400-411.

5. D. D. Sleator y R. E. Tarjan, «Self-adjusting Heaps», *SIAM Journal on Computing* **15** (1986), 52-69.

6. J. T. Stasko y J. S. Vitter, «Pairing Heaps: Experiments and Analysis», *Communications of the ACM* **32** (1987), 234-249.

7. M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, Benjamin/Cummings Publishing Co., Redwood City, Calif. (1994).