CAPÍTULO

20 Una cola de prioridad: el montículo binario

La cola de prioridad es una estructura de datos fundamental que sólo permite el acceso al elemento mínimo. En este capítulo se estudia una implementación de la cola de prioridad, el elegante *montículo binario*. El montículo binario soporta la inserción de nuevos elementos y la eliminación del elemento mínimo en tiempo logarítmico en el caso peor. Solamente usa un vector y es muy simple de implementar.

En este capítulo estudiaremos:

- Las propiedades básicas de los montículos binarios.
- Cómo se pueden codificar las operaciones `insertar` y `eliminarMin` para que se ejecuten en tiempo logarítmico.
- Un algoritmo lineal de construcción de montículos.
- Una implementación en Java.
- Un algoritmo de ordenación fácilmente implementable: el *método del montículo* (*heapsort*), que se ejecuta en un tiempo $O(N \log N)$ sin requerir ninguna memoria adicional.
- Cómo se pueden utilizar los montículos para implementar la ordenación externa.

20.1 Ideas básicas

Usando listas enlazadas o vectores, siempre hay alguna de las operaciones que exije tiempo lineal.

Como ya se indicó en la Sección 6.8, la cola de prioridad soporta acceso y eliminación del elemento mínimo, usando, respectivamente, `buscarMin` y `eliminarMin`. Podríamos usar una lista enlazada, llevando a cabo las inserciones por delante de la lista en tiempo constante, pero entonces encontrar y/o eliminar el elemento mínimo requeriría un recorrido lineal de la lista. Alternativamente, podríamos obligar a que la lista estuviera siempre ordenada, lo cual haría que el acceso y la eliminación del elemento mínimo fueran más baratos, pero entonces las inserciones serían lineales.

Otra forma de implementar las colas de prioridad consiste en usar un árbol binario de búsqueda, en el que las dos operaciones mencionadas requieren, en media, un tiempo $O(N \log N)$. Sin embargo, ésta es habitualmente una mala elección,

Los árboles binarios de búsqueda no equilibrados no tienen un buen caso peor. Los árboles de búsqueda equilibrados exigen mucho trabajo.

pues normalmente la entrada no es lo suficientemente aleatoria. Podríamos usar un árbol binario de búsqueda equilibrado, pero las estructuras estudiadas en el Capítulo 18 son tediosas de implementar y en la práctica resultan poco eficientes. (Sin embargo, en el Capítulo 21 se estudia una estructura de datos, el árbol de ensanchamiento, que ha demostrado ser una buena alternativa en algunas situaciones.)

La cola de prioridad tiene propiedades intermedias entre las de las colas y las de los árboles binarios de búsqueda.

Puesto que la cola de prioridad soporta solamente algunas de las operaciones de los árboles de búsqueda, no debería resultar más costosa de implementar que éstos. Por otra parte, la cola de prioridad es más potente que una simple cola, porque podemos implementar una cola usando una cola de prioridad de la siguiente manera: insertamos cada elemento junto con una indicación del momento de inserción, con lo que una operación `eliminarMin` en base al momento de inserción, implementa en realidad una operación `quitarPrimero`. En consecuencia, es razonable esperar que la implementación tenga propiedades intermedias entre las de las colas y las de los árboles de búsqueda. Esto último se consigue con el montículo binario, el cual:

- se puede implementar usando un simple vector (al igual que la cola),
- soporta `insertar` y `eliminarMin` en tiempo $O(\log N)$ en el caso peor (un término medio entre el rendimiento de la cola y el del árbol binario de búsqueda), y
- soporta `insertar` en tiempo constante en el caso medio y `buscarMin` en tiempo constante en el caso peor (como la cola).

El *montículo binario* es el método clásico utilizado para implementar las colas de prioridad.

Al igual que los árboles de búsqueda equilibrados del Capítulo 18, el montículo binario tiene dos propiedades, una relacionada con su estructura y otra relacionada con la ordenación. Y al igual que los árboles de búsqueda equilibrados, una operación sobre un montículo binario puede destruir alguna de estas propiedades, por lo que la aplicación de una operación sobre un montículo binario no debe terminar hasta que ambas propiedades se restablezcan, lo cual, afortunadamente, resulta fácil de conseguir. (En este capítulo, el término *montículo* se refiere siempre al montículo binario.)

20.1.1 Propiedad estructural

El montículo es un *árbol binario completo,* lo cual permite implementarlo usando un vector, y garantiza que su profundidad es logarítmica.

La única estructura que proporciona cotas de tiempo logarítmicas es el árbol, por lo que parece natural que el montículo organice sus datos como tal. Como queremos que la cota logarítmica se mantenga en el caso peor, el árbol debería estar equilibrado.

Un *árbol binario completo* es un árbol completamente lleno, con la excepción del nivel inferior, que debe llenarse de izquierda a derecha. La característica distintiva del árbol binario completo es que no faltan nodos en el árbol. En la Figura 20.1 se muestra un ejemplo de árbol binario completo con diez elementos. Si el nodo J fuera hijo derecho de E, el árbol ya no sería completo, pues faltaría un nodo.

Los árboles completos tienen algunas propiedades que son muy útiles. En primer lugar, su altura (longitud del camino más largo) es a lo sumo $\lfloor \log N \rfloor$. Esto es así, porque un árbol completo de altura H tiene entre 2^H y $2^{H+1} - 1$ nodos. Esto significa que podemos esperar un comportamiento logarítmico en el caso peor, si restringimos los cambios en la estructura a los caminos de la raíz a una hoja.

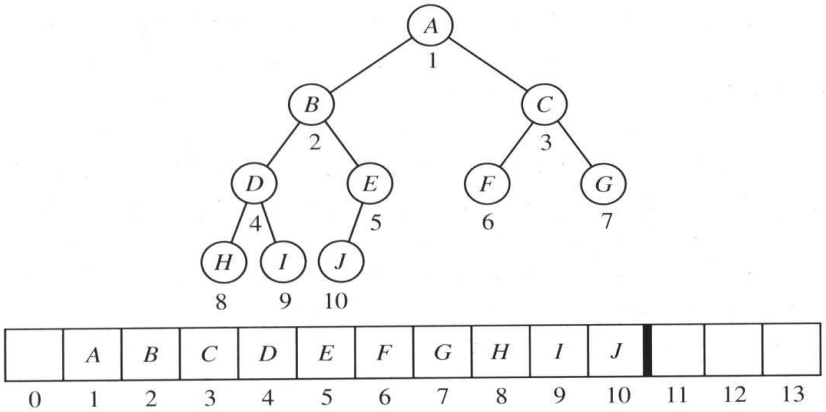


Figura 20.1 Árbol binario completo y su representación mediante un vector.

En segundo lugar, pero no por ello menos importante, en un árbol binario completo no se necesitan referencias a los hijos `izquierdo` y `derecho`, ya que, como se muestra en la Figura 20.1, podemos representarlo sin ambigüedad almacenando su recorrido por niveles en un vector. Colocamos la raíz en la posición 1 (la posición 0 se deja libre por una razón que se explicará más adelante). Necesitamos también mantener un entero que nos dice cuántos nodos hay actualmente en el árbol. Entonces, dado un elemento situado en la posición i del vector, sabemos que su hijo izquierdo se encuentra en la posición $2i$, aunque si dicha cantidad sobrepasa el número de nodos en el árbol, entonces sabemos que no existe tal hijo izquierdo. Análogamente, el hijo derecho se encuentra inmediatamente a continuación del hijo izquierdo, es decir, en la posición $2i + 1$, de nuevo comprobándose su existencia mediante una comparación con el número de nodos del árbol. Finalmente, su padre se encuentra en la posición $\lfloor i/2 \rfloor$. Observemos que todos los nodos, excepto la raíz, tienen un padre. Si la raíz tuviera un padre, el cálculo lo situaría en la posición 0. Por eso reservamos dicha posición para colocar un elemento falso que sirva como padre de la raíz, lo que simplificará algunas operaciones.

El padre se encuentra en la posición $\lfloor i/2 \rfloor$, el hijo izquierdo en la $2i$ y el derecho en la $2i + 1$.

El uso de un vector para almacenar un árbol recibe el nombre de *representación implícita*. Como resultado de usar esta implementación, no solamente no son necesarias las referencias a los hijos, sino que además las operaciones de recorrido de los árboles son muy simples y muy probablemente lo bastante rápidas en la mayoría de los computadores. El montículo consistirá en un vector de objetos y un entero que representa el tamaño actual del montículo.

El uso de un vector para almacenar un árbol recibe el nombre de *representación implícita*.

En este capítulo, para hacer los algoritmos más sencillos de visualizar, los montículos se dibujan como árboles, aunque la implementación de estos árboles utilizará un vector. No usamos la representación implícita para todos los árboles de búsqueda, ya que esto supondría algunos problemas adicionales, que se esbozan en el Ejercicio 20.8.

20.1.2 Propiedad de ordenación de los montículos

La propiedad que permite realizar rápidamente operaciones sobre los montículos es la *propiedad de ordenación de los montículos*. Puesto que queremos encontrar rápidamente el elemento mínimo, tendría sentido que el elemento más pequeño se

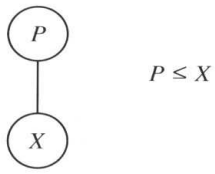


Figura 20.2 Propiedad de ordenación de los montículos.

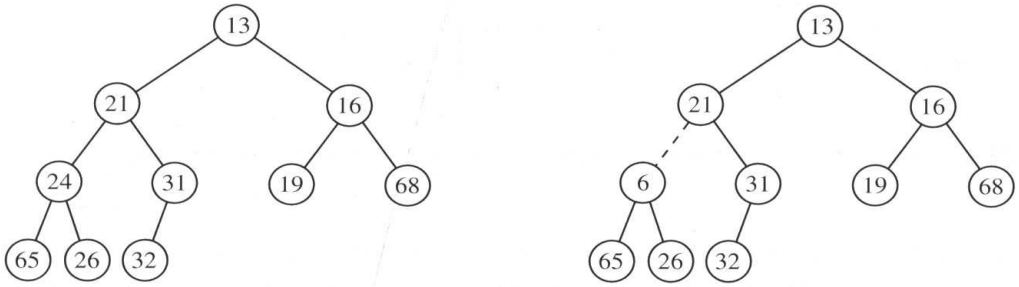


Figura 20.3 Dos árboles completos (sólo el de la izquierda es un montículo).

La *propiedad de ordenación de los montículos* establece que el elemento almacenado en el padre de un nodo nunca es mayor que el elemento almacenado en dicho nodo.

encontrara situado en la raíz. Si consideramos que cualquier subárbol también debería ser un montículo (recursivamente), entonces cada nodo debe ser menor que todos sus descendientes. Aplicando este razonamiento obtenemos la propiedad de ordenación de los montículos.

PROPIEDAD DE ORDENACIÓN DE UN MONTÍCULO
En un montículo, para cada nodo X con padre P, se cumple que el dato en P es menor o igual que el dato en X.

El padre de la raíz puede almacenarse en la posición 0 y podemos darle un valor de menos infinito.

En la Figura 20.2 se ilustra la propiedad de ordenación de los montículos. En la Figura 20.3, el árbol de la izquierda es un montículo, pero el de la derecha no lo es (la línea punteada muestra el lugar en el que se viola la propiedad). Observe que la raíz no tiene padre. En la representación implícita, podemos colocar el valor $-\infty$ en la posición 0 para eliminar este caso especial cuando implementamos los montículos. Por la propiedad de ordenación, vemos que el menor elemento siempre se puede encontrar en la raíz, de modo que `buscarMin` se puede ejecutar en tiempo constante. Los *montículos maximales* soportan acceso al máximo en lugar de al mínimo. Para implementarlos basta con hacer unos mínimos cambios.

20.1.3 Operaciones permitidas

Ahora que hemos fijado la representación, podemos empezar a escribir código. Ya sabemos que nuestro montículo soporta las operaciones básicas `insertar`, `buscarMin` y `eliminarMin`, y también las habituales `esVacio` y `vaciar`. Añadiremos además algunas operaciones más. La Figura 20.4 muestra la sección pública de la clase e ilustra algunos de estos métodos adicionales. La Figura 20.5 contiene la parte privada de la clase.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase MonticuloBinario
7  //
8  // CONSTRUCCIÓN: con un centinela con un valor menos infinito
9  //
10 // ******************OPERACIONES PÚBLICAS*******************
11 // void insertar( x )        --> Inserta x
12 // Comparable eliminarMin( ) --> Elimina y devuelve el menor
13 // Comparable buscarMin( )   --> Devuelve el menor elemento
14 // boolean esVacia( )        --> Devuelve true si está vacío
15 // void vaciar( )            --> Elimina todos los elementos
16 // void introducir( x )      --> Inserta x (perezosamente)
17 // ************************ERRORES**************************
18 // buscarMin y eliminarMin lanzan DesbordamientoInferior si vacío
19
20 /**
21  * Implementa un montículo binario.
22  * Permite inserción perezosa y proporciona un método lineal de
23  * construcción de montículos.
24  * Observe que todas las comparaciones están basadas en compara.
25  */
26 public class MonticuloBinario implements ColaPrioridad
27 {
28     public MonticuloBinario( Comparable infNeg )
29       { /* Figura 20.7 */ }
30     public void insertar( Comparable x )
31       { /* Figura 20.12 */ }
32     public void introducir( Comparable x )
33       { /* Figura 20.12 */ }
34     public Comparable buscarMin( ) throws DesbordamientoInferior
35       { /* Figura 20.6 */ }
36     public Comparable eliminarMin( ) throws DesbordamientoInferior
37       { /* Figura 20.16 */ }
38     public boolean esVacia( )
39       { return tamActual == 0; }
40     public void vaciar( )
41       { tamActual = 0; }
```

Figura 20.4 Esqueleto de la clase `MonticuloBinario` (parte 1: sección pública).

```
42     private int tamActual;     // Número de elementos del montículo
43     private boolean ordenado; // True si cumple propiedad ordenación
44     private Comparable [ ] vector; // El vector del montículo
45     private static final int CAPACIDAD = 11;
46
47     private void obtenerVector( int nuevoTamMax )
48       { /* Figura 20.8 */ }
49     private void comprobarTam( )
50       { /* Figura 20.11 */ }
51     private void hundir( int hueco )
52       { /* Figura 20.17 */ }
53     private void arreglarMonticulo( )
54       { /* Figura 20.19 */ }
55 }
```

Figura 20.5 Esqueleto de la clase `MonticuloBinario` (parte 2: sección privada).

El método `introducir` añade un elemento, pero al contrario que `insertar`, no garantiza que se mantenga la propiedad de ordenación del montículo. Es útil cuando queremos añadir muchos elementos antes de acceder al elemento mínimo.

El método `arreglarMonticulo` restablece el orden en el montículo. Debido a que `arreglarMonticulo` es costoso, su uso solamente está justificado si se realizan muchas operaciones `introducir` entre dos accesos al elemento mínimo.

Comenzamos examinando los métodos públicos. El constructor se declara en la línea 28, mientras que el método `insertar` aparece en la 30. Añade un nuevo elemento `x` al montículo, realizando las operaciones necesarias para mantener la propiedad de ordenación del montículo. En la línea 32 se añade un método nuevo, `introducir`, el cual añade un nuevo elemento `x` al montículo, pero no garantiza que se mantenga la propiedad de ordenación del montículo. ¿Para qué querríamos `introducir` un elemento en el montículo? La respuesta es que `introducir` es un método mucho más sencillo de implementar que `insertar`. Por supuesto, tan pronto como realizamos una operación `introducir` , ya no tenemos un montículo, por lo que no debemos esperar que funcionen correctamente las operaciones `buscarMin` o `eliminarMin`. Por ejemplo, si queremos `introducir` un nuevo mínimo, no tenemos garantías de que se situará en la raíz (de hecho es de esperar que no lo haga).

En consecuencia, usar `introducir` parece una idea tonta. Pero no lo es, porque hay muchas aplicaciones en las que podemos tener que `introducir` muchos elementos antes de que se produzca la siguiente llamada a `eliminarMin`. En tal caso, no necesitamos mantener el montículo ordenado hasta que se produzca la llamada a `eliminarMin`. La operación `arreglarMonticulo`, declarada en la línea 53, reestablece el orden en el montículo en tiempo lineal, sin importar lo desordenado que se encuentre. Luego, por ejemplo, si queremos colocar N elementos en el montículo antes de ejecutar la primera operación `eliminarMin`, es más eficiente ejecutar N operaciones `introducir` y una `arreglarMonticulo` que N operaciones `insertar`. Sin embargo, no usaremos `introducir` en sustitución de sólo unos pocos `insertar`. El método `arreglarMonticulo` es privado, y su uso es transparente.

El resto de las operaciones son como esperamos. En la línea 34 se declara `buscarMin`, que devuelve el elemento mínimo. El método `eliminarMin`, en la línea 36, devuelve y elimina el elemento mínimo. También se proporcionan, en las líneas 38 a 41, los habituales `esVacio` y `vaciar`.

Una vez procesamos una operación `introducir`, el usuario no puede realizar una operación `buscarMin` ni `eliminarMin` sin realizar antes una operación `arreglarMonticulo`. En lugar de confiar en que el usuario haga esta llamada, mantenemos un atributo `ordenado`, que toma el valor `false` cuando una aplicación de `introducir` produce una violación del orden en el montículo. La Figura 20.6 muestra que `buscarMin` comprueba si se satisface la propiedad de ordenación de montículo, y en caso de no ser así, produce automáticamente una llamada a `arreglarMonticulo` para restablecer la propiedad.

```
 1  /**
 2   * Busca el elemento más pequeño en la cola de prioridad.
 3   * @return el elemento más pequeño.
 4   * @exception DesbordamientoInferior si la cola está vacía.
 5   */
 6  public Comparable buscarMin( ) throws DesbordamientoInferior
 7  {
 8      if( esVacia( ) )
 9          throw new DesbordamientoInferior ( "Montículo vacío" );
10      if( !ordenado )
11          arreglarMonticulo( );
12      return vector[ 1 ];
13  }
```

Figura 20.6 Rutina `buscarMin`.

```
1  /**
2   * Construye el montículo binario.
3   * @param infNeg un valor menor o igual que todos los demás.
4   */
5  public MonticuloBinario( Comparable infNeg )
6  {
7      tamActual = 0;
8      ordenado = true;
9      obtenerVector( CAPACIDAD );
10     vector[ 0 ] = infNeg;
11 }
```

Figura 20.7 Constructor para `MonticuloBinario`.

```
1  /**
2   * Método privado para reservar memoria para el vector.
3   * Incluye una posición extra para el centinela.
4   * @param nuevoTamMax la capacidad del montículo.
5   */
6  private void obtenerVector( int nuevoTamMax )
7  {
8      vector = new Comparable[ nuevoTamMax + 1 ];
9  }
```

Figura 20.8 Reserva de memoria para el vector (método llamado por el constructor y durante la duplicación del tamaño del vector).

El resto de atributos son los que ya sospechábamos: un vector al que se asigna memoria dinámicamente y un entero que almacena el tamaño actual del montículo. El constructor, mostrado en la Figura 20.7, inicializa los atributos, asignando `true` a `ordenado` y reservando después memoria para el vector mediante una llamada a `obtenerVector`, en la línea 9. Después, en la línea 10, se coloca el centinela en la posición 0. El usuario debe proporcionar al constructor `infNeg` como un parámetro. El cuerpo de `obtenerVector`, mostrado en la Figura 20.8, consta de una sola línea.

20.2 Implementación de las operaciones básicas

La propiedad de ordenación parece buena hasta el momento, puesto que permite un rápido acceso al elemento mínimo. Ahora debemos mostrar que podemos soportar eficientemente las operaciones `insertar` y `eliminarMin`, haciéndolo en un tiempo logarítmico. Es fácil, conceptual y prácticamente, realizar las dos operaciones requeridas. Todo el trabajo implicado se encuentra en mantener la propiedad de ordenación del montículo.

La inserción se implementa creando un hueco en la siguiente posición disponible, para *reflotarlo* después hasta que el nuevo elemento se pueda colocar en él sin producir una violación de la ordenación del montículo con respecto a su padre.

20.2.1 `insertar`

Para insertar un elemento X en el montículo, primero debemos añadir un nodo al árbol. La única opción es crear un hueco en la siguiente posición disponible; en caso contrario, el árbol no sería completo y violaríamos la propiedad estructural.

Si X se puede colocar en ese hueco sin violar la propiedad de ordenación del montículo, lo colocamos ahí y hemos terminado. Si no, desplazamos el elemento situado en el padre del nodo a dicho nodo, moviendo así el hueco hacia la raíz. Continuamos con este proceso hasta que se pueda colocar X en el hueco. La Figura 20.9 muestra que para insertar 14, creamos un hueco en la siguiente posición disponible. Como insertar 14 en el hueco, produciría una violación del orden, colocamos 31 en el hueco. En la Figura 20.10 se continúa con esta estrategia hasta encontrar la posición correcta para 14.

El método `insertar` usa el centinela para evitar el caso especial de la raíz.

Llamamos a esta estrategia general *reflotamiento*, porque el nuevo elemento es reflotado hasta encontrar la posición adecuada. La rutina, `comprobarTam`, mostrada en la Figura 20.11, duplicará el tamaño del vector cuando ello sea necesario. La Figura 20.12 muestra las rutinas que añaden elementos al montículo. La rutina `introducir` es bien corta; simplemente añade el nuevo elemento `x` en la siguiente posición disponible. Si se produce una violación del orden, se le asigna `false` a `ordenado`. El método `insertar` implementa el reflotamiento usando un bucle. La instrucción de la línea 30 incrementa el tamaño actual y coloca el hueco en el nodo que acabamos de añadir. Ejecutamos el bucle de la línea 31 mientras el elemento del nodo padre sea mayor que `x`. La línea 32 mueve el elemento del padre al hueco tras lo cual la tercera componente del bucle `for` reflota el hueco hacia el padre. Cuando el bucle termina, la línea 33 coloca `x` en el hueco. El centinela situado en la posición O garantiza la terminación del bucle `for`.

La inserción requiere un tiempo constante en el caso medio y logarítmico en el caso peor.

El tiempo necesario para realizar la inserción podría ser $O(\log N)$ si el elemento a insertar fuera el nuevo mínimo, porque en tal caso se reflotaría hasta la raíz. En media, el reflotamiento termina pronto: se ha demostrado que en media se requieren 2,6 comparaciones para llevar a cabo una inserción, de modo que en media `insertar` mueve un elemento 1,6 niveles.

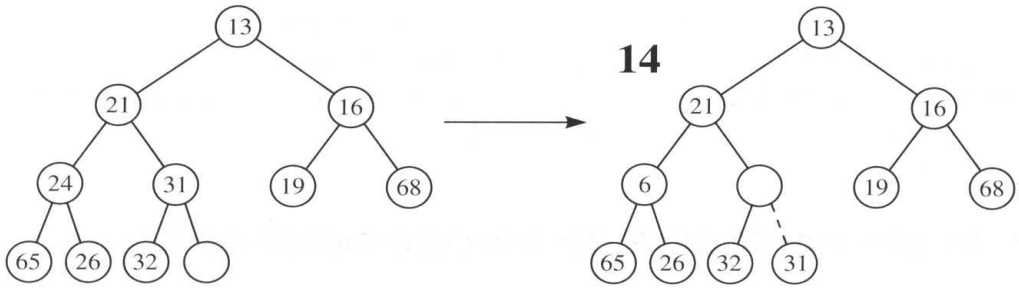


Figura 20.9 Intento de inserción de 14, creando el hueco y flotándolo hacia arriba.

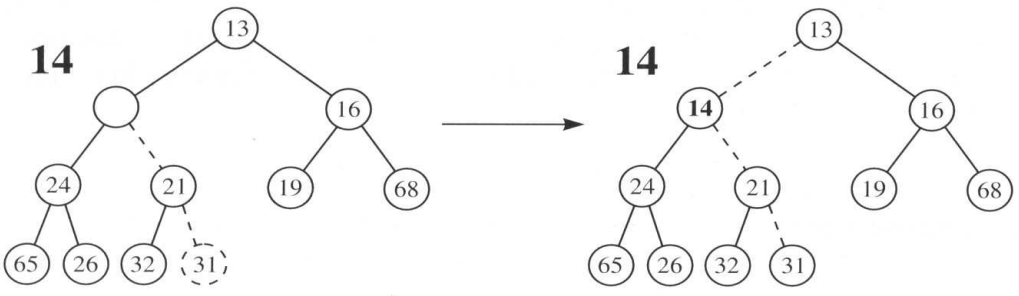


Figura 20.10 Los dos pasos que faltan para insertar 14 en el montículo de la figura anterior.

```
1  /**
2   * Método privado que duplica el vector cuando está lleno.
3   */
4  private void comprobarTam( )
5  {
6      if( tamActual == vector.length - 1 )
7      {
8          Comparable [ ] vectorAnt = vector;
9          obtenerVector( tamActual * 2 );
10         for( int i = 0; i < vectorAnt.length; i++ )
11             vector[ i ] = vectorAnt[ i ];
12     }
13 }
```

Figura 20.11 Método privado `comprobarTam` que dobla el tamaño del vector si es necesario.

```
1  /**
2   * Inserta en la cola de prioridad, sin mantener
3   * el orden del montículo. Se permiten elementos duplicados.
4   * @param x el elemento a insertar.
5   */
6  public void introducir( Comparable x )
7  {
8      comprobarTam( );
9      vector[ ++tamActual ] = x;
10     if( x.menorQue( vector[ tamActual / 2 ] ) )
11         ordenado = false;
12 }
13
14 /**
15  * Inserta en la cola de prioridad, manteniendo el orden.
16  * Se permiten elementos duplicados.
17  * @param x el elemento a insertar.
18  */
19 public void insertar( Comparable x )
20 {
21     if( !ordenado )
22     {
23         introducir( x );
24         return;
25     }
26
27     comprobarTam( );
28
29         // Reflotamiento
30     int hueco = ++tamActual;
31     for( ; x.menorQue( vector[ hueco / 2 ] ); hueco /= 2 )
32         vector[ hueco ] = vector[ hueco / 2 ];
33     vector[ hueco ] = x;
34 }
```

Figura 20.12 Métodos `introducir` e `insertar`.

20.2.2 `eliminarMin`

La eliminación del mínimo implica colocar el último elemento en un hueco que se crea en la raíz. El hueco se *hunde* en el árbol a través de los hijos menores hasta que el elemento se pueda colocar sin violar la propiedad de ordenación del montículo.

El método `eliminarMin` es logarítmico, tanto en el caso peor como en el caso medio.

La operación `eliminarMin` se realiza de forma similar a las inserciones. Como ya hemos visto, encontrar el mínimo es fácil; la parte difícil es eliminarlo. Cuando se elimina el mínimo, se crea un hueco en la raíz. Puesto que el montículo tiene ahora un elemento menos, la propiedad estructural nos dice que se debe eliminar el último nodo. La Figura 20.13 muestra la situación: el elemento mínimo es 13, la raíz tiene un hueco y el último elemento debe colocarse en algún lugar del montículo.

Si el último elemento se pudiera colocar en el hueco, ya habríamos terminado. Pero esto es imposible, a menos que el tamaño del árbol sea 2 o 3, porque los elementos de la parte inferior son más grandes que los del segundo nivel. Debemos hacer lo mismo que en la inserción: poner algún elemento en el hueco y después mover el hueco. La única diferencia está en que en `eliminarMin`, nos movemos hacia abajo. Para hacer esto, buscamos el hijo del hueco con el elemento más pequeño. Si ese elemento es menor que el que estamos intentando colocar, movemos el hijo al hueco, empujando así el hueco un nivel hacia abajo. En la Figura 20.14, colocamos el hijo más pequeño (14) en el hueco, moviendo el hueco un nivel hacia abajo. Repetimos esto de nuevo, colocando 19 en el hueco y creando un nuevo hueco un nivel más abajo. Entonces colocamos 26 en el hueco y creamos un nuevo hueco en el último nivel. Finalmente, podemos colocar 31 en el hueco, como se muestra en la Figura 20.15. Es fácil ver que ésta es una operación logarítmica en el caso peor. Llamamos a este proceso *hundimiento*. No es sorprendente que el hundimiento raramente termine antes de bajar uno o dos niveles, por lo que la operación es también logarítmica en el caso medio.

La Figura 20.16 muestra el método `eliminarMin`. Las comprobaciones de montículo vacío y de ordenación se realizan automáticamente a través de la llamada

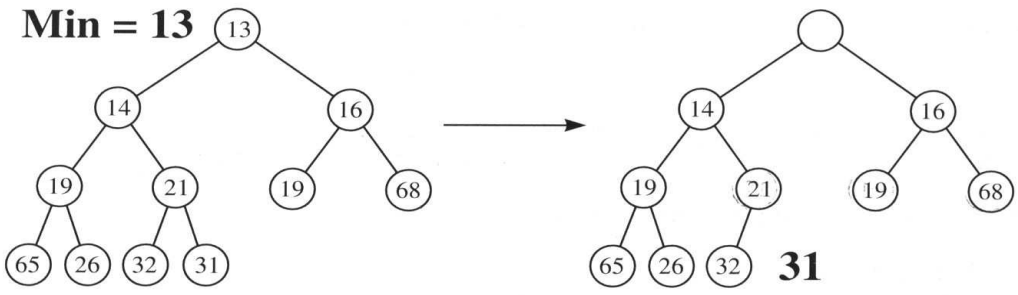


Figura 20.13 Creación del hueco en la raíz cuando se elimina el elemento mínimo.

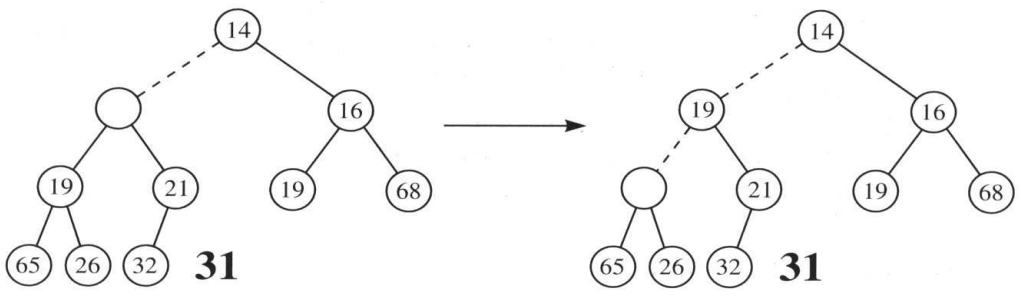


Figura 20.14 Los dos pasos siguientes de `eliminarMin`.

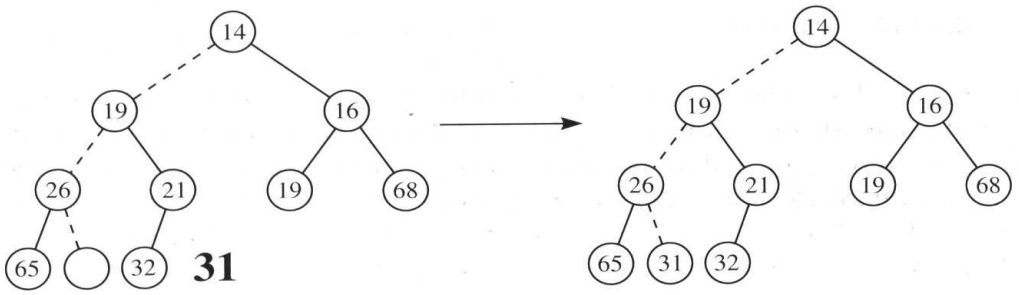


Figura 20.15 Últimos dos pasos de `eliminarMin`.

```
1  /**
2   * Elimina el elemento mínimo de la cola de prioridad.
3   * @exception DesbordamientoInferior si la cola está vacía.
4   */
5  public Comparable eliminarMin( ) throws DesbordamientoInferior
6  {
7      Comparable elemMin = buscarMin( );
8      vector[ 1 ] = vector[ tamActual-- ];
9      hundir( 1 );
10
11     return elemMin;
12 }
```

Figura 20.16 Método `eliminarMin`.

a `buscarMin` en la línea 7. El verdadero trabajo lo lleva a cabo `hundir`, que se muestra en la Figura 20.17. El código aquí mostrado es similar en espíritu al del reflotamiento en el método `insertar`. Debido a que hay dos hijos en lugar de un padre, el código se complica un poco más. El método `hundir` toma un único argumento que nos dice dónde se va a colocar el hueco. Entonces el elemento situado en el hueco se elimina y comienza el hundimiento.

```
1  /**
2   * Método interno para hundir en el montículo.
3   * @param hueco el índice donde comienza el hundimiento.
4   */
5  private void hundir( int hueco )
6  {
7      int hijo;
8      Comparable tmp = vector[ hueco ];
9
10     for( ; hueco * 2 <= tamActual; hueco = hijo )
11     {
12         hijo = hueco * 2;
13         if( hijo != tamActual && vector[ hijo + 1 ].
14                         menorQue( vector[ hijo ] ) )
15             hijo++;
16         if( vector[ hijo ].menorQue( tmp ) )
17             vector[ hueco ] = vector[ hijo ];
18         else
19             break;
20     }
21     vector[ hueco ] = tmp;
22 }
```

Figura 20.17 Método `hundir` usado en `eliminarMin` y `arreglarMonticulo`.

En el caso de `eliminarMin`, `hueco` estará en la posición 1. El bucle `for` de la línea 10 termina cuando no hay hijo izquierdo. La tercera componente mueve el hueco al hijo. En las líneas 13 a 15 se busca el hijo con el elemento más pequeño. Nótese que hemos de ser cuidadosos porque el último nodo de un montículo de tamaño par es hijo único, por lo que no siempre podemos asumir que hay dos hijos. En base a ello realizamos el test de la línea 13.

20.3 `arreglarMonticulo`: construcción en tiempo lineal del montículo

Se puede ejecutar `arreglarMonticulo` en tiempo lineal aplicando una rutina de hundimiento de los nodos en orden inverso al recorrido por niveles.

La operación `arreglarMonticulo` toma un árbol completo sin orden y restablece el mismo. Queremos que sea una operación lineal, pues en tal caso se podrían hacer N inserciones en tiempo $O(N \log N)$. Esperamos poder alcanzar un tiempo $O(N)$ porque N inserciones sucesivas requieren un tiempo total $O(N)$ en media, basándonos en el resultado del final de la Sección 20.2.1. N inserciones sucesivas llevan a cabo más trabajo de lo que necesitamos, ya que tras cada inserción se mantiene el orden, mientras que ahora esto es solamente necesario en un instante concreto.

La solución abstracta más sencilla se obtiene viendo el montículo como una estructura definida recursivamente, como se muestra en la Figura 20.18. Comenzaremos por llamar recursivamente a `arreglarMonticulo` sobre los submontículos izquierdo y derecho. Tras ello, tenemos garantizado que la propiedad de ordenación del montículo se mantiene en todas partes excepto, quizás, en la raíz. Podemos establecer el orden completamente llamando a `hundir` con la raíz como argumento. La rutina recursiva trabaja garantizando que cuando aplicamos `hundir(i)`, todos los descendientes del nodo `i` han sido procesados por la correspondiente llamada recursiva a `hundir`. Sin embargo, la recursión no es necesaria, como se hace evidente a partir del comentario siguiente: si llamamos a `hundir` sobre cada nodo en sentido inverso al recorrido por niveles, cuando realicemos la llamada con `i` se habrán procesado todos los descendientes del nodo `i` con una llamada a `hundir`. Esto nos conduce a un algoritmo increíblemente simple para `arreglarMonticulo`, que se muestra en la Figura 20.19. Observemos que no hace falta ejecutar `hundir` sobre las hojas, por lo que comenzamos por el nodo de mayor índice que no sea una hoja.

El primer árbol de la Figura 20.20 es el árbol sin ordenar. Los otros siete árboles en las Figuras 20.20 a 20.23 muestran el resultado de cada una de las siete lla-

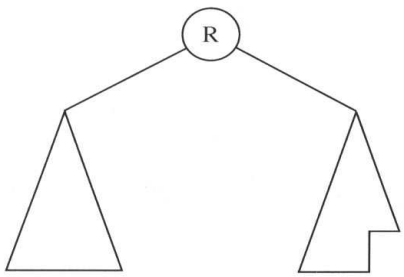


Figura 20.18 Visión recursiva del montículo.

```
1  /**
2   * Restablece la propiedad de ordenación del montículo tras una
3   * serie de operaciones introducir. Se ejecuta en tiempo lineal.
4   */
5  private void arreglarMonticulo( )
6  {
7      for( int i = tamActual / 2; i > 0; i-- )
8          hundir( i );
9      ordenado = true;
10 }
```

Figura 20.19 Implementación del método lineal `arreglarMonticulo`.

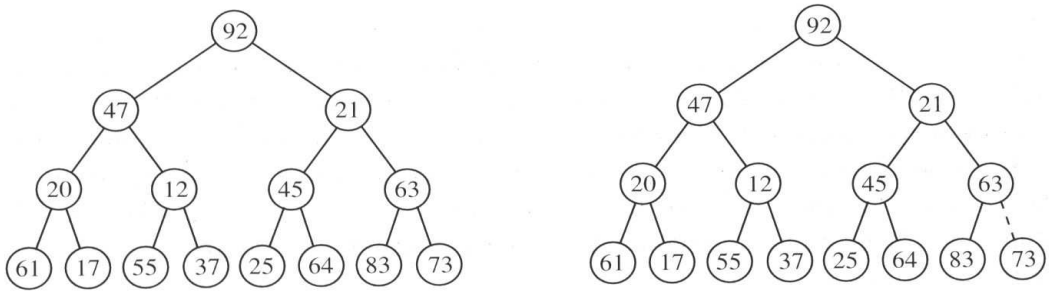


Figura 20.20 Montículo inicial (a la izquierda); y tras la llamada a `hundir(7)` (a la derecha).

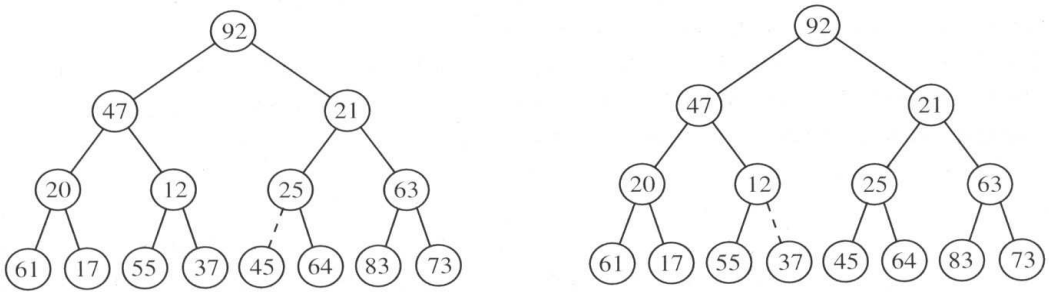


Figura 20.21 Montículo tras la llamada a `hundir(6)` (a la izquierda); y tras la llamada a `hundir(5)` (a la derecha).

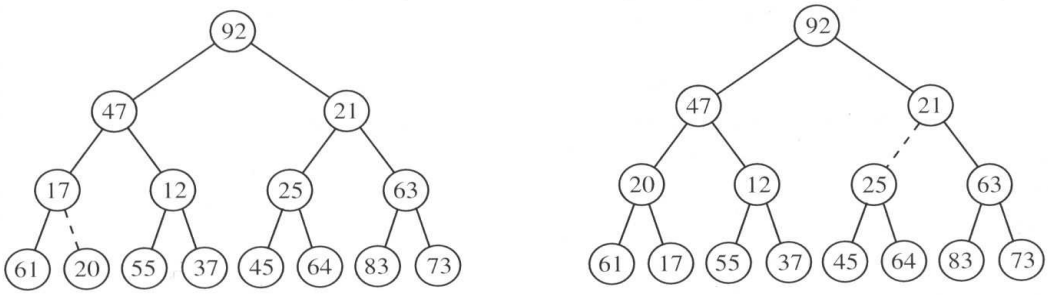


Figura 20.22 Montículo tras la llamada a `hundir(4)` (a la izquierda); y tras la llamada a `hundir(3)` (a la derecha).

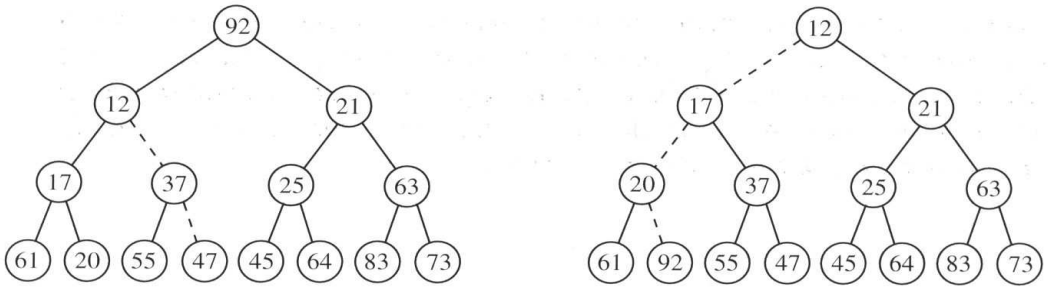


Figura 20.23 Montículo tras la llamada a `hundir(2)` (a la izquierda); y tras la llamada a `hundir(1)` (a la derecha), terminando así la ejecución de `arreglarMonticulo`.

madas a `hundir`. Cada línea punteada se corresponde con dos comparaciones: una para buscar el hijo menor y la otra para comparar el hijo menor con el nodo. Observemos que hay solamente diez líneas punteadas en el algoritmo completo (podría haber habido sólo una más), lo que se corresponde con 20 comparaciones.

La cota lineal se puede demostrar calculando la suma de las alturas de todos los nodos en el montículo.

Para acotar el tiempo de ejecución de `arreglarMonticulo`, debemos acotar el número de líneas punteadas, lo cual se puede conseguir calculando la suma de las alturas de todos los nodos del montículo, que es igual al máximo número de líneas punteadas. Es de esperar que este número sea pequeño porque la mitad de los nodos son hojas, las cuales tienen altura 0, y una cuarta parte tiene altura 1. Luego, solamente una cuarta parte de los nodos (los que no están incluidos en los dos casos anteriores) pueden contribuir en más de 1 unidad de altura. En particular, hay solamente un nodo que contribuye con la máxima altura, $\lfloor \log N \rfloor$.

Demostramos la cota para los árboles perfectos usando un argumento de marcado.

Para obtener una cota de tiempo lineal para `arreglarMonticulo`, necesitamos demostrar que la suma de las alturas de los nodos de un árbol binario completo es $O(N)$. Esto se demuestra en el Teorema 20.1. Demostramos la cota para árboles perfectos usando un argumento de marcado.

Teorema 20.1 *Dado un árbol binario perfecto de altura H que contiene $N = 2^{H+1} - 1$ nodos, se cumple que la suma de alturas de sus nodos es $N - H - 1$.*

Demostración *Usamos un argumento de marcado del árbol. (También se podría hacer un cálculo más directo usando la fuerza bruta, como en el Ejercicio 20.10.) Para cada nodo en el árbol con altura h, marcamos h aristas del árbol en la siguiente forma: bajamos por la arista izquierda y después solamente por aristas derechas, marcando cada arista por la que pasamos. Un ejemplo nos lo da un árbol perfecto de altura 4. Los nodos de altura 1 tienen su arista izquierda marcada, como se muestra en la Figura 20.24. Después, los nodos de altura 2 tienen marcada una arista izquierda y luego una derecha hacia abajo en el árbol, lo cual se muestra en la Figura 20.25. En la Figura 20.26, se marcan tres aristas para cada nodo de altura 3: la primera es la arista izquierda que sale del nodo, y después las aristas derechas en el camino hacia abajo en el árbol. Finalmente, en la Figura 20.27 vemos que se han marcado 4 aristas: la izquierda que sale del nodo y después las tres derechas siguientes en el camino hacia abajo en el árbol.*

Observe que una arista nunca se marca dos veces y que se marcan todas las aristas excepto las del camino de la derecha. Puesto que hay $N - 1$ aristas (a cada nodo excepto a la raíz le llega una arista) y H aristas en el camino de la derecha, el número de aristas marcadas es $N - 1 - H$, lo que concluye la demostración del teorema.

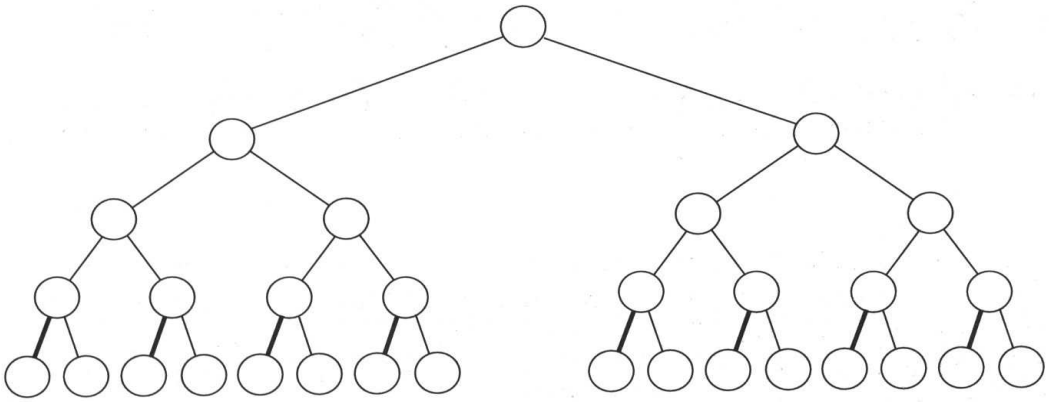


Figura 20.24 Marcado de las aristas izquierdas para los nodos de altura 1.

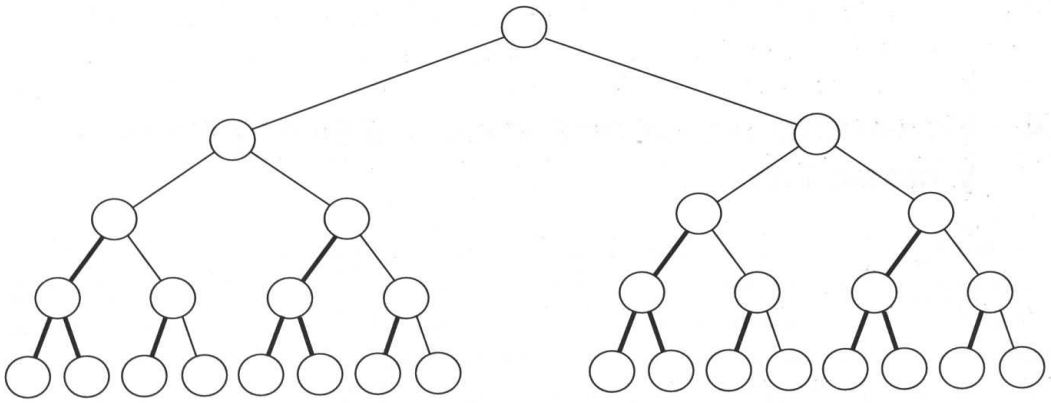


Figura 20.25 Marcado de la primera arista izquierda y de la derecha situada a continuación, para los nodos de altura 2.

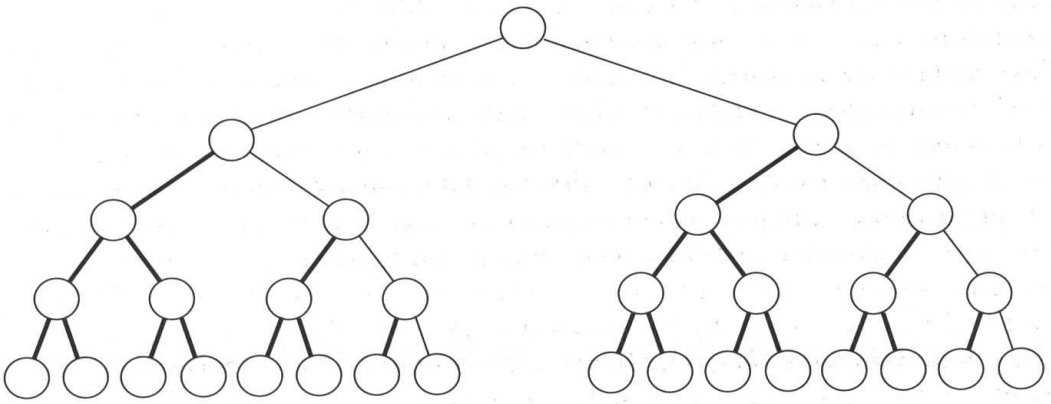


Figura 20.26 Marcado de la primera arista izquierda y de las dos aristas derechas situadas a continuación, para los nodos de altura 3.

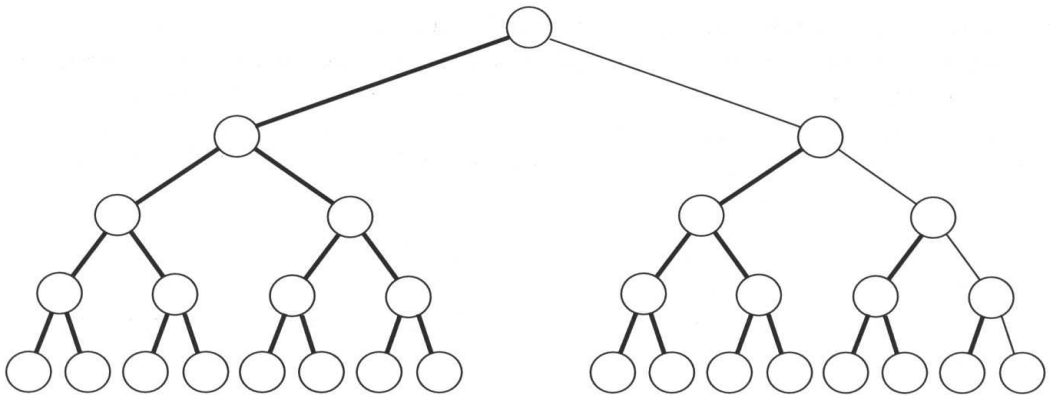


Figura 20.27 Marcado de la primera arista izquierda y de las tres aristas derechas situadas a continuación, para los nodos de altura 4.

Un árbol binario completo no es un árbol perfecto, pero el resultado que hemos obtenido es una cota superior de la suma de las alturas de los nodos de un árbol binario completo. Puesto que un árbol completo tiene entre 2^H y $2^{H+1} - 1$ nodos, este teorema implica que la suma es $O(N)$. Un argumento más cuidadoso establece que la suma de las alturas es $N - v(N)$, donde $v(N)$ es el número de unos en la representación binaria de N. Se deja como Ejercicio 20.12 la demostración de este resultado.

20.4 Operaciones avanzadas: `reducirClave` y `mezclar`

En el Capítulo 22 se estudian unas colas de prioridad que soportan dos operaciones adicionales. La operación `reducirClave`reduce el valor de un elemento en la cola de prioridad. Se supone conocida la posición del elemento. En un montículo binario, esto se puede conseguir fácilmente reflotando hasta restablecer el orden en el montículo. Sin embargo, debemos tener cuidado, ya que hemos supuesto que la posición de cada elemento se almacena de forma separada. Todos los elementos implicados en el reflotamiento ven sus posiciones modificadas. El método `reducirClave`es útil en la implementación de algoritmos sobre grafos (por ejemplo el algoritmo de Dijkstra, visto en la Sección 14.3).

El algoritmo `mezclar` combina dos colas de prioridad. Debido a que el montículo está basado en un vector, lo mejor que podemos esperar conseguir con una mezcla es copiar los elementos del montículo más pequeño en el grande y realizar después algunos arreglos. Aun así, esto llevará un tiempo al menos lineal por operación. Si usáramos árboles generales con nodos conectados mediante referencias a los hijos, podríamos reducir la cota a coste logarítmico por operación. La operación de mezcla se usa en el diseño avanzado de algoritmos.

Se puede usar una cola de prioridad para ordenar elementos en tiempo $O(N \log N)$. Un algoritmo basado en esta idea es el *método del montículo*.

20.5 Ordenación interna: método del montículo

Se puede usar una cola de prioridad para ordenar N elementos de la siguiente forma:

1. Insertar cada elemento en un montículo binario.
2. Extraer cada elemento llamando a `eliminarMin` N veces. El resultado está ordenado.

Usando las observaciones de la sección anterior, podemos implementar este proceso de forma más eficiente:

1. `introducir` cada elemento en un montículo binario.
2. Aplicar `arreglarMonticulo`.
3. Llamar a `eliminarMin` N veces: los elementos saldrán del montículo en orden.

El paso 1 requiere un tiempo total lineal y el paso 2 también requiere un tiempo lineal. En el paso 3, cada llamada a `eliminarMin` exige un tiempo logarítmico, con lo que N llamadas tardan un tiempo $O(N \log N)$. En consecuencia, tenemos un algoritmo de ordenación $O(N \log N)$ en el caso peor, lo que es al menos igual de bueno que un algoritmo basado en comparaciones, como se vió en la Sección 8.8. Un problema del algoritmo tal y como lo hemos presentado es que la ordenación de un vector requiere la construcción de una estructura de datos montículo, que ya conlleva en sí misma la sobrecarga de implementarse como un vector. Sería preferible emular el montículo, en lugar de usar toda la parafernalia de la clase de los montículos. Supondremos para el resto de esta discusión que vamos a hacer esto.

Pero, incluso aunque no usemos directamente la clase de los montículos, tenemos el problema de tener que utilizar un segundo vector, ya que tenemos que almacenar en ese segundo vector el orden en el que los elementos van saliendo del equivalente al montículo, para después copiarlo en el original. Se ha doblado la cantidad de memoria utilizada, lo cual puede ser crítico en algunas aplicaciones. Observemos que el tiempo extra utilizado en copiar el segundo vector en el primero, es solamente $O(N)$, por lo que a diferencia de lo que sucedía en la ordenación por mezcla, el vector adicional no afecta de forma significativa al tiempo de ejecución. El problema es el espacio.

Usando las partes vacías del vector podemos llevar a cabo la ordenación *in situ*.

Una forma inteligente de evitar el uso de un segundo vector hace uso del hecho de que después de cada operación `eliminarMin`, el montículo tiene un elemento menos. En consecuencia, la última celda del vector está disponible para almacenar el elemento eliminado. Por ejemplo, supongamos que tenemos un montículo de 6 elementos. La primera operación `eliminarMin` produce A_1. Ahora el montículo tiene 5 elementos, por lo que podemos colocar A_1 en la posición 6. La siguiente operación `eliminarMin` produce A_2. Puesto que el montículo tiene ahora solamente 4 elementos, podemos colocar A_2 en la posición 5.

Si usamos un montículo maximal, obtenemos los elementos en orden creciente.

Bajo esta estrategia, el vector contendrá, después de la última operación `eliminarMin`, los elementos en orden *decreciente*. Si queremos el vector en orden *creciente*, podemos cambiar la propiedad de ordenación de forma que cada elemento padre sea mayor que sus hijos. En tal caso tendríamos un montículo maximal. Por ejemplo, supongamos que queremos ordenar la secuencia 59, 36, 58, 21, 41, 97, 31, 16, 26, 53. Después de haber introducido los elementos en el montículo y de haber aplicado `arreglarMonticulo`, obtenemos lo reflejado en la Figura 20.28. (Observemos que en este caso no hay centinela: asumimos que los datos comienzan en la posición 0, como es habitual en otros algoritmos de ordenación del Capítulo 8.)

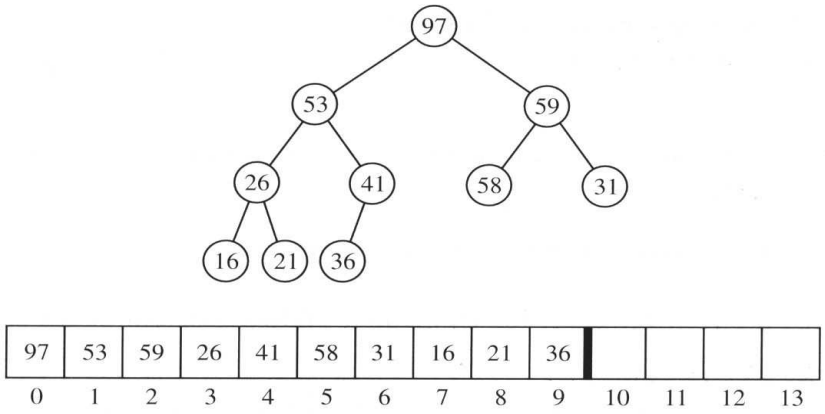


Figura 20.28 Montículos maximal tras la aplicación de `arreglarMonticulo`.

La Figura 20.29 muestra el montículo resultante tras la primera aplicación de `eliminarMax`. El último elemento del montículo es 21; 97 se ha colocado en una parte del vector que técnicamente ya no forma parte del montículo.

La Figura 20.30 muestra que tras la segunda aplicación de `eliminarMax`, 16 se convierte en el último elemento. Ahora quedan solamente ocho elementos en el montículo. El máximo elemento, 59, que ha sido eliminado, se coloca en una posición del vector que ya no pertenece al montículo. Después de ejecutar otras siete operaciones `eliminarMax` más, el montículo contendrá sólo un elemento, pero los elementos que aparecen en el vector se encuentran ordenados de forma creciente.

Se requieren pocos cambios para el algoritmo heapsort porque la raíz está situada en la posición 0.

La implementación del método del montículo (`ordenaViaMonticulo`) es simple porque las operaciones básicas se corresponden con las operaciones sobre los montículos. Hay tres pequeñas diferencias. En primer lugar, puesto que estamos usando un montículo maximal, necesitamos invertir todas las comparaciones, pasando de > a <. En segundo lugar, ya no podemos asumir que hay un centinela en la posición 0, porque todos los demás algoritmos de ordenación almacenan datos en la posición 0 y `ordenViaMonticulo` no debería ser diferente en este sentido. Aunque ya no se necesita el centinela (porque no hay operaciones de reflotamiento), su ausencia afecta a los cálculos de las posiciones de los hijos y del padre. En

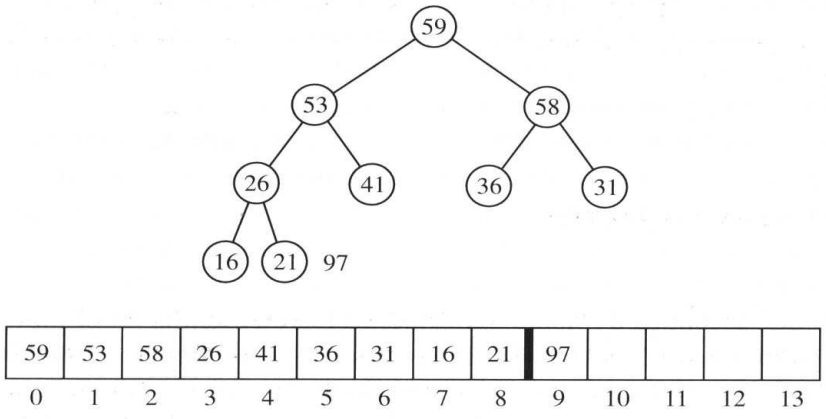
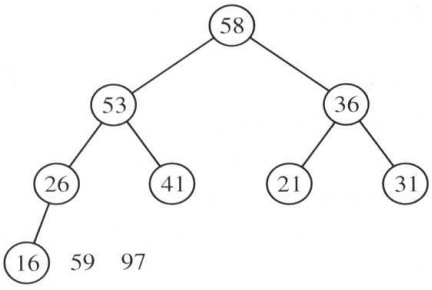


Figura 20.29 Montículo después de la primera aplicación de `eliminarMax`.



58	53	36	26	41	21	31	16	59	97				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figura 20.30 Montículo después de la segunda aplicación de `eliminarMax`.

```
1  /**
2   * Método estándar de ordenación mediante un montículo
3   * @param v un vector de objetos de la clase Comparable
4   */
5  public static void ordenaViaMonticulo( Comparable [ ] v )
6  {
7      for( int i = v.length / 2; i >= 0; i-- )  // Construcción
8          hundir2( v, i, v.length );
9      for( int i = v.length - 1; i > 0; i-- )   // eliminarMax
10     {
11         intercambiarReferencias( v, 0, i );
12         hundir2( v, 0, i );
13     }
14 }
```

Figura 20.31 Rutina `ordenaViaMonticulo`.

concreto, dado un nodo en la posición i, su padre está en la posición $(i - 1)/2$, su hijo izquierdo en la $2i + 1$, y el derecho en la siguiente al hijo izquierdo. En tercer lugar, `hundir2` necesita ser informado del tamaño actual del montículo (que se reduce en 1 en cada iteración de `eliminarMax`). La implementación de `hundir2` se deja como Ejercicio 20.20. Suponiendo que hemos escrito `hundir2`, entonces el algoritmo `ordenViaMonticulo` se puede expresar fácilmente, como se muestra en la Figura 20.31.

20.6 Ordenación externa

Las *ordenaciones externas* se usan cuando la cantidad de datos es demasiado grande para cargarlos juntos en memoria principal.

Hasta el momento, todos los algoritmos de ordenación examinados requieren que la entrada quepa en la memoria principal. Sin embargo, hay aplicaciones para las que la entrada es demasiado grande para ser cargarda completa en memoria. Esta sección estudia algoritmos *de ordenación externa*, diseñados para tratar entradas de gran tamaño.

20.6.1 Por qué necesitamos nuevos algoritmos

La mayor parte de los algoritmos de ordenación interna toman ventaja del hecho de que se puede acceder directamente a la memoria. El método Shellsort compara los elementos `v[i]` y `v[i-desp]` en una unidad de tiempo. El método del montículo compara `v[i]` y `v[hijo=i*2]` en una unidad de tiempo.

El algoritmo de ordenación rápida, con selección del pivote mediante la mediana de tres, requiere la comparación de `v[primero]`, `v[centro]` y `v[ultimo]` en un número constante de unidades de tiempo. Si la entrada se encuentra en una cinta, todas esas operaciones pierden la eficiencia requerida, pues solamente podemos acceder a los elementos en una cinta de forma secuencial. Incluso si los datos están en un disco, hay una notable pérdida de eficiencia en la práctica debido al retraso producido por el necesario giro del disco y el movimiento de la cabeza hasta el lugar adecuado.

Para ver lo lentos que pueden llegar a ser los accesos externos, podríamos crear un fichero aleatorio grande pero no lo suficiente como para no caber en memoria principal. Cuando leemos el fichero y lo ordenamos usando un algoritmo eficiente, el tiempo para leer la entrada es significativo en comparación con el tiempo que se tarda en ordenarla, incluso aunque la operación de ordenación requiere un tiempo $O(N \log N)$ (o incluso peor para el método Shellsort) y la lectura de la entrada sea sólo $O(N)$.

20.6.2 Modelo de ordenación externa

Suponemos que las ordenaciones se realizan sobre una cinta, donde sólo está permitido el acceso secuencial a la entrada.

La amplia variedad de dispositivos de almacenamiento masivo hacen que la ordenación externa dependa del dispositivo mucho más que la ordenación interna. Los algoritmos considerados aquí trabajan sobre cintas, que constituyen probablemente el medio de almacenamiento más restrictivo. Puesto que el acceso a un elemento en una cinta se realiza recorriendo toda la cinta hasta la posición adecuada, sólo se puede acceder de forma eficiente a una cinta de forma secuencial (en cualquier sentido).

Supondremos que disponemos al menos de tres cintas para llevar a cabo la ordenación. Necesitamos dos para realizar una ordenación eficiente, la tercera simplifica las cosas. Si solamente tuviéramos una cinta, tendríamos un grave problema: cualquier algoritmo requeriría $\Omega(N)$ accesos a la cinta.

El algoritmo básico de ordenación externa usa repetidamente mezclas de dos direcciones. Cada grupo ordenado es una *carrera*. Como resultado de una pasada, se dobla la longitud de la carrera hasta que eventualmente quede sólo una carrera.

20.6.3 El algoritmo sencillo

El algoritmo básico de ordenación externa usa la rutina de mezcla usada en mergesort. Supongamos que tenemos cuatro cintas, $A1$, $A2$, $B1$ y $B2$, que son dos cintas de entrada y dos de salida. Dependiendo del punto en el que nos encontremos en el algoritmo, las cintas A y B son de entrada o de salida. Supondremos que los datos están inicialmente en $A1$, y además que la memoria puede almacenar (y ordenar) M registros de una vez. El primer paso natural es leer M registros de una vez de la cinta de entrada, ordenar internamente estos registros y después escribir los registros ordenados en $B1$ o en $B2$ (de forma alternada). Cada conjunto de registros ordenados se llama una *carrera*. Cada vez que está hecho esto, rebobinamos todas las cintas. Supongamos que tenemos la misma entrada que en nuestro

ejemplo del Shellsort. La configuración inicial se muestra en la Figura 20.32. Si $M = 3$, después de construir las carreras, las cintas contendrán los datos como se muestra en la Figura 20.33.

Ahora $B1$ y $B2$ contienen un grupo de carreras. Tomamos las primeras carreras de cada cinta, las mezclamos y escribimos el resultado, que es una carrera el doble de larga, en $A1$. Después tomamos la siguiente carrera de las dos cintas, las mezclamos y escribimos el resultado en $A2$. Continuamos con este proceso, alternando la salida sobre $A1$ y $A2$ hasta que o bien $B1$ o bien $B2$ esté vacía. En este punto, o bien las dos están vacías o hay una (probablemente corta) carrera restante. En este último caso, copiamos esta carrera en la cinta apropiada. Rebobinamos todas las cintas y repetimos los mismos pasos, esta vez usando las cintas A como entrada y las B como salida. Esto nos conducirá a carreras de longitud $4M$. Continuamos con este proceso hasta obtener una carrera de longitud N, en cuyo momento la carrera representa la entrada ordenada. Las Figuras 20.34 a 20.36 muestran cómo trabaja este proceso sobre nuestro ejemplo de entrada.

Necesitamos $\lceil \log (N/M) \rceil$ pasadas sobre la entrada para completar la ordenación.

El algoritmo requerirá $\lceil \log (N/M) \rceil$ pasadas, más la primera para la construcción de las carreras iniciales. Por ejemplo, si tenemos 10 millones de registros de 128 bytes cada uno y 4 MB de memoria principal, entonces la primera pasada creará 320 carreras. Necesitaríamos 9 pasadas más para completar la ordenación. Esta fórmula también nos indica que nuestro ejemplo de la Figura 20.33 requiere $\lceil \log (13/3) \rceil$ pasadas más, es decir 3.

A1	81	94	11	96	12	35	17	99	28	58	41	75	15
A2													
B1													
B2													

Figura 20.32 Configuración inicial de la cinta.

A1													
A2													
B1	11	81	94	17	28	99	15						
B2	12	35	96	41	58	75							

Figura 20.33 Distribución de las carreras de longitud 3 en dos cintas.

A1	11	12	35	81	94	96	15						
A2	17	28	41	58	75	99							
B1													
B2													

Figura 20.34 Cintas después de la primera ronda de mezclas (la longitud de la carrera es 6).

A1												
A2												
B1	11	12	17	28	35	41	58	75	81	94	96	99
B2	15											

Figura 20.35 Cintas después de la segunda ronda de mezclas (la longitud de la carrera es 12).

A1	11	12	15	17	28	35	41	58	75	81	94	96	99
A2													
B1													
B2													

Figura 20.36 Cintas después de la tercera ronda de mezclas.

20.6.4 Mezcla multiaria

La mezcla *K*-aria reduce el número de pasadas. La implementación obvia usa *2K* cintas.

Si tuviéramos cintas adicionales, entonces podríamos esperar reducir el número de pasadas requeridas para ordenar nuestra entrada. Hacemos esto extendiendo la mezcla básica de dos cintas a una mezcla *K*-aria.

Mezclar dos carreras consiste en recorrer cada cinta de entrada hasta el principio de las carreras a mezclar. Entonces se toma el elemento más pequeño de los dos (aquellos sobre los que nos encontramos en cada una de las cintas) y se coloca en una cinta de salida avanzando la correspondiente cinta de entrada. Si hay *K* cintas de entrada, la estrategia trabaja de la misma forma; la única diferencia está en que es ligeramente más complicado encontrar el elemento más pequeño entre *K* elementos. Podemos encontrar el más pequeño usando una cola de prioridad. Para obtener el siguiente elemento a escribir en la cinta de salida, llevamos a cabo una operación `eliminarMin`. Se avanza la correspondiente cinta de entrada, y si aún no se ha completado la carrera de esa cinta de entrada, insertamos el nuevo elemento en la cola de prioridad. La Figura 20.37 muestra cómo se distribuye la entrada del ejemplo anterior en tres cintas. Las Figuras 20.38 y 20.39 muestran las dos pasadas de la mezcla 3-aria que completan la ordenación.

A1								
A2								
A3								
B1	11	81	94	41	58	75		
B2	12	35	96	15				
B3	17	28	99					

Figura 20.37 Distribución inicial de carreras de longitud 3 en tres cintas.

A1	11	12	17	28	35	81	94	96	99
A2	15	41	58	75					
A3									
B1									
B2									
B3									

Figura 20.38 Situación después de una ronda de mezcla 3-aria (la longitud de la carrera es 9).

A1												
A2												
A3												
B1	11	12	15	17	28	35	41	58	75	81	94	96	99
B2												
B3												

Figura 20.39 Situación después de dos rondas de mezclas 3-arias.

Después de la fase de construcción de las carreras iniciales, el número de pasadas requeridas usando una mezcla K-aria es $\lceil \log_K (N/M) \rceil$, porque la longitud de las carreras se hace K veces más grande cada vez. Para nuestro ejemplo, se tiene $\lceil \log_3 13/3 \rceil = 2$. Si tuviéramos 10 cintas, entonces $K = 5$, y en tal caso, en el ejemplo grande de la sección anterior, las 320 carreras requerirían $\log_5 320 = 4$ pasadas.

20.6.5 Mezcla multifase

La *mezcla multifase* implementa una mezcla K-aria con $K + 1$ cintas.

La estrategia de la mezcla K-aria desarrollada en la sección anterior requiere el uso de $2K$ cintas. Esto podría ser prohibitivo en algunas aplicaciones. Es posible hacerlo con solamente $K + 1$ cintas, a lo que se llama *mezcla multifase*. Un ejemplo de ello sería realizar una mezcla 2-aria usando solamente tres cintas.

Supongamos que tenemos tres cintas —$T1$, $T2$ y $T3$— y un fichero de entrada en $T1$ que produce 34 carreras. Una opción consiste en colocar 17 carreras en cada una de las cintas $T2$ y $T3$. Entonces podríamos mezclar este resultado en $T1$, obteniendo entonces una cinta con 17 carreras. El problema está en que puesto que todas las carreras están en una sola cinta, ahora debemos poner algunas de ellas en la cinta $T2$ para poder realizar la siguiente mezcla. La forma lógica de hacer esto consiste en copiar las primeras ocho carreras de $T1$ en $T2$ y después realizar la mezcla.

	Carrera	Después					
	Const.	T3+T2	T1+T2	T1+T3	T2+T3	T1+T2	T1+T3	T2+T3
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	2

Figura 20.40 Número de carreras usando la mezcla multifase.

Esto tiene el efecto de añadir media pasada extra por cada pasada realizada. La cuestión es, ¿podemos hacerlo mejor?

La distribución de las carreras afecta a la eficiencia. La mejor distribución está relacionada con los números de Fibonacci.

Un método alternativo consiste en dividir las 34 carreras de forma no uniforme. Supongamos que colocamos 21 carreras en $T2$ y 13 en $T3$. Entonces podríamos mezclar 13 carreras produciendo el resultado sobre $T1$ antes de que $T3$ se vacíe. Entonces podemos rebobinar $T1$ y $T3$, y mezclar después $T1$ con 13 carreras, y $T2$ con 8 carreras produciendo el resultado sobre $T3$. Tras ello mezclaríamos $T1$ y $T3$, y así sucesivamente. La Figura 20.40 muestra el número de carreras en cada cinta después de cada pasada.

La distribución original de las carreras supone una gran diferencia. Por ejemplo, si se colocan 22 carreras en $T2$ y 12 en $T3$, después de la primera mezcla, obtenemos 12 carreras en $T1$ y 10 en $T2$. Tras una nueva mezcla hay 10 carreras en $T1$ y 2 carreras en $T3$. En este punto, la ejecución se hace más lenta porque solamente podemos mezclar dos conjuntos de carreras antes de agotar $T3$. Entonces $T1$ tiene 8 carreras y $T2$ tiene 2 carreras. De nuevo solamente podemos mezclar dos conjuntos de carreras, obteniendo $T1$ con 6 carreras y $T3$ con 2 carreras. Después de tres pasadas más, $T2$ tiene dos carreras y las demás cintas están vacías. Debemos copiar 1 carrera en otra cinta. Después mezclamos y terminamos.

Se observa que la primera distribución considerada es óptima. Si el número de carreras es un número de Fibonacci F_N, entonces la mejor forma de distribuirlas consiste en dividirlas en dos números de Fibonacci, F_{N-1} y F_{N-2}. En caso contrario, la cinta debería rellenarse con carreras falsas para conseguir un número de carreras igual a un número de Fibonacci. Los detalles de cómo colocar el conjunto inicial de carreras en las cintas se deja como Ejercicio 20.19. Podemos extender esto a las mezclas K-arias, en las que necesitaríamos para la distribución óptima números de Fibonacci de orden K. Un número de Fibonacci de orden K se define como la suma de los K números de Fibonacci de orden K anteriores, como indican las siguientes ecuaciones:

$$F^{(K)}(N) = F^{(K)}(N-1) + F^{(K)}(N-2) + \cdots + F^{(K)}(N-K)$$

$$F^{(K)}(0 \leqslant N \leqslant K-2) = 0$$

$$F^{(K)}(K-1) = 1$$

20.6.6 Selección del reemplazo

La última cuestión que consideramos es la construcción de las carreras. La estrategia usada hasta el momento es sencilla: leemos tantos elementos como sea posible y los ordenamos, escribiendo el resultado en una cinta. Ésta parece la mejor apro-

ximación posible, hasta que nos damos cuenta de que tan pronto como se escribe el primer elemento en la cinta de salida, la memoria que estaba utilizando está disponible para otro elemento. Si el siguiente elemento de la cinta es mayor que el que acaba de escribirse en la cinta, entonces se puede incluir en la carrera.

Usando esta observación, podemos dar un algoritmo para producir carreras, llamado habitualmente *selección del reemplazo*. Inicialmente se leen M elementos y se colocan en una cola de prioridad (en la memoria principal), usando operaciones `introducir` y una operación `arreglarMonticulo` al final. Llevamos a cabo una operación `eliminarMin`, escribiendo el elemento menor en la cinta de salida. Leemos el siguiente elemento de la cinta de entrada. Si es mayor que el elemento que acabamos de escribir, podemos añadirlo a la cola de prioridad usando una operación `insertar`; en caso contrario, no puede formar parte de la carrera actual. Puesto que la cola de prioridad tiene un elemento menos, este elemento se almacenará en el espacio que no forma parte del montículo (espacio muerto) de la cola de prioridad hasta que se complete la carrera, y se utilizará en la siguiente. Almacenar un elemento en el espacio muerto es exactamente lo que se hacía en heapsort. Continuamos haciendo esto hasta que el tamaño de la cola de prioridad se haga 0, en cuyo momento se acaba la carrera. Comenzamos una nueva carrera construyendo una nueva cola de prioridad utilizando una operación `arreglarMonticulo` sobre todos los elementos situados en el espacio muerto.

Si somos listos, podemos hacer la longitud de las carreras que construimos inicialmente de mayor longitud que la capacidad de la memoria principal disponible. Llamamos a esta técnica *selección del reemplazo*.

La Figura 20.41 muestra la construcción de la carrera para el pequeño ejemplo que hemos estado utilizando, con $M = 3$. Los elementos reservados para la siguiente carrera aparecen sombreados. Los elementos 11, 94 y 81 se colocan usando `arreglarMonticulo`. Se produce la salida de 11 y entonces se coloca 96 en el montículo mediante una inserción, pues es mayor que 11. El elemento 81 sale a

	3 elementos del vector del montículo				
	vector(1)	vector(2)	vector(3)	Salida	Siguiente elemento Lectura
Carrera 1	11	94	81	11	96
	81	94	96	81	12
	94	96	12	94	35
	96	35	12	96	17
	17	35	12	Final de la carrera	Reconstrucción
Carrera 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75
	58	99	75	58	15
	75	99	15	75	Final de la cinta
	99		15	99	
			15	Final de la carrera	Reconstrucción
Carrera 3	15				15

Figura 20.41 Ejemplo de construcción de una carrera.

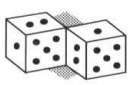
continuación, y entonces se lee 12. Como 12 es menor que 81, no puede incluirse 1 en la carrera actual, por lo que se coloca en el espacio muerto del montículo. El montículo ahora solamente contiene 94 y 96. Después de producirse su salida, sólo tenemos elementos en el espacio muerto, por lo que construimos un nuevo montículo comenzando así con la segunda carrera. En este ejemplo, la selección del reemplazo produce solamente 3 carreras, comparadas con las 5 obtenidas mediante ordenación. Gracias a ello, una mezcla 3-aria acaba en una pasada en lugar de en dos. Si la entrada está distribuida aleatoriamente, se puede demostrar que la selección del reemplazo puede producir carreras de longitud media $2M$. Para nuestro ejemplo grande, tendríamos 160 carreras en lugar de 320, por lo que una mezcla 5-aria aun requeriría cuatro pasadas. En este caso no nos hemos ahorrado ninguna pasada, aunque si tuviéramos suerte podríamos ahorrárnosla de tener 125 carreras o menos. Puesto que la ordenación externa tarda tanto, cada pasada que nos ahorremos puede tener una gran influencia en el tiempo total de ejecución.

Como hemos visto, es posible que la selección del reemplazo no sea mejor que el algoritmo estándar. Pero, la entrada de la que partimos está con frecuencia casi ordenada, en cuyo caso la selección del reemplazo produce unas pocas carreras, anormalmente largas. Este tipo de entrada es habitual en las ordenaciones externas y hace que la selección del reemplazo sea extremadamente valiosa.

Resumen

En este capítulo hemos presentado una implementación elegante de la cola de prioridad. El montículo binario usa solamente un vector, y soporta las operaciones básicas en tiempo logarítmico, en el caso peor. El montículo nos facilita un algoritmo de ordenación conocido, el *método del montículo* o *heapsort*. Los Ejercicios 20.22 y 20.23 le piden comparar la eficiencia del heapsort con la del quicksort. Generalmente hablando, heapsort es más lento que el quicksort, pero es más sencillo de implementar. Finalmente, hemos visto que las colas de prioridad son estructuras de datos importantes en la ordenación externa.

Esto completa la implementación de las estructuras de datos fundamentales clásicas. En la Parte V se examinan estructuras de datos más sofisticadas, comenzando con el árbol de ensanchamiento, un árbol binario de búsqueda con algunas propiedades especiales.



Elementos del juego

árbol binario completo Árbol completamente lleno, sin que falte ningún nodo. Un montículo es un árbol binario completo, lo que permite su representación mediante un vector y garantiza una profundidad logarítmica.

carrera Grupo ordenado en la ordenación externa. Al final de la ordenación tenemos solamente una carrera.

heapsort Algoritmo basado en la idea de que se puede usar una cola de prioridad para ordenar elementos en un tiempo $O(N \log N)$.

hundimiento La eliminación del elemento mínimo implica colocar el último elemento en un hueco creado en la raíz. El hueco se va hundiendo en el árbol a

través de los hijos menores hasta que se pueda colocar el elemento sin violar la propiedad de ordenación del montículo.

mezcla multiaria Mezcla K-aria que reduce el número de pasadas. La implementación obvia usa $2K$ cintas.

mezcla multifase Implementa una mezcla K-aria usando $K + 1$ cintas.

montículo binario Estructura clásica usada para implementar las colas de prioridad. Tiene dos propiedades: una estructural y otra de ordenación.

montículo maximal Montículo que soporta accesos al máximo en lugar de al mínimo.

operación `arreglarMonticulo` Proceso de restablecer la propiedad de ordenación de un montículo. Se puede hacer en tiempo lineal aplicando una rutina de hundimiento a todos los nodos en sentido inverso al recorrido por niveles.

operación `introducir` Operación que añade un elemento, pero que al contrario que `insertar`, no garantiza que se mantenga la ordenación del montículo. Es útil si pretendemos añadir muchos elementos antes de acceder al mínimo.

ordenación externa Método de ordenación usado cuando la cantidad de datos es demasiado grande para caber en la memoria principal.

propiedad de ordenación del montículo Propiedad que se cumple en un montículo (minimal), que consiste en que el elemento del padre de un nodo nunca es mayor que el elemento del propio nodo.

reflotamiento La inserción se implementa creando un hueco en la primera posición disponible, flotándolo después hasta que se pueda colocar en él el nuevo elemento sin que ello suponga una violación del orden respecto a su padre.

representación implícita Uso de un vector para almacenar un montículo.

selección del reemplazo La longitud de las carreras construidas inicialmente puede ser mayor que la cantidad de memoria principal disponible. Si podemos almacenar M elementos en la memoria principal, con esta técnica podemos esperar carreras de longitud $2M$.

Errores comunes

1. El montículo binario requiere un centinela en la posición 0. Un error común es utilizar un centinela inapropiado.
2. La parte más difícil del montículo binario es el caso del hundimiento cuando solamente existe uno de los hijos. Ya que es una situación rara, es difícil descubrir una implementación incorrecta.
3. Si se ha ejecutado una operación `introducir`, la propiedad de ordenación del montículo debe restablecerse antes de acceder al elemento mínimo.
4. Para heapsort, los datos comienzan en la posición 0, por lo que los hijos del nodo i se encuentran en las posiciones $2i + 1$ y $2i + 2$.

En Internet

El montículo binario está disponible en el directorio **DataStructures**. El nombre del fichero es el siguiente:

BinaryHeap.java Contiene la implementación del montículo binario.

Ejercicios

Cuestiones breves

20.1. Describa las propiedades estructural y de ordenación de los montículos binarios.

20.2. En un montículo binario, para un elemento en la posición i, ¿dónde se encuentran su padre y sus hijos izquierdo y derecho?

20.3. Muestre los resultados de insertar los elementos 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13 y 2, uno a uno, en un montículo inicialmente vacío. Después muestre el resultado de usar en su lugar el algoritmo lineal `arreglarMonticulo`.

20.4. ¿Dónde podría haber estado la undécima línea punteada en las Figuras 20.20 a 20.23?

20.5. Un montículo maximal soporta las operaciones `insertar`, `eliminarMax` y `buscarMax` (pero no `eliminarMin` o `buscarMin`). Describa en detalle cómo se pueden implementar los montículos maximales.

20.6. Muestre el resultado del algoritmo de ordenación heapsort sobre la entrada del Ejercicio 20.3 después de la construcción inicial y de dos operaciones posteriores `eliminarMax`.

20.7. ¿Es heapsort un algoritmo de ordenación estable? Es decir, si hay elementos duplicados, ¿retienen dichos elementos duplicados la ordenación inicial entre ellos?

Problemas teóricos

20.8. Un árbol binario completo con N elementos usa las posiciones 1 a N de un vector. Determine lo largo que puede llegar a ser el vector en los siguientes casos:

a) Un árbol binario con dos niveles adicionales (es decir, está ligeramente desequilibrado).
b) Un árbol binario con el nodo más profundo a profundidad 2 log N.
c) Un árbol binario con el nodo más profundo a profundidad 4,1 log N.
d) El peor caso de árbol binario.

20.9. Demuestre las siguientes afirmaciones con respecto al elemento máximo en un montículo:

a) Debe ser una de las hojas.
b) Hay exactamente $\lceil N/2 \rceil$ hojas.
c) Se debe examinar cada hoja para encontrarlo.

20.10. Demuestre el Teorema 20.1 usando directamente una suma. Haga lo siguiente:

a) Demuestre que hay 2^i nodos de altura $H - i$.
b) Escriba la ecuación que define la suma de las alturas usando la parte (a).
c) Evalúe la suma de la parte (b).

20.11. Verifique que la suma de las alturas de un árbol binario perfecto es $N - v(N)$, donde $v(N)$ es el número de unos en la representación binaria de N.

20.12. Demuestre la cota del Ejercicio 20.11 usando un argumento de inducción.

20.13. En el heapsort se usan, en el caso peor, $O(N \log N)$ comparaciones. Derive la constante multiplicativa (es decir, decida si son $N \log N$, $2N \log N$ o $3N \log N$, etc.).

20.14. Demuestre que hay entradas que fuerzan al método `hundir2` de heapsort a bajar todo el camino hasta una hoja. *Pista:* trabaje hacia atrás.

20.15. Un *d-montículo* es una estructura de datos implícita similar al montículo binario, con la diferencia de que los nodos tienen d hijos. Un d-montículo es, por tanto, menos profundo que un montículo binario, pero encontrar el hijo menor requiere examinar d hijos en lugar de dos. Con esto en mente determine el tiempo de ejecución (en términos de d y de N) de las operaciones `insertar` y `eliminarMin` para un d-montículo.

20.16. Un *montículo mini-max* es una estructura de datos que soporta `eliminarMin` y `eliminarMax` con coste logarítmico. La estructura es idéntica a la del montículo binario. La propiedad de ordenación del montículo mini-max es que para cada nodo X a profundidad par, el dato almacenado en X es el menor de su subárbol, mientras que para cada nodo X a profundidad impar, el dato almacenado en X es el mayor de su subárbol. La raíz se encuentra a profundidad par. Haga lo siguiente:

a) Dibuje un posible montículo mini-max para los elementos 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10. Observe que hay muchos montículos posibles.
b) Determine cómo encontrar los elementos máximo y mínimo.
c) Proporcione un algoritmo para insertar un nuevo nodo en un montículo mini- max.
d) Proporcione un algoritmo para ejecutar `eliminarMin` y `eliminarMax`.
e) Proporcione un algoritmo para ejecutar `arreglarMonticulo` en tiempo lineal.

20.17. Un *2-D-montículo* es una estructura de datos que permite que cada nodo tenga dos claves individuales. Se puede ejecutar `eliminarMin` con respecto a cualquiera de dichas claves. La propiedad de ordenación de un 2-D-montículo consiste en que para cada nodo X a profundidad par, el elemento almacenado en X tiene la clave #1 más pequeña de su subárbol, mientras que para cada nodo X a profundidad impar, el elemento almacenado en X tiene la clave #2 más pequeña de su subárbol. Haga lo siguiente:

a) Dibuje un posible 2-D-montículo para los elementos (1, 10), (2, 9), (3, 8), (4, 7) y (5, 6).
b) Explique cómo encontrar el elemento con mínima clave #1.
c) Explique cómo encontrar el elemento con mínima clave #2.
d) Proporcione un algoritmo para insertar un nuevo elemento en el 2-D-montículo.
e) Proporcione un algoritmo para ejecutar `eliminarMin` con respecto a cualquiera de las claves.
f) Proporcione un algoritmo para ejecutar `arreglarMonticulo` en tiempo lineal.

20.18. Un *montárbol* es un árbol binario de búsqueda en el que cada nodo almacena un elemento, dos hijos y una prioridad aleatoria generada en la construcción del nodo. Los nodos del árbol obedecen el orden habitual de los árboles binarios de búsqueda, pero también deben mantener la ordenación de tipo montículo con respecto a las prioridades. El montárbol es una buena alternativa a los árboles binarios equilibrados porque el equilibrio se basa en prioridades aleatorias en lugar de en los valores de sus elementos. En consecuencia los resultados de los árboles binarios de búsqueda para el caso medio se aplican también aquí. Haga lo siguiente:

a) Muestre que una colección de elementos distintos, cada uno de los cuales tiene una prioridad distinta, se puede representar mediante un único montárbol.
b) Muestre cómo llevar a cabo una inserción en un montárbol usando un algoritmo ascendente.
c) Muestre cómo llevar a cabo una inserción en un montárbol usando un algoritmo descendente.
d) Muestre cómo llevar a cabo una eliminación en un montárbol.

20.19. Explique cómo colocar el conjunto inicial de carreras en dos cintas cuando el número de cintas no es un número de Fibonacci.

Problemas prácticos

20.20. Escriba la rutina `hundir2` con la siguiente declaración. (Recuerde que el montículo maximal comienza en la posición 0, no en la 1):

```
private static void hundir2( Comparable [ ] v,
                             int indice, int tamanyo );
```

Prácticas de programación

20.21. Escriba un programa para comparar el tiempo de ejecución de N operaciones `introducir` seguidas por una operación `arreglarMonticulo` con el de N operaciones `insertar` separadas. Ejecute su programa para entradas ordenadas, ordenadas de forma inversa y aleatorias.

20.22. Implemente tanto heapsort como quicksort y compare su eficiencia sobre entradas ordenadas y aleatorias. Use en las pruebas distintos tipos de datos.

20.23. Supongamos que tenemos un hueco en el nodo X. La rutina `hundir2` compara con el hijo de X y después mueve el hijo hacia arriba si es mayor (en el caso de un montículo maximal) que el elemento a colocar, empujando por tanto el hueco hacia abajo. La rutina termina cuando es seguro colocar el elemento a insertar en el hueco. Consideremos la siguiente estrategia alternativa para `hundir2`. Movemos los elementos hacia arriba y el hueco hacia abajo lo más lejos posible, sin comprobar que se puede insertar el nuevo elemento. Esto colocaría la nueva celda en una de las hojas y probablemente violaría el orden. Para arreglar el orden, reflote la nueva celda en la forma habitual. Se espera que el reflotamiento se repe-

tirá en media solamente uno o dos niveles. Escriba una rutina para introducir esta idea. Compare el tiempo de ejecución con el de la implementación estándar de heapsort.

20.24. Implemente una ordenación externa.

20.25. Diseñe un applet que ilustre el funcionamiento del montículo binario.

Bibliografía

El montículo binario se describió por primera vez, en el contexto del algoritmo heapsort, en [8]. El algoritmo lineal `arreglarMonticulo` se ha tomado de [4]. En [7] se proporcionan resultados precisos del número de comparaciones y movimientos de datos usados por heapsort en los casos peor, mejor y medio. En los Capítulos 21 y 22 se discuten implementaciones avanzadas de las colas de prioridad. La ordenación externa se discute en detalle en [6]. El Ejercicio 20.15 se resuelve en [5], el 20.16 en [2] y el 20.17 en [3]. Los montárboles se describen en [1].

1. C. Aragon y R. Seidel, «Randomized Search Trees», *Algorithmica* **16** (1996), 464-497.
2. M. D. Atkinson, J. R. Sack, N. Santoro, y T. Strothotte, «Min-Max Reaps and Generalized Priority Queues», *Communications of the ACM* **29** (1986), 996-1000.
3. Y. Ding y M. A. Weiss, «The k-d Heap: An Efficient Multi-dimensional Priority Queue», *Proceedings of the Third Workshop on Algorithms and Data Structures* (1993), 302-313.
4. R. W. Floyd, «Algorithm 245: Treesort 3», *Communications of the ACM* **7** (1964), 701.
5. D. B. Johnson, «Priority Queues with Update and Finding Minimum Spanning Trees», *Information Processing Letters* **4** (1975), 53-57.
6. D. E. Knuth, *The Art of Computer Programming. Volume* 3: *Sorting and Searching*, 2.ª ed., Addison-Wesley, Reading, Mass. (1997).
7. R. Schaffer y R. Sedgewick, «The Analysis of Heapsort», *Journal of Algorithms* **14** (1993), 76-100.
8. J. W. J. Williams, «Algorithm 232: Heapsort», *Communications of the ACM* **7** (1964), 347-348.