CAPÍTULO

18 Árboles binarios de búsqueda

Para entradas de gran tamaño, el acceso en tiempo lineal de las listas enlazadas es prohibitivo. Este capítulo examina una opción alternativa a la lista enlazada: el *árbol binario de búsqueda*. El árbol binario de búsqueda es una estructura de datos sencilla que puede considerarse como una extensión del algoritmo de búsqueda binaria que permite tanto inserciones como eliminaciones. El tiempo de ejecución de muchas de sus operaciones es, en media, $O(\log N)$. Desgraciadamente, su eficiencia en el caso peor es $O(N)$.

En este capítulo veremos:

- En qué consiste el árbol binario de búsqueda básico.
- Cómo incluir búsquedas por posición en el orden (esto es, la operación `encontrarKesimo`).
- Tres modos diferentes de eliminar el caso peor $O(N)$ (los árboles AVL, los árboles rojinegros y los AA-árboles).
- Cómo puede mejorarse la eficiencia de las consultas en una base de datos de gran tamaño, empleando B-árboles.

18.1 Ideas básicas

Por lo general, un elemento se busca por su clave. Por ejemplo, un estudiante puede buscarse en la base de datos utilizando su DNI. En este caso, el número de DNI se considera la clave del elemento.

El *árbol binario de búsqueda* es un árbol binario que satisface la propiedad de la búsqueda ordenada. Esto significa que para cada nodo X del árbol, los valores de todas las claves de su subárbol izquierdo son menores que la clave de X y los valores de todas las claves de su subárbol derecho son mayores que la clave de X. En la Figura 18.1, el árbol de la izquierda es un árbol binario de búsqueda, mientras que el árbol de la derecha no lo es (la clave 8 no debería pertenecer al subárbol izquierdo de la clave 7). La propiedad que define los árboles binarios de búsqueda implica que todos los elementos del árbol pueden ordenarse de forma consistente (el recorrido en orden simétrico de la estructura muestra los elementos ordenados en forma ascendente). Dicha propiedad no permite, en principio la exis-

Para cualquier nodo del *árbol binario de búsqueda*, todos los nodos con clave menor que la suya están en su subárbol izquierdo, y todos los nodos con clave mayor se encuentran en su subárbol derecho. No se permiten elementos duplicados.

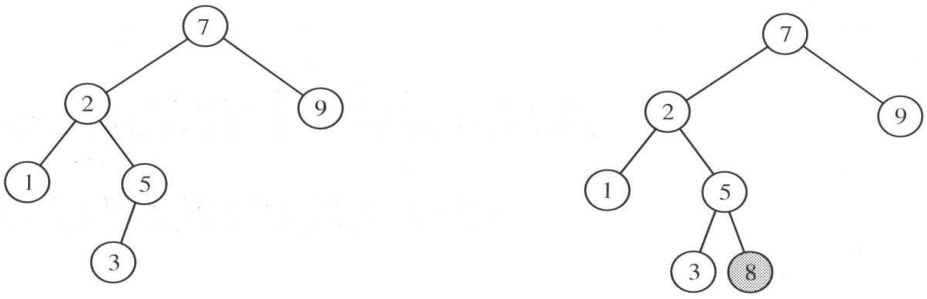

Figura 18.1 Dos árboles binarios (sólo el de la izquierda es de búsqueda).

tencia de elementos duplicados, pero es sencillo generalizarla de modo que se permita la repetición de claves. Generalmente, es mejor guardar los elementos con la misma clave en una estructura secundaria. Si los elementos son duplicados exactos, la mejor opción suele ser tener uno solo y mantener un contador con el número de repeticiones.

18.1.1 Las operaciones

`buscar` se ejecuta desplazándonos por las ramas izquierda o derecha, dependiendo del resultado de las comparaciones.

La mayoría de las operaciones de los árboles binarios de búsqueda son sencillas de comprender. Podemos realizar una operación `buscar` comenzando en la raíz y desplazándonos repetidamente por las ramas izquierda o derecha, dependiendo del resultado de las comparaciones. Por ejemplo, para encontrar el elemento 5 en el árbol binario de búsqueda de la Figura 18.1, empezamos en el 7 y vamos por la rama izquierda. Esto nos conduce al 2, y desde aquí vamos a la derecha. Así, llegamos al 5. Para encontrar el 6, hubiéramos seguido el mismo camino. Cuando estuviésemos en el 5, iríamos por la derecha, encontrando una referencia `null`, por lo que el 6 no está en el árbol. La Figura 18.2 muestra que el 6 puede insertarse en el lugar donde ha finalizado su búsqueda fallida.

`buscarMin` se ejecuta recorriendo los nodos izquierdos hasta llegar a un nodo que no tenga hijo izquierdo. `buscarMax` es similar.

El árbol binario de búsqueda soporta de modo eficiente las operaciones `buscarMin` y `buscarMax`. Para ejecutar `buscarMin`, partimos de la raíz y recorremos repetidamente todos los nodos izquierdos hasta llegar a un nodo que no tenga hijo izquierdo. Dicha hoja es el elemento mínimo del árbol. `buscarMax` es análoga, salvo por el hecho de que el recorrido es ahora por la derecha. Nótese que el coste de ambas operaciones es proporcional al número de nodos del camino de búsqueda. El coste en media es logarítmico, pero puede ser lineal en el peor de los casos. Esto se explica posteriormente en este capítulo.

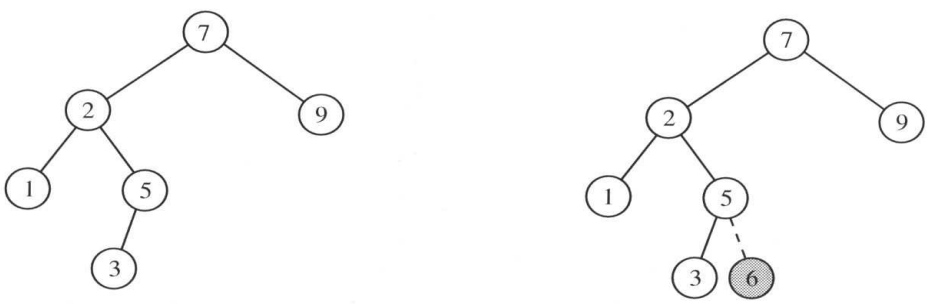

Figura 18.2 Los árboles binarios de búsqueda antes y después de insertar el 6.

La operación más complicada es `eliminar`. Una vez que hemos encontrado el nodo que debe ser borrado, es preciso considerar varias alternativas. El problema es que la eliminación de un nodo puede desconectar varias partes del árbol. Debemos ser cuidadosos en su manipulación de modo que se mantenga siempre la propiedad de la búsqueda ordenada. Además, es necesario evitar que el árbol tenga demasiada profundidad, puesto que, como ya hemos comentado anteriormente, la profundidad del árbol influye en el tiempo de ejecución de los algoritmos.

`eliminar` es complicado ya que los nodos que no son hojas mantienen el árbol conectado, y no deseamos desconectarlo.

Si el nodo a borrar es una hoja, su eliminación no desconectará el árbol, por lo que puede ser eliminado sin más. Si el nodo tiene un solo hijo, podemos eliminarlo después de que su padre haya ajustado sus referencias hacia los hijos, saltando al nodo eliminado. Esto se muestra en la Figura 18.3, en la que eliminamos el 5. Observe que esto implica que `eliminarMin` y `eliminarMax` no son excesivamente complejos, ya que los nodos afectados son hojas o tienen un único hijo. Nótese también que la raíz es un caso especial porque no tiene padre. Sin embargo, cuando implementemos el método `eliminar`, éste se tratará en principio de forma uniforme, sin ser necesario un tratamiento especial.

Si un nodo tiene un único hijo, puede ser eliminado haciendo que su padre pase a referenciar a dicho hijo. La raíz es un caso especial ya que no tiene padre.

El caso complicado llega cuando debemos eliminar un nodo con dos hijos. La estrategia general es reemplazar el elemento de ese nodo con el menor elemento de su subárbol derecho (que, como ya hemos comentado, se encuentra fácilmente) y eliminar después el nodo del menor elemento (que se considera vacío desde el punto de vista lógico). La segunda operación `eliminar` es sencilla, ya que el nodo mínimo de un árbol no tiene hijo izquierdo. La Figura 18.4 muestra un árbol

Un nodo con dos hijos se sustituye por el menor elemento de su subárbol derecho. Después se elimina el nodo correspondiente a dicho menor elemento

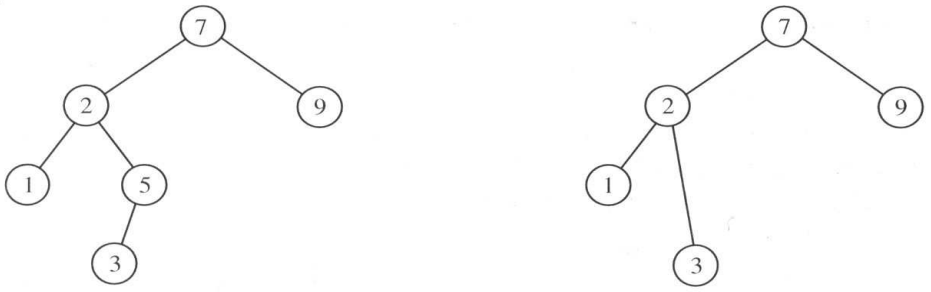

Figura 18.3 Los árboles binarios de búsqueda antes y después de eliminar el 5 (que tenía un único hijo).

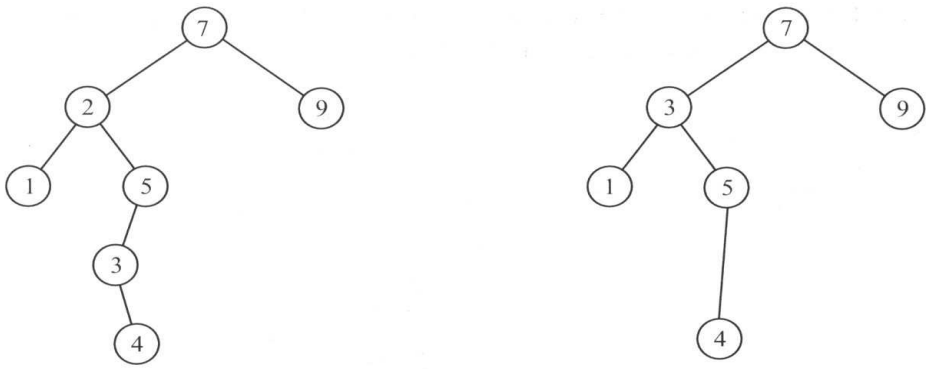

Figura 18.4 Los árboles binarios de búsqueda antes y después de eliminar el 2 (que tenía dos hijos).

inicial y el resultado tras eliminar el nodo 2. Sustituimos dicho nodo por el nodo de menor valor (3) de su subárbol derecho. Nótese que en cualquiera de los casos, la eliminación de un nodo no incrementa la profundidad del árbol.

18.1.2 Implementación en Java

Empleamos una única clase `NodoBinario` para todos los árboles de este capítulo.

En principio, los árboles binarios de búsqueda son sencillos de implementar. Unas pequeñas simplificaciones evitan que el código se complique en exceso. Para empezar, la Figura 18.5 muestra la clase `NodoBinario`. Como es usual, el acceso a los atributos es amistoso, pero la clase sólo es accesible desde dentro de su paquete. En las líneas 31 a 33 hemos incluido algunos atributos adicionales, ya que deseamos emplear las mismas declaraciones en árboles binarios de búsqueda más complejos, que serán discutidos más tarde en este capítulo. Por tanto, dichos atributos adicionales no se emplean en las implementaciones de esta sección. Una opción sería omitirlos de momento y emplear posteriormente la herencia, pero esto oscurece demasiado los conceptos básicos. En consecuencia, la clase `NodoBinario`

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4
5  // Nodo básico almacenado en todos los árboles binarios de búsqueda.
6  // Incluye los campos para todas las variaciones.
7  // Esta clase no es accesible fuera del
8  // paquete EstructurasDatos.
9
10 class NodoBinario
11 {
12         // Constructores
13     NodoBinario( Comparable elDato )
14     {
15         this( elDato, null, null );
16     }
17
18     NodoBinario( Comparable elDato, NodoBinario li, NodoBinario ld )
19     {
20         dato      = elDato;
21         izquierdo = li;
22         derecho   = ld;
23     }
24
25       // Atributos amistosos; accesibles por otras rutinas del paquete
26     Comparable dato;          // La información del nodo
27     NodoBinario izquierdo;    // Hijo izquierdo
28     NodoBinario derecho;      // Hijo derecho
29
30       // Información de equilibrio; se emplea una en cada tipo de árbol
31     int  tamanyo = 1;  // Para árboles binarios de búsqueda con rango
32     int  color   = 1;  // Para árboles rojinegros
33     int  nivel   = 1;  // Para AA-árboles
34 }
```

Figura 18.5 La clase de los nodos de los árboles binarios de búsqueda.

contiene la lista usual de atributos (el dato y dos referencias), más los atributos adicionales empleados más tarde. Se permiten varias formas para la construcción de un nodo.

En las Figuras 18.6 y 18.7 se muestra el esqueleto de la clase `ArbolBinarioBusqueda`. El principal atributo es una referencia a la raíz del árbol, llamada `raiz`.

El atributo `raiz` apunta a la raíz del árbol o es `null` si el árbol está vacío.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase ArbolBinarioBusqueda
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ************************OPERACIONES PÚBLICAS************************
11 // void insertar( x )       --> Inserta x
12 // void eliminar( x )       --> Elimina x
13 // void eliminarMin( )      --> Elimina el menor elemento
14 // Comparable buscar( x )   --> Devuelve el elemento que ajusta con x
15 // Comparable buscarMin( )  --> Devuelve el menor elemento
16 // Comparable buscarMax( )  --> Devuelve el mayor elemento
17 // boolean esVacio( )       --> Devuelve true si vacío; si no, false
18 // void vaciar( )           --> Elimina todos los elementos
19 // void imprimirArbol( )    --> Imprime el árbol ordenadamente
20 // ****************************ERRORES*****************************
21 // Muchas rutinas lanzan ElementoNoEncontrado en condiciones degeneradas
22 // insertar lanza ElementoDuplicado si el elemento está en el árbol
23
24 /**
25  * Implementa un árbol binario de búsqueda sin equilibrar.
26  * Nótese que todas las comparaciones se basan en el método compara.
27  */
28 public class ArbolBinarioBusqueda implements ArbolBusqueda
29 {
30     public ArbolBinarioBusqueda( )
31       { raiz = null; }
32
33     public void insertar( Comparable x ) throws ElementoDuplicado
34       { raiz = insertar( x, raiz ); }
35     public void eliminar( Comparable x ) throws ElementoNoEncontrado
36       { raiz = eliminar( x, raiz ); }
37     public void eliminarMin( ) throws ElementoNoEncontrado
38       { raiz = eliminarMin( raiz ); }
39     public Comparable buscarMin( ) throws ElementoNoEncontrado
40       { return buscarMin( raiz ).dato; }
41     public Comparable buscarMax( ) throws ElementoNoEncontrado
42       { return buscarMax( raiz ).dato; }
43     public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
44       { return buscar( x, raiz ).dato; }
45     public boolean esVacio( )
46       { return raiz == null; }
47     public void vaciar( )
48       { raiz = null; }
49     public void imprimirArbol( )
50       { imprimirArbol( raiz ); }
```

Figura 18.6 Esqueleto de la clase `ArbolBinarioBusqueda` (parte 1).

```
1  /**
2   * Método interno para buscar el menor elemento de un subárbol.
3   * @param t la raíz del subárbol.
4   * @return el nodo que contiene el menor elemento.
5   * @exception ElementoNoEncontrado si el subárbol está vacío.
6   */
7  protected NodoBinario
8  buscarMin( NodoBinario t ) throws ElementoNoEncontrado
9  {
10     if( t == null )
11         throw new ElementoNoEncontrado( "buscarMin de ArbolBusqueda" );
12
13     while( t.izquierdo != null )
14         t = t.izquierdo;
15     return t;
16 }
17
18 /**
19  * Método interno para buscar el mayor elemento de un subárbol.
20  * @param t la raíz del subárbol.
21  * @return el nodo que contiene el mayor elemento.
22  * @exception ElementoNoEncontrado si el subárbol está vacío.
23  */
24 protected NodoBinario
25 buscarMax( NodoBinario t ) throws ElementoNoEncontrado
26 {
27     if( t == null )
28         throw new ElementoNoEncontrado( "buscarMax de ArbolBusqueda" );
29
30     while( t.derecho != null )
31         t = t.derecho;
32     return t;
33 }
```

Figura 18.9 `buscarMin` y `buscarMax` de los árboles binarios de búsqueda.

```
1  /**
2   * Método interno para insertar en un subárbol.
3   * @param x el elemento a insertar.
4   * @param t la raíz del subárbol.
5   * @return la nueva raíz.
6   * @exception ElementoDuplicado si el elemento que ajusta
7   *            con x ya está en el subárbol de raíz t.
8   */
9  protected NodoBinario
10 insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
11 {
12     if( t == null )
13         t = new NodoBinario( x, null, null );
14     else if( x.compara( t.dato ) < 0 )
15         t.izquierdo = insertar( x, t.izquierdo );
16     else if( x.compara( t.dato ) > 0 )
17         t.derecho = insertar( x, t.derecho );
18     else
19         throw new ElementoDuplicado( "insertar de ArbolBusqueda" );
20     return t;
21 }
```

Figura 18.10 Versión recursiva del método `insertar` de los árboles binarios de búsqueda.

Si el árbol no es vacío tenemos tres posibilidades. Si el elemento a insertar es menor que el elemento del nodo `t`, llamamos recursivamente a `insertar` sobre el subárbol derecho. Si es mayor, llamamos recursivamente a `insertar` sobre el subárbol izquierdo. Todo esto se codifica en las líneas 14 a 17. Nótese que devolvemos el resultado de la llamada recursiva a `insertar`. El tercer caso es que el elemento a insertar coincida con el elemento del nodo `t`; en este caso lanzamos una excepción.

Las restantes rutinas corresponden a las operaciones de eliminación. Como se ha indicado anteriormente en este capítulo, la operación `eliminarMin` es sencilla, ya que el nodo mínimo no tiene hijo izquierdo. Sólo debemos saltar sobre el nodo eliminado. Puede parecer que en cada paso se necesita conocer el padre del nodo actual, pero una vez más, empleando la recursión podemos evitar la referencia explícita al padre. El código se muestra en la Figura 18.11.

Si el árbol `t` es vacío, el método `eliminarMin` falla. En caso contrario, si `t` tiene hijo izquierdo, eliminamos de forma recursiva el menor elemento del subárbol izquierdo, por medio de la llamada recursiva de la línea 13. Si alcanzamos la línea 14, sabemos que en ese momento estamos situados sobre el menor nodo del árbol. Esto significa, en particular, que `t` es la raíz de un subárbol que no tiene hijo izquierdo. Si hacemos que `t` sea igual a `t.derecho` entonces `t` es ahora la raíz de un subárbol que ha perdido su menor elemento. Esto es lo que hacemos en la línea 15. Como antes, debemos devolver la raíz del árbol resultante. Pero, ¿al hacerlo no desconectamos el árbol? De nuevo, la respuesta es no. Si `t` era `raiz`, se devuelve el nuevo `t`, que se asigna a `raiz` en el método público, y el proceso es seguro. Si `t` no era `raiz` entonces es `p.izquierdo`, donde `p` es el padre de `t` en el momento de la llamada recursiva. El método que tiene a `p` como su parámetro (en otras palabras, el método que hizo la llamada al método actual) cambia `p.izquierdo` al nuevo `t`. Así, el hijo `izquierdo` del padre apunta a `t`, y el árbol queda conectado. Lo que hemos hecho es mantener el padre en la pila de la recursión en lugar de controlarlo explícitamente en un bucle iterativo.

En las rutinas `eliminar` debemos devolver la raíz del nuevo subárbol. Al efecto, mantenemos almacenado el padre en la pila de las llamadas recursivas.

Una vez empleado este truco en el caso más sencillo, podemos adaptarlo para la rutina general `eliminar`. Esto se muestra en la Figura 18.12. Si el árbol es vacío, el método `eliminar` falla, lanzándose una excepción en la línea 13. En caso

```
1  /**
2   * Método interno para eliminar el menor elemento de un subárbol.
3   * @param t la raíz del subárbol.
4   * @return la nueva raíz.
5   * @exception ElementoNoEncontrado si el subárbol está vacío.
6   */
7  protected NodoBinario
8  eliminarMin( NodoBinario t ) throws ElementoNoEncontrado
9  {
10     if( t == null )
11         throw new ElementoNoEncontrado( "eliminarMin de ArbolBusqueda" );
12     if( t.izquierdo != null )
13         t.izquierdo = eliminarMin( t.izquierdo );
14     else
15         t = t.derecho;
16     return t;
17 }
```

Figura 18.11 Método `eliminarMin` de la clase `ArbolBinarioBusqueda`.

```
1  /**
2   * Método interno para eliminar en un subárbol.
3   * @param x el elemento a eliminar.
4   * @param t la raíz del subárbol.
5   * @return la nueva raíz.
6   * @exception ElementoNoEncontrado si el elemento que ajusta
7   *        con x está en el subárbol de raíz t.
8   */
9  protected NodoBinario
10 eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
11 {
12     if( t == null )
13         throw new ElementoNoEncontrado( "eliminar de ArbolBusqueda" );
14     if( x.compara( t.dato ) < 0 )
15         t.izquierdo = eliminar( x, t.izquierdo );
16     else if( x.compara( t.dato ) > 0 )
17         t.derecho = eliminar( x, t.derecho );
18     else if( t.izquierdo != null && t.derecho != null ) // Dos hijos
19     {
20         t.dato = buscarMin( t.derecho ).dato;
21         t.derecho = eliminarMin( t.derecho );
22     }
23     else // Cambio de raíz
24         t = ( t.izquierdo != null ) ? t.izquierdo : t.derecho;
25     return t;
26 }
```

Figura 18.12 Método `eliminar` de la clase `ArbolBusquedaBinario`.

contrario, si el elemento a eliminar no se ajusta al nodo actual, llamamos recursivamente a `eliminar` modificando su parámetro, haciendo que éste sea el hijo izquierdo o el derecho, según corresponda. En otro caso alcanzamos la línea 18, habiendo encontrado el nodo a eliminar.

La codificación de `eliminar` emplea algunos trucos, pero no es demasiado complicada si empleamos recursión. Los casos de las hojas, de un único hijo o de una raíz con un único hijo se tratan todos juntos en la línea 24.

Recordemos que cuando el nodo a eliminar tiene dos hijos, lo reemplazamos con el menor elemento del subárbol derecho y después eliminamos dicho elemento mínimo. Esto se codifica en las líneas 20 y 21. En caso contrario, tenemos uno o ningún hijo. Si existe hijo izquierdo, entonces se iguala `t` a su hijo izquierdo, tal y como se haría en `eliminarMax`. Si no, sabemos que no hay hijo izquierdo y podemos igualar `t` a su hijo derecho. Todo esto se codifica de forma breve y sencilla en la línea 24, en la que también se cubre el caso de las hojas. Tras cualquiera de estas alternativas, el método `eliminar` devuelve la raíz del subárbol en la línea 25.

Existen dos detalles en esta implementación que debemos comentar. En primer lugar, durante las operaciones básicas de `insertar`, `buscar`, o `eliminar`, se hacen dos comparaciones por nodo, con el propósito de distinguir entre los casos <, = y >. Pero podríamos conseguir lo mismo con una sola comparación por nodo. La estrategia es muy similar a la seguida en el algoritmo de búsqueda binaria de la Sección 5.6. La adecuación de esta técnica para los árboles binarios de búsqueda se discute en la Sección 18.6.2, donde se trata el algoritmo de eliminación de los AA-árboles.

El segundo punto es que no tenemos por qué emplear la recursión para la inserción de elementos. De hecho, una implementación recursiva es, probablemente, más lenta que una no recursiva. En la Sección 18.5.3 se discute una implementación iterativa de `insertar`, en el contexto de los árboles rojinegros.

18.2 Búsqueda por posición en el orden

El árbol binario de búsqueda nos permite encontrar los elementos mínimo o máximo en tiempo equivalente al de una operación arbitraria `buscar`. En algunas ocasiones es importante poder acceder al K-ésimo menor elemento, para un valor arbitrario K. Esto se consigue fácilmente si mantenemos información sobre el tamaño de cada nodo del árbol.

Podemos implementar la operación `buscarKesimo` si actualizamos el tamaño de los nodos en cada modificación del árbol.

Recuerde, de la Sección 17.1, que el tamaño de un nodo es su número de descendientes (incluyéndole a él mismo). Supongamos que se desea encontrar el K-ésimo menor elemento y que K es mayor o igual que 1 y menor o igual que la cantidad de nodos del árbol. La Figura 18.13 muestra que existen tres casos posibles, dependiendo de la relación entre K y el tamaño del subárbol izquierdo, denotado por S_L. Si K es igual a $S_L + 1$ entonces el K-ésimo menor elemento es la raíz, y hemos terminado. Si K es menor o igual que S_L entonces el K-ésimo menor elemento debe estar en el subárbol izquierdo, y podemos encontrarlo de forma recursiva. Una vez más, la recursión no es obligatoria; se emplea para simplificar la descripción del algoritmo. En el caso restante, el K-ésimo menor elemento es el $(K - S_L - 1)$-ésimo elemento más pequeño del subárbol derecho, y también puede encontrarse de forma recursiva.

El mayor esfuerzo se concentra en mantener el tamaño de los nodos durante los cambios. Estas modificaciones se producen en las operaciones `insertar`, `eliminar` y `eliminarMin`. En principio, el mantenimiento de la información es bastante simple. Durante una inserción, cada nodo del camino hasta el punto de inserción gana un nodo en su subárbol. Como consecuencia, el tamaño de cada uno de ellos se incrementa en una unidad, y el nuevo nodo insertado tiene tamaño 1. En `eliminarMin`, cada nodo del camino hasta el mínimo pierde un nodo en su subárbol, y así, su tamaño decrece en una unidad. Durante una operación `eliminar`, todos los nodos del camino hasta el que va a ser eliminado también pierden un nodo en sus subárboles. Como consecuencia de todo ello, podemos mantener fácilmente el tamaño de los nodos con una pequeña cantidad adicional de esfuerzo.

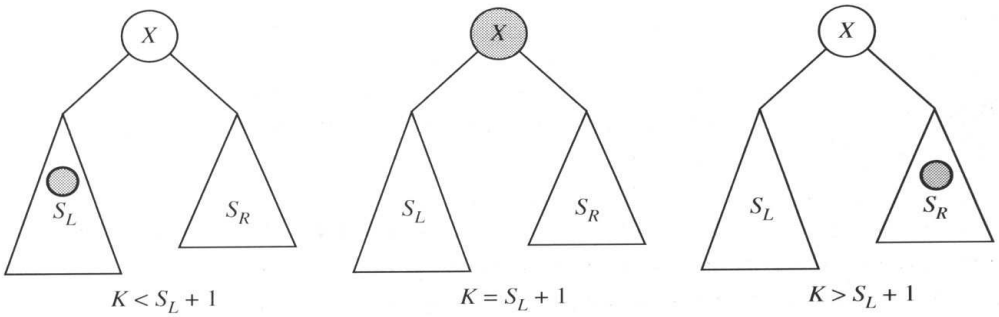

Figura 18.13 Empleo del atributo `tamanyo` para implementar el método `buscarKesimo`.

18.2.1 Implementación en Java

Derivamos una nueva clase que permite realizar búsquedas por posición en el orden.

Desde el punto de vista lógico, los únicos cambios necesarios son añadir la operación `buscarKesimo` y mantener la información del atributo `tamanyo` en las rutinas `insertar`, `eliminar` y `eliminarMin`. Derivamos una nueva clase a partir de `ArbolBinarioBusqueda`, cuyo esqueleto se muestra en la Figura 18.14.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase ABBConRango
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ********************OPERACIONES PÚBLICAS*********************
11 // void insertar( x )       --> Inserta x
12 // void eliminar( x )       --> Elimina x
13 // void eliminarMin( )      --> Elimina el menor elemento
14 // Comparable  buscar( x )  --> Devuelve el elemento que ajusta con x
15 // Comparable buscarMin( ) --> Devuelve el menor elemento
16 // Comparable buscarMax( ) --> Devuelve el mayor elemento
17 // Comparable buscarKesimo( int k )
18 //                          --> Buscar K-ésimo menor elemento
19 // boolean esVacio( )       --> Devuelve true si vacío; si no, false
20 // void vaciar( )           --> Elimina todos los elementos
21 // void imprimirArbol( )    --> Imprime el árbol ordenadamente
22 // ************************ERRORES*****************************
23 // Muchas rutinas lanzan ElementoNoEncontrado en condiciones
      degeneradas
24 // insertar lanza ElementoDuplicado si el elemento está en el árbol
25
26 /**
27  * Implementa un árbol binario de búsqueda sin equilibrar.
28  * Nótese que todas las comparaciones se basan en el método compara.
29  */
30 public class ABBConRango extends ArbolBinarioBusqueda
31 {
32   public Comparable buscarKesimo( int k ) throws ElementoNoEncontrado
33     { return buscarKesimo( k, raiz ).dato; }
34
35   // Los métodos internos insertar, eliminar y eliminarMin se
36   // sobrescriben. Se añade el método interno para buscarKesimo
37 }
```

Figura 18.14 Esqueleto de la clase de los árboles binarios de búsqueda con búsqueda por posición.

Debemos redefinir los métodos ocultos. Los métodos públicos pueden permanecer sin cambios.

Veamos los nuevos métodos públicos. Como el constructor no se define explícitamente, se emplea el constructor generado por defecto; los atributos heredados son inicializados por el constructor de `ArbolBinarioBusqueda`. El nuevo método público `buscarKesimo` se declara en la línea 32 y llama al método privado correspondiente. Los métodos públicos, como `insertar`, no necesitan programarse de nuevo, ya que sus cuerpos son idénticos a las versiones en `ArbolBinarioBusqueda`.

`buscarKesimo` se implementa fácilmente una vez que se cuenta con los atributos de tamaño.

Los que cambian son los métodos de ayuda privados. Necesitamos que el `insertar` público invoque al correspondiente método de ayuda privado. Observe, sin embargo, que los métodos privados no deben tener el atributo `static`, ya que la decisión debe hacerse empleando ligado dinámico, en lugar de ligado estático.

La operación `buscarKesimo` de la Figura 18.15 se ha escrito de forma recursiva, aunque claramente no necesita serlo. Esta rutina sigue la descripción del algoritmo, línea a línea. La comprobación de la referencia `null` en la línea 11 es

```
1  /**
2   * Método interno para buscar el K-ésimo menor elemento de un subárbol.
3   * @param k el orden del elemento deseado (1 es el menor elemento).
4   * @return el nodo que contiene el K-ésimo menor elemento del subárbol.
5   * @exception ElementoNoEncontrado si k es menor que
6   *     1 o mayor que el tamaño del subárbol.
7   */
8  protected NodoBinario
9  buscarKesimo( int k, NodoBinario t ) throws ElementoNoEncontrado
10 {
11   if( t == null )
12     throw new ElementoNoEncontrado( "buscarKesimo de ABBR" );
13   int tamanyoIzquierdo = ( t.izquierdo != null ) ? t.izquierdo.tamanyo : 0;
14
15   if( k <= tamanyoIzquierdo )
16     return buscarKesimo( k, t.izquierdo );
17   if( k == tamanyoIzquierdo + 1 )
18     return t;
19   else
20     return buscarKesimo( k - tamanyoIzquierdo - 1, t.derecho );
21 }
```

Figura 18.15 Operación `buscarKesimo` de los árboles binarios de búsqueda con búsqueda por posición.

necesaria, ya que `k` puede no ser válido. En la línea 13 se calcula el tamaño del subárbol izquierdo. Si éste existe, el acceso a su atributo `tamanyo` nos da la respuesta pedida. Si no existe, podemos considerar que su tamaño es cero. Nótese que este test se realiza después de asegurar que la referencia `t` no es `null`.

`insertar` se muestra en la Figura 18.16. La parte que potencialmente tiene truco es que cuando la inserción tiene éxito, queremos en efecto incrementar el

`insertar` y `eliminar` utilizan un hábil truco para no actualizar la información del tamaño de los nodos cuando la operación no tiene éxito.

```
1  /**
2   * Método interno para insertar en un subárbol, ajustando
3   *     el tamaño según corresponda.
4   * @param x el elemento a insertar.
5   * @param t la raíz del subárbol.
6   * @return la nueva raíz.
7   * @exception ElementoDuplicado si el elemento que ajusta con
8   *     x ya está en el subárbol de raíz t.
9   */
10 protected NodoBinario
11 insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
12 {
13     if( t == null )
14         return new NodoBinario( x, null, null );
15     else if( x.compara( t.dato ) < 0 )
16         t.izquierdo = insertar( x, t.izquierdo );
17     else if( x.compara( t.dato ) > 0 )
18         t.derecho = insertar( x, t.derecho );
19     else
20         throw new ElementoDuplicado( "insertar de ABBR" );
21
22     t.tamanyo++;
23     return t;
24 }
```

Figura 18.16 Operación `insertar` de los árboles binarios de búsqueda con búsqueda por posición.

```
1  /**
2   * Método interno para eliminar el menor elemento de un subárbol,
3   *      ajustando el tamaño de los nodos cuando corresponda.
4   * @param t la raíz del subárbol.
5   * @return la nueva raíz.
6   * @exception ElementoNoEncontrado si el subárbol está vacío.
7   */
8  protected NodoBinario
9  eliminarMin( NodoBinario t ) throws ElementoNoEncontrado
10 {
11     if( t == null )
12         throw new ElementoNoEncontrado( "eliminarMin de ABBR" );
13     if( t.izquierdo == null )
14         return t.derecho;
15     t.izquierdo = eliminarMin( t.izquierdo );
16
17     t.tamanyo--;
18     return t;
19 }
```

Figura 18.17 Operación `eliminarMin` de los árboles binarios de búsqueda con búsqueda por posición.

```
1  /**
2   * Método interno para eliminar de un subárbol, ajustando
3   *     el tamaño de los nodos cuando corresponda.
4   * @param x el elemento a eliminar.
5   * @param t la raíz del subárbol.
6   * @return la nueva raíz.
7   * @exception ElementoNoEncontrado si no hay ningún elemento
8   *     que ajuste con x en el subárbol de raíz t.
9   */
10 protected NodoBinario
11 eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
12 {
13     if( t == null )
14         throw new ElementoNoEncontrado( "eliminar de ABBR" );
15     if( x.compara( t.dato ) < 0 )
16         t.izquierdo = eliminar( x, t.izquierdo );
17     else if( x.compara( t.dato ) > 0 )
18         t.derecho = eliminar( x, t.derecho );
19     else if( t.izquierdo != null && t.derecho != null ) // Dos hijos
20     {
21         t.dato = buscarMin( t.derecho ).dato;
22         t.derecho = eliminarMin( t.derecho );
23     }
24     else
25         return ( t.izquierdo != null ) ? t.izquierdo : t.derecho;
26     t.tamanyo--;
27     return t;
28 }
```

Figura 18.18 Operación `eliminar` de los árboles binarios de búsqueda con búsqueda por posición.

atributo `tamanyo` de `t` y devolver la nueva raíz del subárbol. Pero si la llamada recursiva falla, dicho atributo no debe modificarse, debiéndose lanzar una excepción. Ahora bien, ¿es posible que en una inserción fallida los tamaños de algunos nodos cambien? La respuesta es no: `tamanyo` sólo se actualiza si la llamada recursiva termina sin lanzar ninguna excepción. Nótese que cuando se genera un nuevo nodo mediante la llamada `new`, el atributo `tamanyo` se inicializa a 1 en el constructor de `NodoBinario`.

La Figura 18.17 muestra que el mismo truco puede utilizarse en `eliminarMin`. Si la llamada recursiva finaliza con éxito, el atributo `tamanyo` se decrementa; si la llamada recursiva falla, `tamanyo` permanece invariante. `eliminar` es parecido, y se muestra en la Figura 18.18.

18.3 Análisis de las operaciones de los árboles binarios de búsqueda

El coste de una operación es proporcional a la profundidad del último nodo accedido. Este coste es logarítmico en un árbol bien equilibrado, pero puede ser lineal en un árbol degenerado.

Es fácil comprobar que el coste de cada operación de los árboles binarios de búsqueda (`insertar`, `buscar` y `eliminar`) es proporcional al número de nodos consultados durante la operación. Así, podemos considerar que el coste del acceso a cada nodo es 1 más su profundidad (recordamos que la profundidad mide el número de arcos en lugar del número de nodos). Todo esto nos da el coste de una búsqueda con éxito.

La Figura 18.19 muestra dos árboles. El de la izquierda es un árbol bien equilibrado con 15 nodos. El coste de acceder a cualquier nodo es de a lo sumo, 4 unidades, aunque algunos nodos requieren menos accesos. Esto es prácticamente análogo a la situación que se produce en el algoritmo de búsqueda binaria. En consecuencia, si el árbol está bien equilibrado, el coste de los accesos es logarítmico.

Desafortunadamente, no tenemos la garantía de que todo el árbol esté bien equilibrado. El segundo árbol de la Figura 18.19 es el ejemplo clásico de árbol no equilibrado. Aquí, los N nodos están en el camino a recorrer hasta llegar al nodo de mayor profundidad, de forma que el coste de la búsqueda en el peor de los casos es $O(N)$. Debido a que el árbol de búsqueda ha degenerado en una lista

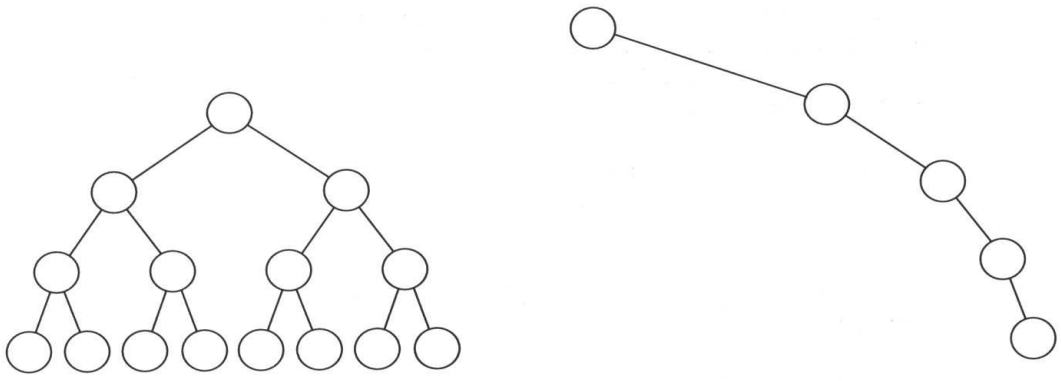

Figura 18.19 El árbol bien equilibrado de la izquierda tiene una profundidad de $\lfloor \log N \rfloor$; el árbol no equilibrado de la derecha tiene una profundidad de $N - 1$.

enlazada, el tiempo medio de la búsqueda en *esta instancia particular* es la mitad del coste en el caso peor, que también es $O(N)$. Así, se tienen dos extremos: en el caso mejor, el coste de los accesos es logarítmico, y en el caso peor es lineal.

Entonces, ¿cuál es la media? ¿Tienden la mayoría de los árboles binarios de búsqueda a inclinarse hacia los casos bien o mal equilibrados, o existe algún punto medio, como $\sqrt{N}$? La respuesta es idéntica a la obtenida en el caso de quicksort: el caso medio es un 38 por ciento peor que el caso mejor.

En media, la profundidad es un 38 por ciento peor que en el caso mejor. Este resultado es idéntico al obtenido al analizar quicksort.

En esta sección demostramos que la profundidad media de los nodos de un árbol binario de búsqueda es logarítmica, asumiendo que cada árbol se genera como resultado de secuencias aleatorias de inserción (sin operaciones de tipo `eliminar`). Para comprender qué significa esto, considere el resultado de insertar tres elementos en el árbol vacío. Como sólo es importante su orden relativo, podemos suponer, sin pérdida de generalidad, que los tres elementos son 1, 2 y 3. Entonces, existen seis posibles órdenes de inserción: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) y (3, 2, 1). En nuestra demostración asumiremos que cada uno de estos casos es equiprobable. Los árboles binarios de búsqueda que resultan de estas inserciones se muestran en la Figura 18.20. Nótese que el árbol de raíz 2 se genera también por la secuencia (2, 1, 3). Así que algunos árboles son más probables que otros y, tal y como se muestra, los árboles bien equilibrados son más probables que los mal equilibrados (aunque esto no es evidente a partir del ejemplo propuesto por ser éste demasiado pequeño).

Comenzamos con la siguiente definición:

DEFINICIÓN: La *longitud del camino interno* de un árbol binario es igual a la suma de las profundidades de sus nodos.

La *longitud del camino interno* se emplea para calcular el coste de una búsqueda con éxito.

Cuando en un árbol dividimos la longitud de su camino interno por el número de nodos del árbol, obtenemos la profundidad media de un nodo. Incrementando esta media en una unidad, obtenemos el coste medio de una búsqueda con éxito en el árbol. Por este motivo deseamos calcular la longitud media del camino interno de un árbol binario de búsqueda, donde la media se calcula considerando equiprobables todas las permutaciones de entrada. Esto se hace fácilmente considerando el árbol desde el punto de vista recursivo y empleando las técnicas mostradas en el análisis del quicksort en la Sección 8.6. La longitud media del camino interno se establece en al Teorema 18.1.

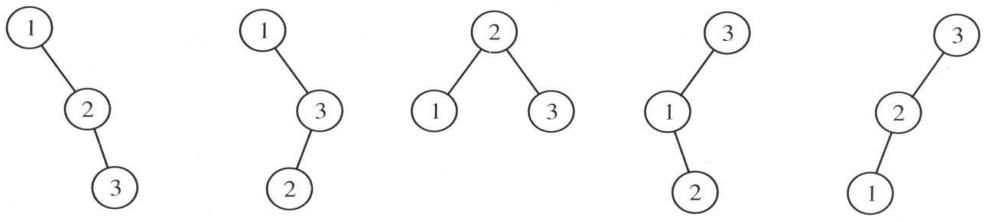

Figura 18.20 Árboles binarios de búsqueda obtenidos de la inserción de la permutación de 1, 2 y 3; la probabilidad de obtener el árbol bien equilibrado del centro es el doble que la de obtener cada uno de los restantes resultados.

Teorema 18.1

La longitud del camino interno de un árbol binario de búsqueda es, aproximadamente en media, $1{,}38N \log N$, *asumiendo que todas las permutaciones son equiprobables.*

Demostración

Sea $D(N)$ *la longitud media del camino interno de los árboles de N nodos.* $D(1) = 0$. *Un árbol T con N nodos está formado por un subárbol izquierdo de i nodos, un subárbol derecho de* $(N - i - 1)$*-nodos, además de una raíz a profundidad* 0 *con* $0 \leqslant i < N$. *Por hipótesis, cada valor de i es igualmente probable. Para un i dado,* $D(i)$ *es la longitud media del camino interno del subárbol izquierdo respecto a su raíz. En T, todos estos nodos están un nivel más abajo. Así, la contribución media de los nodos del subárbol izquierdo a la longitud media del camino interno de T es* $(1/N)\Sigma_{i=0}^{N-1} D(i)$, *más 1 para cada nodo del subárbol izquierdo. Lo mismo se verifica para el subárbol derecho. Obtenemos así la recurrencia* $D(N) = (2/N)(\Sigma_{i=0}^{N-1} D(i)) + N - 1$, *que es idéntica a la recurrencia obtenida para quicksort, en la Sección 8.6, y que allí resolvimos. Como consecuencia, obtenemos que la longitud media del camino interno es* $O(N \log N)$.

El algoritmo de inserción implica que el coste de una inserción es igual al coste de una búsqueda sin éxito, el cual se mide usando la *longitud del camino externo*. En algunas ocasiones, durante una inserción o en una búsqueda sin éxito, se alcanza el test `t==null`. Recordemos que en un árbol de N nodos existen $N + 1$ referencias `null`. La longitud del camino externo es el número total de nodos a los que se accede, incluyendo el nodo `null` para cada una de esas $N + 1$ referencias `null`. En ocasiones el nodo `null` recibe el nombre de *nodo externo*, lo que explica el término *longitud del camino externo*. Como se muestra posteriormente en este mismo capítulo, a veces es conveniente emplear un sustituto para el nodo `null`.

La *longitud del camino externo* se emplea para medir el coste de una búsqueda sin éxito.

DEFINICIÓN: La *longitud del camino externo* de un árbol binario de búsqueda es la suma de los costes de acceso a todas las referencias `null`. Para estos propósitos, la referencia `null` de las hojas se considera como un nodo.

Dividiendo la longitud media del camino externo entre $N + 1$, obtenemos el coste medio de una inserción o de una búsqueda sin éxito. Al igual que en el algoritmo de búsqueda binaria, el coste medio de una búsqueda sin éxito es ligeramente superior al coste de una búsqueda exitosa. Ésta es la conclusión del Teorema 18.2

Teorema 18.2

Para cualquier árbol T, si $IPL(T)$ *es la longitud de su camino interno y* $EPL(T)$ *la longitud de su camino externo, entonces, si T tiene N nodos, se tiene* $EPL(T) = IPL(T) + 2N$.

Demostración

El teorema se demuestra por inducción y se deja como ejercicio al lector en el Ejercicio 18.8.

enlazada, el tiempo medio de la búsqueda en *esta instancia particular* es la mitad del coste en el caso peor, que también es $O(N)$. Así, se tienen dos extremos: en el caso mejor, el coste de los accesos es logarítmico, y en el caso peor es lineal.

Entonces, ¿cuál es la media? ¿Tienden la mayoría de los árboles binarios de búsqueda a inclinarse hacia los casos bien o mal equilibrados, o existe algún punto medio, como $\sqrt{N}$? La respuesta es idéntica a la obtenida en el caso de quicksort: el caso medio es un 38 por ciento peor que el caso mejor.

En media, la profundidad es un 38 por ciento peor que en el caso mejor. Este resultado es idéntico al obtenido al analizar quicksort.

En esta sección demostramos que la profundidad media de los nodos de un árbol binario de búsqueda es logarítmica, asumiendo que cada árbol se genera como resultado de secuencias aleatorias de inserción (sin operaciones de tipo `eliminar`). Para comprender qué significa esto, considere el resultado de insertar tres elementos en el árbol vacío. Como sólo es importante su orden relativo, podemos suponer, sin pérdida de generalidad, que los tres elementos son 1, 2 y 3. Entonces, existen seis posibles órdenes de inserción: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) y (3, 2, 1). En nuestra demostración asumiremos que cada uno de estos casos es equiprobable. Los árboles binarios de búsqueda que resultan de estas inserciones se muestran en la Figura 18.20. Nótese que el árbol de raíz 2 se genera también por la secuencia (2, 1, 3). Así que algunos árboles son más probables que otros y, tal y como se muestra, los árboles bien equilibrados son más probables que los mal equilibrados (aunque esto no es evidente a partir del ejemplo propuesto por ser éste demasiado pequeño).

Comenzamos con la siguiente definición:

DEFINICIÓN: La *longitud del camino interno* de un árbol binario es igual a la suma de las profundidades de sus nodos.

La *longitud del camino interno* se emplea para calcular el coste de una búsqueda con éxito.

Cuando en un árbol dividimos la longitud de su camino interno por el número de nodos del árbol, obtenemos la profundidad media de un nodo. Incrementando esta media en una unidad, obtenemos el coste medio de una búsqueda con éxito en el árbol. Por este motivo deseamos calcular la longitud media del camino interno de un árbol binario de búsqueda, donde la media se calcula considerando equiprobables todas las permutaciones de entrada. Esto se hace fácilmente considerando el árbol desde el punto de vista recursivo y empleando las técnicas mostradas en el análisis del quicksort en la Sección 8.6. La longitud media del camino interno se establece en al Teorema 18.1.

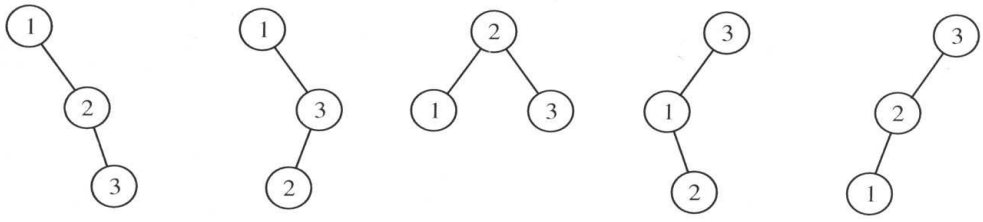

Figura 18.20 Árboles binarios de búsqueda obtenidos de la inserción de la permutación de 1, 2 y 3; la probabilidad de obtener el árbol bien equilibrado del centro es el doble que la de obtener cada uno de los restantes resultados.

Teorema 18.1

La longitud del camino interno de un árbol binario de búsqueda es, aproximadamente en media, $1{,}38N \log N$, asumiendo que todas las permutaciones son equiprobables.

Demostración

Sea $D(N)$ la longitud media del camino interno de los árboles de N nodos. $D(1) = 0$. Un árbol T con N nodos está formado por un subárbol izquierdo de i nodos, un subárbol derecho de $(N - i - 1)$-nodos, además de una raíz a profundidad 0 con $0 \leqslant i < N$. Por hipótesis, cada valor de i es igualmente probable. Para un i dado, $D(i)$ es la longitud media del camino interno del subárbol izquierdo respecto a su raíz. En T, todos estos nodos están un nivel más abajo. Así, la contribución media de los nodos del subárbol izquierdo a la longitud media del camino interno de T es $(1/N)\Sigma_{i=0}^{N-1} D(i)$, más 1 para cada nodo del subárbol izquierdo. Lo mismo se verifica para el subárbol derecho. Obtenemos así la recurrencia $D(N) = (2/N)(\Sigma_{i=0}^{N-1} D(i)) + N - 1$, que es idéntica a la recurrencia obtenida para quicksort, en la Sección 8.6, y que allí resolvimos. Como consecuencia, obtenemos que la longitud media del camino interno es $O(N \log N)$.

La *longitud del camino externo* se emplea para medir el coste de una búsqueda sin éxito.

El algoritmo de inserción implica que el coste de una inserción es igual al coste de una búsqueda sin éxito, el cual se mide usando la *longitud del camino externo*. En algunas ocasiones, durante una inserción o en una búsqueda sin éxito, se alcanza el test `t==null`. Recordemos que en un árbol de N nodos existen $N + 1$ referencias `null`. La longitud del camino externo es el número total de nodos a los que se accede, incluyendo el nodo `null` para cada una de esas $N + 1$ referencias `null`. En ocasiones el nodo `null` recibe el nombre de *nodo externo*, lo que explica el término *longitud del camino externo*. Como se muestra posteriormente en este mismo capítulo, a veces es conveniente emplear un sustituto para el nodo `null`.

DEFINICIÓN: La *longitud del camino externo* de un árbol binario de búsqueda es la suma de los costes de acceso a todas las referencias `null`. Para estos propósitos, la referencia `null` de las hojas se considera como un nodo.

Dividiendo la longitud media del camino externo entre $N + 1$, obtenemos el coste medio de una inserción o de una búsqueda sin éxito. Al igual que en el algoritmo de búsqueda binaria, el coste medio de una búsqueda sin éxito es ligeramente superior al coste de una búsqueda exitosa. Ésta es la conclusión del Teorema 18.2

Teorema 18.2

Para cualquier árbol T, si $IPL(T)$ es la longitud de su camino interno y $EPL(T)$ la longitud de su camino externo, entonces, si T tiene N nodos, se tiene $EPL(T) = IPL(T) + 2N$.

Demostración

El teorema se demuestra por inducción y se deja como ejercicio al lector en el Ejercicio 18.8.

Las operaciones `eliminar` aleatorias no preservan la aleatoriedad de un árbol. Los efectos no son completamente comprensibles desde el punto de vista teórico, pero parece que son relativamente despreciables en la práctica.

Es tentador concluir sin más que de estos resultados se sigue que el tiempo medio de ejecución de todas las operaciones es $O(\log N)$. Esto parece ser cierto en la práctica, pero no ha sido establecido analíticamente debido a que en la demostración de los resultados anteriores no se ha tenido en cuenta el efecto de las operaciones de eliminación. De hecho, un examen más detallado sugiere que nuestro algoritmo de eliminación probablemente planteará problemas, ya que `eliminar` siempre reemplaza un nodo borrado con dos hijos con un nodo del subárbol derecho. Podría parecer que el efecto de este reemplazamiento es el desequilibrio del árbol, inclinándolo hacia la izquierda. De hecho se ha demostrado que si se construye un árbol binario de búsqueda de forma aleatoria y se hacen sobre él N^2 pares de combinaciones aleatorias `insertar`/`eliminar` (en las que el orden de `insertar` y `eliminar` sea también aleatorio), entonces los árboles de búsqueda binarios resultantes tendrán una profundidad de $O(\sqrt{N})$. Sin embargo, no se ha podido demostrar que si nos limitamos a una cantidad razonable de operaciones `insertar` y `eliminar` aleatorias el árbol se desequilibre de forma apreciable. De hecho, y de forma un tanto sorprendente, para árboles de búsqueda pequeños, el algoritmo `eliminar` parece equilibrar el árbol. Como consecuencia, parece razonable asumir que para datos de entrada aleatorios, todas las operaciones tienen, en media, coste logarítmico, aunque esto no haya sido probado formalmente. En el Ejercicio 18.26 se describen algunas estrategias alternativas de eliminación.

El problema más importante no es el posible desequilibrio causado por el algoritmo `eliminar`, sino el hecho de que la secuencia de entrada esté ordenada, es entonces cuando nos encontramos en el caso peor. Cuando esto ocurre, tenemos un grave problema: el coste de cada operación es lineal (para secuencias de N operaciones) en lugar de logarítmico. La situación se corresponde a la diferencia entre el algoritmo quicksort y el de la ordenación por inserción. El tiempo de ejecución resultante es completamente inaceptable.

Más aún, no son sólo problemáticas las entradas ordenadas, sino cualquier entrada que contenga secuencias largas no aleatorias. Una solución a este problema es insistir en una condición estructural adicional llamada *equilibrio*: no se permite que ningún nodo esté a demasiada profundidad.

Los *árboles binarios de búsqueda bien equilibrados* añaden una propiedad estructural que garantiza una profundidad logarítmica en el caso peor. Las modificaciones son más lentas, pero los accesos son más rápidos.

Existen numerosos métodos para implementar *árboles binarios de búsqueda bien equilibrados*. Muchos de ellos son bastante más complicados que los árboles binarios de búsqueda estándar, y en media, consumen más tiempo en las inserciones y las eliminaciones. Sin embargo, evitan los problemas de los embarazosos casos simples. Además, al generar árboles equilibrados, se consigue que los accesos sean más eficientes. Normalmente, las longitudes de sus caminos internos están bastante más cerca del caso óptimo $N \log N$ que lo que supondría el $1{,}38N \log N$ que se alcanzaba en el caso ordinario, de modo que el tiempo de búsqueda medio viene a ser un 25 por ciento menor que en dicho caso.

18.4 Árboles AVL

El primer tipo de árbol binario de búsqueda bien equilibrado fue el *árbol AVL* (debe su nombre a sus autores Adelson-Velskii y Landis). Los árboles AVL ilustran las ideas básicas de una amplia clase de árboles binarios de búsqueda bien equilibrados. Se tratan de árboles binarios de búsqueda con una condición adicional de

equilibrio. Dicha condición debe ser fácil de mantener, asegurando que la profundidad del árbol sea siempre $O(\log N)$. La idea más sencilla es exigir que los subárboles izquierdo y derecho tengan la misma profundidad. La recursión nos indica que dicha idea se aplica a todos los nodos del árbol, ya que cada uno de ellos es a su vez la raíz de algún subárbol. Esta condición asegura que la profundidad del árbol es logarítmica. Sin embargo, es demasiado restrictiva ya que es complicado mantener la condición de equilibrio al insertar nuevos elementos. Así, los árboles AVL emplean una noción de equilibrio que es algo más débil, pero lo suficientemente fuerte para garantizar la profundidad logarítmica.

El *árbol AVL* fue el primer tipo de árbol binario de búsqueda bien equilibrado. Tiene una gran importancia histórica y muestra muchas de las ideas básicas empleadas en otros esquemas.

18.4.1 Propiedades

DEFINICIÓN: Un *árbol AVL* es un árbol binario de búsqueda con una propiedad adicional de equilibrio, según la cual, las alturas de los hijos derecho e izquierdo sólo pueden diferir, a lo sumo, en una unidad. Como es usual, tomamos -1 como la altura del árbol vacío.

La Figura 18.21 muestra dos árboles binarios de búsqueda. El árbol de la izquierda satisface la condición de los árboles AVL y por tanto es un árbol AVL. El árbol de la derecha, que se obtiene al insertar 1 empleando el algoritmo ordinario, no es un árbol AVL, ya que los hijos izquierdos de los nodos más oscuros son dos unidades más profundos que los correspondientes hijos derechos. Si se inserta el elemento 13 empleando el algoritmo ordinario de inserción en un árbol binario de búsqueda, el nodo 16 tampoco cumpliría la condición. Esto es debido a que la altura del subárbol izquierdo sería 1, mientras que la del subárbol derecho sería -1.

Cada nodo de un árbol AVL tiene subárboles cuyas alturas difieren en, como mucho, una unidad. La altura del árbol vacío es -1

La condición de equilibrio AVL implica que el árbol tiene siempre una profundidad logarítmica. Para demostrarlo, basta mostrar que un árbol de altura H debe tener, por lo menos, C^H nodos, donde C es una constante mayor que 1. En otras palabras, el número mínimo de nodos de un árbol crece exponencialmente con la altura. Así, la profundidad máxima de un árbol con N elementos es $\log_C N$. El Teorema 18.3 muestra que todo árbol AVL de altura H tiene *muchos* nodos.

La altura de un árbol AVL es, como mucho, un 44 por ciento mayor que la mínima posible.

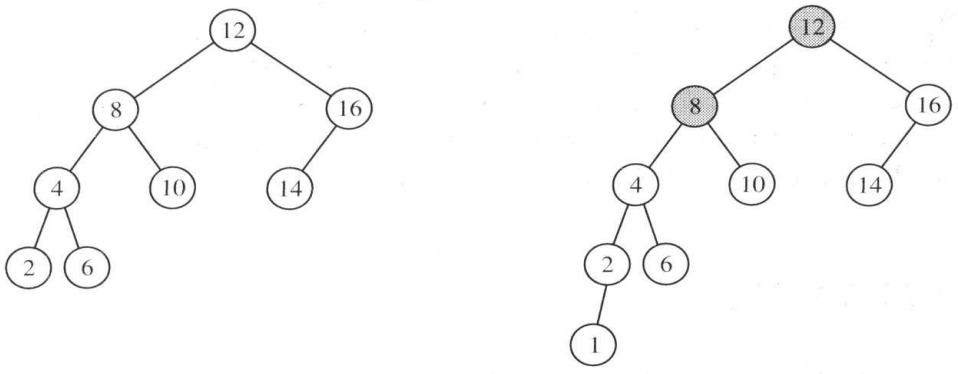

Figura 18.21 Dos árboles binarios de búsqueda: el árbol de la izquierda es un árbol AVL, mientras que el de la derecha no lo es (los nodos no equilibrados aparecen en un tono más oscuro).

Teorema 18.3 *Un árbol AVL de altura H tiene por lo menos $F_{H+3} - 1$ nodos, donde F_i es el i-ésimo número de Fibonacci (véase la Sección 7.3.4).*

Demostración *Sea S_H el tamaño del menor árbol AVL de altura H. Claramente $S_0 = 1$ y $S_1 = 2$. La Figura 18.22 muestra que el menor árbol AVL de altura H debe tener subárboles de tamaño $H - 1$ y $H - 2$. Esto es debido a que al menos un subárbol tiene altura $H - 1$ y la condición de equilibrio implica que las alturas de los subárboles pueden diferir a lo sumo en una unidad. Dichos subárboles deben tener el menor número de nodos posible para su altura, luego $S_H = S_{H-1} + S_{H-2} + 1$. Es sencillo completar la demostración empleando inducción.*

Del Ejercicio 7.8, tenemos $F_i \approx \phi^i/\sqrt{5}$ donde $\phi = (1 + \sqrt{5})/2 \approx 1{,}618$. Como consecuencia, un árbol AVL de altura H tiene, aproximadamente, $\phi^{H+3}/\sqrt{5}$ nodos. Así, su profundidad es, a lo sumo, logarítmica. Más exactamente, la altura de un árbol AVL satisface

$$H < 1{,}44 \log (N + 2) - 1{,}328 \qquad \textbf{(18.1)}$$

luego la altura en el caso peor es a lo sumo un 44 por ciento mayor que la mínima posible en los árboles binarios.

La profundidad de un nodo típico de un árbol AVL está muy próxima a la óptima, log *N*.

La profundidad media de un nodo en un árbol AVL construido aleatoriamente, tiende a acercarse a log *N*. El valor exacto aún no ha sido establecido analíticamente. Todavía no se sabe si es de la forma $\log N + C$ o de la forma $(1 + \varepsilon) \log N + C$, para algún ε, que podría ser aproximadamente 0,01. Las simulaciones no han sido capaces de demostrar de modo convincente que una forma sea más plausible que la otra.

Una modificación en un árbol AVL puede destruir el equilibrio. Debemos restablecer el equilibrio del árbol antes de dar por concluida la operación.

Una consecuencia de estas argumentaciones es que todas las operaciones de búsqueda en un árbol AVL tienen cota logarítmica en el caso peor. La dificultad estriba en que las operaciones que modifican el árbol, como `insertar` y `eliminar`, no son tan simples como antes. Esto es debido a que una inserción (o una eliminación) puede destruir el equilibrio de varios nodos del árbol, tal y como se muestra en la Figura 18.21. Se debe recuperar el equilibrio del árbol antes de considerar finalizada la operación. El algoritmo de inserción se describe aquí, dejándose como ejercicio en el Ejercicio 18.10 el algoritmo de eliminación.

Sólo los nodos del camino desde el punto de inserción hasta la raíz pueden ver alterado su equilibrio.

Una observación clave es que tras una inserción, sólo los nodos que se encuentran en el camino desde el punto de inserción hasta la raíz pueden ver alterado su equilibrio, ya que sólo se modifican subárboles de dichos nodos. Esto mismo

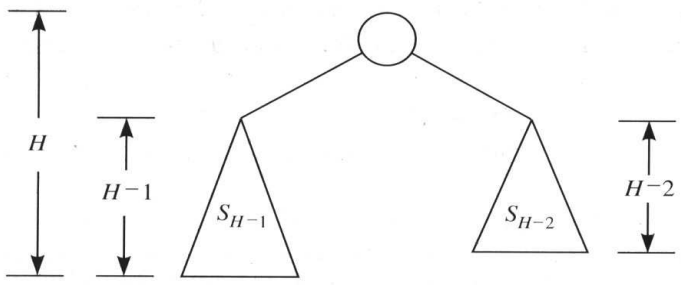

Figura 18.22 Árbol mínimo e altura *H*.

sucede en muchos de los algoritmos de los árboles de búsqueda equilibrados. Al ir recorriendo el camino hacia la raíz y actualizando la información del equilibrio, podemos encontrar un nodo cuyo nuevo equilibrio incumpla la condición AVL. Esta sección muestra cómo recuperar el equilibrio del árbol desde el primer nodo (es decir, el más profundo) que incumple la condición AVL, demostrando que el mecanismo propuesto garantiza que el árbol completo obtenido satisface la propiedad AVL.

Supongamos que el nodo cuyo equilibrio debemos ajustar es X. Como cualquier nodo tiene a lo sumo dos hijos, y la diferencia de las profundidades de los subárboles de X es 2, el incumplimiento de la propiedad puede darse en uno de los cuatro casos siguientes:

Si arreglamos adecuadamente el equilibrio del nodo desequilibrado más profundo, recuperaremos el equilibrio correcto de todo el árbol. Existen cuatro casos que debemos considerar; dos de ellos son simétricos a los otros dos.

1. Una inserción en el subárbol izquierdo del hijo izquierdo de X.
2. Una inserción en el subárbol derecho del hijo izquierdo de X.
3. Una inserción en el subárbol izquierdo del hijo derecho de X.
4. Una inserción en el subárbol derecho del hijo derecho de X.

Los casos 1 y 4 son simétricos respecto de X, al igual que los casos 2 y 3. Como consecuencia, sólo existen dos casos básicos, aunque desde la perspectiva de la programación, debamos seguir distinguiendo cuatro casos (y bastantes subcasos especiales).

El primer caso, en el que la inserción se produce en los «márgenes» (es decir, a la izquierda de la izquierda o a la derecha de la derecha), se recupera con una *rotación simple* del árbol. Una rotación simple intercambia los papeles de los padres y de los hijos, manteniendo la ordenación del árbol. El segundo caso, en el que la inserción se produce «dentro» (es decir, a la izquierda de la derecha o a la derecha de la izquierda), se resuelve de forma más compleja mediante una *rotación doble*. Éstas son las operaciones básicas sobre árboles, empleadas en numerosas ocasiones en los algoritmos de equilibrado de árboles. El resto de esta sección describe estas rotaciones y demuestra que ambas son suficientes para mantener la condición de equilibrio.

El equilibrio se recupera mediante rotaciones del árbol. Una *rotación simple* intercambia los papeles de los padres y los hijos manteniendo la ordenación del árbol.

18.4.2 Rotación simple

La Figura 18.23 muestra la rotación simple que resuelve el caso 1. El gráfico que muestra el árbol antes del ajuste es el de la izquierda; a la derecha se muestra el árbol después de dicho ajuste. Veamos lo que vamos a hacer. El nodo k_2 incumple la propiedad de equilibrio AVL ya que su subárbol izquierdo es dos niveles más profundo que su subárbol derecho (en esta sección empleamos las líneas punteadas para marcar los niveles). La situación descrita es la única posible dentro del caso 1 que permite al nodo k_2 satisfacer la propiedad AVL antes de la inserción e incumplirla después. El subárbol A ha crecido en un nivel, provocando que sea, exactamente, dos niveles más profundo que C. B no puede estar al mismo nivel que el nuevo A, ya que si no k_2 no estaría bien equilibrado antes de la inserción. B tampoco puede estar al mismo nivel que C, ya que, de estarlo, k_1 sería el primer nodo en el camino que incumpliría la condición de equilibrio AVL (y estamos suponiendo que lo es k_2).

La rotación simple resuelve los casos extremos (casos 1 y 4). Realizamos la rotación entre un nodo y su hijo. El resultado es un árbol binario de búsqueda que satisface la propiedad AVL.

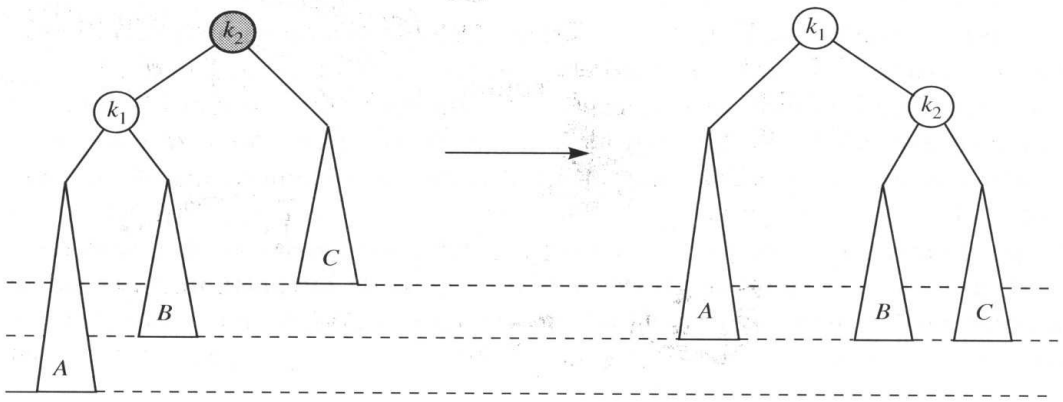

Figura 18.23 Rotación simple para tratar el caso 1.

Para reequilibrar de forma correcta el árbol, queremos subir A un nivel y bajar C otro tanto. Nótese que de hecho esto es algo más de lo que requiere la propiedad AVL. Para hacerlo, debemos reorganizar los nodos en un árbol de búsqueda equivalente, tal y como se muestra en la Figura 18.23. Ello queda ilustrado por el siguiente razonamiento abstracto: consideramos el árbol como una estructura flexible, ponemos un peso en el nodo k_2, cerramos los ojos, y dejamos que actúe la gravedad reequilibrando la balanza. El resultado es que k_1 será la nueva raíz.

La propiedad de los árboles binarios de búsqueda indica que en el árbol inicial, $k_2 > k_1$, así que en el nuevo árbol k_2 se convierte en el hijo derecho de k_1. A y C continúan como hijo izquierdo de k_1 e hijo derecho de k_2, respectivamente. El subárbol B, que contiene los elementos entre k_1 y k_2, puede colocarse en el nuevo árbol como hijo izquierdo de k_2, satisfaciéndose así todos los requerimientos de la ordenación.

Una rotación es suficiente para resolver los casos 1 y 4 en los árboles AVL.

Para hacer todo esto sólo se precisan los cambios de las referencias de los hijos mostrados en la Figura 18.24, que producen un nuevo árbol binario que ahora es un AVL. Esto es debido a que A sube un nivel, B permanece en el mismo y C desciende un nivel. k_1 y k_2 no sólo satisfacen la propiedad AVL, sino que además sus subárboles son de la misma altura. Más aún, la nueva altura del subárbol completo es *exactamente la misma* que la altura del subárbol previo a la inserción, que ha provocado el crecimiento de A. Así, no es necesario realizar ninguna actualización más de las alturas de los nodos que se encuentran en el camino hacia la raíz, y como consecuencia, *no se necesitan más rotaciones*. Esta rotación simple se uti-

```
1  /**
2   * Rotación de un nodo de un árbol binario con hijo izquierdo.
3   * En los árboles AVL, es la rotación simple para el caso 1.
4   */
5  static NodoBinario conHijoIzquierdo( NodoBinario k2 )
6  {
7      NodoBinario k1 = k2.izquierdo;
8      k2.izquierdo = k1.derecho;
9      k1.derecho = k2;
10     return k1;
11 }
```

Figura 18.24 Código para la rotación simple (caso 1).

liza mucho en otros algoritmos de equilibrado presentados en este capítulo. Debido a ello, la hacemos formar parte del paquete `Rotaciones`.

La Figura 18.25 muestra que después de la inserción de 1 en el árbol AVL, el nodo 8 está desequilibrado. Claramente estamos en el primer caso, ya que 1 está en el subárbol izquierdo del hijo izquierdo de 8. En consecuencia, realizamos una rotación simple entre 8 y 4, obteniéndose el árbol de la derecha. Como ya hemos mencionado anteriormente en esta sección, el caso 4 es simétrico a éste. La rotación adecuada se ilustra en la Figura 18.26, y el código que la implementa se muestra en la Figura 18.27. Este método también forma parte del paquete `Rotaciones`.

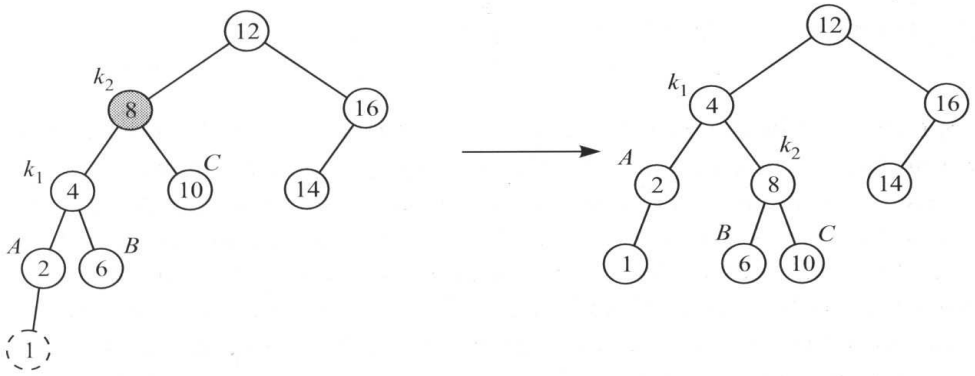

Figura 18.25 La rotación simple arregla el árbol AVL tras la inserción de 1.

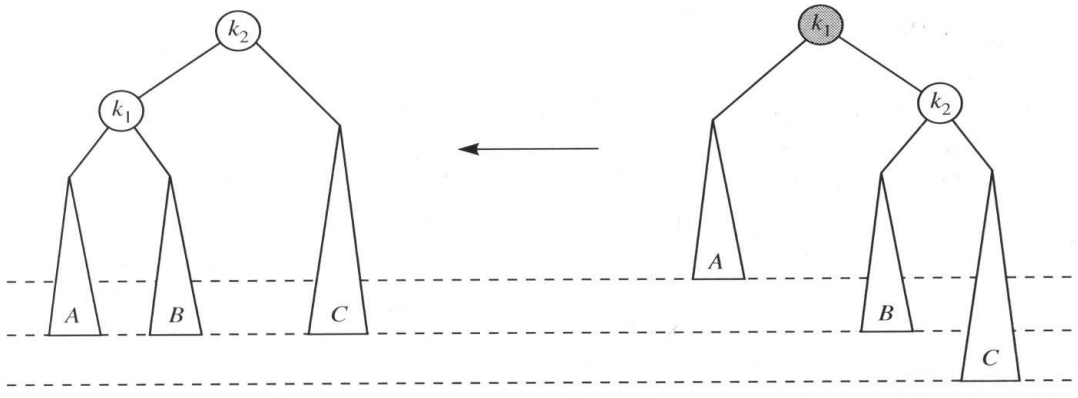

Figura 18.26 Rotación simple simétrica para resolver el caso 4.

```
1  /**
2   * Rotación de un nodo de un árbol binario con hijo derecho.
3   * En los árboles AVL, es la rotación simple para el caso 4.
4   */
5  static NodoBinario conHijoDerecho( NodoBinario k1 )
6  {
7      NodoBinario k2 = k1.derecho;
8      k1.derecho = k2.izquierdo;
9      k2.izquierdo = k1;
10     return k2;
11 }
```

Figura 18.27 Código para la rotación simple (caso 4).

18.4.3 Rotación doble

La rotación simple no resuelve los casos interiores (2 y 3). Estos casos requieren una *rotación doble*, que manipula tres nodos y cuatro subárboles.

La rotación simple tiene un problema. Tal y como muestra la Figura 18.28, no funciona en el caso 2 (ni, por simetría, en el caso 3). El problema estriba en que el subárbol Q es demasiado profundo; una rotación simple no reduce lo suficiente su profundidad. La *rotación doble* que resuelve el problema se muestra en la Figura 18.29.

El hecho de que en la Figura 18.28 se inserte un elemento en el subárbol Q garantiza que no está vacío. Asumiremos que tiene una raíz y dos subárboles (posiblemente vacíos), así que podemos ver el árbol como cuatro subárboles conectados por tres nodos. Renombramos dichos subárboles como A, B, C y D. Como sugiere la Figura 18.29, exactamente uno de los árboles B o C es dos niveles más profundo que D, aunque no podemos asegurar cuál de ellos es. Esto carece de importancia por lo que, en la figura, tanto B como C se han dibujado 1,5 niveles por debajo de D.

Para recuperar el equilibrio, observamos que es imposible dejar k_3 como raíz, y además que la rotación entre k_3 y k_1 no funciona, tal y como se comprueba en la Figura 18.28. La única alternativa es colocar a k_2 como nueva raíz. Esto fuerza a k_1 a ser el hijo izquierdo de k_2 y a k_3 a ser el hijo derecho de k_2. Todo ello determina completamente las posiciones de los cuatro subárboles. Es fácil comprobar que el árbol resultante cumple la propiedad AVL. Además, como en el caso de la rotación simple, esto restablece la altura que tenía el árbol antes de la inserción, garantizando que la recuperación del equilibrio y la modificación de alturas han sido completadas.

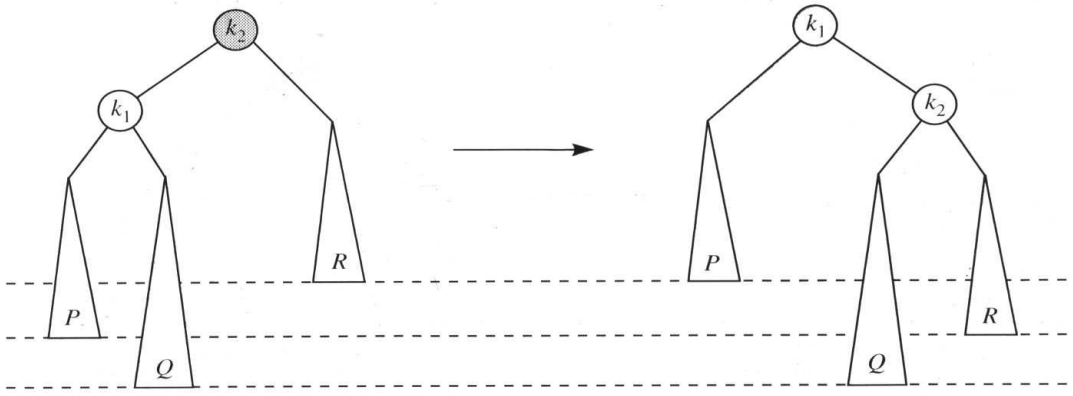

Figura 18.28 La rotación simple no arregla el caso 2.

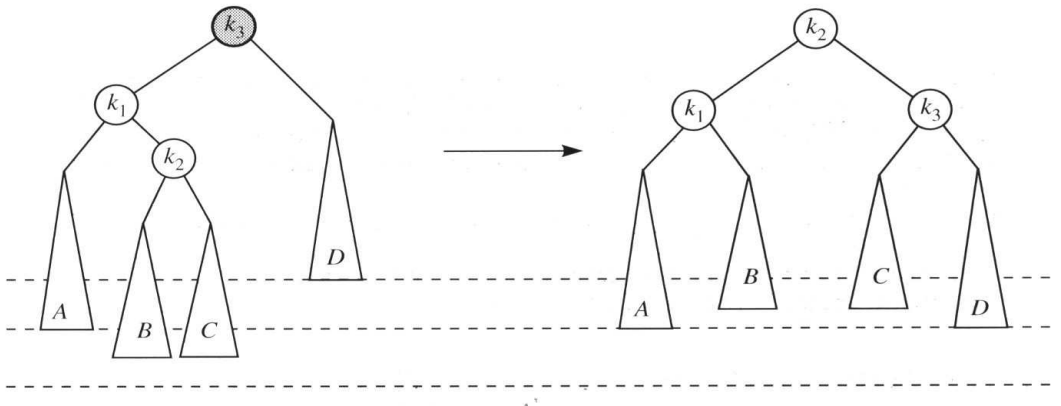

Figura 18.29 Rotación doble izquierda-derecha para arreglar el caso 2.

Como ejemplo, la Figura 18.30 muestra el resultado de insertar 5 en un árbol AVL. El desequilibrio de las alturas se produce en el nodo 8, por lo que estamos en el caso 2. Realizamos una rotación doble en ese nodo, generando el árbol de la derecha.

La Figura 18.31 muestra el caso simétrico 3, que también puede ser arreglado con una rotación doble. Por último, nótese que aunque la rotación doble parezca compleja, es equivalente a hacer lo siguiente:

Una rotación doble es equivalente a dos rotaciones simples.

- Una rotación simple entre el hijo de X y su nieto, seguida de
- una rotación simple entre X y su nuevo hijo.

Así es posible obtener un código muy compacto para implementar la rotación doble del caso 2, que se muestra en la Figura 18.32. El código para el caso simétrico 3 se encuentra en la Figura 18.33.

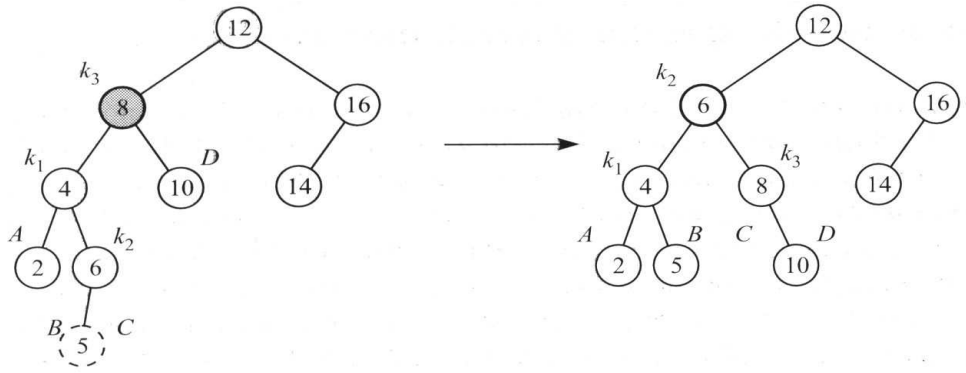

Figura 18.30 La rotación doble arregla el árbol AVL tras la inserción de 5.

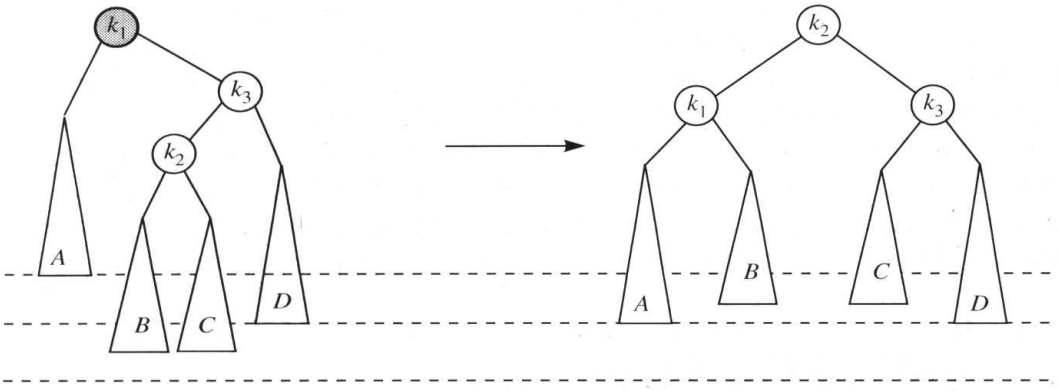

Figura 18.31 Rotación doble izquierda-derecha para arreglar el caso 3.

```
1  /**
2   * Rotación doble de un nodo de un árbol binario; primero
      el hijo izquierdo con su hijo derecho;
3   * después el nodo k3 con su nuevo hijo izquierdo.
4   * En los árboles AVL, es la rotación doble para resolver el caso 2.
5   */
6  static NodoBinario dobleConHijoIzquierdo( NodoBinario k3 )
7  {
8      k3.izquierdo = conHijoDerecho( k3.izquierdo );
9      return conHijoIzquierdo( k3 );
10 }
```

Figura 18.32 Código para la rotación doble (caso 2).

```
1  /**
2   * Rotación doble de un nodo de un árbol binario; primero
3   * el hijo derecho con su hijo izquierdo; después el nodo k1
4   * con su nuevo hijo derecho.
5   * En los árboles AVL, es la rotación doble para resolver el caso 3.
6   */
7  static NodoBinario dobleConHijoDerecho( NodoBinario k1 )
8  {
9      k1.derecho = conHijoIzquierdo( k1.derecho );
10     return conHijoDerecho( k1 );
11 }
```

Figura 18.33 Código para la rotación doble (caso 3).

18.4.4 Resumen de la inserción en un árbol AVL

Una implementación directa del criterio que gobierna los árboles AVL resulta bastante sencilla, aunque no es demasiado eficiente. Han sido descubiertos otros tipos de árboles de búsqueda equilibrados mejores, por lo que en la práctica no vale la pena implementar los árboles AVL.

Concluiremos con un breve resumen de cómo se implementa la inserción en un árbol AVL. La forma más sencilla de hacerlo es emplear un algoritmo recursivo. Para insertar un nuevo nodo con clave X en un árbol AVL T, se le inserta recursivamente en el subárbol adecuado de T (llamado T_{LR}). Si la altura de T_{LR} no cambia, ya hemos acabado. En caso contrario, si se produce en T un desequilibrio de las alturas, realizamos la correspondiente rotación simple o doble, dependiendo de X y de las claves de T y T_{LR}, tras lo que habremos terminado (ya que la altura original es la misma que la que se obtiene tras la rotación). Esta descripción recursiva se ha explicado cuidadosamente, por resultar más fácil de entender y contener todos los elementos esenciales de cualquier implementación alternativa. Sin embargo, resulta mejorable. Por ejemplo, observamos que en cada nodo comparamos las alturas de sus dos subárboles. En general, es más eficiente almacenar directamente en el nodo el resultado de la comparación en lugar de guardar su altura. Esto evita cálculos repetitivos acerca del equilibrio. Por otra parte, en esta ocasión la recursión implica una sobrecarga substancial respecto a la versión iterativa. Esto es debido a que, hemos de recorrer el árbol hacia abajo para después volver hasta la raíz, en lugar de detenernos tan pronto como la rotación se realice. Como consecuencia, en la práctica, se emplean otros esquemas de equilibrio de árboles.

18.5 Árboles rojinegros

Los *árboles rojinegros* son una buena alternativa a los árboles AVL. Los detalles de la implementación producen un código más eficiente ya que las rutinas de inserción y eliminación emplean un solo recorrido descendente. No se permite que haya nodos consecutivos de color rojo, y todos los caminos deben tener el mismo número de nodos negros.

Una alternativa históricamente popular a los árboles AVL son los *árboles rojinegros*. Al igual que en los árboles AVL, las operaciones sobre los árboles rojinegros son logarítmicas en el caso peor. La principal ventaja de los árboles rojinegros es que las rutinas de inserción y de eliminación pueden efectuarse con un único recorrido descendente. Esto contrasta con los árboles AVL, en los que es necesario un recorrido descendente para establecer el lugar de la inserción y otro recorrido ascendente para actualizar las alturas de los nodos y, posiblemente, ajustar su equilibrio. Como resultado, una cuidadosa implementación no recursiva de los árboles rojinegros es más simple y eficiente que una implementación de los árboles AVL.

Un árbol rojinegro es un árbol binario de búsqueda que verifica las siguientes propiedades de orden:

1. Cada nodo está coloreado con los colores rojo o negro.
2. La raíz es negra.
3. Si un nodo es rojo, sus hijos deben ser negros.
4. Todos los caminos desde un nodo a una referencia `null` deben contener el mismo número de nodos negros.

En las representaciones gráficas los nodos rojos se presentan sombreados.

En nuestras representaciones, en blanco y negro, de los árboles rojinegros, los nodos rojos se mostrarán sombreados. En la Figura 18.34 se presenta un árbol rojinegro. Todo camino desde la raíz a una referencia `null` contiene tres nodos negros.

Se puede demostrar por inducción que si cualquier camino desde la raíz a una referencia `null` contiene B nodos negros, entonces el árbol debe tener, al menos, $2^B - 1$ nodos negros. Más aún, como la raíz es negra y no pueden existir dos nodos consecutivos de color rojo en un camino, la altura de un árbol rojinegro es a lo sumo $2\log(N + 1)$. Como consecuencia, está garantizado que la búsqueda es una operación logarítmica.

Tenemos garantizado que la profundidad de los árboles rojinegros es logarítmica. Normalmente, la profundidad es la misma que para los árboles AVL.

La dificultad, como siempre, es que algunas operaciones pueden modificar el árbol y destruir así sus propiedades de coloreado. Esto dificulta la inserción y convierte a la eliminación en una operación especialmente complicada. En la próxima sección, se implementa la inserción y se estudia el algoritmo de eliminación.

18.5.1 Inserción ascendente

Recordemos que un nuevo elemento se inserta siempre en el árbol como una hoja. Si coloreamos el nuevo elemento de negro, estamos seguros de incumplir la cuarta propiedad, ya que así generamos un camino con más nodos negros. Como consecuencia, el nuevo nodo debe ser, en principio, rojo. De esta forma, si el padre es negro ya hemos acabado; así, la inserción de 25 en el árbol de la Figura 18.34 es trivial. Pero si el padre es rojo, estaríamos incumpliendo la tercera propiedad, al haber dos nodos rojos consecutivos. En este caso debemos ajustar el árbol para hacer que se cumpla la propiedad 3 sin que deje de cumplirse la cuarta propiedad. Las operaciones básicas que emplearemos son los cambios de color y las rotaciones.

Los nuevos elementos deben ser de color rojo. Si el padre también es rojo debemos colorear de nuevo y/o realizar rotaciones para eliminar los nodos rojos consecutivos.

Existen varios casos (cada uno con su correspondiente simétrico) que debemos considerar cuando el padre es rojo. En primer lugar, supongamos que el hermano del padre es negro (adoptamos el acuerdo de que las referencias `null` son negras).

Si el hermano del padre es negro, la situación se arregla con una rotación simple o doble, al igual que en los árboles AVL.

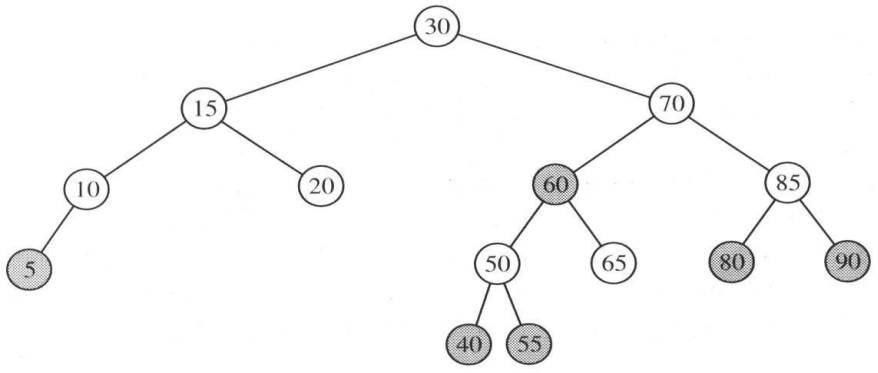

Figura 18.34 Ejemplo de árbol rojinegro; la secuencia de inserciones es 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5 y 55. Los nodos sombreados son rojos.

Esto se aplicaría en la inserción de 3 u 8, pero no en la inserción de 99. Sea X la nueva hoja añadida, P su padre, S el hermano del padre (si existe) y G el abuelo. En este caso sólo X y P son rojos. G es negro ya que en caso contrario *antes* de la inserción existirían dos nodos rojos consecutivos —un incumplimiento de la propiedad 3—. Adoptando la terminología de los árboles AVL, decimos que respecto a G, X es un nodo exterior o interior. Si X es un nieto exterior, la propiedad 3 se recupera mediante una rotación simple entre su padre y su abuelo, con algunos cambios de color. Si X es un nieto interior, es suficiente una rotación doble y algunos cambios de color. La rotación simple se muestra en la Figura 18.35 y la doble en la Figura 18.36. Aunque X es una hoja, hemos ilustrado un caso más general, que permite que X esté en medio del árbol. Esta rotación más general será empleada más tarde, en este mismo capítulo.

Antes de continuar, observemos por qué estas rotaciones son correctas. Necesitamos asegurar que no existen dos nodos rojos consecutivos. Por ejemplo, en la Figura 18.36 es fácil ver que las únicas posibles instancias de nodos rojos consecutivos se encontrarían entre P y uno de sus hijos, o entre G y C. Sin embargo, las raíces de A, B y C deben ser negras; de lo contrario en el árbol de partida se incumpliría la propiedad 3. En el árbol inicial, existe un único nodo negro en el camino desde la raíz del árbol hasta A, B y C, y dos nodos negros en el camino desde la raíz hasta D y E. Es fácil comprobar que éste es todavía el caso después de la rotación y la recoloración.

Si el hermano del padre fuese rojo, después de haber arreglado las cosas habríamos introducido nodos rojos consecutivos en un nivel superior. Necesitaremos recorrer el árbol hacia arriba para terminar de resolver el problema.

Pero, ¿qué sucede si S es rojo y se intenta insertar el elemento 79 en el árbol de la Figura 18.34? Entonces no funcionan ni la rotación simple ni la rotación doble, ya que ambas incluyen en sus resultados nodos rojos consecutivos. De hecho, es

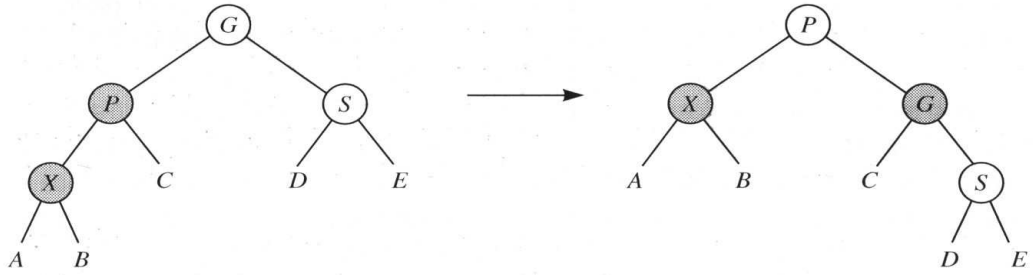

Figura 18.35 Si S es negro y X es un nieto externo, la propiedad 3 se recupera con una rotación simple entre su padre y su abuelo, con los cambios adecuados de color.

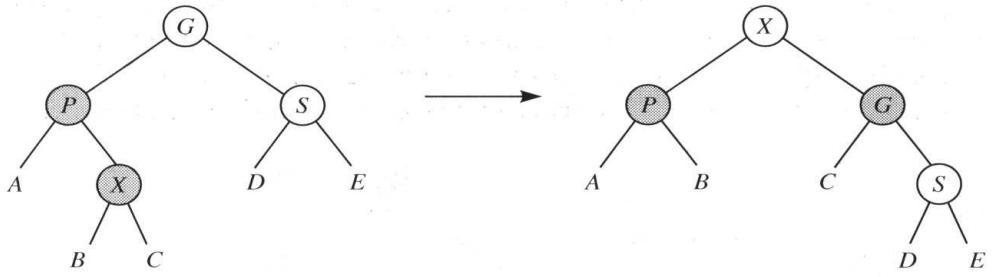

Figura 18.36 Si S es negro y X es un nieto interior, una rotación doble en la que intervengan X, su padre y su abuelo, junto a los cambios de color apropiados, solucionará el incumplimiento de la tercera propiedad.

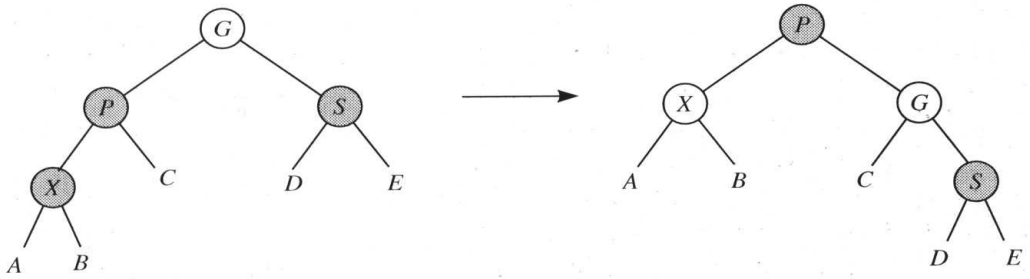

Figura 18.37 Si *S* es rojo, una rotación simple entre el padre y el abuelo, junto a los cambios de color apropiados, restablece la tercera propiedad entre *X* y *P*.

fácil comprobar que en este caso debe haber tres nodos en los caminos hasta *D* y *E*, y sólo uno de ellos puede ser negro. Esto nos indica que tanto *S* como la nueva raíz del subárbol deben ser rojas. Por ejemplo en la Figura 18.37, se muestra el caso de la rotación simple que se produce cuando *X* es un nieto exterior. Aunque, en este caso parece funcionar, aparece un problema: ¿qué sucede si el padre de la raíz del subárbol (es decir, el bisabuelo inicial de *X*) también es rojo? Podemos filtrar este mecanismo hacia atrás, hasta que lleguemos a la raíz (que se coloreará en negro) o ya no se tengan nodos rojos consecutivos. Pero entonces deberíamos realizar un recorrido ascendente hacia la raíz, igual que en los árboles AVL.

18.5.2 Árboles rojinegros descendentes

Para evitar hacer un doble recorrido del árbol, mientras descendemos por el árbol nos aseguramos que el hermano del padre correspondiente no sea rojo. Esto puede conseguirse con cambios de color y/o rotaciones.

Para evitar la eventual necesidad de tener que hacer un doble recorrido del árbol, aplicamos un procedimiento descendente mientras se busca el punto de inserción. Más concretamente, se garantiza que cuando llegamos a una hoja e insertamos un nodo, *S* no será rojo. Entonces basta con añadir una hoja roja y, si es necesario, emplear una sola rotación (simple o doble). El proceso es conceptualmente sencillo.

En el recorrido hacia abajo, cuando se encuentra un nodo *X* con dos hijos rojos, convertimos *X* en rojo y sus dos hijos en negro. La Figura 18.38 muestra este cambio de color. Es sencillo comprobar que el número de nodos negros en los caminos por debajo de *X* permanece invariante. Sin embargo, si el padre de *X* es rojo, introducimos dos nodos rojos consecutivos. Pero en este caso, podemos realizar la rotación simple de la Figura 18.35 o la rotación doble de la Figura 18.36. Ahora bien, ¿qué sucedería si el hermano del padre de *X* fuese también rojo? En realidad, *esto no puede suceder*. Si en el camino hacia las hojas encontramos un nodo *Y* con dos hijos rojos, sabemos que los nietos de *Y* deben ser negros. Y como los hijos de *Y* se colorean en negro, incluso después de la rotación que puede pro-

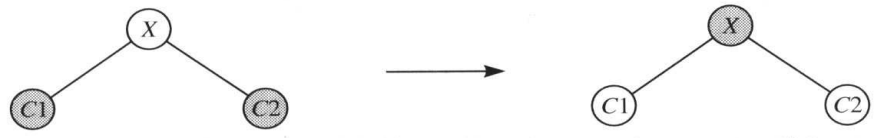

Figura 18.38 Cambio de color; sólo si el padre de *X* es rojo se continúa el proceso con una rotación.

ducirse, no encontraremos otro nodo rojo en dos niveles. Así, cuando llegamos a X, si el padre de X es rojo entonces no es posible que el hermano del padre de X sea rojo también.

Como ejemplo, supongamos que deseamos insertar 45 en el árbol de la Figura 18.34. En el camino hacia las hojas del árbol, encontramos el nodo 50 que tiene dos hijos rojos. Entonces hacemos un cambio de color, coloreando 50 en rojo, y 40 y 55 en negro. El resultado se muestra en la Figura 18.39. Sin embargo, ahora 50 y 60 son rojos. En consecuencia, hacemos una rotación simple (ya que 50 es un nodo exterior) entre 60 y 70, convirtiendo 60 en la raíz negra del subárbol derecho de 30 y coloreando 70 en rojo. Todo esto se muestra en la Figura 18.40. Entonces continuamos el proceso, realizando los mismos cambios si encontramos en el camino otros nodos con dos hijos rojos. En nuestro ejemplo no hay ninguno más.

Cuando alcanzamos una hoja, insertamos 45 como un nodo rojo, y como el padre es negro, ya hemos terminado. En la Figura 18.41 se muestra el árbol que se obtiene. Si el padre hubiese sido rojo, habríamos necesitado hacer una rotación.

Tal y como muestra la Figura 18.41, el árbol rojinegro que se obtiene está, por lo general, muy bien equilibrado. Las distintas pruebas sugieren que el número de nodos consultados en una búsqueda de un árbol rojinegro medio es casi idéntico al número de nodos consultados en los árboles AVL, aunque estáticamente las propiedades de equilibrio de los árboles rojinegros son ligeramente más débiles. La

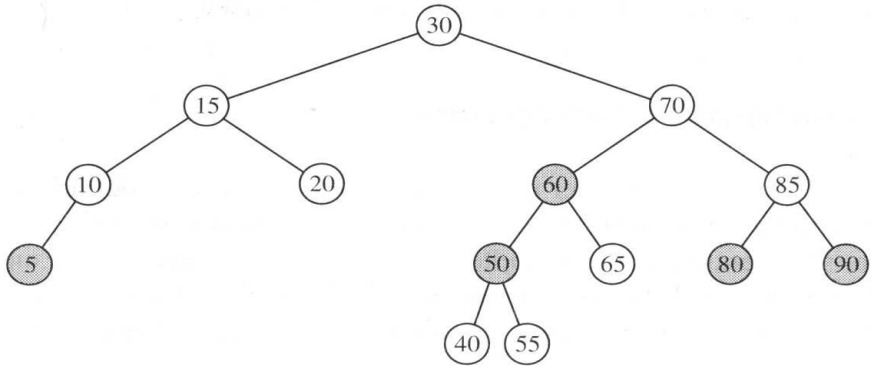

Figura 18.39 El cambio de color de 50 supone una incorrección; debido a que ésta se produce en un nodo exterior, para arreglarla basta una rotación simple.

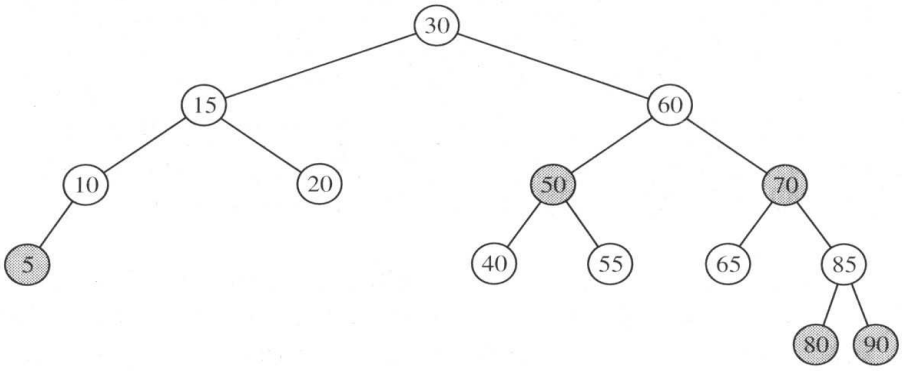

Figura 18.40 Resultado de la rotación simple que arregla el fallo producido en el nodo 50.

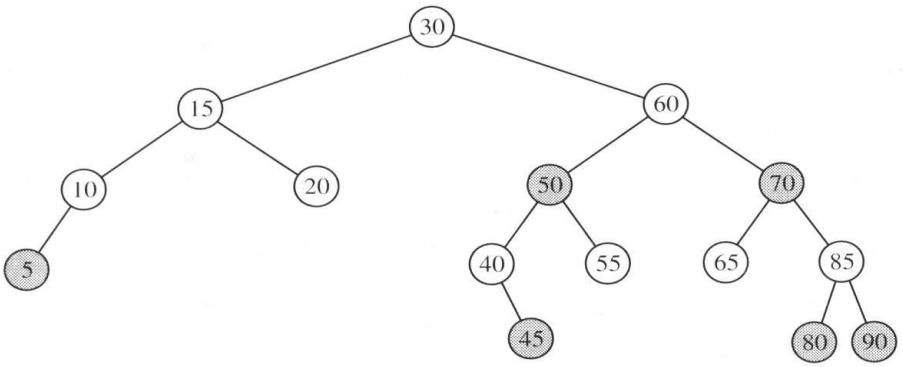

Figura 18.41 Inserción de 45 como nodo rojo.

ventaja de los árboles rojinegros es la sobrecarga relativamente pequeña que se necesita para realizar una inserción y el hecho de que, en la práctica, las rotaciones son bastante infrecuentes.

18.5.3 Implementación en Java

Eliminamos los casos especiales empleando un centinela para la referencia nula y una pseudo-raíz. Esto supone mínimas modificaciones en la mayoría de todas las rutinas.

La implementación concreta es compleja no sólo por la gran cantidad de rotaciones posibles. También es complicada por la posibilidad de que algunos subárboles (como el subárbol derecho de 10) pueden estar vacíos y por el caso especial de la raíz (la cual, entre otras cosas, no tiene padre). Para eliminar los casos especiales, empleamos dos testigos o centinelas:

- Se emplea `nodoNulo` en lugar de la referencia `null`, `nodoNulo` es siempre de color negro.
- Se emplea `cabecera` como pseudo-raíz. Tendrá un valor clave $-\infty$ y un hijo derecho que apuntará a la verdadera raíz.

En el camino hacia las hojas, mantenemos referencias al nodo actual, a su padre, a su abuelo y a su bisabuelo.

A causa de estos elementos, incluso las operaciones básicas necesitan ser modificadas. Como consecuencia, no tiene sentido emplear la herencia de la clase `ArbolBinarioBusqueda`. De modo que, esta clase está escrita partiendo de cero. Su sección pública se muestra en la Figura 18.42 y la sección privada en la Figura 18.43. Recordamos que, por defecto, el constructor de `NodoBinario` inicializa el atributo `color` a 1. Por tanto, `NEGRO` debe ser 1. Las líneas 59 y 60 declaran los centinelas que han sido explicados anteriormente. En la rutina `insertar` se emplean cuatro referencias: `actual`, `padre`, `abuelo` y `bisabuelo`. Su declaración como atributos estáticos (en las líneas 66 a 69) indica que son esencialmente variables globales. Esto es debido a que, como mostraremos más tarde, es conveniente tenerlos compartidos por las rutinas `insertar` y `reorientacion`. El método `eliminar` está sin implementar.

El resto de rutinas son similares a los métodos correspondientes de `ArbolBinarioBusqueda`, excepto en que se tratan adecuadamente los nodos centinelas. En concreto, el constructor debe codificarse con el valor $-\infty$, y `vaciar` y `esVacio`, mostrados en las líneas 45 a 48, necesitan emplear `nodoNulo` (en lugar de `null`) y `cabecera.derecho` (en lugar de la típica `raiz`).

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase ArbolRojoNegro
7  //
8  // CONSTRUCCIÓN: con un centinela con valor menos infinito
9  //
10 // ************************OPERACIONES PÚBLICAS**********************
11 // void insertar( x )       --> Inserta x
12 // void eliminar( x )       --> Elimina x
13 // void eliminarMin( )      --> Elimina el elemento más pequeño
14 // Comparable buscar( x )   --> Devuelve el elemento que ajusta con x
15 // Comparable buscarMin( )  --> Devuelve el elemento más pequeño
16 // Comparable buscarMax( )  --> Devuelve el elemento más grande
17 // boolean esVacio( )       --> Devuelve true si vacío; si no, false
18 // void vaciar( )           --> Elimina todos los elementos
19 // void imprimirArbol( )    --> Imprime el árbol de forma ordenada
20 // ****************************ERRORES*****************************
21 // Muchas rutinas lanzan ElementoNoEncontrado en condiciones degeneradas
22 // insertar lanza ElementoDuplicado si el elemento ya está en el árbol
23
24 /**
25  * Implementa los árboles rojinegros.
26  * Nótese que los "encajes" se basan en el método compara.
27  */
28 public class ArbolRojoNegro implements ArbolBusqueda
29 {
30     public ArbolRojoNegro( Comparable negInf )
31       { /* Figura 18.44 */ }
32
33     public void insertar( Comparable x ) throws ElementoDuplicado
34       { /* Figura 18.48 */ }
35     public void eliminar( Comparable x ) throws ElementoNoEncontrado
36       { /* No se implementa */ }
37     public void eliminarMin( ) throws ElementoNoEncontrado
38       { /* No se implementa */ }
39     public Comparable buscarMin( ) throws ElementoNoEncontrado
40       { /* Figura 18.46 */ }
41     public Comparable buscarMax( ) throws ElementoNoEncontrado
42       { /* Similar a buscarMin */ }
43     public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
44       { /* Figura 18.47 */ }
45     public boolean esVacio( )
46       { return cabecera.derecho == nodoNulo; }
47     public void vaciar( )
48       { cabecera.derecho = nodoNulo; }
49     public void imprimirArbol( )
50       { imprimirArbol( cabecera.derecho ); }
```

Figura 18.42 Esqueleto de la clase `ArbolRojoNegro` (parte 1: sección pública).

El constructor de `ArbolRojoNegro` se muestra en la Figura 18.44. Únicamente crea los dos centinelas e inicializa las referencias de sus hijos `izquierdo` y `derecho` apuntando a `nodoNulo`. Es preciso un inicializador estático para crear e inicializar `nodoNulo`.

```
51      private void imprimirArbol( NodoBinario t )
52        { /* Figura 18.45 */ }
53      private void reorientacion( Comparable item )
54        { /* Figura 18.49 */ }
55      private NodoBinario rotar( Comparable item,
56                                 NodoBinario padre )
57        { /* Figura 18.50 */ }
58
59      private NodoBinario cabecera;
60      private static NodoBinario nodoNulo;
61
62      private static final int NEGRO = 1;     // NEGRO debe ser 1
63      private static final int ROJO  = 0;
64
65          // Empleados en la rutina de inserción y las auxiliares
66      private static NodoBinario actual;
67      private static NodoBinario padre;
68      private static NodoBinario abuelo;
69      private static NodoBinario bisabuelo;
70  }
```

Figura 18.43 Esqueleto de la clase `ArbolRojoNegro` (parte 2: sección privada).

```
1   /**
2    * Construye el árbol.
3    * @param negInf un valor menor o igual que todos los demás.
4    */
5   public ArbolRojoNegro( Comparable negInf )
6   {
7       cabecera           = new NodoBinario( negInf );
8       cabecera.izquierdo = cabecera.derecho = nodoNulo;
9   }
10
11  static                 // Inicializador estático de nodoNulo
12  {
13      nodoNulo           = new NodoBinario( null );
14      nodoNulo.izquierdo = nodoNulo.derecho = nodoNulo;
15  }
```

Figura 18.44 Constructor de `ArbolRojoNegro` e inicializador estático.

La Figura 18.45 muestra el sencillo cambio que supone el uso de los centinelas. El test sobre `null` es reemplazado por el test sobre `nodoNulo`. El método `buscarMin` se muestra en la Figura 18.46.

El test sobre `null` se sustituye por el test sobre `nodoNulo`. Al objeto de evitar comprobaciones adicionales, cuando se realiza una búsqueda, colocamos x en el centinela `nodoNulo`.

```
1   /**
2    * Método interno para imprimir un subárbol en orden.
3    * @param t la raíz del subárbol.
4    */
5   private void imprimirArbol( NodoBinario t )
6   {
7       if( t != nodoNulo )
8       {
9           imprimirArbol( t.izquierdo );
10          System.out.println( t.dato.toString( ) );
11          imprimirArbol( t.derecho );
12      }
13  }
```

Figura 18.45 `imprimirArbol` de la clase `ArbolRojoNegro`.

```
1  /**
2   * Busca el elemento más pequeño del árbol.
3   * @return el menor elemento.
4   * @exception ElementoNoEncontrado si el árbol es vacío.
5   */
6  public Comparable buscarMin( ) throws ElementoNoEncontrado
7  {
8     if( esVacio( ) )
9       throw new ElementoNoEncontrado( "buscarMin de ArbolRojoNegro" );
10
11    NodoBinario itr = cabecera.derecho;
12    while( itr.izquierdo != nodoNulo )
13       itr = itr.izquierdo;
14
15    return itr.dato;
16 }
```

Figura 18.46 Método `buscarMin` de `ArbolRojoNegro`; observe el uso de `cabecera` y `nodoNulo`.

La rutina `buscar`, mostrada en la Figura 18.47, emplea otro truco muy común. Antes de iniciar la búsqueda, colocamos `x` en el centinela `nodoNulo`. Hemos garantizado así el ajuste con `x`, incluso si `x` no es encontrado. Si el ajuste se produce en `nodoNulo`, concluimos que el elemento no ha sido encontrado. Emplearemos este truco en el método `insertar`.

```
1  /**
2  * Buscar un elemento en el árbol.
3  * @param x el elemento a buscar.
4  * @return el elemento buscado.
5  * @exception ElementoNoEncontrado si en el árbol no
6  *        no hay ningún elemento que ajuste con x.
7  */
8  public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
9  {
10    nodoNulo.dato = x;
11    actual = cabecera.derecho;
12
13    for( ; ; )
14    {
15       if( x.menorQue( actual.dato ) )
16         actual = actual.izquierdo;
17       else if( actual.dato.menorQue( x ) )
18         actual = actual.derecho;
19       else if( actual != nodoNulo )
20         return actual.dato;
21       else
22         throw new ElementoNoEncontrado( "buscar de ArbolRojoNegro" );
23     }
24 }
```

Figura 18.47 Método `buscar` de `ArbolRojoNegro`; obsérvese el uso de `cabecera` y `nodoNulo`.

El método `insertar`, se muestra en la Figura 18.48, se obtiene directamente de nuestra descripción. El bucle `while`, codificado en las líneas 12 a 21, desciende por el árbol y «repara» los nodos con dos hijos rojos llamando al método `reorientacion`, tal y como se muestra en la Figura 18.49. Para hacerlo, no sólo mantenemos información acerca del nodo actual, sino también de su padre, abuelo y bisabuelo. Nótese que después de la rotación, los valores almacenados en el abuelo y el bisabuelo no son correctos. Sin embargo, nos hemos asegurado que serán correctamente colocados la próxima vez que se necesiten.

El código es relativamente reducido, si se consideran la gran cantidad de casos y el hecho de que la implementación no es recursiva. Ésta es la razón por la cual los árboles rojinegros funcionan satisfactoriamente en la práctica.

Cuando el bucle termina, `x` es encontrado (indicado por `actual != nodoNulo`) o no. Si lo encontramos, se lanza una excepción en la línea 25. En caso contrario, `x` no está en el árbol y es preciso hacerle hijo de `padre`. Creamos un nuevo nodo (como el nuevo nodo `actual`), lo unimos al padre e invocamos a `reorientacion`, todo ello en las líneas 26 a 33.

El código empleado para realizar una rotación simple se muestra en el método `rotar` de la Figura 18.50. Debido a que el árbol obtenido debe unirse al padre, `rotar` tiene como parámetro el nodo padre. En lugar de guardar la información de qué tipo de rotación realizamos (esto es, si es hacia la derecha o hacia la izquier-

`rotar` considera cuatro posibilidades. El uso del operador `?:` reduce el código, pero es lógicamente equivalente al test `if else`.

```
1  /**
2   * Inserta en el árbol.
3   * @param x el elemento a insertar.
4   * @exception ElementoDuplicado si un elemento que
5   *     ajusta con x ya está en el árbol.
6   */
7  public void insertar( Comparable x ) throws ElementoDuplicado
8  {
9      actual = padre = abuelo = cabecera;
10     nodoNulo.dato = x;
11
12     while( actual.dato.compara( x ) != 0 )
13     {
14         bisabuelo = abuelo; abuelo = padre; padre = actual;
15         actual = x.menorQue( actual.dato ) ?
16                     actual.izquierdo : actual.derecho;
17
18             // Comprueba si hay dos hijos ROJOs; resolviéndolo en su caso
19         if( actual.izquierdo.color == ROJO && actual.derecho.color == ROJO )
20             reorientacion( x );
21     }
22
23         // La inserción falla si el elemento ya está en el árbol
24     if( actual != nodoNulo )
25         throw new ElementoDuplicado( "insertar de ArbolRojoNegro" );
26     actual = new NodoBinario( x, nodoNulo, nodoNulo );
27
28         // Enlace al padre
29     if( x.menorQue( padre.dato ) )
30         padre.izquierdo = actual;
31     else
32         padre.derecho = actual;
33     reorientacion( x );
34 }
```

Figura 18.48 Rutina `insertar` de la clase `ArbolRojoNegro`.

```
1  /**
2   * Rutina invocada durante una inserción si un nodo tiene dos
3   * hijos ROJOs. Realiza cambios de color y rotaciones.
4   * @param item el elemento a insertar.
5   */
6  private void reorientacion( Comparable item )
7  {
8          // Cambio de color
9      actual.color = ROJO;
10     actual.izquierdo.color = NEGRO;
11     actual.derecho.color = NEGRO;
12
13     if( padre.color == ROJO ) // Tiene que rotar
14     {
15         abuelo.color = ROJO;
16         if( item.menorQue( abuelo.dato ) !=
17                         item.menorQue( padre.dato ) )
18             padre = rotar( item, abuelo ); // Comenzar rotación doble
19         actual = rotar( item, bisabuelo );
20         actual.color = NEGRO;
21     }
22     cabecera.derecho.color = NEGRO; // Colorea la raíz en NEGRO
23 }
```

Figura 18.49 Rutina `reorientacion`: invocada cuando un nodo tiene dos hijos rojos o cuando se inserta un nuevo nodo.

```
1  /**
2   * Rutina interna que realiza una rotación simple o doble.
3   * Invocada por reorientacion.
4   * @param item el elemento de reorientacion.
5   * @param padre el padre de la raíz del subárbol que debe rotarse.
6   * @return la raíz del subárbol rotado.
7   */
8  private NodoBinario
9  rotar( Comparable item, NodoBinario padre )
10 {
11     if( item.menorQue( padre.dato ) )
12         return padre.izquierdo = item.menorQue(
13                                     padre.izquierdo.dato ) ?
14             Rotaciones.conHijoIzquierdo( padre.izquierdo ) : // II
15             Rotaciones.conHijoDerecho( padre.izquierdo )  ; // ID
16     else
17         return padre.derecho = item.menorQue(
18                                    padre.derecho.dato ) ?
19             Rotaciones.conHijoIzquierdo( padre.derecho )  : // DI
20             Rotaciones.conHijoDerecho( padre.derecho )    ; // DD
21 }
```

Figura 18.50 Rutina para realizar la rotación apropiada.

da) mientras descendemos por el árbol, pasamos `x` como parámetro. Como se espera un número reducido de rotaciones a lo largo de la inserción, hacerlo de este modo es, además de simple, rápido.

`reorientacion` llama a `rotar` cuando es necesario, para efectuar una rotación simple o doble. Como una rotación doble no es más que la combinación de

dos rotaciones simples, podemos preguntar si estamos en un caso interior y si es así, realizar una rotación extra entre el nodo actual y su padre (pasando su abuelo a la rutina `rotar`). En cualquier caso, hacemos una rotación entre el padre y el abuelo (pasando el bisabuelo a `rotar`). Esto es codificado, de forma sutil, en las líneas 16 a 19 de la Figura 18.49.

18.5.4 Eliminación descendente

La eliminación en los árboles rojinegros también puede implementarse de modo descendente. No es necesario decir que en este caso una implementación concreta es bastante complicada, ya que el mismo algoritmo `eliminar` de los árboles no equilibrados no es trivial. El algoritmo usual de eliminación en los árboles binarios de búsqueda elimina nodos que son hojas o tienen un único hijo. Recuerde que nunca se borran nodos con dos hijos; sólo se reemplaza su contenido.

La eliminación es bastante compleja. La idea básica consiste en asegurar que el nodo eliminado es siempre rojo.

Si el nodo que vamos a borrar es rojo no hay ningún problema. Sin embargo, si es negro su eliminación hará que se incumpla la cuarta propiedad. La solución al problema consiste en asegurar que cualquier nodo que eliminemos será rojo.

A lo largo de esta discusión, X será el nodo actual, T su hermano y P su padre. Comenzamos por colorear la raíz centinela en rojo. Según descendemos por el árbol, intentamos asegurar que X sea rojo. Cuando llegamos a un nuevo nodo, estamos seguros de que P es rojo (de forma inductiva, gracias al invariante que tratamos de mantener) y de que X y T son negros (ya que no podemos tener dos nodos rojos consecutivos).

Existen dos casos principales, además de las variantes simétricas usuales (que son omitidas).

En primer lugar supongamos que X tiene dos hijos rojos. Existen tres subcasos, los cuales dependen de los hijos de T:

1. T tiene dos hijos negros: cambio de color (Figura 18.51).
2. T tiene un hijo exterior rojo: realizar una rotación simple (Figura 18.52).
3. T tiene un hijo interior rojo: realizar una rotación doble (Figura 18.53).

Una observación detenida de las rotaciones nos conduce a la conclusión de que si T tiene dos hijos rojos, tanto una rotación simple como una rotación doble funcionarían correctamente, de forma que es más lógico decantarse por la rotación simple. Observe que si X es una hoja entonces sus dos hijos son negros, así que siempre podemos emplear alguno de estos tres mecanismos para conseguir que su color sea rojo.

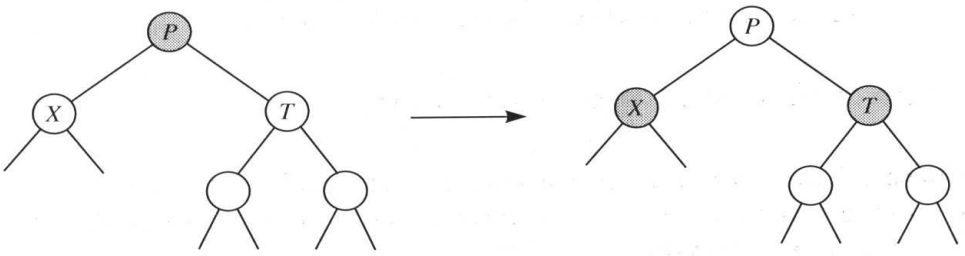

Figura 18.51 X tiene dos hijos negros cuyos hermanos son también negros; debemos hacer un cambio de color.

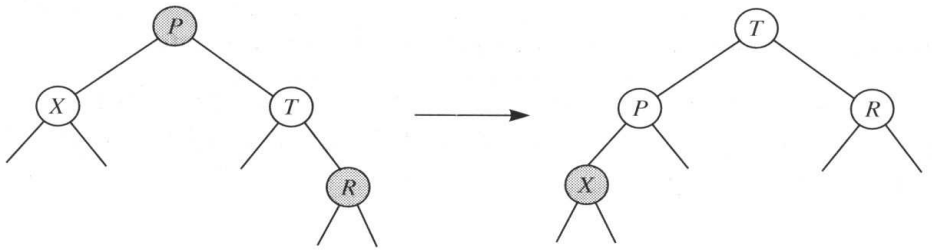

Figura 18.52 *X* tiene dos hijos negros y el hijo exterior de su hermano es rojo; debemos hacer una rotación simple.

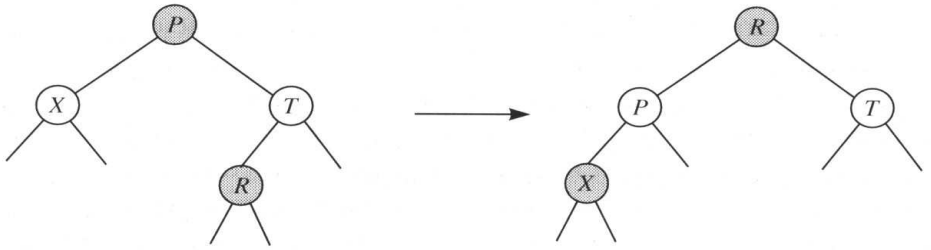

Figura 18.53 *X* tiene dos hijos negros y el hijo interior de su hermano es rojo; debemos hacer una rotación doble.

El segundo caso es que alguno de los hijos de X sea rojo. Nótese que debido a que las rotaciones del primer caso siempre colorean X de rojo, si X tiene un hijo de su mismo color, introduciríamos nodos rojos consecutivos. Por tanto, necesitamos una solución alternativa. En este caso, descendemos al siguiente nivel, obteniendo nuevos X, T y P. Si tenemos suerte, estaremos en un nodo rojo (esto puede suceder el 50 por ciento de las veces), y arreglamos el problema convirtiendo el nodo actual nuevo en rojo. En caso contrario, estamos en la situación mostrada en la Figura 18.54. Esto es, el X actual es negro, T es rojo y P es negro. Podemos realizar una rotación entre T y P, haciendo que el nuevo padre de X sea rojo y su nuevo abuelo sea negro. Ahora X ya no es rojo, pero estamos de nuevo en el punto de partida (aunque en un nivel inferior). Esto es suficientemente bueno, ya que prueba que podemos descender de modo iterativo por el árbol. Así, siempre que alcancemos un nodo rojo o un nodo con dos hijos negros hemos ganado. Esto está garantizado en el algoritmo de eliminación, ya que se alcanza una de las dos situaciones siguientes:

- X es una hoja. Esta situación ya fue considerada por el caso principal ya que X tiene dos hijos negros.

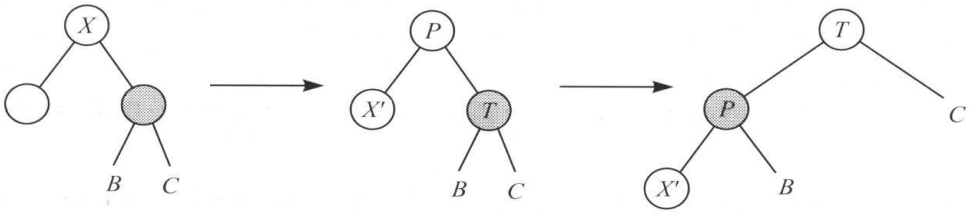

Figura 18.54 *X* es negro y al menos un hijo suyo es rojo; si descendemos al siguiente nivel y caemos en un nodo rojo, todo está bien; en caso contrario, realizamos una rotación entre su padre y un hermano.

- *X* tiene un solo hijo. Si éste es negro, se aplica el caso principal y si es rojo, eliminamos *X*, si es necesario, y coloreamos su hijo en negro.

En algunas ocasiones se utiliza como alternativa la *eliminación perezosa*, en la que los elementos eliminados se marcan simplemente como tales. Sin embargo, consume espacio de forma innecesaria y complica otras rutinas (*véase* el Ejercicio 18.24).

18.6 AA-Árboles

Los *AA-árboles* son el método elegido cuando se necesitan árboles equilibrados, se permite una implementación más inmediata y se requieren eliminaciones.

Debido a la gran cantidad de rotaciones necesarias, los árboles rojinegros necesitan de algunos trucos para su codificación, con el subsiguiente peligro de cometer errores. En particular, la operación `eliminar` es bastante complicada. Esta sección describe un tipo de árbol de búsqueda bastante simple pero con un equilibrado bastante competitivo, conocido como *AA-árbol*. Los AA-árboles son el método elegido cuando se necesitan árboles equilibrados, se permite una implementación más inmediata (con el subsiguiente coste) y se requieren eliminaciones. Los AA-árboles añaden una condición adicional a las impuestas por los árboles rojinegros: los hijos izquierdos no pueden ser rojos.

Esta restricción adicional simplifica en gran medida los algoritmos de los árboles rojinegros, por dos razones. En primer lugar, elimina aproximadamente la mitad de los casos de reestructuración. En segundo lugar, simplifica el algoritmo `eliminar`, al no tener que considerar uno de sus casos más complicados. Más exactamente, si un nodo interno tiene un único hijo, éste debe ser un hijo derecho rojo, ya que los hijos izquierdos rojos están ahora prohibidos y un hijo único negro incumpliría la cuarta propiedad de los árboles rojinegros. De este modo, siempre podemos reemplazar un nodo interno por el nodo más pequeño de su subárbol derecho. Este nodo más pequeño será una hoja o tendrá un hijo rojo, que puede ser fácilmente eliminado.

El *nivel* de un nodo de un AA-árbol es el número de enlaces izquierdos que hay en el camino hasta el centinela `nodoNulo`.

Para simplificar aún más la implementación, representamos la información del equilibrado de un modo más directo. En lugar de guardar un color en cada nodo, almacenamos el *nivel* de dicho nodo. El nivel de un nodo se define como sigue:

- El nivel es 1 si el nodo es una hoja.
- El nivel es el mismo que el de su padre si el nodo es rojo.
- El nivel es una unidad menor que el nivel de su padre si el nodo es negro.

El nivel representa el número de enlaces izquierdos que se encuentran en el camino hacia el centinela `nodoNulo`.

En un AA-árbol, un *enlace horizontal* es un enlace entre un nodo y un hijo suyo del mismo nivel. Un enlace horizontal debe dirigirse siempre a la derecha y no pueden existir enlaces horizontales consecutivos.

El resultado es un *AA-árbol*. Si traducimos las condiciones estructurales de colores a niveles, sabemos que el nivel del hijo izquierdo debe ser una unidad menor que su padre y el del hijo derecho debe ser cero o una unidad menor que el de su padre (pero no más). Llamaremos *enlaces horizontales* a las conexiones entre un nodo y un hijo suyo del mismo nivel. Entonces, las propiedades de color implican lo siguiente:

1. Los enlaces horizontales son referencias de lados derechos (ya que sólo los hijos derechos son rojos).
2. No existen dos enlaces horizontales consecutivos (ya que no pueden existir dos nodos rojos consecutivos).

3. Los nodos del nivel 2, o superiores, deben tener dos hijos.
4. Si un nodo no tiene un enlace horizontal derecho entonces sus dos hijos están al mismo nivel.

La Figura 18.55 muestra un ejemplo de AA-árbol. La raíz de dicho árbol es el nodo de clave 30. La búsqueda se realiza aplicando el algoritmo usual. Como ya es habitual, `insertar` y `eliminar` son más complejos, ya que los algoritmos naturales de los árboles de búsqueda pueden incumplir las propiedades de los enlaces horizontales. Sin embargo, y no sorprendentemente, las rotaciones pueden solucionar todos los problemas planteados por estas rutinas.

18.6.1 Inserción

La inserción se realiza empleando el algoritmo recursivo usual más dos llamadas a métodos.

La inserción de un nuevo elemento siempre se hace en el nivel más bajo. Como es habitual, esto puede causar problemas. En la Figura 18.55 la inserción de 2 podría crear un enlace horizontal izquierdo, mientras que la inserción de 45 generaría dos enlaces horizontales derechos consecutivos. Como consecuencia, después de insertar un nodo en el nivel más profundo, pueden necesitarse algunas rotaciones para restablecer las propiedades de los enlaces horizontales.

Los enlaces horizontales izquierdos se eliminan mediante un `giro` (rotación entre un nodo y su hijo izquierdo). Los enlaces horizontales derechos consecutivos se eliminan con un `reparto` (rotación entre un nodo y su hijo derecho). `giro` precede a `reparto`.

En ambos casos, una única rotación simple arregla el problema. Eliminamos los enlaces horizontales izquierdos haciendo una rotación entre el nodo y su hijo izquierdo y eliminamos los enlaces horizontales derechos consecutivos realizando una rotación entre el primero y el segundo de los (tres) nodos conectados mediante los dos enlaces. Estos procedimientos se llaman `giro` y `reparto`, respectivamente.

`giro` se muestra en la Figura 18.56 y `reparto` en la Figura 18.57. Un `giro` elimina un enlace horizontal izquierdo. Sin embargo, puede generar enlaces horizontales derechos consecutivos, ya que el hijo derecho de *X* puede ser también horizontal. En este caso, invocaremos primero a `giro` y después a `reparto`. Después de `reparto` se incrementa el nivel del nodo del centro. Esto puede causar proble-

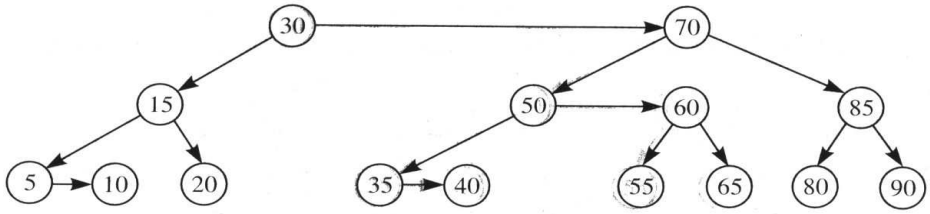

Figura 18.55 AA-árbol obtenido de la inserción de 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55 y 35.

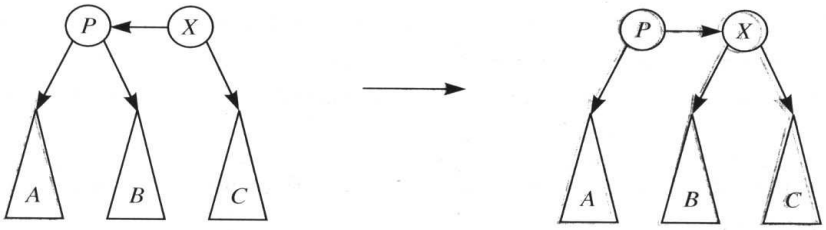

Figura 18.56 `giro` es una rotación simple entre *X* y *P*.

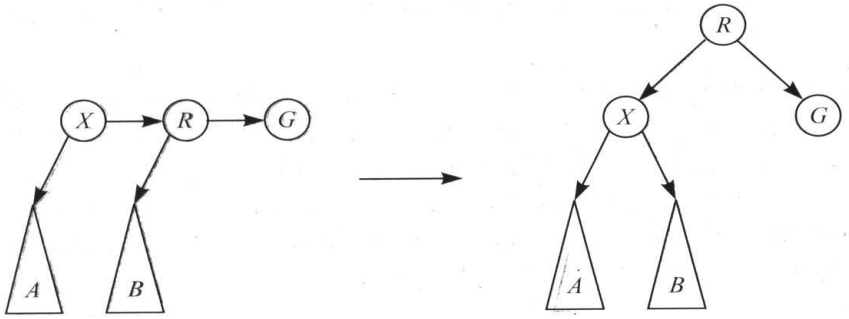

Figura 18.57 `reparto` es una rotación simple entre *X* y *R*; obsérvese que el nivel de *R* aumenta.

mas al padre inicial de *X*, al crear un enlace horizontal izquierdo o enlaces horizontales derechos consecutivos: ambos problemas pueden solucionarse aplicando la estrategia `giro/reparto` en el camino de regreso a la raíz. Esto se hace automáticamente si se usa recursión, así que una implementación recursiva de `insertar` es sólo dos llamadas más larga que la correspondiente rutina de los árboles de búsqueda no equilibrados.

Éste es un algoritmo un tanto extraño, pues es más complicado seguirlo sobre el papel que implementarlo en un computador.

Para ver el algoritmo en acción, mostramos el resultado de insertar el elemento 45 en el AA-árbol de la Figura 18.55. En la Figura 18.58, cuando añadimos 45 en el último nivel, se forman dos enlaces horizontales consecutivos. Los pares `giro/reparto` se aplican en el camino de vuelta a la raíz, siempre que sea necesario. En particular, se necesita un `reparto` en el nodo 35 a causa de los enlaces horizontales derechos consecutivos. El resultado se muestra en la Figura 18.59. Cuando la recursión vuelve al nodo 50, encontramos un enlace horizontal izquierdo. Realizamos entonces un `giro` en 50 para eliminar el enlace horizontal (el resultado puede verse en la Figura 18.60) y un `reparto` en 40 para eliminar los enlaces horizontales derechos consecutivos. En la Figura 18.61 se muestra el resultado

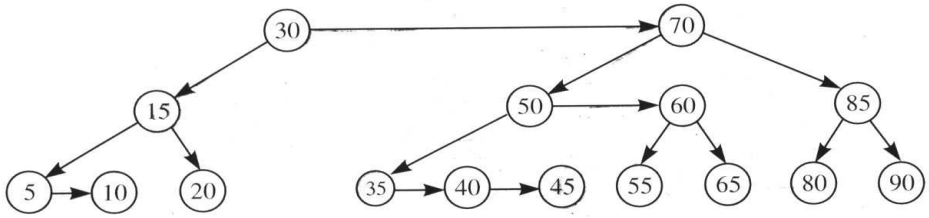

Figura 18.58 Situación después de insertar 45 en nuestro árbol ejemplo; aparecen dos enlaces horizontales derechos consecutivos tras el nodo 35.

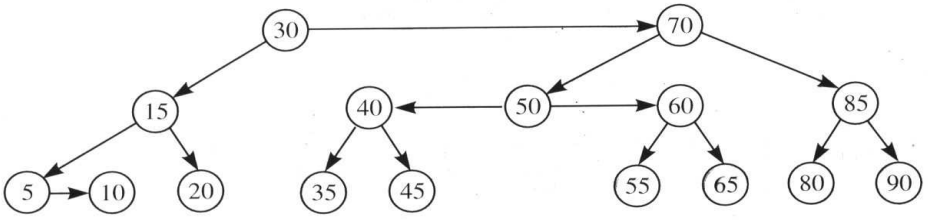

Figura 18.59 Situación después del `reparto` en 35; éste genera un enlace horizontal izquierdo en 50.

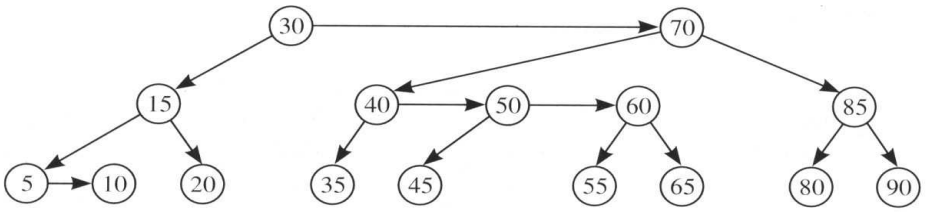

Figura 18.60 Situación después del `giro` en 50; éste genera dos nodos horizontales consecutivos comenzando en 40.

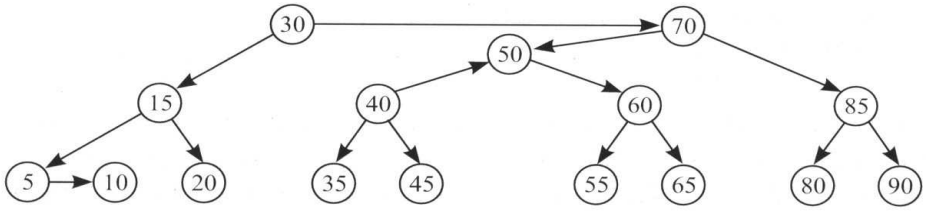

Figura 18.61 Situación después del `reparto` en 40; 50 está ahora al mismo nivel que 70, introduciéndose así un enlace horizontal izquierdo prohibido.

después del `reparto`: ahora el nodo 50 está en el nivel 3 y es el hijo horizontal izquierdo de 70. Necesitamos ejecutar ahora un nuevo par `giro/reparto`. El `giro` en 70 elimina el enlace horizontal izquierdo en el nivel superior, pero genera enlaces horizontales derechos consecutivos, tal y como se ilustra en la Figura 18.62. Cuando se aplica el último intercambio, los nodos horizontales consecutivos desaparecen y 50 se convierte en la nueva raíz del árbol. El resultado se muestra en la Figura 18.63.

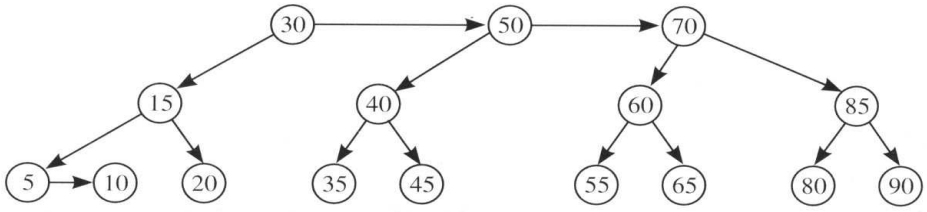

Figura 18.62 Situación después del `giro` en 70; éste crea dos enlaces horizontales consecutivos tras 30.

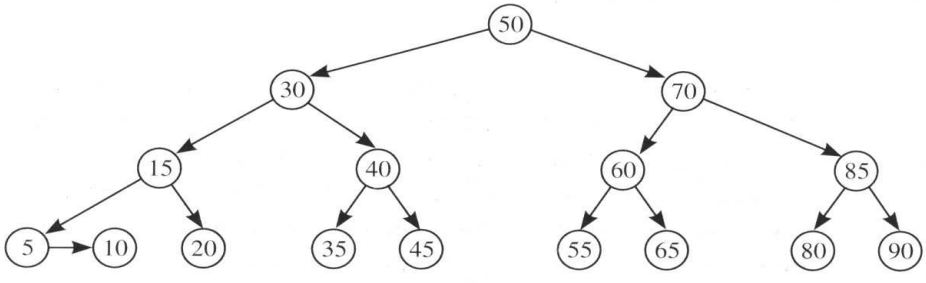

Figura 18.63 Situación después del `reparto` en 30; la inserción se ha completado.

18.6.2 Eliminación

La eliminación se simplifica, ya que el caso de un único hijo sólo puede producirse en el nivel 1 y además deseamos emplear recursión.

En los árboles binarios de búsqueda ordinarios, el algoritmo `eliminar` se divide en tres casos: el elemento a eliminar puede ser una hoja, tener un solo hijo o tener dos hijos. En los AA-árboles tratamos el caso de un único hijo igual que el caso de dos hijos, ya que el primero de ellos sólo puede aparecer en el nivel 1. Más aún, el caso de dos hijos también es sencillo, si se garantiza que el nodo empleado como valor de reemplazamiento está en el nivel 1 y tiene, en el peor de los casos, un solo enlace horizontal derecho. De este modo, todo gira entorno a `eliminar` un nodo del nivel 1. Claramente, esto puede afectar al equilibrio del árbol (considere, por ejemplo, la eliminación de 10 en la Figura 18.63).

Sea T el nodo actual y considere que estamos empleando recursión. Si la eliminación ha reducido el nivel de uno de los hijos de T hasta dos unidades menos que el nivel de T, entonces éste también ha de ser reducido (sólo el hijo introducido en la llamada recursiva puede haber sido afectado, pero por simplicidad, no guardamos información de ello). Además, si T tiene un enlace horizontal derecho, entonces éste también debe descender de nivel. En este punto podemos tener hasta seis nodos en el mismo nivel: T, el hijo derecho horizontal de T, llamado R, los dos hijos de R, y sus hijos derechos horizontales. La Figura 18.64 muestra la situación posible más sencilla.

Después de una eliminación recursiva, tres `giros` y dos `repartos` garantizarán el equilibrio del árbol.

Después de eliminar el elemento 1, los nodos 2 y 5 se convierten en nodos del nivel-1. En primer lugar, arreglamos el enlace horizontal izquierdo que se encuentra ahora entre los nodos 5 y 3. Esto requiere esencialmente dos rotaciones: una entre los nodos 5 y 3 y otra entre los nodos 5 y 4. En este caso, el nodo actual T no interviene. Por otro lado, si la eliminación procede del lado derecho, entonces el nodo izquierdo de T puede convertirse en horizontal; esto requeriría una rotación doble similar (comenzando en T). Para evitar distinguir todos los casos, invocaremos siempre a `giro` tres veces. Una vez hecho esto, son suficientes dos llamadas a `reparto` para recolocar los arcos horizontales.

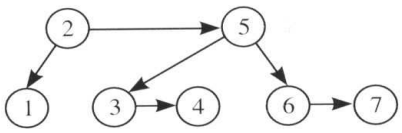

Figura 18.64 Cuando se elimina el elemento 1, todos los nodos están en el nivel 1, apareciendo así enlaces horizontales izquierdos.

18.6.3 Implementación en Java

Las rutinas de los AA-árboles son relativamente sencillas (comparadas con las de los árboles rojinegros).

El esqueleto de la clase de los AA-árboles se muestra en las Figuras 18.65 y 18.66. Gran parte de ella duplica implementaciones de árboles anteriores. Una vez más, empleamos el centinela `nodoNulo`; sin embargo, ahora no necesitamos una pseudo-raíz. El constructor, que no mostramos, genera el `nodoNulo`, igual que en el caso de los árboles rojinegros, y hace que `raiz` le apunte. `nodoNulo` está en el nivel 0. Las rutinas emplean diversos métodos privados auxiliares.

`insertar` se muestra en la Figura 18.67. Como hemos mencionado anteriormente en este capítulo, este método es idéntico al `insertar` recursivo de los ár-

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase AA-Arbol
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ********************OPERACIONES PÚBLICAS*********************
11 // Las mismas que en ArbolBinarioBusqueda; se omiten por brevedad
12 // ***************************ERRORES*****************************
13 // Muchas rutinas lanzan ElementoNoEncontrado en condiciones degeneradas
14 // Insertar lanza ElementoDuplicado si el elemento ya está en el árbol
15
16 /**
17  * Implementa los AA-árboles.
18  * Nótese que todas las comparaciones se basan en el método compara.
19  */
20 public class AA_Arbol implements ArbolBusqueda
21 {
22    public AA_Arbol( )
23      { raiz = nodoNulo; }
24
25    public void insertar( Comparable x ) throws ElementoDuplicado
26      { raiz = insertar( x, raiz ); }
27    public void eliminar( Comparable x ) throws ElementoNoEncontrado
28      { raiz = eliminar( x, raiz ); }
29    public void eliminarMin( ) throws ElementoNoEncontrado
30      { eliminar( buscarMin( ) ); }
31    public Comparable buscarMin( ) throws ElementoNoEncontrado
32      { /* Implementación habitual; ver el código en Internet */ }
33    public Comparable buscarMax( ) throws ElementoNoEncontrado
34      { /* Implementación habitual; ver el código en Internet */ }
35    public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
36      { /* Implementación habitual; ver el código en Internet */ }
37    public boolean esVacio( )
38      { return raiz == nodoNulo; }
39    public void vaciar( )
40      { raiz = nodoNulo; }
41    public void imprimirArbol( )
42        { imprimirArbol( raiz ); }
```

Figura 18.65 Esqueleto de la clase de los AA_árboles (parte 1).

boles binarios de búsqueda, excepto por el hecho de que incluye una llamada a `giro` seguida de una llamada a `reparto`. En las Figuras 18.68 y 18.69, `giro` y `reparto` se implementan de una forma sencilla empleando las rotaciones de árboles de las que disponemos. Por último, `eliminar` se encuentra en la Figura 18.70.

Para ayudarnos en la tarea, mantenemos dos variables, `nodoBorrado` y `ultimoNodo`, que son permanentes ya que se declaran como estáticas. Cuando recorremos un hijo derecho, ajustamos `nodoBorrado`. Debido a que invocamos recursivamente a `eliminar` hasta que llegamos al final (no preguntamos acerca de la igualdad en el recorrido descendente), tenemos garantizado que si el elemento que debemos eliminar está en el árbol, `nodoBorrado` apuntará al nodo que lo contiene.

```
43      private NodoBinario
44      insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
45        { /* Figura 18.67 */ }
46      private NodoBinario
47      eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
48        { /* Figura 18.70 */ }
49      private void imprimirArbol( NodoBinario t )
50        { /* Implementación habitual; ver el código en Internet */ }
51      private NodoBinario giro( NodoBinario t )
52        { /* Figura 18.68 */ }
53      private NodoBinario reparto( NodoBinario t )
54        { /* Figura 18.69 */ }
55
56      private NodoBinario raiz;
57      private static NodoBinario nodoNulo;
58          static     // Inicializador estático de nodoNulo
59          {
60              nodoNulo = new NodoBinario( null );
61              nodoNulo.izquierdo = nodoNulo.derecho = nodoNulo;
62              nodoNulo.nivel = 0;
63          }
64
65      private static NodoBinario nodoBorrado;
66      private static NodoBinario ultimoNodo;
67  }
```

Figura 18.66 Esqueleto de la clase de los AA_árboles (parte 2).

```
1   /**
2    * Método interno para insertar en un subárbol.
3    * @param x el elemento a insertar.
4    * @param t la raíz del árbol.
5    * @return la nueva raíz.
6    * @exception ElementoDuplicado si el elemento que ajusta
7    *            con x ya está en el subárbol de raíz t.
8    */
9   private NodoBinario
10  insertar( Comparable x, NodoBinario t ) throws ElementoDuplicado
11  {
12      if( t == nodoNulo )
13          t = new NodoBinario( x, nodoNulo, nodoNulo );
14      else if( x.compara( t.dato ) < 0 )
15          t.izquierdo = insertar( x, t.izquierdo );
16      else if( x.compara( t.dato ) > 0 )
17          t.derecho = insertar( x, t.derecho );
18      else
19          throw new ElementoDuplicado( "insertar de AA_Arbol" );
20
21      t = giro( t );
22      t = reparto( t );
23      return t;
24  }
```

Figura 18.67 Método `insertar` de la clase `AA_Arbol`.

```
1  /**
2   * Giro primitivo para AA-árboles.
3   * @param t la raíz del árbol.
4   * @return la nueva raíz tras la rotación.
5   */
6  private NodoBinario giro( NodoBinario t )
7  {
8      if( t.izquierdo.nivel == t.nivel )
9          t = Rotaciones.conHijoIzquierdo( t );
10     return t;
11 }
```

Figura 18.68 Método `giro` de la clase `AA_Arbol`.

```
1  /**
2   * Reparto primitivo para los AA-árboles.
3   * @param t la raíz del árbol.
4   * @return la nueva raíz después de la rotación.
5   */
6  private NodoBinario reparto( NodoBinario t )
7  {
8      if( t.derecho.derecho.nivel == t.nivel )
9      {
10         t = Rotaciones.conHijoDerecho( t );
11         t.nivel++;
12     }
13     return t;
14 }
```

Figura 18.69 Método `reparto` de la clase `AA_Arbol`.

`nodoBorrado` referenciará el nodo que contiene el elemento `x` (si `x` está en el árbol) o `nodoNulo` si `x` no está en el árbol. `ultimoNodo` apuntará al nodo de sustitución. Empleamos comparaciones entre dos en lugar de comparaciones entre tres.

Nótese que esta misma técnica puede emplearse en la rutina `buscar` para sustituir las comparaciones entre tres realizadas en cada nodo por comparaciones entre dos en cada nodo, más un test de igualdad adicional al final. `ultimoNodo` referencia el nodo del nivel–1 en el que finaliza la búsqueda. Como no nos detenemos hasta que llegamos al final, si el elemento se encuentra en el árbol, `ultimoNodo` apunta al nodo del nivel–1 que contiene el valor de reemplazamiento que debe ser eliminado.

Cuando termina una llamada recursiva, si estamos en el nivel 1, colocamos el valor del nodo en el nodo interno que debe ser sustituido, tras lo que podemos sobrepasar el nodo del nivel–1. En caso contrario, estamos en un nivel superior y necesitamos comprobar si se incumple la condición de equilibrio. Si es así, restablecemos el equilibrio y hacemos tres llamadas a `giro` y dos llamadas a `reparto`. Como hemos comentado anteriormente, esto garantiza que se recuperan las propiedades de los AA-árboles.

18.7 B-Árboles

Hasta este momento hemos supuesto que podemos almacenar las estructuras de datos completas en la memoria principal del computador. Supongamos ahora que tenemos más datos de los que podemos guardar en memoria principal, por lo que debemos almacenar los datos en disco. Cuando esto sucede las reglas del juego cambian, ya que a la hora de evaluar la complejidad de los algoritmos la notación O resulta insuficiente.

```
1  /**
2   * Método interno para eliminar elementos de un subárbol.
3   * @param x el elemento a eliminar.
4   * @param t la raíz del árbol.
5   * @return la nueva raíz.
6   * @exception ElementoNoEncontrado si ningún elemento del árbol
7   *        de raíz t ajusta con x.
8   */
9  private NodoBinario
10 eliminar( Comparable x, NodoBinario t ) throws ElementoNoEncontrado
11 {
12     if( t != nodoNulo )
13     {
14         // Paso 1: Busca descendiendo por el árbol y actualiza
15         //         ultimoNodo y nodoBorrado
16         ultimoNodo = t;
17         if( x.menorQue( t.dato ) )
18             t.izquierdo = eliminar( x, t.izquierdo );
19         else
20         {
21             nodoBorrado = t;
22             t.derecho = eliminar( x, t.derecho );
23         }
24         // Paso 2: Si x está al final del árbol,
25         //         lo eliminamos
26         if( t == ultimoNodo )
27         {
28             if( nodoBorrado == nodoNulo ||
29                  x.compara( nodoBorrado.dato ) != 0 )
30                 throw new ElementoNoEncontrado( "de AA_Arbol eliminar" );
31             nodoBorrado.dato = t.dato;
32             nodoBorrado = nodoNulo;
33             t = t.derecho;
34         }
35         // Paso 3: En otro caso, no estamos al final;
36         //         debemos restablecer el equilibrio
37         else
38             if( t.izquierdo.nivel < t.nivel - 1 ||
39                 t.derecho.nivel < t.nivel - 1 )
40             {
41                 if( t.derecho.nivel > --t.nivel )
42                     t.derecho.nivel = t.nivel;
43                 t = giro( t );
44                 t.derecho = giro( t.derecho );
45                 t.derecho.derecho = giro( t.derecho.derecho );
46                 t = reparto( t );
47                 t.derecho = reparto( t.derecho );
48             }
49     }
50     return t;
51 }
```

Figura 18.70 Eliminación en un AA-árbol.

El problema es que el análisis en notación *O* asume que el coste de todas las operaciones es similar. Pero, esto en particular deja de ser cierto cuando interviene la E/S de disco. Por ejemplo, una máquina 25-MIPS supuestamente ejecuta 25 millones de instrucciones por segundo. Esto supone ir muy deprisa, principalmente

Cuando los datos son demasiado grandes para almacenarlos en memoria, el número de accesos a disco adquiere importancia. Un acceso a disco es increíblemente costoso en comparación con una instrucción habitual del computador.

por que la velocidad depende en gran medida de las propiedades eléctricas. Por contra, un disco es, en buena parte, un dispositivo mecánico. Por ello, su velocidad depende sobre todo del tiempo de giro y del tiempo que tarda en mover su cabeza lectora. La mayoría de discos giran a 3.600 RPM (discos más rápidos giran a 5.400 RPM). Así, en un minuto se realizan 3.600 revoluciones, o lo que es lo mismo, se produce una revolución cada 1/60 de segundo o 16.7 ms. En media, debemos esperar que, para encontrar lo que estamos buscando, el disco girará media vuelta, luego, si ignoramos otros factores, se tiene un tiempo de acceso de 8.3 ms. Ésta es una estimación generosa, pues los tiempos de acceso habituales varían entre 9 y 11 ms. Como consecuencia, podemos realizar unos 120 accesos a disco por segundo. Esto suena muy bien, hasta que lo comparamos con la velocidad del procesador. Lo que nos encontramos entonces es que 25 millones de instrucciones equivalen a 120 accesos a disco. O visto de otro modo, que un acceso a disco tiene el coste de 200.000 instrucciones. Desde luego, todos son cálculos aproximados, pero las velocidades relativas están muy claras: los accesos a disco son increíblemente costosos. Más aún, las velocidades de los procesadores están creciendo mucho más rápido que las velocidades de los discos (son sus *tamaños* los que sí están creciendo también deprisa), así que podemos permitirnos hacer un montón de cálculos si con ellos conseguimos ahorrarnos un acceso a disco. En la mayoría de los casos, es el número de accesos a disco el que gobierna el tiempo de ejecución. Así que si dividimos por dos el número de accesos a disco dividimos también por dos el tiempo de ejecución.

Veamos como se comporta normalmente un árbol de búsqueda sobre un disco. Supongamos que se desea tener acceso a los registros de conducción de los ciudadanos de Madrid. Asumimos que se tienen 2.000.000 elementos, que cada clave ocupa 32 bytes (es un nombre), y que cada registro supone 256 bytes. Asumimos que todo esto no cabe en la memoria principal y que somos 1 de los 20 usuarios de un sistema (luego disponemos de 1/20 de los recursos). Así, en un segundo, podemos ejecutar un millón de instrucciones o realizar seis accesos a disco.

Cualquier comportamiento logarítmico es inaceptable. Necesitamos realizar búsquedas con sólo tres o cuatro accesos a disco. En las modificaciones nos podríamos permitir algunos más.

Entonces el uso de un árbol binario de búsqueda no equilibrado puede suponer un desastre. En el peor de los casos, puede llegar a tener profundidad lineal, lo que nos llevaría a 2.000.000 accesos a disco. En media, una búsqueda exitosa requeriría $1{,}38 \log N$ accesos a disco, y como $\log 2.000.000 \approx 22$, una búsqueda media necesitaría 31 accesos a disco o 5 segundos. El comportamiento de un árbol rojinegro es algo mejor. El caso peor de $2 \log N$ es poco probable y el caso habitual está muy cercano a $\log N$. Como consecuencia, un árbol rojinegro emplearía, en media, 23 accesos a disco, requiriendo 4 segundos.

Los *árboles de búsqueda M-arios* permiten ramificaciones de M hijos. En tanto aumenta el número de hijos disminuye su profundidad.

Deseamos reducir el número de accesos a disco hasta una constante muy pequeña, como tres o cuatro. Estamos abiertos a escribir un código bastante complicado ya que las instrucciones máquina son, en este contexto, esencialmente gratis, a menos que abusemos de ellas de una forma ridículamente irrazonable. Debe quedar claro que un árbol binario de búsqueda no resolverá nunca el problema, ya que la profundidad habitual del árbol rojinegro está muy próxima a la altura óptima. No podemos ir más allá de $\log N$ empleando un árbol de búsqueda binario. La solución es intuitivamente sencilla: si tenemos mayor grado de ramificación, tenemos menor altura. Así, mientras un árbol binario perfecto de 31 nodos tiene 5 niveles, un árbol 5-ario de 31 nodos tiene sólo tres niveles, tal y como se muestra en la Figura 18.71. Un *árbol de búsqueda M-ario* permite ramificaciones de M hijos. A medida que el grado de ramificación aumenta, la profundidad disminuye. Cuando

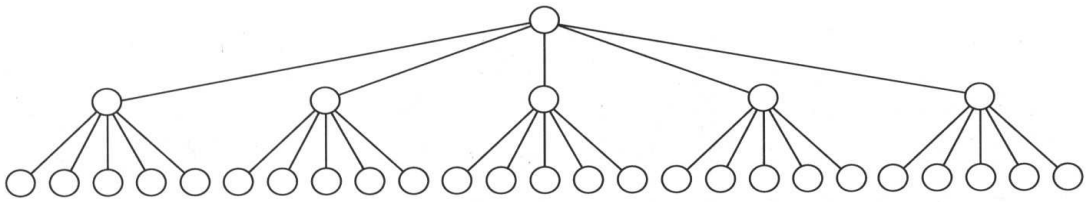

Figura 18.71 Un árbol 5-ario de 31 nodos tiene sólo tres niveles.

la altura de un árbol binario completo es, aproximadamente, $\log_2 N$, la altura de un árbol M-ario completo es, más o menos, $\log_M N$.

Podemos definir los árboles de búsqueda M-arios de forma similar a como definimos los árboles de búsqueda binarios. En este último caso, necesitamos una clave para decidir cuál de las dos ramas tomar. En un árbol de búsqueda M-ario, son necesarias $M - 1$ claves para decidir cuál de las ramas debemos utilizar. Para conseguir que este esquema sea eficiente en el peor de los casos, es preciso asegurarnos que el árbol M-ario esté equilibrado de alguna forma. En caso contrario, y al igual que los árboles binarios de búsqueda, podría degenerar en una lista enlazada. Ahora necesitamos una condición de equilibrio aún más restrictiva, ya que tampoco podemos permitirnos que nuestro árbol M-ario degenere en un árbol binario, ya que en tal caso volveríamos a los log N accesos.

El *B-árbol* es la estructura de datos más popular para realizar la búsqueda en disco.

Una forma de implementar todo esto es emplear *B-árboles*. En esta sección describimos el concepto básico de B-árbol[1]. Se conocen muchas variaciones y mejoras, y su implementación es ligeramente complicada, ya que es necesario distinguir bastantes casos. Sin embargo, es sencillo comprobar que, en principio, el uso de un B-árbol garantiza tener que realizar pocos accesos a disco.

Los B-árboles satisfacen cinco propiedades fundamentales.

Un B-árbol de orden M es un árbol M-ario que verifica las siguientes propiedades[2]:

1. Los datos se almacenan en las hojas.
2. Los nodos internos contienen $M - 1$ claves para guiar la búsqueda: la clave i representa la menor clave en el subárbol $i + 1$.
3. La raíz es una hoja o tiene entre 2 y M hijos.
4. Todos los nodos internos, excepto la raíz, tienen entre $\lceil M/2 \rceil$ y M hijos.
5. Todas las hojas se encuentran a la misma profundidad y tienen entre $\lceil L/2 \rceil$ y L hijos para un cierto valor fijo L (la discusión sobre cómo elegir L se realizará a continuación).

Los nodos deben estar, al menos, medio llenos. Esto garantiza que el árbol no degenerará en un simple árbol binario o ternario.

En la Figura 18.72 se muestra un ejemplo de B-árbol de orden 5. Nótese que todos los nodos internos tienen entre tres y cinco hijos (y por tanto, entre dos y cuatro claves); la raíz podría tener sólo dos hijos. Aquí tenemos $L = 5$. En este ejemplo L y M tienen los mismos valores, pero esto no es necesario, en general. Como L es 5, cada hoja guarda entre tres y cinco datos. Es necesario que los nodos estén medio llenos para poder garantizar que el B-árbol no degenerará en un simple árbol binario o ternario. Aunque existen varias definiciones de B-árboles en las que cambia esta estructura, en muchas de ellas en un grado muy pequeño, esta definición es una de las más extendidas.

[1] Lo que describimos aquí se conoce popularmente como B$^+$-árbol.

[2] Las reglas 3 y 5 deben relajarse durante las primeras L inserciones.

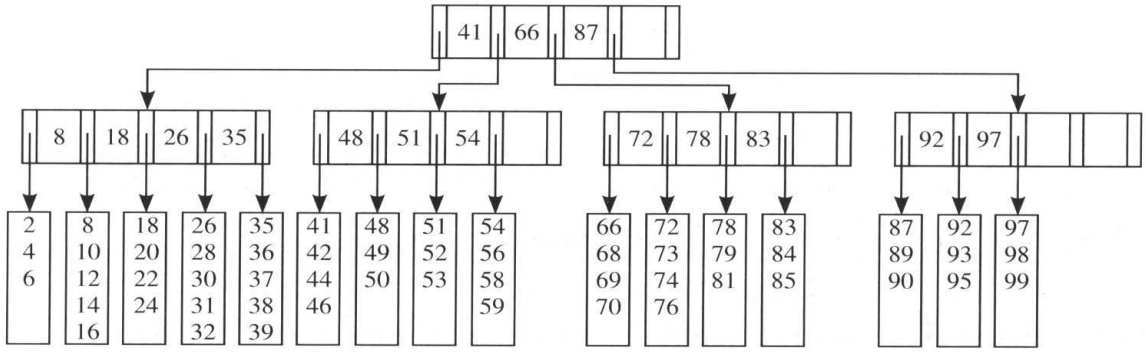

Figura 18.72 Un B-árbol de orden 5.

Escogemos los valores máximos de M y L que permiten almacenar un nodo en un bloque de disco.

Cada nodo representa un bloque de disco, así que escogemos M y L basándonos en el tamaño de los elementos que deben guardarse. Como ejemplo, supongamos que un bloque almacena 8.192 bytes. En nuestro ejemplo cada clave ocupa 32 bytes. En un B-árbol de orden M, tendríamos $M - 1$ claves para un total de $32M - 32$ bytes, además de M ramas. Como cada rama es, esencialmente, el número de otro bloque de disco, podemos suponer que cada una de ellas ocupa 4 bytes. Es decir, las ramas ocupan $4M$ bytes. La memoria total necesaria para un nodo interno es $36M - 32$. El mayor valor de M para el que esta cifra no supera 8.192 es 228. Así que elegiremos $M = 228$. Como cada registro de datos es de 256 bytes, seríamos capaces de almacenar 32 de ellos en un bloque, por lo que $L = 32$. De este modo hemos garantizado que cada hoja tiene entre 16 y 32 registros de datos y que cada nodo interno (excepto la raíz) puede ramificarse en al menos 114 hijos. Como tenemos 2.000.000 registros, existen a lo sumo 62.500 hojas. Como consecuencia, en el caso peor, las hojas estarán en el nivel 4. Más concretamente, el número de accesos en el caso peor viene dado aproximadamente por $\log_{M/2} N$, más o menos 1. (Por ejemplo, la raíz y el primer nivel pueden almacenarse en la memoria principal, por lo que los accesos a disco sólo serían necesarios para el nivel 3 e inferiores.)

La cuestión pendiente es cómo insertar y eliminar elementos en un B-árbol. Las ideas básicas se esbozan a continuación. Observe que vuelven a aparecer muchos de los aspectos ya estudiados.

Si la hoja tiene espacio libre para un nuevo elemento, lo insertamos en ella y hemos terminado.

Comenzamos estudiando la inserción. Supongamos que se desea insertar el elemento 57 en el B-árbol de la Figura 18.72. Una búsqueda descendente por el árbol muestra que dicho elemento aún no está en el árbol y que podemos añadirlo en el nivel de las hojas como quinto hijo. Nótese que es posible que para hacerlo debamos reorganizar la información en la correspondiente hoja. Sin embargo, el coste de dicha reorganización es despreciable, si lo comparamos con el tiempo de los accesos a disco, en los que, en este caso, se incluye una escritura en disco.

Si la hoja está llena, podemos insertar un nuevo elemento, dividiendo dicha hoja y generando dos nuevos nodos medio llenos.

Desde luego, esto es bastante sencillo ya que la hoja aún no está llena. Supongamos ahora que se desea insertar el elemento 55. La Figura 18.73 muestra el problema: la hoja donde queremos guardar 55 ya está llena. La solución es sencilla: como tenemos $L + 1$ elementos, los repartimos en dos hojas, en las que queda garantizado que podemos almacenar el menor número posible de elementos. En este caso, creamos dos nuevas hojas con tres elementos cada una. Se necesitan dos accesos a disco para escribir dichas hojas y un tercero para actualizar el padre. Note que en el padre cambian las dos claves y las dos ramas, pero lo hacen de una

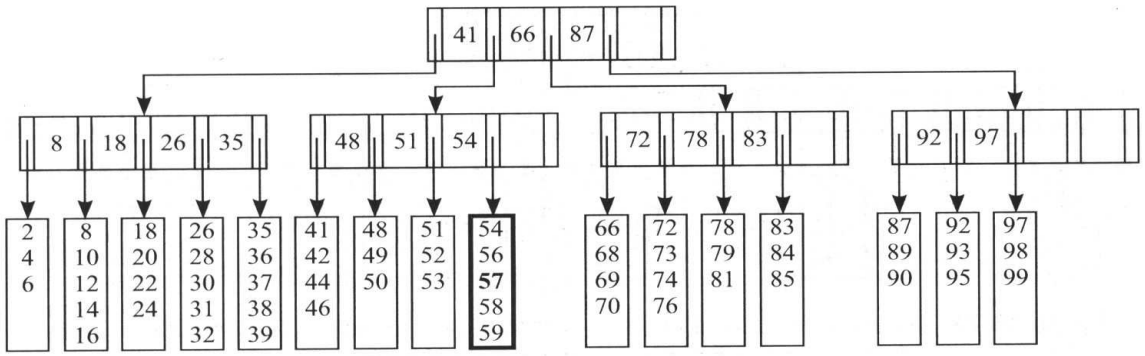

Figura 18.73 El B-árbol resultante tras de la inserción de 57 en el árbol de la Figura 18.72.

forma controlada, que puede calcularse fácilmente. El B-árbol obtenido se muestra en la Figura 18.74. Aunque la división de nodos consume tiempo, ya que necesita dos escrituras en disco adicionales, es una situación relativamente rara. Por ejemplo, si L es 32, cuando dividimos un nodo se crean dos hojas con 16 y 17 elementos, respectivamente. Para la hoja con 17 elementos, podemos realizar 15 inserciones más sin que sea necesario ningún desdoblamiento adicional. Visto de otro modo, por cada división de un nodo hay, aproximadamente, $L/2$ inserciones sin desdoblamiento.

La división de nodos genera un hijo extra para el padre de la hoja. Si el padre ya ha completado su número de hijos, también le dividimos.

La división en el ejemplo anterior del nodo resuelve el problema, ya que el padre no tiene completo su conjunto de hijos. Pero, ¿qué ocurriría en caso contrario? Supongamos que, por ejemplo, insertamos el elemento 40 en el B-árbol de la Figura 18.74. Debemos dividir la hoja que contiene las claves entre 35 y 39, y ahora 40, en dos nuevas hojas. Pero al hacerlo, el padre tendría seis hijos, y sólo se permite un máximo de cinco. La solución es dividir también el padre. El resultado se muestra en la Figura 18.75. Cuando el padre se divide, debemos actualizar los valores de las claves y también el padre del padre, empleando dos escrituras en disco adicionales (así que esta inserción cuesta cinco escrituras en disco). Sin embargo, una vez más, las claves cambian de una forma controlada, aunque ciertamente, debido a la gran cantidad de casos el código no es nada sencillo.

Cuando se divide un nodo interno, como sucede en este caso, su padre gana un hijo. ¿Qué ocurre si el padre ha llegado al límite permitido de hijos? En tal caso continuamos la división de nodos hasta que encontremos un padre que no necesite

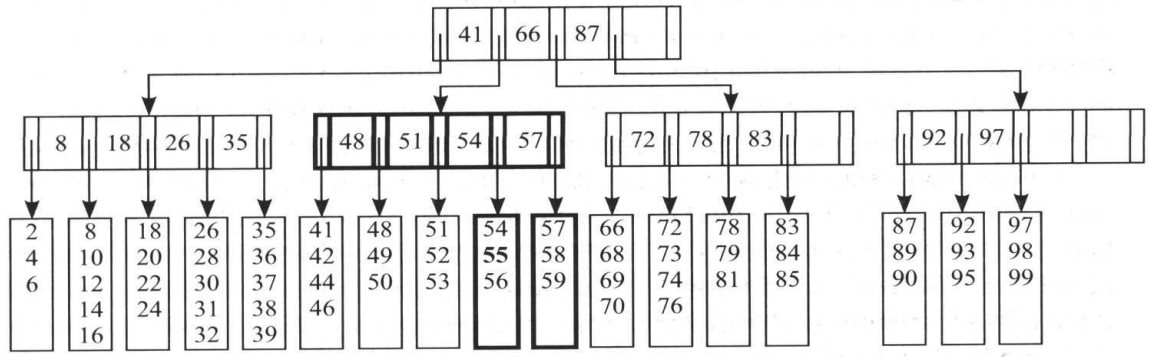

Figura 18.74 La inserción de 55 en el B-árbol de la Figura 18.73 provoca la división en dos nuevas hojas.

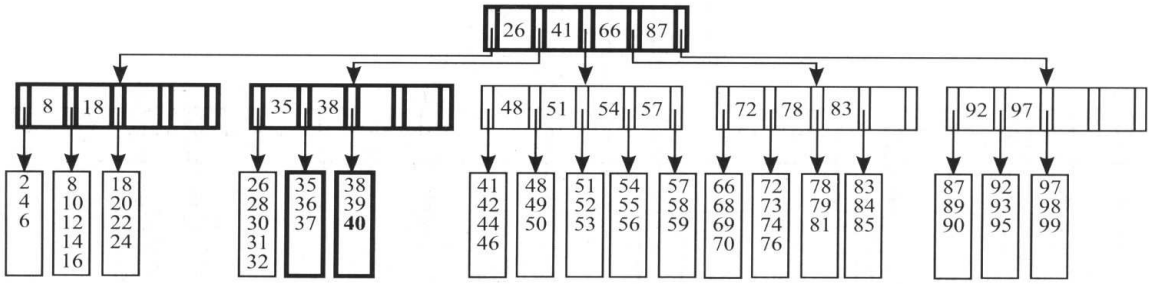

Figura 18.75 La inserción de 40 en el B-árbol de la Figura 18.74 provoca la división en dos nuevas hojas y la división del padre.

Debemos continuar dividiendo nodos en el camino hacia la raíz (aunque esto sucede raramente). En el peor de los casos, dividimos la raíz, creando una nueva raíz con dos hijos.

ser dividido o bien la raíz del árbol. Observe que esta idea ya ha sido utilizada en los árboles ascendentes rojinegros y en los AA-árboles. Si dividimos la raíz, obtenemos dos raíces. Obviamente, esto es inaceptable, pero en lugar de ello podemos generar una nueva raíz que ahora tenga como hijos las dos raíces previamente obtenidas. Ésta es la razón por la que se permite que la raíz pueda tener sólo dos hijos. Ésta es también la única manera en la que un B-árbol puede ganar altura. No es necesario comentar que la división de todos los nodos hasta llegar a la raíz es una situación excepcional. Esto es debido a que, por ejemplo, en un árbol con cuatro niveles la raíz tendrá que haber sido dividida tres veces a lo largo de toda la secuencia de inserciones (suponiendo que no se ha producido ninguna eliminación). De hecho, tener que dividir un nodo que no sea una hoja es una situación bastante rara.

Existen otras formas de manejar el desbordamiento en el número de hijos. Una técnica alternativa es dar un hijo en adopción a un vecino que tenga sitio para él. Por ejemplo, para insertar 29 en el B-árbol de la Figura 18.75, podemos hacer sitio moviendo el elemento 32 a la siguiente hoja. Esta técnica requiere una modificación del padre, ya que las claves se ven afectadas. Sin embargo, tiende a mantener llenos los nodos, ahorrando espacio a largo plazo.

La eliminación funciona al revés. Si una hoja pierde un hijo, es posible que necesitemos combinarla con otra hoja. La combinación de nodos puede continuar a lo largo del camino hasta la raíz, aunque esto es improbable. En el caso peor, la raíz pierde uno de sus dos hijos. En tal caso la eliminamos y empleamos el otro hijo como nueva raíz.

Podemos realizar una eliminación buscando el elemento que debemos borrar y después eliminándolo. El problema es que si la hoja en la que se encuentra sólo tiene el mínimo número de elementos permitido, ahora nos quedamos por debajo de dicho mínimo. Podemos arreglar esta situación adoptando el hijo de un vecino, si es que el vecino no tiene también el mínimo número de hijos. Si no es así, podemos combinar la hoja con dicho vecino para obtener una hoja completa. Desafortunadamente, esto significa que el padre ha perdido un hijo. Si esto causa que el padre también esté bajo mínimos, reiteramos el mismo procedimiento. Este proceso puede repetirse a lo largo del camino hacia la raíz. La raíz no puede tener un único hijo (permitir tal cosa sería claramente una tontería), así que si, como resultado de la adopción, se queda con un solo hijo, lo que haremos será eliminarle, convirtiendo a su hijo en la nueva raíz del árbol. Ésta es la única manera de que un B-árbol pierda altura. Por ejemplo, supongamos que se desea eliminar el elemento 99 en el B-árbol de la Figura 18.75. Como la hoja tiene sólo dos elementos y el vecino tiene también el mínimo de tres, combinamos los elementos en una nueva hoja de cinco elementos. Como resultado, el padre tiene ahora dos hijos. Sin embargo, puede adoptar otro de un vecino, ya que éste tiene cuatro hijos. Así, ambos tendrán tres hijos. El resultado se muestra en la Figura 18.76.

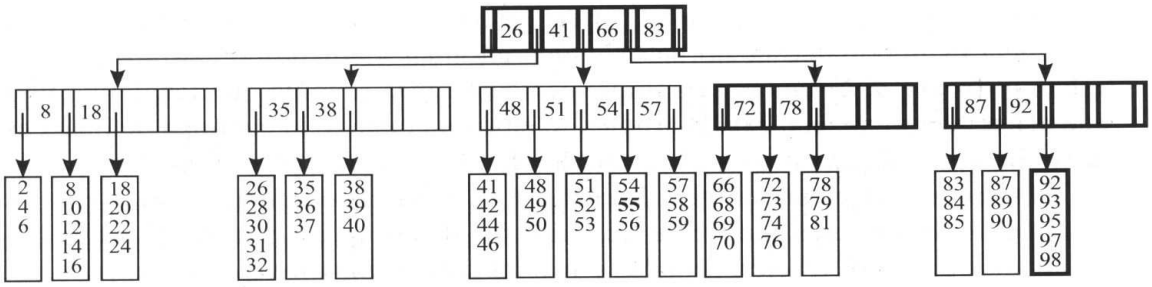

Figura 18.76 B-árbol resultante tras la inserción de 99 en el B-árbol de la Figura 18.75.

Resumen

Los árboles binarios de búsqueda son muy importantes dentro del diseño de algoritmos. Soportan casi todas las operaciones más útiles, y su coste logarítmico en media es muy reducido. Las implementaciones no recursivas de los árboles son algo más rápidas que las correspondientes versiones recursivas, pero estas últimas son más elegantes y más fáciles de entender y depurar. El problema que plantean los árboles de búsqueda es que su eficiencia depende en gran medida de la aleatoriedad de la entrada. Si no se da el caso, el tiempo de ejecución puede crecer extraordinariamente, hasta el punto de que los árboles de búsqueda se hacen más costosos que las listas enlazadas.

Hemos visto varias maneras de solucionar este problema. Todas ellas incluyen la reestructuración del árbol para asegurar que en cada nodo se da algún tipo de equilibrio. La reestructuración se realiza a través de rotaciones que preservan la propiedad de los árboles binarios de búsqueda. Para estos árboles, habitualmente el coste de una búsqueda es menor que el coste en un árbol binario de búsqueda no equilibrado, ya que el nodo central tiende a estar cerca de la raíz. Sin embargo, los costes de la inserción y la eliminación suelen ser algo mayores. Las variaciones en el equilibrio de los árboles se distinguen en la cantidad de esfuerzo necesario para implementar las operaciones que los modifican.

El esquema clásico es el árbol AVL, en el que para cada nodo, las alturas de sus hijos derecho e izquierdo deben diferir, a lo sumo, en una unidad. El problema que presentan en la práctica los árboles AVL es que exigen considerar gran cantidad de casos diferentes, haciendo que el coste de cada inserción o eliminación sea relativamente alto. En este capítulo se han estudiado dos alternativas. La primera son los árboles rojinegros descendentes. Su principal ventaja es que la recuperación de la condición de equilibrio puede implementarse en un único paso, descendiendo por el árbol, en sustitución de la solución tradicional, que desciende por el árbol para regresar después al punto de partida.

Esto conduce a un código más simple y a una ejecución más rápida de la que permiten los árboles AVL. La segunda alternativa son los AA-árboles, que son similares a los árboles rojinegros ascendentes. Su ventaja principal es una implementación recursiva relativamente sencilla de las rutinas de inserción y eliminación. Ambas emplean nodos centinelas para eliminar los casos especiales un tanto fastidiosos.

Se deben emplear árboles binarios de búsqueda no equilibrados cuando se está seguro de que los datos son razonablemente aleatorios y que la cantidad de los

mismos es relativamente pequeña. Se deben emplear los árboles rojinegros si la eficiencia es importante (y no necesitamos eliminar elementos). Los AA-árboles deben utilizarse si se quiere una implementación más sencilla que tenga una más que aceptable eficiencia. Los B-árboles se emplean cuando la cantidad de datos es demasiado grande como para almacenarlos juntos en la memoria principal.

El Capítulo 21 examina otra alternativa: el *árbol de ensanchamiento*. Ésta es una opción muy interesante frente a los árboles de búsqueda equilibrados que tiene además una codificación sencilla y resulta muy competitiva en la práctica. El capítulo siguiente estudia las tablas hash, que son un método completamente diferente de implementar las operaciones de búsqueda.

Elementos del juego

AA-árbol Árbol de búsqueda equilibrado empleado cuando se necesita un caso peor $O(\log N)$, se aceptan implementaciones poco elaboradas y se requiere poder eliminar elementos.

árbol AVL Árbol binario de búsqueda con la propiedad adicional de equilibrio que consiste en que, en cada nodo del árbol, las alturas de sus subárboles izquierdo y derecho difieren a lo sumo en una unidad. Como primer tipo de árbol equilibrado de búsqueda, tiene una gran importancia histórica. Además, ilustra muchas de las ideas que se emplean en otros esquemas.

árbol binario de búsqueda equilibrado Árbol que añade una propiedad estructural para garantizar profundidad logarítmica en el caso peor. Las actualizaciones son más lentas que en los árboles binarios de búsqueda, pero los accesos son más rápidos.

árbol binario de búsqueda Estructura de datos que permite la inserción, búsqueda y eliminación en un tiempo medio de $O(\log N)$. Para cualquier nodo del árbol, todos los nodos con una clave menor que él están en el subárbol izquierdo y todos los nodos con una clave mayor están el subárbol derecho. No se permite la existencia de duplicados.

árbol M-ario Árbol que permite ramificaciones en M hijos. A medida que el grado de ramificación crece, la profundidad disminuye.

árbol rojinegro Árbol de búsqueda equilibrado que es una buena alternativa al árbol AVL, ya que puede emplearse un único paso descendente durante las rutinas de inserción y de eliminación. Los nodos se colorean como rojos o negros de un modo restrictivo que garantiza la profundidad logarítmica. Una detallada codificación genera una implementación más rápida.

B-árbol La estructura de datos más popular para la búsqueda de datos en disco. Existen diversas variantes de la misma idea.

eliminación perezosa Método de eliminación que marca los elementos como borrados, pero no los elimina realmente.

enlace horizontal En un AA-árbol, es una conexión entre un nodo y un hijo suyo del mismo nivel. Un enlace horizontal debe ir siempre hacia la derecha y no pueden existir dos enlaces horizontales consecutivos.

`giro` y `reparto` Un `giro` elimina los enlaces horizontales izquierdos, realizando una rotación entre un nodo y su hijo izquierdo. Un `reparto` elimina enlaces horizontales consecutivos, realizando una rotación entre un nodo y su hijo derecho.

longitud del camino externo Suma del coste de acceso a todos los nodos externos del árbol de un árbol binario. Mide el coste de una búsqueda no exitosa.
longitud del camino interno Suma de las profundidades de los nodos de un árbol binario. Mide el coste de una búsqueda exitosa.
nivel de un nodo En un AA-árbol, es el número de enlaces izquierdos que hay en el camino desde el nodo hasta el centinela `nodoNulo`.
nodo externo Es el nodo `null`.
rotación doble Equivale a dos rotaciones simples.
rotación simple Intercambia los papeles de un padre y un hijo manteniendo la ordenación del árbol. Se emplea para recuperar el equilibrio de un árbol.

Errores comunes

1. Si se emplea un árbol de búsqueda no equilibrado cuando la secuencia de datos de entrada no es aleatoria se obtendrá una eficiencia pobre.
2. La implementación correcta de la operación `eliminar` presenta algunos trucos ingeniosos, especialmente en el caso de los árboles de búsqueda equilibrados.
3. La eliminación perezosa es una buena alternativa a la rutina `eliminar` estándar, pero entonces debemos modificar otros métodos, como `buscarMin`.
4. El código de los árboles de búsqueda equilibrados tiende casi siempre a presentar errores.
5. Si olvidamos devolver una referencia a la nueva raíz del subárbol en los métodos privados auxiliares `insertar` y `eliminar`, estaremos cometiendo un error. El valor de retorno debe ser asignado a `raiz`.
6. El empleo de centinelas con un código que no saque partido de ellos puede conducir a bucles infinitos. Un caso muy común es hacer un test acerca de `null` cuando se emplea el centinela `nodoNulo`.

En Internet

Todo el código de este capítulo forma parte del directorio **DataStructures**. Aquí están los nombres de los ficheros:

AATree.java	Contiene la implementación de los AA-árboles. Es la versión inglesa de la clase `AA_arbol`.
BinaryNode.java	Contiene la implementación de la clase `NodoBinario`.
BinarySearchTree.java	Contiene la implementación de los árboles binarios de búsqueda no equilibrados. Es la versión inglesa de la clase `ArbolBinarioBusqueda`.
BinarySearchTreeWithRank.java	Contiene los árboles binarios de búsqueda no equilibrados con búsqueda por posición. Es la versión inglesa de la clase `ABBConRango`.

mismos es relativamente pequeña. Se deben emplear los árboles rojinegros si la eficiencia es importante (y no necesitamos eliminar elementos). Los AA-árboles deben utilizarse si se quiere una implementación más sencilla que tenga una más que aceptable eficiencia. Los B-árboles se emplean cuando la cantidad de datos es demasiado grande como para almacenarlos juntos en la memoria principal.

El Capítulo 21 examina otra alternativa: el *árbol de ensanchamiento*. Ésta es una opción muy interesante frente a los árboles de búsqueda equilibrados que tiene además una codificación sencilla y resulta muy competitiva en la práctica. El capítulo siguiente estudia las tablas hash, que son un método completamente diferente de implementar las operaciones de búsqueda.

Elementos del juego

AA-árbol Árbol de búsqueda equilibrado empleado cuando se necesita un caso peor $O(\log N)$, se aceptan implementaciones poco elaboradas y se requiere poder eliminar elementos.

árbol AVL Árbol binario de búsqueda con la propiedad adicional de equilibrio que consiste en que, en cada nodo del árbol, las alturas de sus subárboles izquierdo y derecho difieren a lo sumo en una unidad. Como primer tipo de árbol equilibrado de búsqueda, tiene una gran importancia histórica. Además, ilustra muchas de las ideas que se emplean en otros esquemas.

árbol binario de búsqueda equilibrado Árbol que añade una propiedad estructural para garantizar profundidad logarítmica en el caso peor. Las actualizaciones son más lentas que en los árboles binarios de búsqueda, pero los accesos son más rápidos.

árbol binario de búsqueda Estructura de datos que permite la inserción, búsqueda y eliminación en un tiempo medio de $O(\log N)$. Para cualquier nodo del árbol, todos los nodos con una clave menor que él están en el subárbol izquierdo y todos los nodos con una clave mayor están el subárbol derecho. No se permite la existencia de duplicados.

árbol M-ario Árbol que permite ramificaciones en M hijos. A medida que el grado de ramificación crece, la profundidad disminuye.

árbol rojinegro Árbol de búsqueda equilibrado que es una buena alternativa al árbol AVL, ya que puede emplearse un único paso descendente durante las rutinas de inserción y de eliminación. Los nodos se colorean como rojos o negros de un modo restrictivo que garantiza la profundidad logarítmica. Una detallada codificación genera una implementación más rápida.

B-árbol La estructura de datos más popular para la búsqueda de datos en disco. Existen diversas variantes de la misma idea.

eliminación perezosa Método de eliminación que marca los elementos como borrados, pero no los elimina realmente.

enlace horizontal En un AA-árbol, es una conexión entre un nodo y un hijo suyo del mismo nivel. Un enlace horizontal debe ir siempre hacia la derecha y no pueden existir dos enlaces horizontales consecutivos.

`giro` y `reparto` Un `giro` elimina los enlaces horizontales izquierdos, realizando una rotación entre un nodo y su hijo izquierdo. Un `reparto` elimina enlaces horizontales consecutivos, realizando una rotación entre un nodo y su hijo derecho.

longitud del camino externo Suma del coste de acceso a todos los nodos externos del árbol de un árbol binario. Mide el coste de una búsqueda no exitosa.
longitud del camino interno Suma de las profundidades de los nodos de un árbol binario. Mide el coste de una búsqueda exitosa.
nivel de un nodo En un AA-árbol, es el número de enlaces izquierdos que hay en el camino desde el nodo hasta el centinela `nodoNulo`.
nodo externo Es el nodo `null`.
rotación doble Equivale a dos rotaciones simples.
rotación simple Intercambia los papeles de un padre y un hijo manteniendo la ordenación del árbol. Se emplea para recuperar el equilibrio de un árbol.

Errores comunes

1. Si se emplea un árbol de búsqueda no equilibrado cuando la secuencia de datos de entrada no es aleatoria se obtendrá una eficiencia pobre.
2. La implementación correcta de la operación `eliminar` presenta algunos trucos ingeniosos, especialmente en el caso de los árboles de búsqueda equilibrados.
3. La eliminación perezosa es una buena alternativa a la rutina `eliminar` estándar, pero entonces debemos modificar otros métodos, como `buscarMin`.
4. El código de los árboles de búsqueda equilibrados tiende casi siempre a presentar errores.
5. Si olvidamos devolver una referencia a la nueva raíz del subárbol en los métodos privados auxiliares `insertar` y `eliminar`, estaremos cometiendo un error. El valor de retorno debe ser asignado a `raiz`.
6. El empleo de centinelas con un código que no saque partido de ellos puede conducir a bucles infinitos. Un caso muy común es hacer un test acerca de `null` cuando se emplea el centinela `nodoNulo`.

En Internet

Todo el código de este capítulo forma parte del directorio **DataStructures**. Aquí están los nombres de los ficheros:

AATree.java	Contiene la implementación de los AA-árboles. Es la versión inglesa de la clase `AA_arbol`.
BinaryNode.java	Contiene la implementación de la clase `NodoBinario`.
BinarySearchTree.java	Contiene la implementación de los árboles binarios de búsqueda no equilibrados. Es la versión inglesa de la clase `ArbolBinarioBusqueda`.
BinarySearchTreeWithRank.java	Contiene los árboles binarios de búsqueda no equilibrados con búsqueda por posición. Es la versión inglesa de la clase `ABBConRango`.

RedBackTree.java	Contiene la implementación de los árboles rojinegros. Es la versión inglesa de la clase `ArbolRojoNegro`.
Rotations.java	Contiene las cuatro rotaciones básicas.

Ejercicios

Cuestiones breves

18.1. Muestre el resultado de insertar los elementos 3, 1, 4, 6, 9, 2, 5 y 7 en un árbol binario de búsqueda inicialmente vacío. Muestre después el resultado de borrar la raíz.

18.2. Dibuje todos los árboles binarios de búsqueda que pueden obtenerse a partir de la inserción de las permutaciones de 1, 2, 3 y 4. ¿Cuántos árboles son? ¿Cuál es la probabilidad de obtener cada uno de ellos si todas las permutaciones son igualmente probables?

18.3. Dibuje todos los árboles AVL que pueden obtenerse a partir de la inserción de las permutaciones de 1, 2 y 3. ¿Cuántos árboles son? ¿Cuál es la probabilidad de obtener cada uno de ellos si todas las permutaciones son igualmente probables?

18.4. Repita el ejercicio 18.3 para cuatro elementos.

18.5. Muestre el resultado de insertar 2, 1, 4, 5, 9, 3, 6 y 7 en un árbol AVL inicialmente vacío. Mostrar después el resultado si se hace en un árbol rojinegro descendente.

18.6. Repita los Ejercicios 18.3 y 18.4 para un árbol rojinegro.

18.7. Discuta las ventajas, y los inconvenientes, de evitar todas las excepciones y emplear `null` para indicar los errores.

Problemas teóricos

18.8. Demuestre el Teorema 18.2.

18.9. Muestre el resultado de insertar ordenadamente los elementos del 1 al 15 en un árbol AVL inicialmente vacío. Generalizar esto (con una demostración) para indicar qué es lo que ocurre cuando se insertan los elementos del 1 al $2^k - 1$ en un árbol AVL inicialmente vacío.

18.10. Diseñe un algoritmo para `eliminar` elementos en un árbol AVL.

18.11. Demuestre que la altura de un árbol rojinegro es a lo sumo $2 \log N$ (aproximadamente) y mostrar una secuencia de inserciones que alcance esta cota.

18.12. Muestre que cualquier árbol AVL puede colorearse como un árbol rojinegro. ¿Todos los árboles rojinegros satisfacen la propiedad de los árboles AVL?

18.13. Demuestre la corrección del algoritmo de eliminación de un AA-árbol.

18.14. Supongamos que el atributo `nivel` de un AA-árbol está representado por un `byte` 8-bit. ¿Cuál es el menor AA-árbol que podría desbordar el atributo `nivel` en la raíz?

18.15. Un B*-árbol de orden M es un B-árbol en el que cada nodo interior tiene entre $2M/3$ y M hijos. Las hojas se llenan en la misma medida. Describa un método para realizar la inserción en un B*-árbol.

Problemas prácticos

18.16. Implemente de forma recursiva `buscar`, `buscarMin` y `buscarMax`.

18.17. Implemente iterativamente `buscarKesimo` empleando la misma técnica que en la rutina `buscar` no recursiva.

18.18. Una representación alternativa que permite realizar la operación `buscarKesimo` es almacenar en cada nodo el tamaño de su subárbol izquierdo más 1. ¿Por qué representa esto una ventaja? Rescriba la clase de los árboles de búsqueda para emplear esta representación.

18.19. Escriba un método sobre árboles binarios de búsqueda que tenga como parámetros dos claves, `inferior` y `superior`, y escriba todos los elementos X que se encuentren dentro del rango especificado por dichas claves. El programa debe ejecutarse en un tiempo medio $O(K + \log N)$, donde K es el número de claves mostradas. Si K es pequeño, sólo debería examinarse una pequeña parte del árbol. Tiene que emplear un método recursivo oculto en lugar de iterar la búsqueda ordenada. También se pide acotar el tiempo de ejecución del algoritmo.

18.20. Escriba un método sobre árboles binarios de búsqueda que tenga como parámetros dos enteros, `inferior` y `superior`, y construya de forma óptima un `ABBConRango` equilibrado, que contenga todos los enteros entre `inferior` y `superior`, ambos inclusive. Todas las hojas deben estar en el mismo nivel (si el tamaño del árbol es una unidad menor que una potencia de 2) o en dos niveles consecutivos. *La rutina debe ejecutarse en tiempo lineal.* Pruebe el método resolviendo el problema Josephus (Sección 13.1).

18.21. Las rutinas para realizar las rotaciones dobles son ligeramente ineficientes ya que hacen cambios innecesarios sobre las referencias de los hijos. Rescriba dichas rutinas para evitar las llamadas a la rutina de la rotación simple.

18.22. Diseñe una implementación iterativa descendente de los AA-árboles. Comparar esta implementación con la del texto, discutiendo la simplicidad y eficiencia de ambas.

18.23. Escriba recursivamente los métodos `giro` y `reparto` de modo que en cada `eliminar` sólo se necesite una llamada a las mismas.

Problemas de programación

18.24. Rehaga la clase de los árboles binarios de búsqueda para implementar la eliminación perezosa. Nótese que esto afecta al resto de rutinas, especialmente a `buscarMin` y `buscarMax`, que ahora deben implementarse recursivamente.

18.25. Implemente los árboles binarios de búsqueda de modo que sólo se emplee una comparación por nivel en las rutinas `buscar`, `insertar` y `eliminar`.

18.26. Escriba un programa para evaluar empíricamente las siguientes estrategias de eliminación de nodos con dos hijos.

a) Sustituirlo por el nodo mayor, X, de T_L y eliminar recursivamente X.

b) Reemplazarlo alternativamente con el mayor nodo de T_L y el menor nodo de T_R, y eliminar recursivamente el nodo correspondiente.

c) Sustituirlo por el mayor nodo de T_L o el menor nodo de T_R, eliminando recursivamente el nodo adecuado, realizando la elección de forma aleatoria.

¿Qué estrategia obtiene los mejores resultados? ¿Cuál de ellas tarda menos tiempo de CPU en procesar una secuencia entera de operaciones?

18.27. Implemente el método `eliminar` para árboles rojinegros.

18.28. Implemente las operaciones de los árboles de búsqueda con búsqueda por posición para los árboles de búsqueda equilibrados que prefiera el lector.

18.29. Implemente los B-árboles sobre la memoria principal.

18.30. Implemente los B-árboles sobre ficheros en disco.

18.31. Diseñe un applet que ilustre las operaciones principales de los árboles binarios de búsqueda, tanto equilibrados como no equilibrados.

Bibliografía

Se puede encontrar más información acerca de los árboles binarios de búsqueda, y en particular sobre sus propiedades matemáticas, en los dos libros de Knuth [18 y 19].

Muchos artículos tratan el tema de la pérdida teórica de equilibrio en los árboles de búsqueda, causada por un algoritmo de eliminación sesgado. El artículo de Hibbard [16] propone el algoritmo original de la eliminación y establece que una eliminación conserva la aleatoriedad de los árboles. En [17] se realiza un análisis completo para árboles con tres nodos y en [3] se realiza dicho análisis para árboles con cuatro nodos. El artículo de Eppinger [10] muestra una evidencia empírica de la no aleatoriedad y los artículos de Culberson y Munro [7 y 8] muestran una evidencia analítica, aunque no una demostración completa para el caso general de inserciones y eliminaciones entremezcladas. La afirmación de que el nodo más profundo de un árbol binario de búsqueda aleatorio es tres veces más profundo que la media se demuestra en [11]; el resultado no es nada sencillo.

Los árboles AVL fueron propuestos por Adelson-Velskii y Landis [2]. Se puede encontrar un algoritmo de eliminación en [19]. Los análisis acerca de los costes de las búsquedas en un AVL están sin completar, pero pueden verse algunos resultados en [20]. El algoritmo descendente de los árboles rojinegros es de [15]; puede encontrarse una descripción más comprensible en [21]. En [12] se da una implementación de los árboles rojinegros descendentes sin nodos centinelas; además se presenta una demostración convincente de la poca utilidad de `nodoNulo`. El AA-árbol está basado en los B-árboles binarios simétricos discutidos en [4]. La implementación del texto es una adaptación de la descripción de [1]. En [13] pueden encontrarse multitud de tipos de árboles binarios de búsqueda equilibrados.

Los B-árboles aparecen por primera vez en [5]. La implementación descrita en el artículo original permite que los datos se almacenen tanto en los nodos internos como en las hojas. La estructura de datos descrita aquí se conoce en diversos contextos como B^+-árboles. En [9] puede encontrarse información acerca de los B*-árboles, descritos en el Ejercicio 18.15. En [6] se presenta un resumen de los distintos tipos de B-árboles. En [14] se muestran resultados empíricos sobre distintos esquemas empleados. En [12] puede verse una implementación en C++.

1. A. Andersson, «Balanced Search Trees Made Simple», *Proceedings of the Third Workshop on Algorithms and Data Structures* (1993), 61-71.

2. U. M. Adelson-Velskii y E. M. Landis, «An Algorithm for the Organization of Information», *Soviet Math. Doklady* **3** (1962), 1259-1263.

3. R. A. Baeza-Yates, «A Trivial Algorithm Whose Analysis Isn't: A Continuation», *BIT* **29** (1989), 88-113.

4. R. Bayer, «Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms», *Acta Informatica* **1** (1972), 290-306.

5. R. Bayer y E. M. McCreight, «Organization and Maintenance of Large Ordered Indices», *Acta Informatica* **1** (1972), 173-189.

6. D. Comer, «The Ubiquitous B-tree», *Computing Surveys* **11** (1979), 121-137.

7. J. Culberson y J. I. Munro, «Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations», *Computer Journal* **32** (1989), 68-75.

8. J. Culberson y J. I. Munro, «Analysis of the Standard Deletion Algorithm in Exact Fit Domain Binary Search Trees», *Algorithmica* **5** (1990), 295-311.

9. K. Culik, T. Ottman, y D. Wood, «Dense Multiway Trees», *ACM Transactions on Database Systems* **6** (1981), 486-512.

10. J. L. Eppinger, «An Empirical Study of Insertion and Deletion in Binary Search Trees», *Communications of the ACM* **26** (1983), 663-669.

11. P. Flajolet y A. Odlyzko, «The Average Height of Binary Search Trees and Other Simple Trees», *Journal of Computer and System Sciences* **25** (1982), 171-213.

12. B. Flamig, *Practical Data Structures in C++*, John Wiley and Sons, New York, New York (1994).

13. G. H. Gonnet y R. Baeza-Yates, *Handbook ofAlgorithms and Data Structures*, 2.ª ed., Addison-Wesley, Reading, Mass. (1991).

14. E. Gudes y S. Tsur, «Experiments with B-tree Reorganization», *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), 200-206.

15. L. J. Guibas y R. Sedgewick, «A Dichromatic Framework for Balanced Trees», *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8-21.

16. T. H. Hibbard, «Sorne Combinatorial Properties of Certain Trees with Applications to Searching and Sorting», *Journal of the ACM* **9** (1962), 13-28.

17. A. T. Jonassen y D. E. Knuth, «A Trivial Algorithm Whose Analysis Isn't», *Journal of Computer and System Sciences* **16** (1978), 301-322.

18. D. E. Knuth, *The Art of Computer Programming*: *Volume 1: Fundamental Algorithms*, 3.ª ed., Addison-Wesley, Reading, Mass. (1997).

19. D. E. Knuth, *The Art of Computer Programming*: *Volume 3: Sorting and Searching,* 2.ª ed., Addison-Wesley, Reading, Mass. (1997).

20. K. Melhorn, «A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions», *SIAM Journal on Computing* **11** (1982), 748-760.

21. R. Sedgewick, *Algorithms in C++*, Addison-Wesley, Reading, Mass. (1992).