CAPÍTULO

17 Árboles

El árbol es una estructura fundamental en la computación. Casi todos los sistemas operativos almacenan los ficheros en árboles o en estructuras de aspecto arbóreo. Los árboles se usan también en diseño de compiladores, procesamiento de textos y algoritmos de búsqueda. En el Capítulo 18 se discute esta última aplicación.

En este capítulo veremos:

- Una definición de árbol general y una discusión sobre cómo se usa en un sistema de ficheros.
- Un estudio de los *árboles binarios*.
- Cómo se implementan las operaciones sobre los árboles usando recursión.
- Cómo se recorre un árbol de forma no recursiva.

17.1 Árboles generales

Los árboles se pueden definir de dos formas: recursiva y no recursivamente. La definición no recursiva es la más directa, por lo que comenzamos por ella. La formulación recursiva nos permite escribir algoritmos simples para manipular árboles.

17.1.1 Definiciones

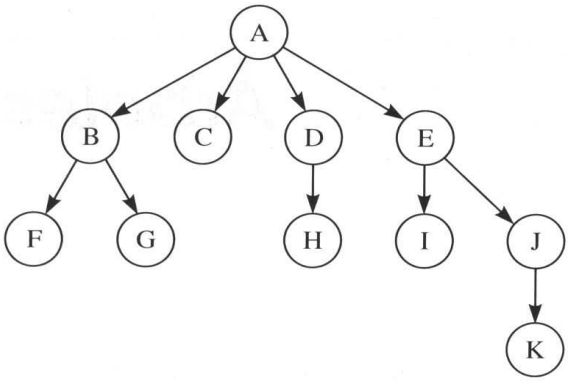
Los *árboles* se pueden definir no recursivamente como un conjunto de nodos y otro de aristas orientadas que los conectan. Los padres e hijos se definen de forma natural. Una arista orientada conecta el *padre* con un *hijo*.

Las *hojas* no tienen hijos.

Recuerde, del Capítulo 6, que un *árbol* consiste en un conjunto de nodos y otro de aristas orientadas que conectan pares de nodos. A lo largo de este libro, solamente consideramos árboles con raíz. Un árbol con raíz tiene las siguientes propiedades:

- Se distingue un nodo como raíz.
- A cada nodo c, exceptuando la raíz, le llega una arista desde exactamente otro nodo p, al cual se le llama *padre* de c. Decimos que c es uno de los *hijos* de p.
- Hay un único camino desde la raíz hasta cada nodo. El número de aristas del mismo recibe el nombre de *longitud del camino*.

La Figura 17.1 muestra un árbol. El nodo raíz es A, y los hijos de A son B, C, D y E. Como A es la raíz, no tiene padre. Todos los demás nodos tienen padre; por



Nodo	Altura	Profundidad
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

Figura 17.1 Un árbol, con información de altura y profundidad.

La *profundidad* de un nodo es la longitud del camino que va desde la raíz hasta el nodo. La *altura* de un nodo es la longitud del camino que va desde el nodo hasta la hoja más profunda bajo él.

ejemplo, el padre de B es A. Los nodos que no tienen hijos reciben el nombre de *hojas*. Las hojas de este árbol son C, F, G, H, I y K. La longitud del camino de A a K es tres (aristas), y la del camino de A a A es de cero aristas.

Un árbol con N nodos debe tener $N - 1$ aristas porque a cada nodo, excepto a la raíz, le llega una arista. La *profundidad* de un nodo en un árbol es la longitud del camino que va desde la raíz hasta ese nodo. En consecuencia, la profundidad de la raíz es siempre 0, y la de cualquier nodo es la de su padre más uno. La *altura* de un nodo es la longitud del camino que va desde el nodo hasta la hoja más profunda bajo él. Luego la altura de E es 2. La altura de cualquier nodo es 1 más que la mayor altura de un hijo suyo. La altura de un árbol se define como la altura de su raíz.

El *tamaño* de un nodo es igual al número de descendientes que tiene (incluyendo dicho nodo).

Los nodos que tienen el mismo padre son *hermanos*, luego B, C, D y E son todos hermanos. Si hay un camino del nodo u al nodo v, entonces decimos que u es un *ascendiente* de v y que v es un *descendiente* de u. Si $u \neq v$, entonces u es un *ascendiente propio* de v y v es un *descendiente propio* de u. El *tamaño* de un nodo es igual al número de descendientes que tiene (incluyendo dicho nodo). El tamaño de B es 3, y el de C es 1. El tamaño de un árbol se define como el tamaño de su raíz, luego el tamaño del árbol de la Figura 17.1 es igual al tamaño de A, que es 11.

Se puede dar una definición alternativa de los árboles en forma recursiva: un árbol es o bien vacío o consiste en una raíz y cero o más subárboles no vacíos T_1, T_2, ..., T_k, cada una de cuyas raíces está conectada por medio de una arista con la

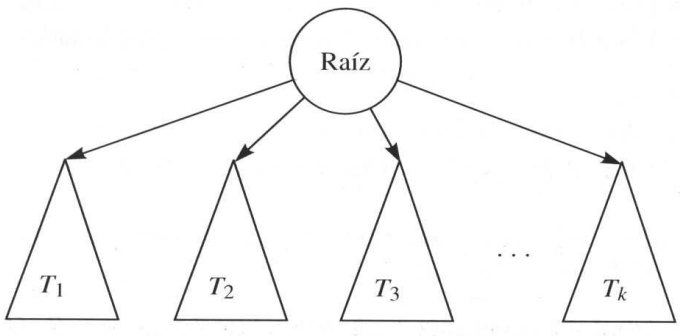


Figura 17.2 Un árbol visto de forma recursiva.

raíz. Una consecuencia de esta definición es que cada nodo de un árbol define un subárbol, que es aquel que le tiene por raíz. Se tiene así una identificación entre nodo y subárbol, que hemos utilizado más arriba al definir el tamaño de un nodo. En algunas clases de árboles, como los *árboles binarios* que se estudian más adelante en el capítulo, se puede permitir que algunos de los subárboles sean vacíos. Esta visión de los árboles se ilustra en la Figura 17.2.

17.1.2 Implementación

Los *árboles generales* se pueden implementar usando el método del *primer hijo/siguiente hermano*, el cual requiere dos referencias por elemento.

Una forma de implementar un árbol consistiría en tener en cada nodo una referencia a cada uno de sus hijos además de a los datos que contiene. Pero debido a que el número de hijos puede variar mucho y dicho número no se conoce a priori, puede que el hecho de conectar a los hijos directamente con el padre en la estructura de datos sea impracticable por la gran cantidad de espacio necesario. La solución es simple: mantener los hijos de cada nodo en una lista enlazada de nodos. Esto hace que cada nodo guarde dos referencias: una a su hijo más a la izquierda (siempre que no sea ya una hoja) y otra a su hermano de la derecha (siempre que no sea ya el situado más a la derecha). A este tipo de implementación se le llama *método primer hijo/siguiente hermano* y se ilustra en la Figura 17.3. Las flechas que apuntan hacia abajo son referencias al `primerHijo`, y las que van de izquierda a derecha son referencias al `siguienteHermano`. Las referencias `null` no se muestran en el dibujo pues hay demasiadas, y pueden deducirse del mismo. En este árbol, el nodo *B* tiene una referencia a su hermano *C* y a su primer hijo *F*, mientras que otros nodos poseen solamente una de estas referencias y algunos de ellos ninguna de las dos. Dada esta representación, es un ejercicio sencillo escribir una clase de árboles que implemente la interfaz de la Figura 6.15.

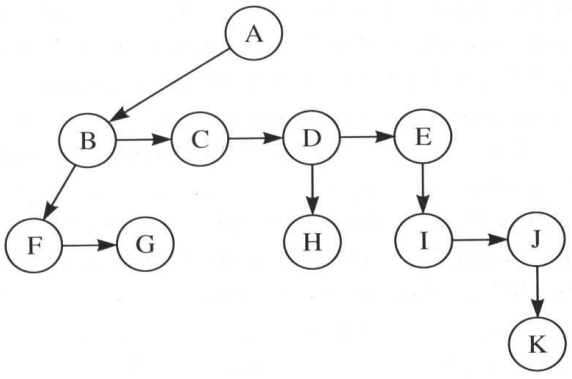


Figura 17.3 Representación primer hijo/siguiente hermano del árbol de la Figura 17.1.

17.1.3 Una aplicación: sistemas de ficheros

Los sistemas de ficheros usan estructuras arbóreas.

Hay muchas aplicaciones de los árboles. Una de las más conocidas es la estructura de directorios en muchos de los sistemas operativos más utilizados, como Unix, VAX/VMS y DOS. La Figura 17.4 muestra un directorio típico en el sistema de ficheros de Unix. La raíz de este directorio es `marca`. (El asterisco

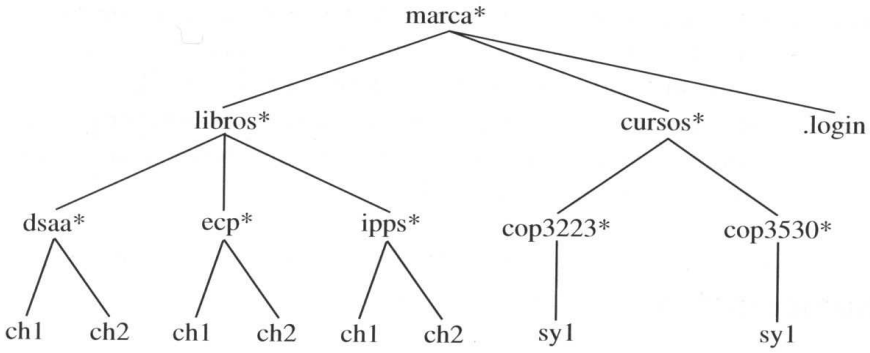


Figura 17.4 Directorio Unix.

junto al nombre indica que `marca` es un directorio.) El directorio `marca` tiene tres hijos —`libros`, `cursos` y `.login`— dos de los cuales son a su vez directorios. Luego `marca` contiene dos subdirectorios y un fichero. El nombre de fichero `marca/libros/dsaa/ch1` se obtiene tomando tres veces el hijo más a la izquierda. Cada `/` indica una arista, produciendo así un nombre de camino. Si el camino comienza en la raíz del sistema completo de ficheros, en lugar de en un directorio arbitrario dentro del sistema, entonces tenemos un nombre de camino completo; en caso contrario tenemos un nombre de camino relativo al directorio actual.

Este sistema de ficheros jerárquico es muy popular porque permite a los usuarios organizar sus datos de una forma estructurada y sencilla. Además, se pueden almacenar dos ficheros con el mismo nombre en distintos directorios, ya que por accederse a ellos a través de caminos distintos desde la raíz, sus nombres de camino completo son distintos. Un directorio en el sistema de ficheros Unix es simplemente un fichero con una lista de todos sus hijos[1], por lo que se pueden recorrer los directorios usando un esquema de iteración que implemente la clase abstracta de la Figura 6.15, es decir, podemos iterar sobre cada hijo. De hecho, en algunos sistemas, si a un directorio se le aplica el comando normal para imprimir un fichero, entonces se pueden ver en la salida los nombres de los ficheros contenidos en ese directorio (junto con otra información que no está en formato ASCII).

Se puede recorrer más fácilmente la estructura de un directorio usando recursión.

Supongamos que queremos listar los nombres de todos los ficheros de un directorio (incluyendo sus subdirectorios). El formato de nuestra salida consistirá en que los nombres de los ficheros de profundidad d aparecerán sangrados d tabuladores. En la Figura 17.5 se muestra un algoritmo simple para hacer esto. La salida de dicho algoritmo para el directorio de la Figura 17.4 se muestra en la Figura 17.6.

Por ahora usaremos pseudocódigo, pero al final de esta sección proporcionamos una implementación en Java. Suponemos la existencia de la clase `SistemaFichero` y de dos métodos, `imprimirNombre` y `esDirectorio`. El método `imprimirNombre` produce como salida el objeto `SistemaFichero` actual con una sangría de d tabuladores, y `esDirectorio` comprueba si el objeto `SistemaFichero` actual es un directorio, en cuyo caso devuelve `true`. Usando estos dos métodos se puede escribir fácilmente la rutina recursiva `listar`, a la cual tenemos que pasarle el parámetro `prof`, que indica el nivel actual del directo-

[1] Cada directorio en el sistema de ficheros de Unix también tiene una entrada (`.`) que hace referencia a sí mismo y otra (`..`) que hace referencia al padre del directorio. Esto introduce un ciclo, por lo que técnicamente, el sistema de ficheros de Unix no es un árbol sino que sólo tiene una estructura semejante a un árbol.

```
1 void listar( int prof = 0 ) // La profundidad es inicialmente 0
2 {
3     imprimirNombre( prof ); // Imprime el nombre del objeto
4     if ( esDirectorio( ) )
5         for cada fichero c en este directorio (para cada hijo)
6             c.listar( prof + 1 );
7 }
```

Figura 17.5 Pseudocódigo para listar el contenido de un directorio y sus subdirectorios en un sistema jerárquico de ficheros.

```
marca
        libros
                dsaa
                        ch1
                        ch2
                ecp
                        ch1
                        ch2
                ipps
                        ch1
                        ch2
        cursos
                cop3223
                        syl
                cop3530
                        syl
                .login
```

Figura 17.6 Listado del contenido del directorio representado por el árbol de la Figura 17.4.

rio relativo a la raíz, es decir, la profundidad a la que nos encontramos. En la llamada inicial a `listar`, el parámetro `prof` toma el valor 0, lo que significa ausencia de sangría al mostrar la raíz. Este parámetro de profundidad es una variable interna y difícilmente una rutina que llamara a `listar` tendría que conocer su valor, esto es, considerarlo como un parámetro. Por ello el pseudocódigo especifica para ella un valor por defecto de 0 (no es legal especificar un valor por defecto en Java).

Es bastante sencillo seguir la lógica del algoritmo: se imprime el objeto actual con el sangrado adecuado y si la entrada es un directorio, procesamos recursivamente todos sus hijos, uno a uno. Estos hijos son un nivel más profundos y por tanto se debe añadir al sangrado un tabulador adicional. Al efecto hacemos la llamada recursiva con `prof + 1`. Es difícil imaginar un fragmento de código más simple que éste para llevar a cabo lo que a priori parece una tarea bastante compleja.

Esta técnica algorítmica se conoce como *recorrido en preorden*, en el cual se lleva a cabo el trabajo sobre el nodo antes de procesar los hijos. Además de ser un algoritmo compacto, se trata de un algoritmo con coste en tiempo lineal, lo cual tiene su mérito. Por qué esto es en efecto así, se discutirá más adelante en el capítulo.

En un *recorrido en preorden*, se lleva a cabo el trabajo sobre el nodo antes de procesar los hijos. El recorrido consume un tiempo constante por nodo.

En un *recorrido en postorden*, se lleva a cabo el trabajo sobre el nodo despues de haber procesado sus hijos. Este recorrido también consume un tiempo constante por nodo.

Otro método habitual para recorrer un árbol es el *recorrido en postorden*, en el cual se lleva a cabo el trabajo sobre un nodo después de haber procesado sus hijos. Como ejemplo, la Figura 17.7 representa la misma estructura de directorio que la de la Figura 17.4. Los números entre paréntesis representan el número de bloques de disco que ocupa cada fichero. Puesto que los directorios son ficheros, también usan bloques de disco (para almacenar los nombres de sus hijos e información sobre ellos).

Supongamos que queremos calcular el número total de bloques que ocupan todos los ficheros en el árbol. La forma más natural de hacer esto es buscando el número total de bloques usados por todos sus hijos (los cuales pueden ser directorios que deben evaluarse recursivamente): `libros` (41), `cursos` (8) y `.login` (2). El número total de bloques es entonces el total en todos los hijos más los bloques usados en la raíz (1), es decir 52. La rutina `tamanyo` de la Figura 17.8 implementa esta estrategia. Si el objeto `SistemaFichero` actual no es un directorio, `tamanyo` simplemente devuelve el número de bloques que utiliza. En caso contrario, se añade el número de bloques del directorio actual al número de bloques encontrados (recursivamente) en todos los hijos. Para ilustrar la diferencia entre los recorridos en preorden y en postorden, en la Figura 17.9 se muestra cómo el algoritmo va produciendo el tamaño de cada directorio (o fichero). En ella vemos que se ha usado un recorrido en postorden clásico porque el tamaño total de un directorio no se conoce hasta que se conocen los de sus hijos. De nuevo el tiempo de ejecución es lineal. En la Sección 17.4 se puede encontrar mucha más información sobre el recorrido de árboles.

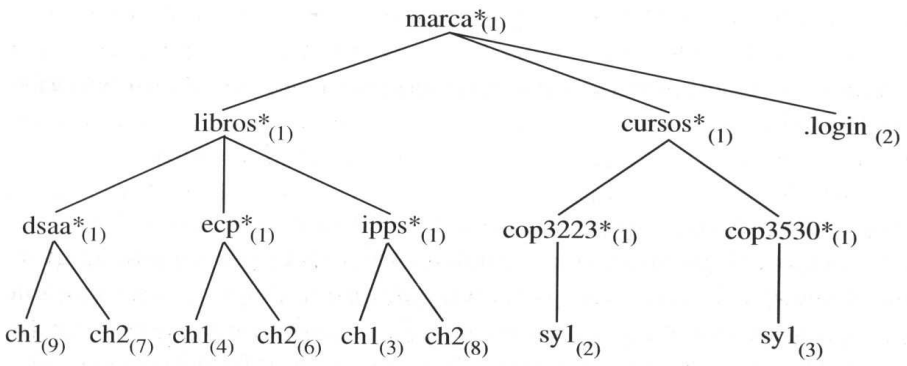


Figura 17.7 Directorio Unix con los tamaños de los ficheros.

```
1  int tamanyo( )
2  {
3      int tamtotal = sizeOfThisFile( );
4
5      if ( isDirectory( ) )
6          for cada fichero en este directorio (para cada hijo)
7              tamtotal += c.tamanyo( );
8
9      return tamtotal;
10 }
```

Figura 17.8 Pseudocódigo para calcular el tamaño total de todos los ficheros de un directorio.

```
                    ch1                 9
                    ch2                 7
              dsaa                     17
                    ch1                 4
                    ch2                 6
              ecp                      11
                    ch1                 3
                    ch2                 8
              ipps                     12
        libros                         41
                    sy1                 2
              cop3223                   3
                    sy1                 3
              cop3530                   4
        cursos                          8
        .login                          2
marca                                  52
```

Figura 17.9 Traza del método `tamanyo`.

Implementación en Java

Java proporciona una clase `File` contenida en el paquete `java.io` que se puede usar para recorrer jerarquías de directorios. Podemos usarla para implementar el pseudocódigo de la Figura 17.5, así como el método `tamanyo`, lo cual se hace en el código disponible en la red. La clase `File` proporciona varios métodos útiles.

Se puede construir un objeto `File` proporcionando un nombre. El método `getName` proporciona el nombre de un objeto `File`. No incluye la parte del camino correspondiente a los directorios, la cual se obtiene usando `getPath`. El método `isDirectory` devuelve `true` si el objeto `File` es un directorio, y `length` devuelve su tamaño en bytes. Si el fichero es un directorio, el método `list` devuelve un vector de valores de tipo `String` que representan los nombres de los ficheros en el directorio (sin incluir `.` ni `..`).

Para implementar el objeto `SistemaFichero` descrito en el pseudocódigo, simplemente extendemos la clase `File` y proporcionamos un constructor, `imprimirNombre` y `listar`, ya que el método `esDirectorio` que necesitábamos ya nos lo proporciona `File` mediante `isDirectory`. Esto se muestra en la Figura 17.10. La única parte difícil son las líneas 36 y 37, donde debemos construir el objeto `SistemaFichero hijo`. El nombre completo del fichero consiste en el nombre del directorio seguido por un separador (`/` en Unix; `\` en DOS) y seguido por el nombre del fichero. Se proporciona también una rutina `main` simple.

17.2 Árboles binarios

Un *árbol binario* es un árbol en el que ningún nodo puede tener más de dos hijos. Puesto que solamente hay dos hijos, podemos llamarles una vez distinguidos uno del otro `izquierdo` y `derecho`. La definición recursiva nos dice en esta ocasión que un árbol binario es o bien vacío o consta de una raíz, un árbol (hijo) izquierdo y otro derecho. Los hijos izquierdo y derecho pueden ser a su vez vacíos, de modo que un nodo con un solo hijo podría tener un hijo izquierdo o un hijo derecho.

Un árbol binario no contiene nodos con más de dos hijos.

```
1  import java.io.*;
2
3  public class SistemaFichero extends File
4  {
5          // Constructor
6      public SistemaFichero( String nombre )
7      {
8          super( nombre );
9      }
10
11         // Imprime el nombre del fichero con sangrado
12     public void imprimirNombre( int prof )
13     {
14         for( int i = 0; i < prof; i++ )
15             System.out.print( "\t" );
16         System.out.println( getName( ) );
17     }
18
19         // Método guía público para listar los ficheros
20     public void listar( )
21     {
22         listar( 0 );
23     }
24
25         // Método recursivo para listar los ficheros de un directorio
26     private void listar( int prof )
27     {
28         imprimirNombre( prof );
29
30         if( isDirectory( ) )
31         {
32             String [ ] hijos = list( );
33
34             for( int i = 0; i < hijos.length; i++ )
35             {
36                 SistemaFichero hijo = new SistemaFichero( getPath( )
37                                       + separatorChar + hijos[ i ] );
38                 hijo.listar( prof + 1 );
39             }
40         }
41     }
42
43         // Rutina main para listar los ficheros del directorio actual
44     public static void main( String [ ] args )
45     {
46         SistemaFichero f = new SistemaFichero( "." );
47         f.listar( );
48     }
49 }
```

Figura 17.10 Implementación en Java del listado de un directorio.

Usaremos repetidamente la definición recursiva en el diseño de algoritmos sobre árboles binarios. Éstos tienen un gran número de usos importantes, dos de los cuales se ilustran en la Figura 17.11.

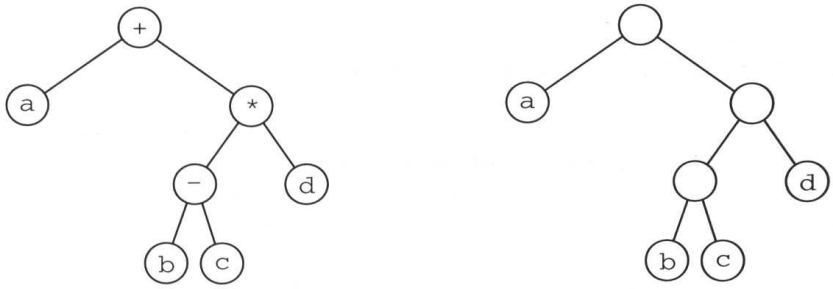


Figura 17.11 Usos de los árboles binarios: a la izquierda un árbol de expresión y a la derecha un árbol de codificación de Huffman.

Un uso de los árboles binarios lo constituyen los *árboles de expresión*, que son la estructura de datos fundamental en el diseño de compiladores. Las hojas de un árbol de expresión son operandos, como constantes o nombres de variables; el resto de los nodos contienen operadores. El árbol será binario en tanto y cuando todos los operadores implicados lo sean. Aunque éste es el caso más simple, es posible que los nodos tengan más de dos hijos (y en el caso de operadores unarios, solamente uno). Podemos evaluar un árbol de expresión T aplicando el operador de la raíz a los valores obtenidos a partir de la evaluación recursiva de sus árboles izquierdo y derecho. En nuestro caso, esto produce el valor de `(a+((b-c)*d))`. En la Sección 11.2 estudiamos la construcción y evaluación de los árboles de expresión.

Un ejemplo de uso de los árboles binarios viene dado por los árboles de expresión, que constituyen la estructura de datos central en el diseño de compiladores.

Un segundo uso de los árboles binarios es el *árbol de codificación de Huffman*, que se usa para implementar un algoritmo simple pero relativamente eficiente de compresión de datos. Cada símbolo del alfabeto se guarda en una hoja. Su código se obtiene siguiendo el camino desde la raíz hasta esa hoja, teniendo en cuenta que bajar por una arista izquierda se corresponde con un 0 y bajar por una derecha se corresponde con un 1. Luego `b` se codifica como 100. En la Sección 12.1 discutimos la construcción del árbol óptimo, que genera la mejor codificación posible.

Los árboles binarios también se utilizan como soporte en los árboles binarios de búsqueda (*véase* Capítulo 18), que permiten inserciones y acceso a los elementos en tiempo logarítmico, y en las colas con prioridad, que soportan acceso y eliminación del mínimo de una colección de elementos. Varias implementaciones eficientes de las colas con prioridad usan árboles, como se discute en los Capítulos 20 a 22.

Un uso importante de los árboles binarios es como soporte en otras estructuras de datos, como los árboles binarios de búsqueda y las colas con prioridad.

La Figura 17.12 proporciona el esqueleto de la clase `NodoBinario`. Las líneas 40 a 42 nos dicen que cada nodo consta de un dato junto con dos punteros a los hijos. Se proporcionan tres constructores. El de la línea 18 tiene cero parámetros. El de la línea 20 se usa para construir un `NodoBinario` dado un dato como parámetro. En estos dos constructores se asigna `null` a las referencias que apuntan a los hijos izquierdo y derecho. El tercer constructor, en la línea 22, inicializa todos los atributos del `NodoBinario`. Observe que estos constructores requieren inicialización o bien de los dos hijos o de ninguno de ellos, con lo que evitamos objetos parcialmente inicializados.

El acceso a los atributos se permite desde otras clases del mismo paquete, por lo que no proporcionamos métodos modificadores ni de acceso. El método `duplicar`, declarado en la línea 36, se usa para generar una copia del árbol cuya raíz es el nodo actual. Las rutinas `tamanyo` y `altura`, declaradas en las líneas 26

Muchas de las rutinas de la clase `NodoBinario` son recursivas. Los métodos de la clase `ArbolBinario` usan las rutinas de `NodoBinario` sobre la `raiz`.

```
1  package EstructurasDatos;
2  // Clase NodoBinario; almacena un nodo en un árbol
3  // CONSTRUCCIÓN: (a) si parámetros, (b) con un Object,
4  //      o (c) un Object, un puntero izquierdo y otro derecho.
5  //
6  // ********************OPERACIONES PÚBLICAS*********************
7  // int tamanyo( )             --> Devuelve el tamaño de un árbol
8  // int altura( )              --> Devuelve la altura de un árbol
9  // void imprimirPostOrden( ) --> Imprime recorrido en postorden
10 // void imprimirOrdenSim( )  --> Imprime recorrido orden simétrico
11 // void imprimirPreOrden( )  --> Imprime el recorrido en preorden
12 // NodoBinario duplicar( )   --> Devuelve una copia del árbol
13 // ************************ERRORES*****************************
14 // Ninguno
15
16 final class NodoBinario
17 {
18     NodoBinario( )
19       { this( null ); }
20     NodoBinario( Object elDato )
21       { this( elDato, null, null ); }
22     NodoBinario( Object elDato, NodoBinario menor,
23                                 NodoBinario mayor )
24       { dato = elDato; izquierdo = menor; derecho = mayor; }
25
26     static int tamanyo( NodoBinario a )
27       { /* Figura 17.19 */ }
28     static int altura( NodoBinario a )
29       { /* Figura 17.21 */ }
30     void imprimirPostOrden( )
31       { /* Figura 17.22 */ }
32     void imprimirOrdenSim( )
33       { /* Figura 17.22 */ }
34     void imprimirPreOrden( )
35       { /* Figura 17.22 */ }
36     NodoBinario duplicar( )
37       { /* Figura 17.17 */ }
38
39     // Atributos amistosos; accesibles por otras rutinas del paquete
40     Object      dato;
41     NodoBinario izquierdo;
42     NodoBinario derecho;
43 }
```

Figura 17.12 Esqueleto de la clase `NodoBinario`.

a 28, calculan las propiedades por ellas nombradas para el nodo referenciado por el parámetro `a`. Estas rutinas se implementarán en la Sección 17.3. También proporcionamos, en las líneas 30 a 34, rutinas que imprimen el contenido del subárbol que tiene como raíz el nodo actual, usando varias estrategias recursivas para recorrerlo. Las formas de recorrer un árbol se estudian en la Sección 17.4. ¿Por qué pasamos un parámetro a las rutinas `tamanyo` y `altura`, pero usamos el objeto actual para los recorridos y el método `duplicar`? En realidad, no hay una razón concreta para esto, es sólo una cuestión de estilo. Aquí presentamos ambos estilos. Las implementaciones mostrarán que las diferencias entre ellos surgen cuando se quiere comprobar si un árbol es `null`.

En esta sección describiremos la implementación de la clase `ArbolBinario`. Se proporciona una clase `NodoBinario` separada para simplificar la implementación de algunas de las rutinas recursivas. En la Figura 17.13 se muestra el esqueleto de la clase `ArbolBinario`. La mayoría de las rutinas son sencillas de implementar pues se limitan a llamar a métodos de la clase `NodoBinario`. En la línea 44 se declara el único atributo, una referencia al nodo raíz.

La clase `NodoBinario` se implementa de forma separada de la clase `ArbolBinario`. El único atributo de la clase `ArbolBinario` es una referencia al nodo raíz.

```
1  package EstructurasDatos;
2  // Clase ArbolBinario; almacena un árbol binario
3  // CONSTRUCCIÓN: (a) sin parámetros o (b) con un objeto a
4  //     colocar en la raíz de un árbol de un único elemento.
5  //
6  // ********************OPERACIONES PÚBLICAS*********************
7  // void imprimirPostOrden( ) --> Imprime recorrido en postorden
8  // void imprimirOrdenSim( )--> Imprime recorrido orden simétrico
9  // void imprimirPreOrden( )--> Imprime el recorrido en preorden
10 // boolean esVacio( )       --> true si el árbol es vacío
11 // void vaciar( )           --> Construye un árbol vacío
12 // void unir( Object raiz, ArbolBinario a1, ArbolBinario a2 )
13 //                          --> Construye un nuevo árbol
14 // ************************ERRORES*****************************
15 // Impresión de mensajes de error por uniones ilegales
16
17 public class ArbolBinario
18 {
19     public ArbolBinario( )
20       { raiz = null; }
21     public ArbolBinario( Object elemRaiz )
22       { raiz = new NodoBinario( elemRaiz ); }
23
24     public void imprimirPreOrden( )
25       { if( raiz != null ) raiz.imprimirPreOrden( ); }
26     public void imprimirOrdenSim( )
27       { if( raiz != null ) raiz.imprimirOrdenSim( ); }
28     public void imprimirPostOrden( )
29       { if( raiz != null ) raiz.imprimirPostOrden( ); }
30     public boolean esVacio( )
31       { return raiz == null; }
32     public void vaciar( )
33       { raiz = null; }
34     public void unir( Object elemRaiz, ArbolBinario a1,
35                                        ArbolBinario a2 )
36       { /* Figura 17.16 */ }
37     public int tamanyo( )
38       { return NodoBinario.tamanyo( raiz ); }
39     public int altura( )
40       { return NodoBinario.altura( raiz ); }
41     public void duplicar( ArbolBinario lder )
42       { if( this != null ) raiz = lder.raiz.duplicar( ); }
43
44     protected NodoBinario raiz;
45 }
```

Figura 17.13 Esqueleto de la clase `ArbolBinario`.

Se proporcionan dos constructores. El de la línea 19 crea un árbol vacío, mientras que el de la línea 21 crea un árbol con un único nodo. En las líneas 24 a 29 se declaran rutinas para recorrer el árbol, las cuales aplican métodos de la clase `NodoBinario` a la `raiz` después de verificar que el árbol no es vacío. Una estrategia alternativa de recorrido de árboles que se puede implementar es el *recorrido por niveles*. En la Sección 17.4 se estudian todas las rutinas de recorrido. En las líneas 32 y 30 se proporcionan rutinas para construir un árbol vacío y para comprobar si un árbol es vacío respectivamente.

Antes de que podamos aplicar el método de la clase `NodoBinario` al nodo apuntado por `raiz`, debemos verificar que `raiz` no es `null`.

Hay dos rutinas más en la clase. En las líneas 41 y 42 definimos el método `duplicar`. Después de comprobar el aliasing, llamamos al método `duplicar` de la clase `NodoBinario` para obtener una copia del árbol `lder`. Entonces asignamos el resultado a la raíz del árbol. Observe que antes de poder aplicar el método de la clase `NodoBinario` al nodo referenciado por `raiz`, debemos verificar que `raiz` no es `null`.

El último método de la clase es la rutina `unir`, la cual usa dos árboles —`a1` y `a2`— y un elemento para crear un nuevo árbol que tiene al elemento como raíz y los dos árboles como hijos izquierdo y derecho. En principio, se trata de una rutina de una sola línea:

```
raiz = new NodoBinario( elemRaiz, a1.raiz, a2.raiz );
```

`unir` sería en principio una rutina de una sola línea, pero hemos de vigilar el aliasing, asegurándonos de que un mismo nodo no aparece en dos subárboles, y comprobar los errores.

Si las cosas fueran siempre tan sencillas, los programadores no encontrarían trabajo. Afortunadamente para nuestras carreras profesionales, hay complicaciones. La Figura 17.14 muestra el resultado del sencillo método `unir` de una sola línea. Se hace patente un problema: los nodos de los árboles `a1` y `a2` se encuentran ahora en dos árboles, los originales y el resultado de la unión. Esta compartición podría ser un problema si quisiéramos eliminar o alterar de alguna otra forma los árboles originales o bien sus apariciones como subárboles, pues cualquier modificación en uno u otro repercutiría inmediatamente en el otro, pues se trata en realidad del mismo objeto. Este en ocasiones podría no ser el efecto pretendido.

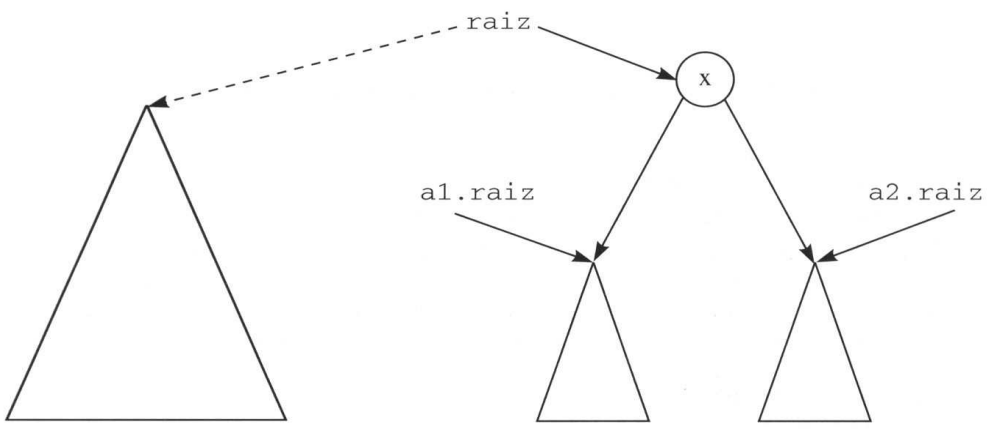


Figura 17.14 Resultado de un método `unir` ingenuo: hay compartición de subárboles.

La solución es en principio sencilla. Podemos asegurar que los nodos no aparecen en dos árboles asignando `null` a `a1.raiz` y `a2.raiz` después de `unir`. Surgen complicaciones cuando consideramos algunas llamadas que contienen aliasing:

Asignamos `null` a las raíces de los árboles originales para que cada nodo esté en un solo árbol.

```
a1.unir( x, a1, a2 );
a2.unir( x, a1, a2 );
a1.unir( x, a3, a3 );
```

Los dos primeros casos son similares, por lo que consideraremos sólo el primero. En la Figura 17.15 se muestra un dibujo de la situación. Puesto que `a1` es un alias del objeto actual, `a1.raiz` y `raiz` son alias. En consecuencia, tras la llamada a `new`, si ejecutamos `a1.raiz = null`, también estamos cambiando `raiz` a `null`. Por tanto, en estos casos hemos de tener mucho cuidado con los alias.

Si un árbol de entrada y el de salida son alias, debemos tener mucho cuidado para evitar que la `raiz` resultante sea la referencia `null`.

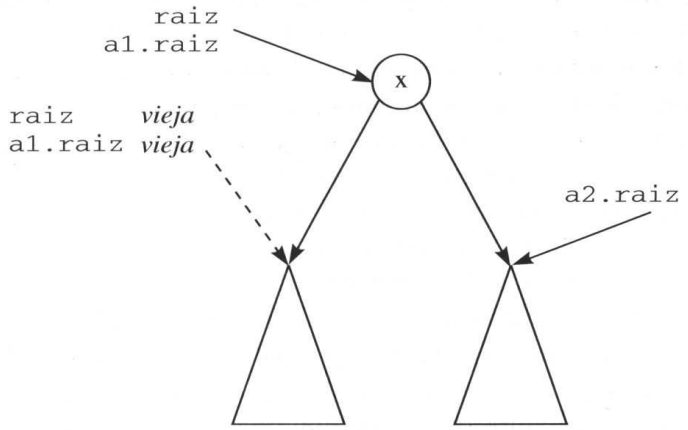


Figura 17.15 Problemas con el aliasing en la operación `unir`; `a1` es también el objeto actual.

```
1  /**
2   * Rutina unir para la clase ArbolBinario
3   * Forma un árbol nuevo a partir de elemRaiz, a1 y a2.
4   * Trata correctamente varias situaciones de aliasing.
5   */
6  public void unir( Object elemRaiz, ArbolBinario a1,
7                                     ArbolBinario a2 )
8  {
9      if( a1.raiz == a2.raiz && a1.raiz != null )
10     {
11         System.err.println( "arbolIzquierdo==arbolDerecho; " +
12                             "unir abortado" );
13         return;
14     }
15
16         // Creación de un nuevo nodo
17     raiz = new NodoBinario( elemRaiz, a1.raiz, a2.raiz );
18
19         // Nos aseguramos de que cada nodo está sólo en un árbol
20     if( this != a1 )
21         a1.raiz = null;
22     if( this != a2 )
23         a2.raiz = null;
24 }
```

Figura 17.16 Rutina unir para la clase `ArbolBinario`.

Si dos árboles de entrada son alias, no deberíamos permitir que se llevara a cabo la operación, a menos que los árboles estén vacíos.

El tercer caso no debería permitirse porque colocaría los nodos de `a3` en dos sitios distintos de `a1`. Ahora bien, si `a3` representara un árbol vacío, el tercer caso sí estaría permitido. Al final resultó bastante peor de lo que esperábamos. El código resultante se muestra en la Figura 17.16. Lo que en un principio era una rutina de una sola línea se ha convertido en una bastante más larga.

17.3 Árboles y recursión

Se usan rutinas recursivas para implementar `tamanyo` y `duplicar`.

Ya que los árboles se pueden definir de forma recursiva, no sorprende el hecho de que la mayoría de las rutinas que implican a los árboles se implementen de forma más sencilla usando recursión. Aquí proporcionamos la implementación recursiva de la mayoría de los métodos restantes de las clases `NodoBinario` y `ArbolBinario`. Las rutinas resultantes son increíblemente compactas.

Puesto que `duplicar` es un método de la clase `NodoBinario`, hacemos las llamadas recursivas sólo una vez se ha comprobado que los subárboles no son `null`.

Comenzamos con el método `duplicar` de la clase `NodoBinario`. Puesto que es un método de la clase `NodoBinario`, sabemos que el árbol que estamos duplicando no es vacío. El algoritmo recursivo es sencillo. Primero creamos un nodo nuevo con el mismo dato que el nodo raíz actual. Después llamamos recursivamente dos veces a `duplicar` para generar el subárbol izquierdo y el derecho. En ambos casos, hacemos las llamadas recursivas después de comprobar que el correspondiente árbol es no vacío. Esta descripción verbal aparece codificada en la Figura 17.17.

El método `tamanyo` se implementa fácilmente de forma recursiva si previamente hacemos un dibujo.

El siguiente método que presentamos es la rutina `tamanyo` de la clase `NodoBinario`, la cual devuelve el tamaño del árbol cuya raíz es el nodo referenciado por `a`, que se pasa como parámetro. Si dibujamos el árbol recursivamente, como se muestra en la Figura 17.18, vemos que el tamaño de un árbol es igual al tamaño del hijo izquierdo más el del hijo derecho más 1, ya que la raíz aporta un nodo. Como en toda rutina recursiva necesitamos un caso base que pueda resolverse directamente. El árbol más pequeño que la rutina `tamanyo` trata es el árbol vacío (correspondiente a `a null`), cuyo tamaño es claramente 0. El resultado está implementado en la Figura 17.19. Podemos comprobar fácilmente que la recursión produce la respuesta correcta para los árboles de tamaño 1.

```
1  /**
2   * Devuelve un puntero al nodo raíz del duplicado del
3   * árbol binario cuya raíz es el nodo actual.
4   */
5  NodoBinario duplicar( )
6  {
7      NodoBinario raiz = new NodoBinario( dato );
8      if( izquierdo != null ) // Si hay un árbol izquierdo
9          raiz.izquierdo = izquierdo.duplicar( ); // Duplicarlo
10     if( derecho != null )   // Si hay un árbol derecho
11         raiz.derecho = derecho.duplicar( );     // Duplicarlo
12     return raiz;            // Devolver el árbol resultante
13 }
```

Figura 17.17 Rutina para devolver una copia del árbol con raíz en el nodo actual.

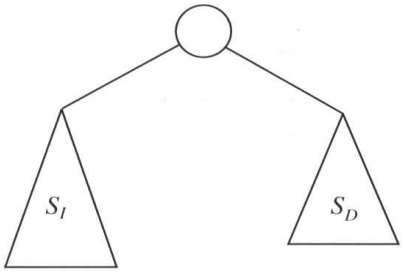


Figura 17.18 Visión recursiva utilizada para calcular el tamaño de un árbol: $S_A = S_I + S_D + 1$.

```
1  /**
2   * Devuelve el tamaño del árbol binario con raíz en a.
3   */
4  static int tamanyo( NodoBinario a )
5  {
6      if( a == null )
7          return 0;
8      else
9          return 1 + tamanyo( a.izquierdo ) + tamanyo( a.derecho );
10 }
```

Figura 17.19 Rutina para calcular el tamaño de un nodo.

La última rutina recursiva de esta sección calcula la altura de un nodo. Es difícil de escribir en forma no recursiva, pero se implementa trivialmente de forma recursiva, una vez nos fijamos en la descripción gráfica de un árbol. La Figura 17.20 muestra un árbol visto de forma recursiva. Supongamos que el hijo izquierdo tiene altura H_I y que el hijo derecho tiene altura H_D. Cualquier nodo que esté d niveles por debajo de la raíz del hijo izquierdo está $d + 1$ niveles por debajo de la raíz del árbol completo. Lo mismo sucede para el hijo derecho. Luego la longitud del camino que lleva al nodo más profundo en el árbol original es 1 más de la longitud de su camino desde la raíz del subárbol correspondiente. En consecuencia, si calculamos dicho valor para ambos subárboles, la respuesta que buscamos es el máximo de dichos valores más 1.

El método `altura` también es fácil de implementar de forma recursiva. La altura de un árbol vacío es −1.

¿Qué sucede con el caso base? De nuevo, corresponde al árbol vacío. La afirmación de que la altura de un árbol vacío es 0, es errónea, porque un árbol de un solo nodo tiene altura 0 (una hoja siempre tiene altura 0). Para que la fórmula recursiva funcione correctamente con árboles de un solo nodo, definimos la altura

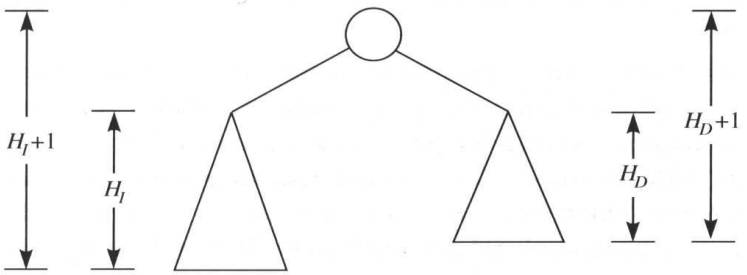


Figura 17.20 Visión recursiva del cálculo de la altura: $H_A = \text{Max}(H_I + 1, H_D + 1)$.

```
1  /**
2   * Devuelve la altura del árbol binario con raíz en a.
3   */
4  static int altura( NodoBinario a )
5  {
6      if( a == null )
7          return -1;
8      else
9          return 1 + Math.max( altura( a.izquierdo ),
10                              altura( a.derecho ) );
11 }
```

Figura 17.21 Rutina para calcular la altura de un nodo.

del árbol vacío como -1. Ahora, como una hoja tiene dos subárboles de altura -1, su altura queda correctamente definida como 0. La rutina resultante se muestra en la Figura 17.21.

17.4 Recorrido de árboles: clases iteradoras

En este capítulo hemos visto cómo se puede usar la recursión para implementar los métodos de los árboles binarios. Cuando se aplica la recursión, calculamos información no sólo sobre un nodo, sino también sobre todos sus descendientes. Decimos entonces que estamos *recorriendo el árbol*. Dos recorridos conocidos que ya hemos visto anteriormente son el recorrido en preorden y el recorrido en postorden.

En un recorrido en preorden, primero se procesa el nodo y después se procesan recursivamente sus hijos. La rutina `duplicar` es un ejemplo de recorrido en preorden, pues lo primero que hacemos es crear la raíz. Después se copian de forma recursiva los hijos izquierdo y derecho.

En un recorrido en postorden, el nodo dado se procesa después de haber procesado recursivamente sus hijos. Dos ejemplos de ello son los métodos `tamanyo` y `altura`. En ambos casos, sólo podemos calcular el valor correspondiente de un nodo dado (por ejemplo, su tamaño o su altura) después de haber calculado el correspondiente valor para sus hijos.

En un *recorrido en orden simétrico*, procesamos el nodo actual entre medias de las llamadas recursivas.

Un tercer recorrido recursivo habitual es el *recorrido en orden simétrico*, en el cual primero se procesa recursivamente el hijo izquierdo, luego se procesa el nodo actual, y finalmente se procesa recursivamente el hijo derecho. Este mecanismo se usa para generar la expresión algebraica correspondiente a un árbol de expresión. Por ejemplo, en la Figura 17.11, el recorrido en orden simétrico genera la expresión `(a+((b-c)*d))`.

Un recorrido simple usando cualquiera de estas estrategias requiere un tiempo lineal.

La Figura 17.22 ilustra rutinas que imprimen los nodos de un árbol binario usando cada uno de los algoritmos recursivos de recorrido de árboles. La Figura 17.23 muestra el orden en que se visitan los nodos para cada una de las tres estrategias. El tiempo de ejecución de todas ellas es lineal. Esto es así, pues en todos los casos cada nodo se imprime exactamente una vez, de modo que el número total de llamadas realizadas (incluyendo el trabajo constante debido a las operaciones de apilar y desapilar de la pila de ejecución interna) es una por nodo, es decir $O(N)$. Luego el tiempo total de ejecución es $O(N)$.

```
1  void imprimirPreOrden( )
2  {
3      System.out.println( dato );            // Nodo
4      if( izquierdo != null )
5          izquierdo.imprimirPreOrden( );     // Izquierdo
6      if( derecho != null )
7          derecho.imprimirPreOrden( );       // Derecho
8  }
9
10 void imprimirPostOrden( )
11 {
12     if( izquierdo != null )
13         izquierdo. imprimirPostOrden( );   // Izquierdo
14     if( derecho != null )
15         derecho. imprimirPostOrden( );     // Derecho
16     System.out.println( dato );            // Nodo
17 }
18
19 void imprimirOrdenSim ( )
20 {
21     if( izquierdo != null )
22         izquierdo. imprimirOrdenSim( );    // Izquierdo
23     System.out.println( dato );            // Nodo
24     if( derecho != null )
25         derecho. imprimirOrdenSim( );      // Derecho
26 }
```

Figura 17.22 Rutinas para imprimir los nodos en preorden, postorden y orden simétrico.

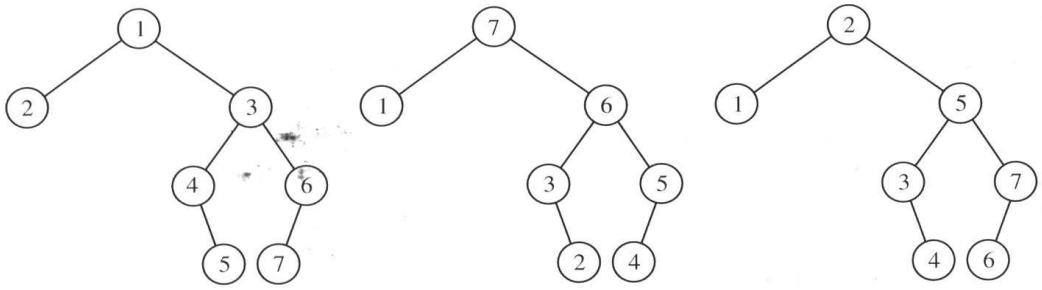


Figura 17.23 Rutas de acceso para los recorridos en preorden, postorden y orden simétrico.

Podemos recorrer un árbol de forma no recursiva, manteniendo la pila nosotros mismos.

¿Debemos necesariamente usar recursión para implementar los recorridos? Claramente la respuesta es no. Dado que, como ya se discutió en la Sección 7.3, la recursión se implementa internamente usando una pila. De modo que podríamos usar nuestra propia pila. El resultado es generalmente un programa algo más rápido, dado que, solamente tenemos que guardar en la pila lo que necesitamos en cada caso, en lugar de hacer que el compilador coloque un registro de activación completo. La diferencia en velocidad entre un algoritmo recursivo y otro no recursivo depende mucho de la plataforma. En la mayoría de los casos, la mejora en la velocidad no justifica el esfuerzo implicado en la eliminación de la recursión. Aun así, vale la pena saber cómo hacerlo por si acaso nuestra plataforma es de las que puede beneficiarse de la eliminación de la recursión, y también porque en ocasiones ver cómo se implementa un programa de forma no recursiva puede hacer que la recursión se entienda más claramente.

Una clase iteradora permite realizar un recorrido paso a paso.

Presentaremos tres clases iteradoras, con el mismo espíritu que en las listas enlazadas. Cada una de ellas nos permitirá ir al primer nodo, avanzar al siguiente nodo, comprobar si hemos llegado al último nodo y acceder al nodo actual. El orden en que se accede a los nodos está determinado por el tipo de recorrido. Implementaremos también un *recorrido por niveles*, en el que los nodos se visitan de arriba abajo y de izquierda a derecha. Este recorrido es inherentemente no recursivo y de hecho usa una cola en lugar de una pila. Hecha esta salvedad, resulta ser bastante parecido al recorrido en preorden.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase ArbolIter; mantiene la "posición actual"
6  //
7  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
8  //
9  // ********************OPERACIONES PÚBLICAS*********************
10 //     primero y avanzar son abstractos; los demás son finales
11 // boolean esValido( ) --> True si estamos en una posición válida
12 // Object recuperar( ) --> Devuelve elemento en posición actual
13 // void primero( )     --> Coloca posición actual en el primero
14 // void avanzar( )     --> Avanza (prefijo)
15 // ************************ERRORES*****************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 /**
19  * Clase iteradora de árboles.
20  */
21 abstract public class ArbolIter
22 {
23     public ArbolIter( ArbolBinario elArbol )
24     {
25         a = elArbol;
26         actual = null;
27     }
28
29     abstract public void primero( );
30
31     final public boolean esValido( )
32     {
33         return actual != null;
34     }
35
36     final public Object recuperar( ) throws ElementoNoEncontrado
37     {
38         if( actual == null )
39          throw new ElementoNoEncontrado( "recuperar de ArbolIter" );
40         return actual.dato;
41     }
42
43     abstract public void avanzar( ) throws ElementoNoEncontrado;
44
45     protected ArbolBinario a;          // El árbol
46     protected NodoBinario actual;      // Posición actual
47 }
```

Figura 17.24 Clase iteradora abstracta para los árboles.

La Figura 17.24 proporciona una clase abstracta para la iteración sobre los árboles. Cada iterador almacenará una referencia al árbol y un puntero al nodo actual, los cuales se declaran respectivamente en las líneas 45 y 46, inicializándose en el constructor. Los dos son declarados como `protected` para permitir acceder a ellos a las clases derivadas. En las líneas 29 a 43 se declaran varios métodos. Los métodos `esValido` y `recuperar` son invariantes en la jerarquía por lo que se proporciona una implementación y se declaran como `final`. Por su parte, cada tipo de iterador debe proporcionar los métodos abstractos `primero` y `avanzar`. Observe que el método `avanzar` difiere del de las listas enlazadas porque lanza una excepción. Se ha hecho así para ilustrar distintas estrategias.

La clase iteradora abstracta contiene un subconjunto de los métodos de la clase iteradora para las listas enlazadas. Cada tipo de recorrido está representado por una clase derivada.

17.4.1 Recorrido en postorden

El recorrido en postorden se implementa usando una pila para almacenar el estado actual. La cima de la pila representará el nodo que estamos visitando en un cierto instante en el recorrido en postorden. Pero podemos encontrarnos en tres situaciones distintas:

1. A punto de hacer una llamada recursiva al hijo izquierdo.
2. A punto de hacer una llamada recursiva al hijo derecho.
3. A punto de procesar el nodo actual.

El recorrido en postorden mantiene una pila que almacena los nodos ya visitados que aún no se han tratado completamente.

En consecuencia, en el curso del recorrido cada nodo se colocará en la pila tres veces. Cuando se desapila un nodo de la pila por tercera vez, podemos marcarlo como el nodo actual a ser visitado.

En caso contrario, el nodo se está desapilando por primera o segunda vez, de modo que aún no está listo para ser visitado, por lo que volvemos a apilarlo en la pila y simulamos una llamada recursiva. Si el nodo está siendo desapilado por primera vez, necesitamos apilar el hijo izquierdo en la pila, si es no vacío. Si está siendo desapilado por segunda vez apilamos el hijo derecho, si es no vacío. En todo caso, desapilamos después, pasando a aplicar el mismo test. Observe que cada vez que desapilamos, estamos simulando la llamada recursiva al hijo apropiado. Si el hijo era vacío y por tanto nunca fue apilado en la pila, entonces cuando desapilemos, estaremos desapilando de nuevo el nodo original.

Cada nodo se coloca en la pila tres veces. La tercera vez que lo desapilemos se considera que el nodo ha sido visitado. En las otras ocasiones simulamos una llamada recursiva.

Eventualmente, o bien el proceso desapila un nodo por tercera vez, o bien la pila queda vacía. En este último caso, hemos recorrido ya todo el árbol. Inicializamos el algoritmo apilando una referencia a la raíz. Un ejemplo de cómo se manipula la pila se muestra en la Figura 17.25.

Cuando la pila queda vacía, es que ya se han visitado todos los nodos.

Un resumen rápido: la pila contiene los nodos que ya hemos recorrido pero que aún no hemos tratado completamente. Cuando un nodo se apila en la pila, el contador es respectivamente 0, 1 o 2 en cada una de las siguientes situaciones:

1. Si estamos a punto de procesar el hijo izquierdo.
2. Si estamos a punto de procesar el hijo derecho.
3. Si estamos a punto de procesar el nodo.

Sigamos la traza de lo que sucede en el recorrido en postorden. Inicializamos el recorrido apilando la raíz `a` en la pila, e inmediatamente se desapila `a` por primera vez, por lo que vuelve a colocarse en la pila y apilamos su hijo izquierdo `b`.

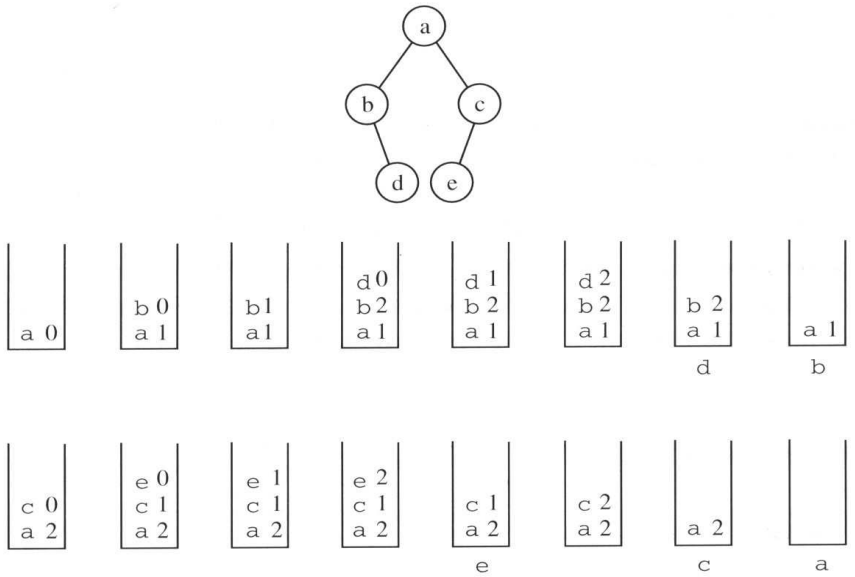


Figura 17.25 Estados de la pila durante el recorrido en postorden.

A continuación se desapila `b` por primera vez, por lo que se coloca de vuelta en la pila. Si existiera el hijo izquierdo de `b` se apilaría, pero como no tiene, se desapila `b` por segunda vez, se vuelve a colocar en la pila y se apila su hijo derecho `d`. Ahora se desapila `d` por primera vez, por lo que se coloca de nuevo en la pila. No se produce una operación de apilar, porque `d` no tiene hijo izquierdo, con lo que `d` se desapila por segunda vez y vuelve a la pila de nuevo. Como tampoco tiene hijo derecho, no se apila nada, sino que `d` se desapila por tercera vez, siendo marcado como nodo visitado. A continuación se desapila `b` por tercera vez, marcándose también como visitado.

Después se desapila `a` por segunda vez, colocándose de nuevo en la pila junto con su hijo derecho `c`. Se desapila `c` por primera vez, por lo que vuelve de nuevo a la pila, apilándose su hijo izquierdo `e`. A continuación `e` es desapilado, apilado, desapilado de nuevo y vuelto a apilar, y finalmente se desapila por tercera vez (esto sucede para todas las hojas). Luego se marca a `e` como nodo visitado. Después se desapila `c` por segunda vez, vuelve a la pila y como no tiene hijo derecho, es inmediatamente desapilado por tercera vez y marcado como visitado. Finalmente se desapila `a` por tercera vez y se marca como visitado. En este momento la pila está vacía, concluyendo el recorrido.

Cada `NodoPl` almacena una referencia a un nodo junto con un contador que nos indica cuantas veces se ha desapilado ya dicho nodo.

La clase `PostOrden`, que se muestra en la Figura 17.26, se implementa directamente usando el algoritmo descrito previamente. La clase `NodoPl` representa los objetos que se colocan en la pila. Sus objetos contienen una referencia a un nodo junto con un entero que almacena el número de veces que se ha desapilado el nodo en cuestión. Los objetos `NodoPl` se inicializan siempre reflejando el hecho de que aún no han sido desapilados.

La clase `PostOrden` se deriva de la clase `ArbolIter`. Añade una pila interna a los atributos heredados. La clase `PostOrden` se inicializa inicializando primero los atributos de `ArbolIter` para después apilar la raíz en la pila. Esto se ilustra en el constructor, en las líneas 23 a 28. El método `primero` se implementa limpiando la pila, apilando la raíz y llamando a `avanzar`.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase PostOrden; mantiene una "posición actual"
6  //      de acuerdo con un recorrido en postorden
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ***********************OPERACIONES PÚBLICAS*********************
11 // boolean esValido( )    --> True en una posición válida del árbol
12 // Object recuperar( )    --> Devuelve elemento en posición actual
13 // void primero( )        --> Coloca posición actual en primer nodo
14 // void avanzar( )        --> Avanza a la siguiente posición
15 // ****************************ERRORES****************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 /**
19  * Clase iteradora PostOrden.
20  */
21 public class PostOrden extends ArbolIter
22 {
23     public PostOrden( ArbolBinario elArbol )
24     {
25         super( elArbol );
26         p = new PilaVec( );
27         p.apilar( new NodoPl( a.raiz ) );
28     }
29
30     public void primero( )
31     {
32         p.vaciar( );
33         if( a.raiz != null )
34             p.apilar( new NodoPl( a.raiz ) );
35         try
36           { avanzar( ); }
37         catch( ElementoNoEncontrado e ) { } // Arbol vacío
38     }
39
40     protected Pila p;       // Pila de objetos NodoPl
41 }
42
43 class NodoPl
44 {
45     NodoBinario nodo;
46     int vecesDesap;
47
48     NodoPl( NodoBinario n )
49       { nodo = n; vecesDesap = 0; }
50 }
```

Figura 17.26 Clase `PostOrden` (clase completa exceptuando `avanzar`).

La Figura 17.27 implementa `avanzar`. Sigue la descripción anterior casi al pie de la letra. La línea 8 comprueba si la pila está vacía. Si lo está, ya hemos completado la iteración y podemos asignar `null` a `actual`, terminando después. Si no, llevamos a cabo repetidos apilados y desapilados hasta que se desapila un

`avanzar` es la única rutina complicada. Su código sigue la descripción anterior casi al pie de la letra.

```
1  /**
2   * Avanza la posición actual al siguiente nodo en el árbol,
3   *    de acuerdo con el esquema de recorrido en postorden.
4   * @exception ElementoNoEncontrado si la posición actual es null.
5   */
6  public void avanzar( ) throws ElementoNoEncontrado
7  {
8      if( p.esVacia( ) )
9      {
10         if( actual == null )
11           throw new ElementoNoEncontrado( "Avanzar de PostOrden" );
12         actual = null;
13         return;
14     }
15
16     NodoPl cnodo;
17
18     for( ; ; )
19     {
20         try
21           { cnodo = ( NodoPl ) p.cimaYDesapilar( ); }
22         catch( DesbordamientoInferior e )
23           { return; } // No puede ocurrir
24
25         if( ++cnodo.vecesDesap == 3 )
26         {
27             actual = cnodo.nodo;
28             return;
29         }
30
31         p.apilar( cnodo );
32         if( cnodo.vecesDesap == 1 )
33         {
34             if( cnodo.nodo.izquierdo != null )
35                 p.apilar( new NodoPl( cnodo.nodo.izquierdo ) );
36         }
37         else // cnodo.vecesDesap == 2
38         {
39             if( cnodo.nodo.derecho != null )
40                 p.apilar( new NodoPl( cnodo.nodo.derecho ) );
41         }
42     }
43 }
```

Figura 17.27 Método `avanzar` para la clase iteradora `PostOrden`.

elemento por tercera vez. Cuando esto sucede, el test de la línea 25 tiene éxito y podemos terminar. En caso contrario, en la línea 31 volvemos a apilar el elemento en la pila (observe que la componente `vecesDesap` ya ha sido incrementada en la línea 25). Entonces simulamos la llamada recursiva. Si el nodo ha sido desapilado por primera vez y tiene un hijo izquierdo, entonces se apila su hijo izquierdo. Si había sido desapilado por segunda vez y tiene un hijo derecho, se apila su hijo derecho. Observe que en todos los casos la construcción del objeto `NodoPl` implica que el nuevo nodo apilado se ha desapilado 0 veces.

Eventualmente el bucle `for` terminará al haberse desapilado algún nodo por tercera vez. Observe que a lo largo de la secuencia completa de iteraciones, puede haber a lo sumo $3N$ operaciones de apilar y desapilar. Ésta es una forma alternativa de establecer la linealidad de los recorridos en postorden.

17.4.2 Recorrido en orden simétrico

El recorrido en orden simétrico es análogo al recorrido en postorden excepto por el hecho de que un nodo se declara visitado cuando se desapila por segunda vez. Antes de concluir con el nodo, el iterador apila su hijo derecho (si existe) en la pila, de

El recorrido en orden simétrico es similar al recorrido en postorden, excepto por el hecho de que cuando se desapila un nodo por segunda vez, se declara ya visitado.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase OrdenSim; mantiene una "posición actual"
6  //     de acuerdo con el recorrido en orden simétrico
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ********************OPERACIONES PÚBLICAS*********************
11 // Las mismas que ArbolIter
12 // ***************************ERRORES*************************
13 // Excepciones lanzadas por accesos o avances ilegales
14
15 public class OrdenSim extends PostOrden
16 {
17     public OrdenSim( ArbolBinario elArbol )
18       { super( elArbol ); }
19
20     public void avanzar( ) throws ElementoNoEncontrado
21     {
22         if( p.esVacia( ) )
23         {
24             if( actual == null )
25                 throw new ElementoNoEncontrado( "avanzar de OrdenSim" );
26             actual = null;
27             return;
28         }
29
30         NodoPl cnodo;
31         for( ; ; )
32         {
33             try
34               { cnodo = ( NodoPl ) p.cimaYDesapilar( ); }
35             catch( DesbordamientoInferior e )
36               { return; }   // No puede suceder
37
38             if( ++cnodo.vecesDesap == 2 )
39             {
40                 actual = cnodo.nodo;
41                 if( cnodo.nodo.derecho != null )
42                     p.apilar( new NodoPl( cnodo.nodo.derecho ) );
43                 return;
44             }
45                 // Se procesa primero
46             p.apilar( cnodo );
47             if( cnodo.nodo.izquierdo != null )
48                 p.apilar( new NodoPl( cnodo.nodo.izquierdo ) );
49         }
50     }
51 }
```

Figura 17.28 Clase iteradora `OrdenSim` completa.

forma que la siguiente llamada a `avanzar` pueda continuar recorriéndolo. Como es muy similar al recorrido en postorden, derivamos la clase `OrdenSim` a partir de la clase `PostOrden` (aun a pesar de que no exista una relación *ES-UN*(*A*)). El único cambio es una pequeña alteración en `avanzar`. La nueva clase se muestra en la Figura 17.28.

17.4.3 Recorrido en preorden

El recorrido en preorden es análogo al recorrido en postorden, excepto por el hecho de que un nodo se declara visitado la primera vez que se desapila, apilándose entonces los hijos derecho e izquierdo.

El recorrido en preorden es análogo al recorrido en postorden, excepto por el hecho de que un nodo se declara visitado la primera vez que se desapila. Al hacerlo, el iterador apila el hijo derecho y luego el hijo izquierdo. Es importante que nos fijemos en el orden de ambas operaciones: queremos procesar primero el hijo izquierdo y después el derecho, por lo que apilamos primero el derecho y luego el izquierdo.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase Preorden; mantiene una "posición actual"
6  //    de acuerdo con el recorrido en preorden
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ********************OPERACIONES PÚBLICAS*********************
11 // boolean esValido( ) --> True en una posición válida del árbol
12 // Object recuperar( )-> Devuelve elemento en posición actual
13 // void primero( )   --> Coloca posición actual en primer nodo
14 // void avanzar( )   --> Avanza a la siguiente posición
15 // ***************************ERRORES*************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 public class PreOrden extends ArbolIter
19 {
20     public PreOrden( ArbolBinario elArbol )
21     {
22         super( elArbol );
23         p = new PilaVec( );
24         p.apilar( a.raiz );
25     }
26
27     public void primero( )
28     {
29         p.vaciar( );
30         if( a.raiz != null )
31             p.apilar( a.raiz );
32         try
33           { avanzar( ); }
34         catch( ElementoNoEncontrado e ) { } // Árbol vacío
35     }
36
37     public void avanzar( ) throws ElementoNoEncontrado
38       { /* Figura 17.30 */ }
39
40     private Pila p; // Pila de objetos NodoBinario
41 }
```

Figura 17.29 Clase `PreOrden` con todos los métodos excepto `avanzar`.

```
1  /**
2   * Avanza la posición actual al siguiente nodo en el árbol,
3   * de acuerdo con el esquema de recorrido en preorden.
4   * @exception ElementoNoEncontrado si la posición actual es null.
5   */
6  public void avanzar( ) throws ElementoNoEncontrado
7  {
8      if( p.esVacia( ) )
9      {
10         if( actual == null )
11             throw new ElementoNoEncontrado( "Avanzar de PreOrden" );
12         actual = null;
13         return;
14     }
15
16     try
17       { actual = ( NodoBinario ) p.cimaYDesapilar( ); }
18     catch( DesbordamientoInferior e )
19       { return; } // No puede suceder
20
21     if( actual.derecho != null )
22         p.apilar( actual.derecho );
23     if( actual.izquierdo != null )
24         p.apilar( actual.izquierdo );
25 }
```

Figura 17.30 Método `avanzar` de la clase iteradora `PreOrden`.

Podríamos derivar la clase `PreOrden` a partir de la clase `OrdenSim` o de la clase `PostOrden`, pero esto supondría un coste innecesario pues en esta ocasión ya no es necesario mantener un contador del número de veces que se ha desapilado cada elemento. En consecuencia, la clase `PreOrden` se deriva directamente de la clase `ArbolIter`. En la Figura 17.29 se muestra el esqueleto resultante con el constructor y el método `primero`.

Tener que desapilar una única vez permite algunas simplificaciones.

En la línea 40 se añade a la colección de atributos de la clase `ArbolIter` una pila de referencias a nodos de árbol. El constructor y el método `primero` son similares a los que ya hemos visto. Como se ilustra en la Figura 17.30, el método `avanzar` es más simple: ya no necesitamos más un bucle `for`, pues tan pronto como en la línea 17 se desapila un nodo, éste se convierte en el nodo actual. En este momento apilamos el hijo derecho y despues el izquierdo, cuando uno u otro exista.

17.4.4 Recorrido por niveles

En un *recorrido por niveles*, los nodos se visitan de forma descendente y de izquierda a derecha. Este recorrido se implementa mediante una cola. Se corresponde con *una búsqueda en anchura*.

Concluimos implementando el *recorrido por niveles*. Este recorrido procesa los nodos comenzando en la raíz y avanzando de forma descendente y de izquierda a derecha. El nombre se deriva del hecho de que primero visitamos los nodos del nivel 0 (la raíz), después los del nivel 1 (los hijos de la raíz), los del nivel 2 (los nietos de la raíz), y así sucesivamente. Un recorrido por niveles se implementa usando una cola en lugar de una pila. La cola almacena los nodos que van a ser visitados. Cuando se visita un nodo, se colocan sus hijos al final de la cola, donde serán visitados después de los nodos que ya están en la cola. Es fácil ver que esto garantiza que los nodos se visitan por niveles. La clase `PorNiveles` se muestra

en las Figuras 17.31 y 17.32 y se asemeja mucho a la clase `PreOrden`. Las únicas diferencias estriban en que usamos una cola en lugar de una pila y que añadimos a la cola el hijo izquierdo y luego el derecho, en lugar de hacerlo al revés. Observe que la cola puede ser muy larga. En el peor de los casos, todos los nodos del último nivel (hasta $N/2$) podrían estar simultáneamente en la cola.

El recorrido por niveles implementa una técnica más general denominada *búsqueda en anchura*. En la Sección 14.2 se presenta un ejemplo de utilización de esta técnica en un marco más general.

```
1  package EstructurasDatos;
2
3  import Excepciones.*;
4
5  // Clase PorNiveles; mantiene una "posición actual"
6  //     de acuerdo con un recorrido por niveles
7  //
8  // CONSTRUCCIÓN: con el árbol al que está ligado el iterador
9  //
10 // ********************OPERACIONES PÚBLICAS********************
11 // boolean esValido( ) --> True en una posición válida del árbol
12 // Object recuperar( ) --> Devuelve elemento en posición actual
13 // void primero( )     --> Coloca posición actual en primer nodo
14 // void avanzar( )     --> Avanza a la siguiente posición
15 // ********************ERRORES*********************************
16 // Excepciones lanzadas por accesos o avances ilegales
17
18 /**
19  * Clase iteradora PorNiveles.
20  */
21 public class PorNiveles extends ArbolIter
22 {
23     public PorNiveles( ArbolBinario elArbol )
24     {
25         super( elArbol );
26         c = new ColaVec( );
27         c.insertar( a.raiz );
28     }
29
30     public void primero( )
31     {
32         c.vaciar( );
33         if( a.raiz != null )
34             c.insertar( a.raiz );
35         try
36           { avanzar( ); }
37         catch( ElementoNoEncontrado e ) { } // Árbol vacío
38     }
39
40     public void avanzar( ) throws ElementoNoEncontrado
41       { /* Figura 17.32 */ }
42
43     private Cola c; // Cola de objetos NodoBinario
44 }
```

Figura 17.31 Clase iteradora `PorNiveles` con la mayoría de sus métodos.

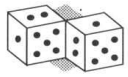
```
1  /**
2   * Avanza la posición actual al siguiente nodo en el árbol,
3   * de acuerdo con el esquema de recorrido por niveles.
4   * @exception ElementoNoEncontrado si la posición actual es null.
5   */
6  public void avanzar( ) throws ElementoNoEncontrado
7  {
8      if( c.esVacia( ) )
9      {
10         if( actual == null )
11           throw new ElementoNoEncontrado( "avanzar de PorNiveles" );
12         actual = null;
13         return;
14     }
15
16     try
17       { actual = ( NodoBinario ) c.quitarPrimero( ); }
18     catch( DesbordamientoInferior E )
19       { return; } // No puede suceder
20
21     if( actual.izquierdo != null )
22         c.insertar( actual.izquierdo );
23     if( actual.derecho != null )
24         c.insertar( actual.derecho );
25 }
```

Figura 17.32 Método `avanzar` para la clase iteradora `PorNiveles`.

Resumen

En este capítulo hemos estudiado los *árboles* y, en particular, los *árboles binarios*. Hemos visto cómo se pueden usar los árboles para implementar los sistemas de ficheros de muchos computadores y también algunas otras aplicaciones, como los árboles de expresión y de codificación, que se estudian con más detalle en la Parte III. Los algoritmos que trabajan sobre árboles hacen un uso extensivo de la recursión. Hemos examinado tres algoritmos recursivos de recorrido de árboles —reorden, postorden y orden simétrico— y hemos visto cómo se pueden implementar de forma no recursiva. También hemos examinado el recorrido por niveles, que forma la base de una importante técnica de búsqueda llamada búsqueda en anchura. En el siguiente capítulo estudiaremos otro tipo fundamental de árbol: el *árbol binario de búsqueda*.

Elementos del juego



altura Longitud del camino que va desde un nodo a la hoja más profunda por debajo de él.

árbol Definido no recursivamente, un árbol es un conjunto de nodos y otro de aristas orientadas que los conectan entre sí. Los árboles se definen de una forma natural de manera recursiva diciendo que un árbol es o bien vacío, o bien consta de una raíz y cero o más subárboles.

árbol binario Árbol en el que ningún nodo puede tener más de dos hijos. Se puede definir recursivamente de manera muy conveniente.

ascendiente y **descendiente** Si hay un camino de un nodo u a otro v, entonces u es ascendiente de v y v es descendiente de u.

ascendiente propio y **descendiente propio** En un camino de un nodo u a otro v, si $u \neq v$, entonces u es un ascendiente propio de v y v es un descendiente propio de u.

hermanos Nodos con el mismo padre.

hoja Nodo de un árbol que no tiene hijos.

método primer hijo/siguiente hermano Implementación de un árbol general en la que cada nodo guarda dos referencias por elemento: una al hijo más a la izquierda (si no es una hoja) y otra al hermano situado a su derecha.

padre e **hijo** Padres e hijos se definen de forma natural. Una arista orientada conecta al padre con cada hijo.

profundidad Longitud del camino de la raíz hasta el nodo.

recorrido en orden simétrico Tipo de recorrido en el que se procesa el nodo actual entre las llamadas recursivas para sus hijos.

recorrido en postorden Tipo de recorrido en el que se visita el nodo después de haber procesado los hijos. Tarda un tiempo constante por nodo.

recorrido en preorden Tipo de recorrido en el que se visita cada nodo antes de procesar sus hijos. El recorrido requiere un tiempo constante por nodo.

recorrido por niveles Tipo de recorrido en el que los nodos se vistan de forma descendiente y de izquierda a derecha. Se implementa usando una cola. Corresponde a una búsqueda en anchura.

tamaño de un nodo Número de descendientes que tiene un nodo (incluyéndole a él mismo).

Errores comunes

1. Permitir que un nodo pertenezca simultáneamente a dos árboles es generalmente una idea poco afortunada pues al realizar cambios en un subárbol se podrían provocar, sin desearlo, cambios simultáneos en varios subárboles.
2. Olvidar la comprobación de que un árbol es vacío es un error común. Si esto sucede en el marco de un algoritmo recursivo, probablemente el programa abortará.
3. Un error común cuando se trabaja con árboles consiste en pensar iterativamente en lugar de hacerlo recursivamente. Diseñe primero los algoritmos de forma recursiva, y sólo después conviértalos a iterativos, de resultar conveniente.

En Internet

Muchos de los ejemplos discutidos en este capítulo se utilizan en el Capítulo 18, en el que se estudian los árboles binarios de búsqueda. El código para las clases iteradoras es parte del paquete `EstructurasDatos` y debe guardarse en el directorio **EstructurasDatos**. Las versiones originales se encuentran en el directorio **DataStructures**. Estos iteradores usan clases `NodoBinario` y `ArbolBinario` ligeramente dife-

rentes de las utilizadas en este capítulo. Esas clases también deben guardarse en el directorio **EstructurasDatos**. Las versiones originales se encuentran en el directorio **DataStructures**.

Las versiones originales de las clases `NodoBinario`, `ArbolBinario` y `SistemaFichero` de este capítulo se encuentran en el directorio **Chapter17**.

BinaryNode.java	Contiene la clase traducida por `NodoBinario` en la Figura 17.12.
BinaryTree.java	Contiene la clase traducida por `ArbolBinario` en la Figura 17.13.
FileSystem.java	Implementa el recorrido de directorios, traducido en la Figura 17.10 por `SistemaFichero`.
InOrder.java	Contiene la clase traducida por `OrdenSim`.
LevelOrder.java	Contiene la clase traducida por `PorNiveles`.
PostOrder.java	Contiene la clase traducida por `PostOrden`.
PreOrder.java	Contiene la clase traducida por `PreOrden`.
TreeIterator.java	Contiene la clase iteradora traducida por `ArbolIter`.

Ejercicios

Cuestiones breves

17.1. Para el árbol de la Figura 17.33, determine lo siguiente:

a) ¿Cúal es el nodo raíz?
b) ¿Qué nodos son las hojas?
c) ¿Cuál es la profundidad del árbol?
d) El resultado de los recorridos en preorden, postorden, orden simétrico y por niveles.

17.2. Para cada nodo del árbol de la Figura 17.33, haga lo siguiente:

a) Nombre el nodo padre.
b) Liste los hijos.
c) Liste los hermanos.
d) Calcule la altura.
e) Calcule la profundidad.
f) Calcule el tamaño.

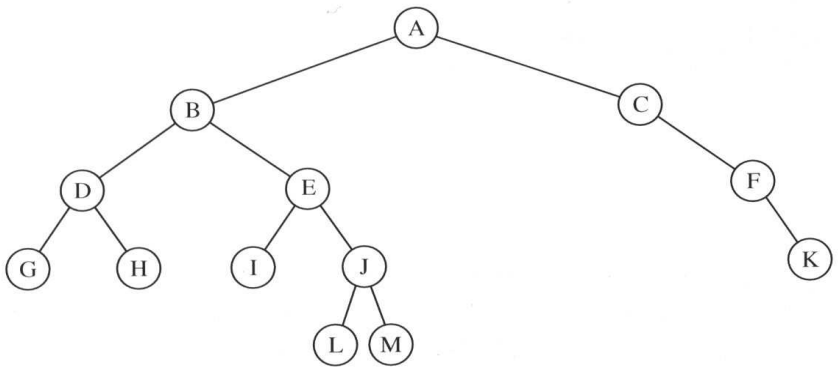


Figura 17.33 Árbol para los ejercicios 17.1 y 17.2.

17.3. ¿Cuál es la salida del método de la Figura 17.34 para el árbol de la Figura 17.25?

17.4. Muestre las operaciones realizadas sobre la pila al recorrer en orden simétrico y en preorden el árbol de la Figura 17.25.

Problemas teóricos

17.5. Muestre que el número máximo de nodos en un árbol binario de altura H es $2^{H+1} - 1$.

17.6. Un nodo completo es un nodo con dos hijos. Demuestre que en un árbol binario, el número de hojas es igual al número de nodos completos más 1.

17.7. ¿Cuántos hijos iguales a `null` hay en un árbol binario de N nodos? ¿Cuántos hay en un árbol M-ario de N nodos?

17.8. Consideremos un árbol binario con hojas l_1, l_2, ..., l_M a profundidades d_1, d_2, ..., d_M, respectivamente. Demuestre que $\Sigma_{i=1}^{M} 2^{-d_i} \leqslant 1$ (llamada *desigualdad de Kraft*) y determine cuando se da la igualdad.

Problemas prácticos

17.9. Escriba métodos eficientes (proporcione las medidas O) que dada una referencia a un árbol binario T calculen lo siguiente:

a) El número de hojas de T.
b) El número de nodos de T que contienen exactamente un hijo distinto de `null`.
c) El número de nodos de T que contienen exactamente dos hijos distintos de `null`.

17.10. Implemente algunas de las rutinas recursivas incluyendo comprobaciones que garanticen que no se realiza ninguna llamada recursiva sobre un subárbol `null`. Compare el tiempo de ejecución con rutinas idénticas que retrasen el test hasta la primera línea del método recursivo.

17.11. Rescriba la clase iteradora de modo que se lance una excepción cuando se aplica `primero` a un árbol vacío. ¿Por qué puede ser esto una idea poco afortunada?

```
1  public void imprimirMisterioso( NodoBinario a )
2  {
3      if( a != null )
4      {
5          System.out.println( a.dato );
6          imprimirMisterioso( a.izquierdo );
7          System.out.println( a.dato );
8          imprimirMisterioso( a.derecho );
9          System.out.println( a.dato );
10     }
11 }
```

Figura 17.34 Programa misterioso para el Ejercicio 17.3.

Prácticas de programación

17.12. Implemente un comando que liste todos los ficheros de un directorio dado, incluyendo todos los subdirectorios, ordenados por tamaño decreciente. Incluya los tamaños y cualquier otra información que pueda obtener de los ficheros.

17.13. Mediante un programa se puede generar automáticamente un árbol binario para autoedición. Esto se podría hacer asignando una coordenada (x, y) a cada nodo del árbol, dibujando un círculo alrededor de dicha coordenada y conectando cada nodo distinto de la raíz con su padre. Supongamos que disponemos de un árbol binario almacenado en memoria y que cada nodo tiene dos atributos extra en los que almacenar sus coordenadas. Supongamos que (0, 0) se corresponde con la esquina superior izquierda. Se pide hacer lo siguiente:

a) La coordenada x se puede calcular usando el número que le corresponde en el recorrido en orden simétrico. Escriba una rutina que haga esto para cada nodo del árbol.

b) La coordenada y se puede calcular usando la profundidad del nodo. Escriba una rutina que haga esto para cada nodo del árbol.

c) Determine, en términos de una unidad imaginaria, cuáles serán las dimensiones del dibujo. También determine cómo ajustar las unidades de forma que la altura del árbol siempre sea aproximadamente dos tercios de su anchura.

d) Demuestre que cuando se usa este sistema, las aristas no se cruzan y que para cada nodo X, todos los elementos del subárbol izquierdo de X aparecen a su izquierda y todos los elementos del subárbol derecho de X aparecen a su derecha.

e) Determine si ambas coordenadas se pueden calcular por medio de un único método recursivo.

f) Escriba un programa de dibujo de árboles de propósito general que convierta un árbol en una secuencia de instrucciones de generación de gráficos del siguiente estilo (los círculos están numerados en el orden en que se pintan):

```
circulo( x, y ); // Dibuja un círculo con centro en (x, y)
dibujaLinea( i, j ); // Conecta el círculo i con el j
```

g) Escriba un programa que lea las instrucciones de generación de gráficos y genere código Java para dibujar sobre un lienzo (observe que tiene que escalar las coordenadas almacenadas en pixels).

17.14. Diseñe un applet que ilustre las distintas estrategias de recorrido de árboles.