En nuestro ejemplo de árbol binario crearemos un árbol binario especial, conocido como **árbol de búsqueda binaria**. Un árbol de búsqueda binaria (sin valores de nodo duplicados) cuenta con la característica de que los valores en cualquier subárbol izquierdo son menores que el valor del nodo padre de ese subárbol, y los valores en cualquier subárbol derecho son mayores que el valor del nodo padre de ese subárbol. En la figura 17.16 se muestra un árbol de búsqueda binaria con 12 valores enteros. Observe que la forma del árbol de búsqueda binaria que corresponde a un conjunto de datos puede variar, dependiendo del orden en el que se inserten los valores en el árbol.

La aplicación de las figuras 17.17 y 17.18 crea un árbol de búsqueda binaria compuesto por valores enteros, y lo recorre (es decir, avanza a través de todos sus nodos) de tres maneras: usando los **recorridos inorden**, **preorden** y **postorden** recursivos. El programa genera 10 números aleatorios e inserta a cada uno de ellos en el árbol. La clase `Arbol` se declara en el paquete `com.deitel.jhtp7.cap17` para fines de reutilización.

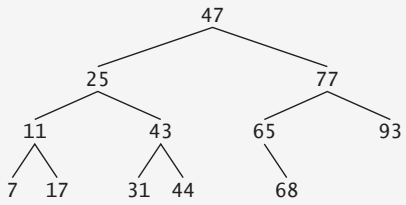

Figura 17.16 | Árbol de búsqueda binario que contiene 12 valores.

```
1  // Fig. 17.17: Arbol.java
2  // Definición de las clases NodoArbol y Arbol.
3  package com.deitel.jhtp7.cap17;
4
5  // definición de la clase NodoArbol
6  class NodoArbol
7  {
8     // miembros de acceso del paquete
9     NodoArbol nodoIzq; // nodo izquierdo
10    int datos; // valor del nodo
11    NodoArbol nodoDer; // nodo derecho
12
13    // el constructor inicializa los datos y hace de este nodo un nodo raíz
14    public NodoArbol( int datosNodo )
15    {
16       datos = datosNodo;
17       nodoIzq = nodoDer = null; // el nodo no tiene hijos
18    } // fin del constructor de NodoArbol
19
20    // localiza el punto de inserción e inserta un nuevo nodo; ignora los valores duplicados
21    public void insertar( int valorInsertar )
22    {
23       // inserta en el subárbol izquierdo
24       if ( valorInsertar < datos )
25       {
26          // inserta nuevo NodoArbol
27          if ( nodoIzq == null )
28             nodoIzq = new NodoArbol( valorInsertar );
29          else // continúa recorriendo el subárbol izquierdo
30             nodoIzq.insertar( valorInsertar );
31       } // fin de if
```

Figura 17.17 | Declaraciones de las clases `NodoArbol` y `Arbol` para un árbol de búsqueda binaria. (Parte 1 de 3).

```
32         else if ( valorInsertar > datos ) // inserta en el subárbol derecho
33         {
34            // inserta nuevo NodoArbol
35            if ( nodoDer == null )
36               nodoDer = new NodoArbol( valorInsertar );
37            else // continúa recorriendo el subárbol derecho
38               nodoDer.insertar( valorInsertar );
39         } // fin de else if
40      } // fin del método insertar
41   } // fin de la clase NodoArbol
42
43   // definición de la clase Arbol
44   public class Arbol
45   {
46      private NodoArbol raiz;
47
48      // el constructor inicializa un Arbol vacío de enteros
49      public Arbol()
50      {
51         raiz = null;
52      } // fin del constructor de Arbol sin argumentos
53
54      // inserta un nuevo nodo en el árbol de búsqueda binaria
55      public void insertarNodo( int valorInsertar )
56      {
57         if ( raiz == null )
58            raiz = new NodoArbol( valorInsertar ); // crea el nodo raíz aquí
59         else
60            raiz.insertar( valorInsertar ); // llama al método insertar
61      } // fin del método insertarNodo
62
63      // comienza el recorrido preorden
64      public void recorridoPreorden()
65      {
66         ayudantePreorden( raiz );
67      } // fin del método recorridoPreorden
68
69      // método recursivo para realizar el recorrido preorden
70      private void ayudantePreorden( NodoArbol nodo )
71      {
72         if ( nodo == null )
73            return;
74
75         System.out.printf( "%d ", nodo.datos ); // imprime los datos del nodo
76         ayudantePreorden( nodo.nodoIzq );       // recorre el subárbol izquierdo
77         ayudantePreorden( nodo.nodoDer );       // recorre el subárbol derecho
78      } // fin del método ayudantePreorden
79
80      // comienza recorrido inorden
81      public void recorridoInorden()
82      {
83         ayudanteInorden( raiz );
84      } // fin del método recorridoInorden
85
86      // método recursivo para realizar el recorrido inorden
87      private void ayudanteInorden( NodoArbol nodo )
88      {
89         if ( nodo == null )
90            return;
```

Figura 17.17 | Declaraciones de las clases `NodoArbol` y `Arbol` para un árbol de búsqueda binaria. (Parte 2 de 3).

```
91
92          ayudanteInorden( nodo.nodoIzq );        // recorre el subárbol izquierdo
93          System.out.printf( "%d ", nodo.datos ); // imprime los datos del nodo
94          ayudanteInorden( nodo.nodoDer );        // recorre el subárbol derecho
95       } // fin del método ayudanteInorden
96
97       // comienza recorrido postorden
98       public void recorridoPostorden()
99       {
100         ayudantePostorden( raiz );
101      } // fin del método recorridoPostorden
102
103      // método recursivo para realizar el recorrido postorden
104      private void ayudantePostorden( NodoArbol nodo )
105      {
106         if ( nodo == null )
107            return;
108
109         ayudantePostorden( nodo.nodoIzq );       // recorre el subárbol izquierdo
110         ayudantePostorden( nodo.nodoDer );       // recorre el subárbol derecho
111         System.out.printf( "%d ", nodo.datos );  // imprime los datos del nodo
112      } // fin del método ayudantePostorden
113   } // fin de la clase Arbol
```

Figura 17.17 | Declaraciones de las clases `NodoArbol` y `Arbol` para un árbol de búsqueda binaria. (Parte 3 de 3).

Analicemos el programa del árbol binario. El método `main` de la clase `PruebaArbol` (figura 17.18) empieza creando una instancia de un objeto `Arbol` vacío y asigna su referencia a la variable `arbol` (línea 10). En las líneas 17 a 22 se generan 10 enteros al azar, cada uno de los cuales se inserta en el árbol binario mediante una llamada al método `insertarNodo` (línea 21). Después el programa realiza recorridos preorden, inorden y postorden (los cuales explicaremos en breve) de `arbol` (líneas 25, 28 y 31, respectivamente).

La clase `Arbol` (figura 17.17, líneas 44 a 113) tiene un campo `private` llamado `raiz` (línea 46); una referencia tipo `NodoArbol` al nodo raíz del árbol. El constructor de `Arbol` (líneas 49 a 52) inicializa `raiz` con `null` para indicar que el árbol está vacío. La clase contiene el método `insertarNodo` (líneas 55 a 61) para insertar un nuevo nodo en el árbol, además de los métodos `recorridoPreorden` (líneas 64 a 67), `recorridoInorden` (líneas 81 a 84) y `recorridoPostorden` (líneas 98 a 101) para empezar recorridos del árbol. Cada uno de estos métodos llama a un método utilitario recursivo para realizar las operaciones de recorrido en la representación interna del árbol. (En el capítulo 15 hablamos sobre la recursividad).

```
1  // Fig. 17.18: PruebaArbol.java
2  // Este programa prueba la clase Arbol.
3  import java.util.Random;
4  import com.deitel.jhtp7.cap17.Arbol;
5
6  public class PruebaArbol
7  {
8     public static void main( String args[] )
9     {
10       Arbol arbol = new Arbol();
11       int valor;
12       Random numeroAleatorio = new Random();
13
14       System.out.println( "Insertando los siguientes valores: " );
15
16       // inserta 10 enteros aleatorios de 0 a 99 en arbol
```

Figura 17.18 | Programa de prueba de un árbol binario. (Parte 1 de 2).

```
17         for ( int i = 1; i <= 10; i++ )
18         {
19            valor = numeroAleatorio.nextInt( 100 );
20            System.out.print( valor + " " );
21            arbol.insertarNodo( valor );
22         } // fin de for
23
24         System.out.println ( "\n\nRecorrido preorden" );
25         arbol.recorridoPreorden(); // realiza recorrido preorden de arbol
26
27         System.out.println ( "\n\nRecorrido inorden" );
28         arbol.recorridoInorden(); // realiza recorrido inorden de arbol
29
30         System.out.println ( "\n\nRecorrido postorden" );
31         arbol.recorridoPostorden(); // realiza recorrido postorden de arbol
32         System.out.println();
33      } // fin de main
34   } // fin de la clase PruebaArbol
```

```
Insertando los siguientes valores:
17 54 3 30 95 69 85 88 16 30

Recorrido preorden
17 3 16 54 30 95 69 85 88

Recorrido inorden
3 16 17 30 54 69 85 88 95

Recorrido postorden
16 3 30 88 85 69 95 54 17
```

Figura 17.18 | Programa de prueba de un árbol binario. (Parte 2 de 2).

El método `insertarNodo` de la clase `Arbol` (líneas 55 a 61) determina primero si el árbol está vacío. De ser así, en la línea 58 se asigna un nuevo objeto `NodoArbol`, se inicializa el nodo con el entero que se insertará en el árbol y se asigna el nuevo nodo a la referencia `raiz`. Si el árbol no está vacío, en la línea 60 se hace una llamada al método `insertar` de `NodoArbol` (líneas 21 a 41). Este método utiliza la recursividad para determinar la posición del nuevo nodo en el árbol e inserta el nodo en esa posición. En un árbol de búsqueda binaria, un nodo puede insertarse solamente como nodo hoja.

El método `insertar` de `NodoArbol` compara el valor a insertar con el valor de datos en el nodo raíz. Si el valor a insertar es menor que los datos del nodo raíz (línea 24), el programa determina si el subárbol izquierdo está vacío (línea 27). De ser así, en la línea 28 se asigna un nuevo objeto `NodoArbol`, se inicializa con el entero que se insertará y se asigna el nuevo nodo a la referencia `nodoIzquierdo`. En caso contrario, en la línea 30 se hace una llamada recursiva a `insertar` para que se inserte el valor en el subárbol izquierdo. Si el valor a insertar es mayor que los datos del nodo raíz (línea 32), el programa determina si el subárbol derecho está vacío (línea 35). De ser así, en la línea 36 se asigna un nuevo objeto `NodoArbol`, se inicializa con el entero que se insertará y se asigna el nuevo nodo a la referencia `nodoDerecho`. En caso contrario, en la línea 38 se hace una llamada recursiva a `insertar` para que se inserte el valor en el subárbol derecho. Si el `valorInsertar` ya se encuentra en el árbol, simplemente se ignora.

Los métodos `recorridoInorden`, `recorridoPreorden` y `recorridoPostorden` llaman a los métodos ayudantes de `Arbol` llamados `ayudanteEnorden` (líneas 87 a 95), `ayudantePreorden` (líneas 70 a 78) y `ayudantePostorden` (líneas 104 a 112), respectivamente, para recorrer el árbol e imprimir los valores de los nodos. Los métodos ayudantes en la clase `Arbol` permiten al programador iniciar un recorrido sin tener que pasar el nodo `raiz` al método. La referencia `raiz` es un detalle de implementación que no debe ser accesible para el programador. Los métodos `recorridoInorden`, `recorridoPreorden` y `recorridoPostorden` simplemente toman la referencia privada `raiz` y la pasan al método ayudante apropiado para iniciar un recorrido del árbol. El caso base para cada método ayudante determina si la referencia que recibe es `null` y, de ser así, regresa inmediatamente.