CAPÍTULO 16

Árboles. Árboles binarios y árboles ordenados

Objetivos

Con el estudio de este capítulo usted podrá:

- Estructurar datos en orden jerárquico.
- Conocer la terminología básica relativa a árboles.
- Distinguir los diferentes tipos de árboles binarios.
- Recorrer un árbol binario de tres formas diferentes.
- Reconocer la naturaleza recursiva de las operaciones con árboles.
- Representar un árbol binario con una estructura enlazada.
- Evaluar una expresión algebraica utilizando un árbol binario.
- Construir un árbol binario ordenado (de búsqueda).

Contenido

16.1. Árboles generales y terminología.
16.2. Árboles binarios.
16.3. Estructura de un árbol binario.
16.4. Árbol de expresión.
16.5. Recorrido de un árbol.
16.6. Implementación de operaciones.
16.7. Árbol binario de búsqueda.
16.8. Operaciones en árboles binarios de búsqueda.
16.9. Diseño recursivo de un árbol binario de búsqueda.

RESUMEN.
BIBLIOGRAFÍA RECOMENDADA.
EJERCICIOS.
PROBLEMAS.

Conceptos clave

- Árbol.
- Árbol binario.
- Árbol binario de búsqueda.
- *Enorden*.
- Hoja.
- Jerarquía.
- *Postorden*.
- *Preorden*.
- Raiz.
- Recorrido.
- Subárbol.

INTRODUCCIÓN

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales*, al contrario que los arrays y las listas enlazadas que constituyen *estructuras lineales*. La estructura de datos árbol generaliza las estructuras lineales vistas en capítulos anteriores.

Los árboles se utilizan para representar fórmulas algebraicas, para organizar objetos en orden de tal forma que las búsquedas son muy eficientes, y en aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de las aplicaciones citadas, los árboles se utilizan en diseño de compiladores, proceso de texto y algoritmos de búsqueda.

En el capítulo se estudiará el concepto de árbol general y los tipos de árboles más usuales, binario y binario de búsqueda o árbol ordenado. Asimismo, se estudiarán algunas aplicaciones típicas del diseño y construcción de árboles.

16.1. ÁRBOLES GENERALES Y TERMINOLOGÍA

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general. La Figura 16.1 representa un ejemplo de árbol general, gráficamente puede verse cómo un árbol invertido, la raíz en la parte más alta de la que salen ramas que llegan a las hojas, que están en la parte baja.

Un **árbol** consta de un conjunto finito de elementos, denominados **nodos** y un conjunto finito de líneas dirigidas, denominadas **ramas**, que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo.

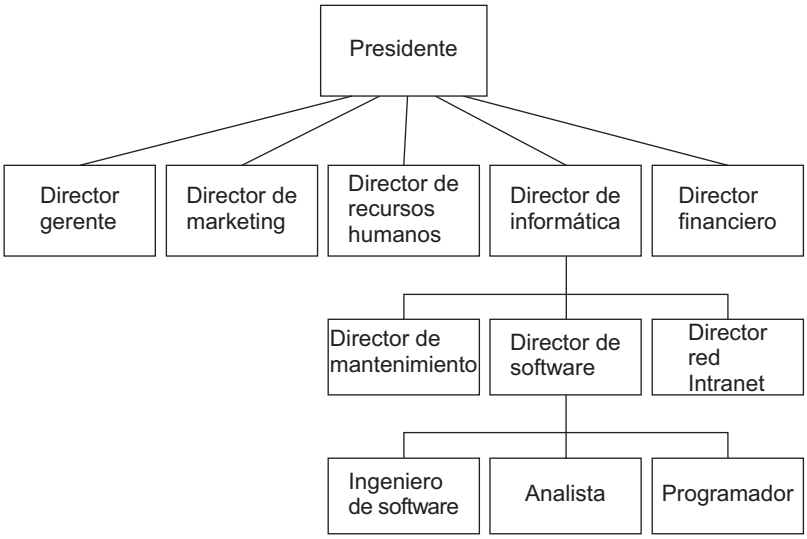


Figura 16.1. Estructura jerárquica tipo árbol.

Definición 1:

Un árbol consta de un conjunto finito de elementos, llamados nodos y un conjunto finito de líneas dirigidas, llamadas ramas, que conectan los nodos.

Definición 2:

Un árbol es:

1. La estructura de datos vacía.
2. O bien un conjunto de uno o más nodos tales que:
   2.1. Hay un nodo diseñado especialmente llamado raíz.
   2.2. Los nodos restantes se dividen en $n \geq 0$ conjuntos disjuntos, `T1...Tn`, tal que cada uno de estos conjuntos es un árbol. A `T1...Tn` se les denomina subárboles del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz.** Obsérvese en la definición 2 que el árbol ha sido definido de modo recursivo ya que los subárboles se definen como árboles. La Figura 16.2 muestra un árbol.

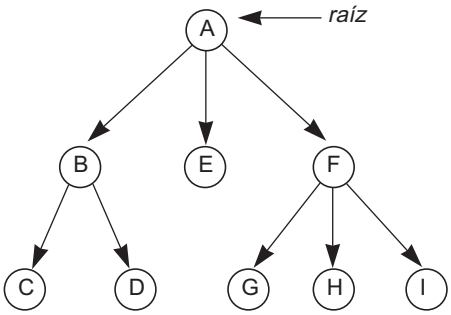


Figura 16.2. Árbol.

16.1.1. Terminología

Además del raíz, existen muchos términos utilizados en la descripción de los atributos de un árbol. En la Figura 16.3, el nodo `A` es el raíz. Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores.

Estos nodos sucesores se llaman **hijos**. Por ejemplo, el nodo `B` es el padre de los hijos `E` y `F`. El padre de `H` es el nodo `D`. Un árbol puede representar diversas generaciones en la familia. Los hijos de un nodo y los hijos de estos hijos se llaman **descendientes** y el padre y abuelos de un nodo son sus **ascendientes**. Por ejemplo, los nodos `E`, `F`, `I` y `J` son descendientes de `B`. Cada nodo no raíz tiene un único padre y cada padre tiene cero o más nodos hijos. Dos o más nodos con el mismo padre se llaman **hermanos**. Un nodo sin hijos, tales como `E`, `I`, `J`, `G` y `H` se llaman nodo **hoja**.

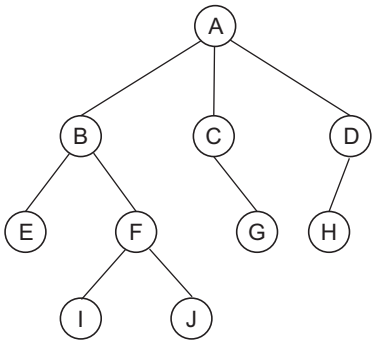


Figura 16.3. Árbol general.

El **nivel** de un nodo es su distancia al nodo raíz. El raíz tiene una distancia cero de sí misma, por ello se dice que el raíz está en el nivel 0. Los hijos del raíz están en el nivel 1, sus hijos están en el nivel 2 y así sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la relación entre niveles y *hermanos*. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos. Por ejemplo, en el nivel 2 (Figura 16.4), `C` y `D` son hermanos, al igual que lo son `G`, `H`, e `I`, pero `D` y `G` no son hermanos ya que ellos tienen diferentes padres.

Un **camino** es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el raíz. En la Figura 16.4, el camino desde el raíz a la hoja `I`, se representa por `AFI`. Incluye dos ramas distintas `AF` y `FI`.

La **altura** o **profundidad** de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición[1] la altura de un árbol vacío es `0`. La Figura 16.4 contiene nodos en tres niveles: `0, 1 y 2`. Su altura es `3`.

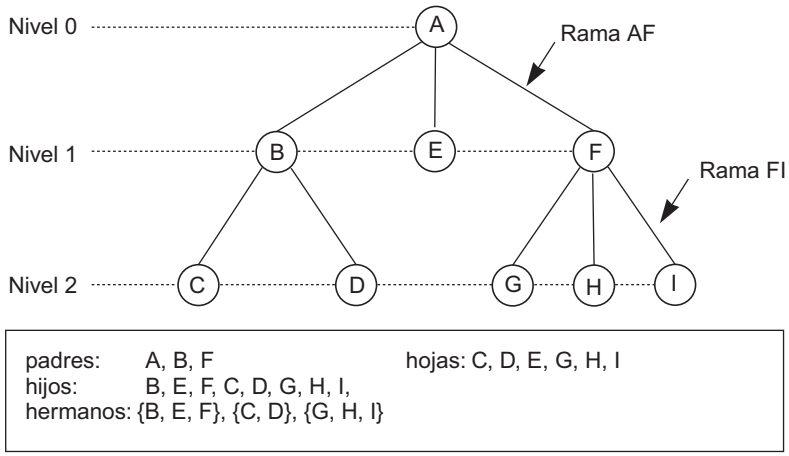


Figura 16.4. Terminología de árboles.

[1] También se suele definir la **profundidad** de un árbol como el nivel máximo de cada nodo. En consecuencia, la profundidad del nodo raíz es 0, la de su hijo 1, etc. Las dos terminologías son aceptadas.

Definición

El nivel de un nodo es su distancia desde el nodo raíz. La altura de un árbol es el nivel de la hoja del camino más largo desde el r aíz más uno.

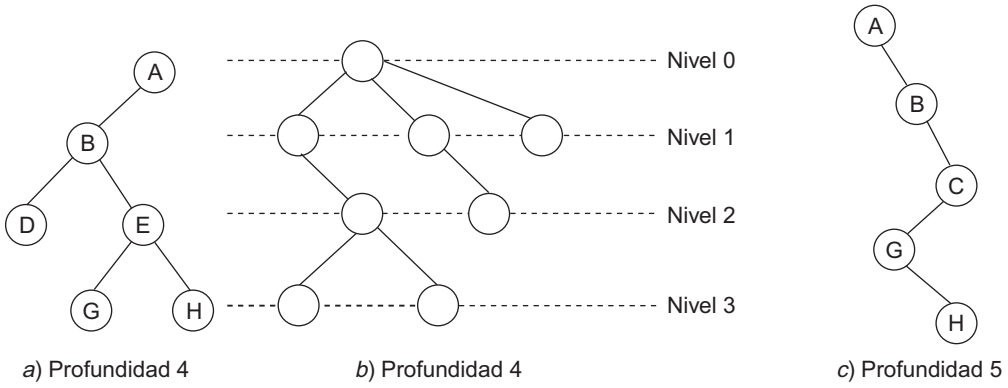


Figura 16.5. Árboles de profundidades diferentes.

Un árbol se divide en subárboles. Un **subárbol** es cualquier estructura conectada por debajo del raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos los descendientes del nodo. El primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En la Figura 16.4, `BCD` es un subárbol al igual que `E` y `FGHI`. Obsérvese, que por esta definición, un nodo simple es un subárbol. Por consiguiente, el subárbol `B` se puede dividir en subárboles `C` y `D` mientras que el subárbol `F` contiene los subárboles `G`, `H` e `I`. Se dice que `G`, `H`, `I`, `C` y `D` son subárboles sin descendientes.

Definición recursiva

El concepto de subárbol conduce a una definición recursiva de un árbol. Un árbol es un conjunto de nodos que:

1. O bien es vacío, o bien,
2. Tiene un nodo deter minado, llamado r aíz, del que jerárquicamente descienden cero o más subárboles, que son también árboles.

Resumen de definiciones

- El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
- Las flechas que conectan un nodo a otro se llaman **arcos** o **ramas**.

- Los **nodos terminales**, esto es, nodos de los cuales no se deduce ningún nodo, se denominan **hojas**.
- Los nodos que no son hojas se denominan **nodos internos**.
- En un árbol una rama va de un nodo n_1 a un nodo n_2, se dice que n_1 es el padre de n_2 y que n_2 es un **hijo** de n_1.
- n_1 se llama **ascendiente** de n_2 si n_1 es el padre de n_2 o si n_1 es el padre de un ascendiente de n_2.
- n_2 se llama **descendiente** de n_1 si n_1 es un ascendiente de n_2.
- Un **camino** de n_1 a n_2 es una secuencia de arcos contiguos que van de n_1 a n_2.
- La **longitud** de un camino es el número de arcos que contiene, o de forma equivalente, el número de nodos del camino menos uno.
- El **nivel** de un nodo es la longitud del camino que lo conecta al nodo raíz.
- La **profundidad** o **altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja más uno.
- Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es, a su vez un árbol.
- Sea S un subárbol de un árbol A: si para cada nodo *n* de SA, SA contiene también todos los descendientes de *n* en A. SA se llama un **subárbol completo** de A.
- Un árbol está **equilibrado** cuando, dado un número máximo *k* de hijos de cada nodo y la **altura del árbol** *h*, cada nodo de nivel $l < h - 2$ tiene exactamente *k* hijos. El árbol está equilibrado perfectamente entre cada nodo de nivel $l < h$ tiene exactamente *k* hijos.

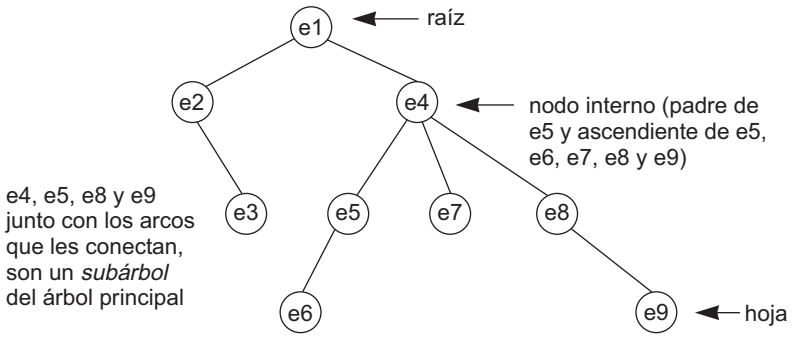


Figura 16.6. Un árbol y sus nodos.

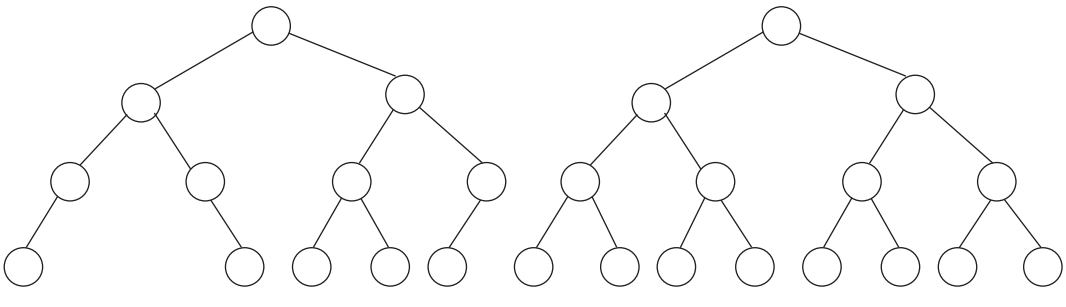


Figura 16.7. *a*) Un árbol equilibrado; *b*) Un árbol perfectamente equilibrado.

16.1.2. Representación gráfica de un árbol

Las formas más frecuentes de representar en papel un árbol son como árbol invertido y como una lista.

Representación como árbol invertido

Es el diagrama o carta de organización utilizada hasta ahora en las diferentes figuras. El nodo raíz se encuentra en la parte más alta de una jerarquía de la que descienden ramas que van a parar a los nodos hijos, y así sucesivamente. La Figura 16.8 presenta esta representación para una descomposición de una computadora.

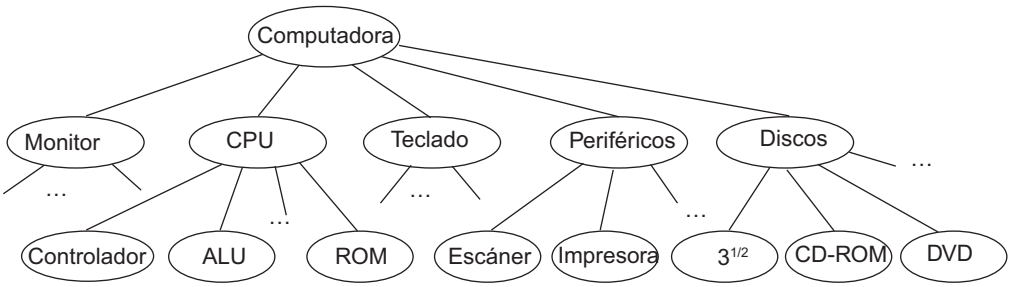


Figura 16.8. Árbol general (computadora).

Representación de lista

Otro formato utilizado para representar árboles es la lista entre paréntesis. Esta es la notación utilizada con expresiones algebraicas. En esta representación, cada paréntesis abierto indica el comienzo de un nuevo nivel; cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol. La notación en paréntesis correspondiente al árbol de la Figura 16.2: `A(B (C, D), E, F (G, H, I))`

EJEMPLO 16.1. Representar el árbol general de la Figura 16.9 en forma de lista.

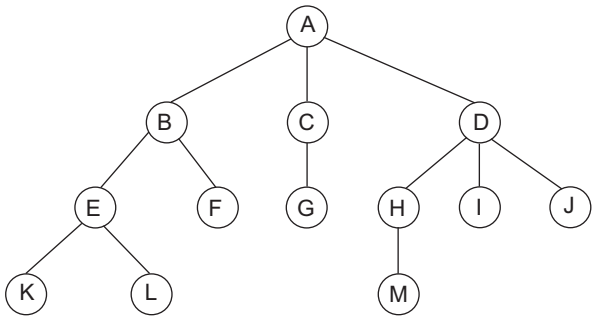


Figura 16.9. Árbol general.

La solución: `A ( B ( E ( K, L), F ), C ( G ), D ( H ( M ), I, J ) ) )`

16.2. ÁRBOLES BINARIOS

Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario, cada nodo puede tener, cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha como *hijo derecho*.

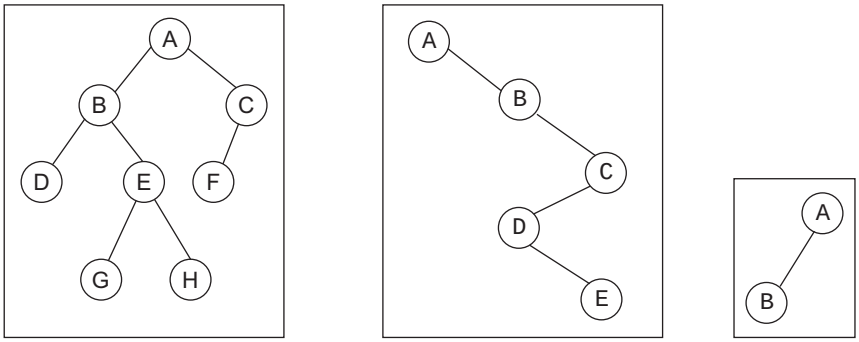


Figura 16.10. Árboles binarios.

Nota

Un árbol binario no puede tener más de dos subárboles. Un nodo no puede tener más de dos hijos.

Un árbol binario es una estructura recursiva. Cada nodo es el raíz de su propio subárbol y tiene hijos, que son raíces de árboles llamados los subárboles derecho e izquierdo del nodo, respectivamente. Un árbol binario se divide en tres subconjuntos disjuntos:

`{R}`	Nodo raíz
`{I1, I2, ...In}`	Subárbol izquierdo de `R`
`{D1, D2, ...Dn}`	Subárbol derecho de `R`

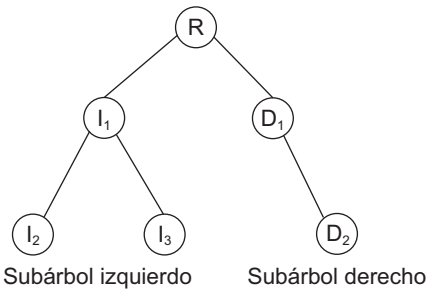


Figura 16.11. Árbol binario.

En cualquier nivel *n*, un árbol binario puede contener de 1 a 2^n nodos. El número de nodos por nivel contribuye a la densidad del árbol.

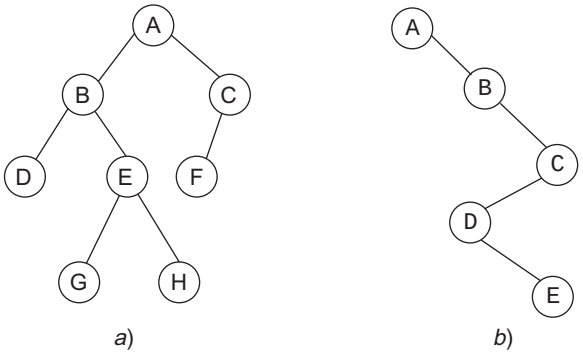


Figura 16.12. Árboles binarios: *a*) profundidad 4; *b*) profundidad 5.

En la Figura 16.12*a*) el árbol A contiene 8 nodos en una profundidad de 4, mientras que el árbol 16.12*b*) contiene 5 nodos y una profundidad 5.

16.2.1. Equilibrio

La distancia de un nodo al raíz determina la eficiencia con la que puede ser localizado. Por ejemplo, dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo sólo un camino de bifurcación o de ramas, el que conduce al nodo deseado. De modo similar, los nodos a nivel 2 de un árbol sólo pueden ser accedidos siguiendo sólo dos ramas del árbol.

La característica anterior nos conduce a una característica muy importante de un árbol binario, su **balance** o **equilibrio**. Para determinar si un árbol está equilibrado, se calcula su factor de equilibrio. El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si la altura del subárbol izquierdo es h_I y la altura del subárbol derecho como h_D, entonces el factor de equilibrio del árbol B se determina por la siguiente fórmula: $B = h_D - h_I$

Utilizando esta fórmula, el equilibrio del nodo raíz los árboles de la Figura 16.12 son *a* 1 y *b* 4.

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también perfectamente equilibrados. Dado que esta definición ocurre raramente se aplica una definición alternativa. Un árbol binario está equilibrado si la altura de sus subárboles difiere en no más de uno y sus subárboles son también equilibrados; por consiguiente, el factor de equilibrio de cada nodo puede tomar los valores: -1, 0, +1.

16.2.2. Árboles binarios completos

Un árbol binario **completo** de profundidad *n* (0 a *n* – 1 niveles) es un árbol en el que para cada nivel, del *0* al nivel *n* – 2, tiene un conjunto lleno de nodos y todos los nodos hoja a nivel *n* – 1 ocupan las posiciones más a la izquierda del árbol.

Un árbol binario completo que contiene 2^n nodos a nivel *n* es un **árbol lleno**. Un árbol lleno es un árbol binario que tiene el máximo número de entradas para su altura. Esto sucede cuando el último nivel está lleno. La Figura 16.13 muestra un árbol binario completo; el árbol de la Figura 16.14*b*) se corresponde con uno lleno.

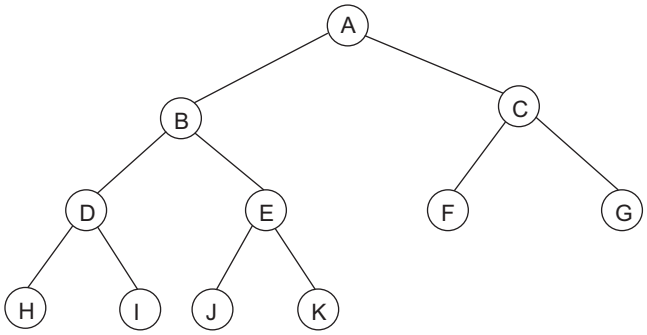


Figura 16.13. Árbol completo (profundidad 4).

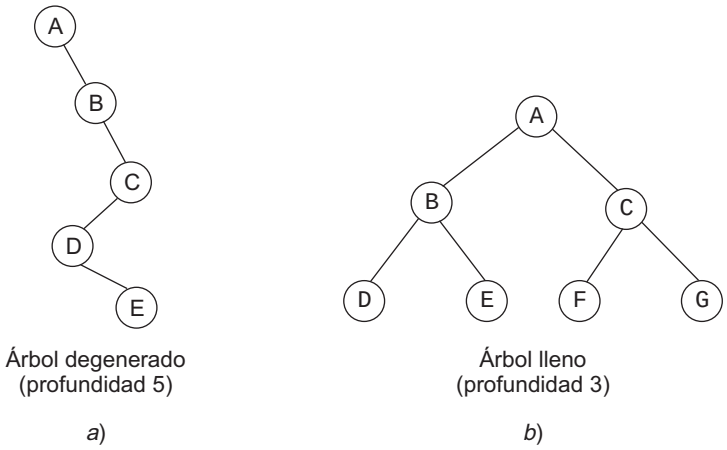


Figura 16.14. Clasificación de árboles binarios: *a*) degenerado; *b*) lleno.

El último caso de árbol es un tipo especial denominado **árbol degenerado** en el que hay un solo nodo hoja (`E`) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

Los árboles binarios completos y llenos de profundidad `k+1` proporcionan algunos datos matemáticos de interés. En cada caso, existe un nodo (2^0) al nivel `0` (raíz), dos nodos (2^1) a nivel `1`, cuatro nodos (2^2) a nivel `2`, etc. A través de los primeros *k-1* niveles se puede demostrar, considerando la suma de los términos de una progresión geométrica de razón `2`, que hay 2^k `- 1` nodos.

$$1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

A nivel *k*, el número de nodos adicionados para un árbol completo está en el rango de un mínimo de 1 a un máximo de 2^k (lleno). Con un árbol lleno, el número de nodos es:

$$1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

El número de nodos, n, en un árbol binario completo de profundidad k+1 (0 a k niveles) cumple la inigualdad:

$$2^k \leq n \leq 2^{k+1} - 1 < 2^{k+1}$$

Aplicando logaritmos a la ecuación con desigualdad anterior

$$k \leq \log_2 (n) < k + 1$$

se deduce que la altura o profundidad de un árbol binario completo de n nodos es:

$$h = |\text{Log}_2 n| + 1$$ (*parte entera* de $\text{Log}_2 n$ *más* 1)

Por ejemplo, un árbol lleno de profundidad 4 (*niveles 0 a 3*) tiene $2^4 - 1 = 15$ nodos.

EJEMPLO 16.2. Calcular la profundidad máxima y mínima de un árbol con 5 nodos .

La profundidad máxima de un árbol con 5 nodos es 5, se corresponde con un árbol degenerado.

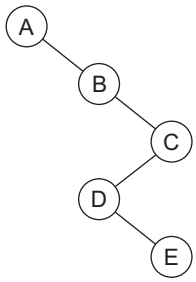


Figura 16.15. Árbol degenerado de raíz A.

La profundidad mínima *h* (número de niveles más uno) de un árbol con 5 nodos, aplicando la inecuación del número de nodos de un árbol binario completo:

$$k \leq log_2 (5) < k + 1$$

como $\log_2(5) = 2.32$, la profundidad h = 3.

EJEMPLO 16.3. Suponiendo que se tiene n = 10.000 elementos que van a ser los nodos de un árbol binario completo. Determinar la profundidad del árbol.

En el árbol binario completo con n nodos, la profundidad del árbol es el valor entero de $\log_2 n + 1$, que es a su vez, la distancia del camino más largo desde el raíz a un nodo más uno.

$$\text{Profundidad} = \text{int} (\log_2 10000) + 1 = \text{int} (13.28) + 1 = 14$$

16.2.3. *TAD* Árbol binario

La estructura de árbol binario constituye un *tipo abstracto de datos*; las operaciones básicas que definen el *TAD árbol binario* son las siguientes.

Tipo de dato	Dato que se almacena en los nodos del árbol.
Operaciones	
CrearÁrbol	Inicia el árbol como vacío.
Construir	Crea un árbol con un elemento raíz y dos ramas, izquierda y derecha que son a su vez árboles.
EsVacio	Comprueba si el árbol no tiene nodos.
Raiz	Devuelve el nodo raíz.
Izquierdo	Obtiene la rama o subárbol izquierdo de un árbol dado.
Derecho	Obtiene la rama o subárbol derecho de un árbol dado.
Borrar	Elimina del árbol el nodo con un elemento determinado.
Pertenece	Determina si un elemento se encuentra en el árbol.

16.2.4. Operaciones en árboles binarios

Algunas de las operaciones típicas que se realizan en árboles binarios son éstas:

- Determinar su altura.
- Determinar su número de elementos.
- Hacer una copia.
- Visualizar el árbol binario en pantalla o en impresora.
- Determinar si dos árboles binarios son idénticos.
- Borrar (eliminar el árbol).
- Si es un árbol de expresión, evaluar la expresión.

Todas estas operaciones se pueden realizar recorriendo el árbol binario de un modo sistemático. El recorrido es la operación de visita al árbol, o lo que es lo mismo, la visita a cada nodo del árbol una vez y sólo una. La visita de un árbol es necesaria en muchas ocasiones, por ejemplo, si se desea imprimir la información contenida en cada nodo. Existen diferentes formas de visitar o recorrer un árbol que se estudiarán más adelante.

16.3. ESTRUCTURA DE UN ÁRBOL BINARIO

Un árbol binario se construye con nodos. Cada nodo debe contener el campo *dato* (datos a almacenar) y dos campos de enlace (*apuntador*), uno al subárbol izquierdo (**izquierdo**, **izdo**) y otro al subárbol derecho (**derecho**, **dcho**). El valor `NULL` indica un árbol o un subárbol vacío.

La Figura 16.16 muestra la representación enlazada de dos árboles binarios de raíz `A`. El primero, es un árbol degenerado a la izquierda; el segundo, es un árbol binario completo de profundidad `4`.

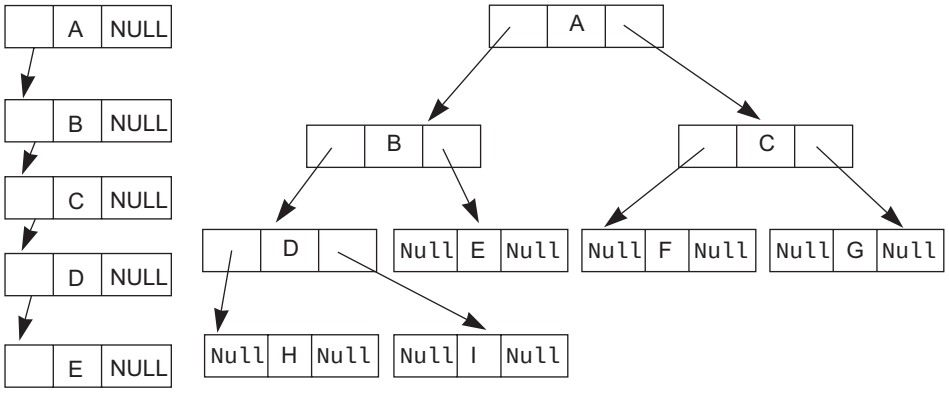


Figura 16.16. Representación enlazada de dos árboles binarios.

Se puede observar que los nodos de un árbol binario que son hojas se caracterizan por tener sus dos campos de enlace a `NULL`.

16.3.1. Representación de un nodo

La clase `Nodo` agrupa a todos los atributos de que consta: `dato`, `izdo` (rama izquierda) y `dcho` (rama derecha). Además, dispone de dos costructores, el primero inicializa el campo `dato` a un valor y los enlaces a `NULL`, en definitiva, se inicializa como hoja. El segundo, inicializa `dato` a un valor y las ramas a dos subárboles. Se incluyen, además, las funciones miembro de acceso y modificación de cada uno de sus atributos.

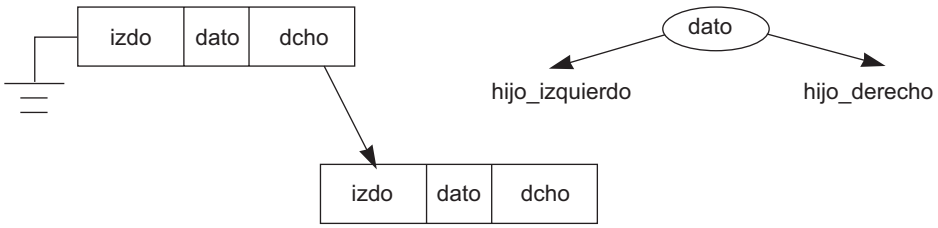


Figura 16.17. Representación gráfica de los campos de un nodo.

```
typedef int Tipoelemento;

class Nodo
{
protected:
    Tipoelemento dato;
    Nodo *izdo;
    Nodo *dcho;
```

```
public:

  Nodo(Tipoelemento valor)
  {
     dato = valor;
     izdo = dcho = NULL;
  }

  Nodo(Tipoelemento valor, Nodo* ramaIzdo, Nodo* ramaDcho)
  {
     dato = valor;
     izdo = ramaIzdo;
     dcho = ramaDcho;
  }
  // operaciones de acceso
  Tipoelemento valorNodo(){ return dato; }
  Nodo* subárbolIzdo(){ return izdo; }
  Nodo* subárbolDcho(){ return dcho; }

  // operaciones de modificación
  void nuevoValor(Tipoelemento d){ dato = d; }
  void ramaIzdo(Nodo* n){ izdo = n; }
  void ramaDcho(Nodo* n){ dcho = n; }
};
```

16.3.2. Creación de un árbol binario

A partir del nodo raíz de un árbol se puede acceder a los demás nodos del árbol, por ello se mantiene la referencia al raíz del árbol. La rama izquierda y derecha son a su vez árboles binarios que tienen su raíz, y así recursivamente hasta llegar a las hojas del árbol. La clase `ArbolBinario` tiene el atributo `raiz`, como un puntero a la clase `Nodo` dos constructores que inicializan `raiz` a `NULL` o a un puntero a un `Nodo` respectivamente. Se añaden, además, los métodos `esVacio()` que decide si un árbol binario está vacío `raizArbol()`, `hijoIzdo()`, `hijoDcho()` para implementar las operaciones de obtener el nodo raíz el hijo izquierdo y derecho respectivamente en caso de que lo tenga. El método `nuevoArbol()`crea un puntero a un `Nodo` con el atributo dato, rama izquierda y derecha pasadas en los argumentos. Se implementan, además, las funciones miembro `Praiz( Nodo *r)`y `Oraiz()` encargadas de poner el atributo raíz al puntero a `Nodo r` y obtener un puntero al nodo raíz.

```
class Arbolbinario
{
protected:
       Nodo *raiz;

public:

   ArbolBinario()
   {
     raiz = NULL;
   }
```

```
    ArbolBinario(Nodo *r)
    {
      raiz = r;
    }

    void Praiz( Nodo *r)
    {
      raiz = r;
    }

    Nodo * Oraiz()
    {
      return raiz;
    }

    Nodo raizArbol()
    {
      if(raiz)
         return *raiz;
      else
         throw " arbol vacio";
    }

    bool esVacio()
    {
      return raiz == NULL;
    }

    Nodo * hijoIzdo()
    {
      if(raiz)
         return raiz->subArbolIzdo();
      else
         throw " arbol vacio";
    }

    Nodo * hijoDcho()
    {
      if(raiz)
         return raiz->subArbolDcho();
      else
         throw " arbol vacio";
    }

    Nodo* nuevoArbol(Nodo* ramaIzqda, Tipoelemento dato, Nodo* ramaDrcha)
    {
        return new Nodo(ramaIzqda, dato, ramaDrcha);
    }
};
```

Así, para crear el árbol binario de la Figura 16.18, se puede utilizar el siguente segmento de código:

```
ArbolBinario a1,a2,a3, a4,a;
Nodo * n1,*n2,*n3, *n4;
```

```
n1 = a1.nuevoArbol(NULL,"Maria",NULL);
n2 = a2.nuevoArbol(NULL,"Rodrigo",NULL);
n3 = a3.nuevoArbol(n1,"Esperanza",n2);

n1 = a1.nuevoArbol(NULL,"Anyora",NULL);
n2 = a2.nuevoArbol(NULL,"Abel",NULL);
n4 = a4.nuevoArbol(n1,"M Jesus",n2);
n1 = a1.nuevoArbol(n3,"Esperanza",n4);
a.Praiz(n1);
```

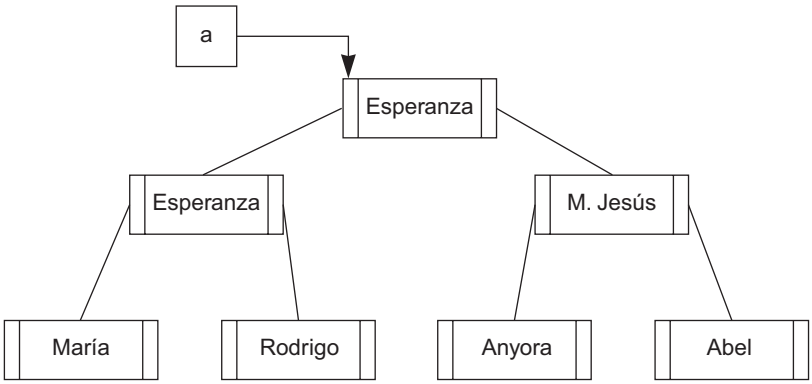


Figura 16.18. Árbol binario de cadenas.

16.4. ÁRBOL DE EXPRESIÓN

Una aplicación muy importante de los árboles binarios son los *árboles de expresiones*. Una **expresión** es una secuencia de *tokens* (componentes de léxicos que siguen unas reglas establecidas). Un *token* puede ser un operando, o bien un operador.

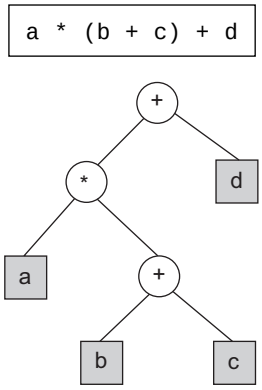


Figura 16.19. Una expresión infija y su árbol de expresión.

La Figura 16.19 representa la expresión *infija* `a * (b + c) + d` junto a su árbol de expresión. El nombre de *infija* es debido a que los operadores se sitúan *entre* los operandos.

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

1. Cada hoja es un operando.
2. Los nodos raíz y nodos internos son operadores.
3. Los subárboles son subexpresiones cuyo nodo raíz es un operador.

Los árboles binarios se utilizan para representar expresiones en memoria; esencialmente, en compiladores de lenguajes de programación. Se observa que los paréntesis de la expresión no aparecen en el árbol, pero están implicados en su forma, y esto resulta muy interesante para la evaluación de la expresión.

Si se supone que todos los operadores tienen dos operandos, se puede representar una expresión mediante un árbol binario cuya raíz contiene un operador y cuyos subárboles izquierdo y derecho, son los operandos izquierdo y derecho respectivamente. Cada operando puede ser una letra (`x, y, a, b` etc.) o una subexpresión representada como un subárbol. La Figura 16.20 muestra un árbol cuya raíz es el operador `*`, su subárbol izquierdo representa la subexpresión `(x + y)` y su subárbol derecho representa la subexpresión `(a - b)`. El nodo raíz del subárbol izquierdo contiene el operador (+) de la subexpresión izquierda y el nodo raíz del subárbol derecho contiene el operador (-) de la subexpresión derecha. Todos los operandos letras se almacenan en nodos hojas.

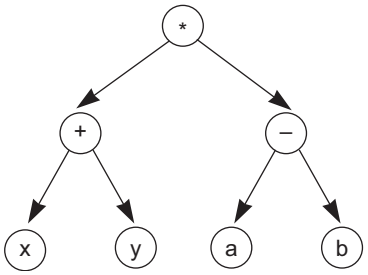


Figura 16.20. Árbol de expresión `(x + y)*(a - b)`.

Utilizando el razonamiento anterior, la expresión `(x * (y - z)) + (a - b)` con paréntesis alrededor de las subexpresiones, forma el árbol binario de la Figura 16.21.

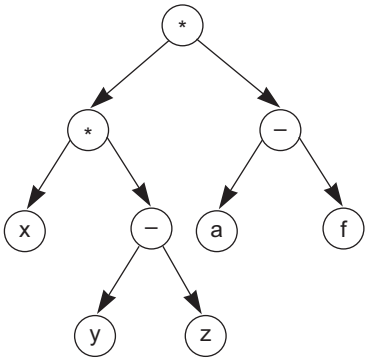
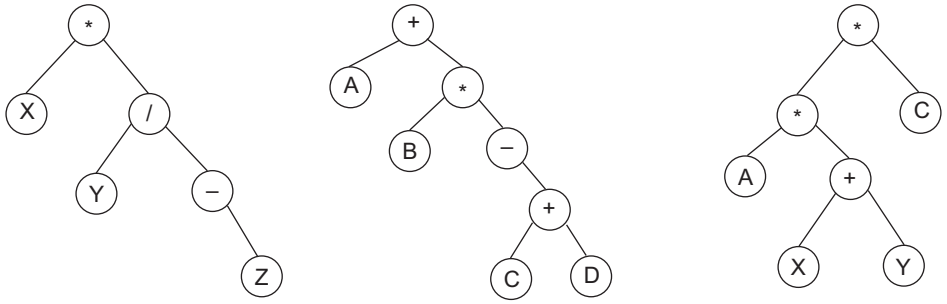


Figura 16.21. Árbol de expresión `(x * (y - z))+(a - b)`.

EJEMPLO 16.4. Deducir las expresiones que representan los árboles binarios de la Figura 16.22.



a) *b*) *c*)

Figura 16.22. Árboles de expresión.

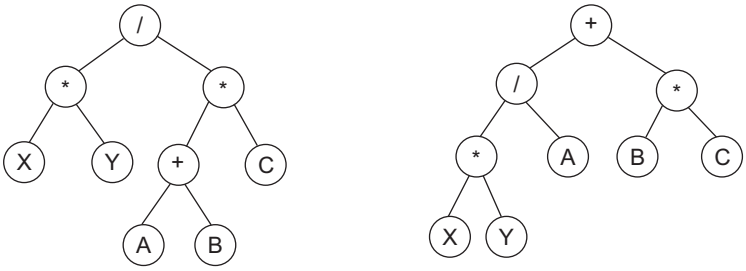
Soluciones

a. `(X * Y /(-Z))`
b. `(A + B * (-(C + D)))`
c. `(A * (X + Y))* C)`

EJEMPLO 16.5. Dibujar la representación en árbol binar io de cada una de las siguientes expresiones:

a. `X * Y /((A + B)* C)`
b. `X * Y /A + B * C`

Soluciones



a) *b*)

Figura 16.23. Árboles de expresión.

16.4.1. Reglas para la construcción de árboles de expresiones

Los árboles de expresiones se utilizan en las computadoras para evaluar expresiones usadas en programas. El algoritmo más sencillo para construir un árbol de expresión es aquél que lee una

expresión completa que contiene paréntesis en la misma. Una expresión con paréntesis es aquélla en que:

1. La prioridad se determina sólo por paréntesis.
2. La expresión completa se sitúa entre paréntesis.

A fin de ver la prioridad en las expresiones, considérese la expresión:

```
a * c + e / g - (b + d)
```

Los operadores con prioridad más alta son `*` y `/`; es decir:

```
(a * c) + (e / g) - (b + d)
```

Los operadores que siguen en orden de prioridad son `+` y `-`, que se evalúan de izquierda a derecha. Por consiguiente, se puede escribir:

```
((a * c) + (e / g)) - (b + d)
```

Por último, la expresión completa entre paréntesis será:

```
(((a * c) + (e / g)) - (b + d))
```

EJEMPLO 16.6. Determinar las expresiones correspondientes de los árboles de expresión de la Figura 16.24.

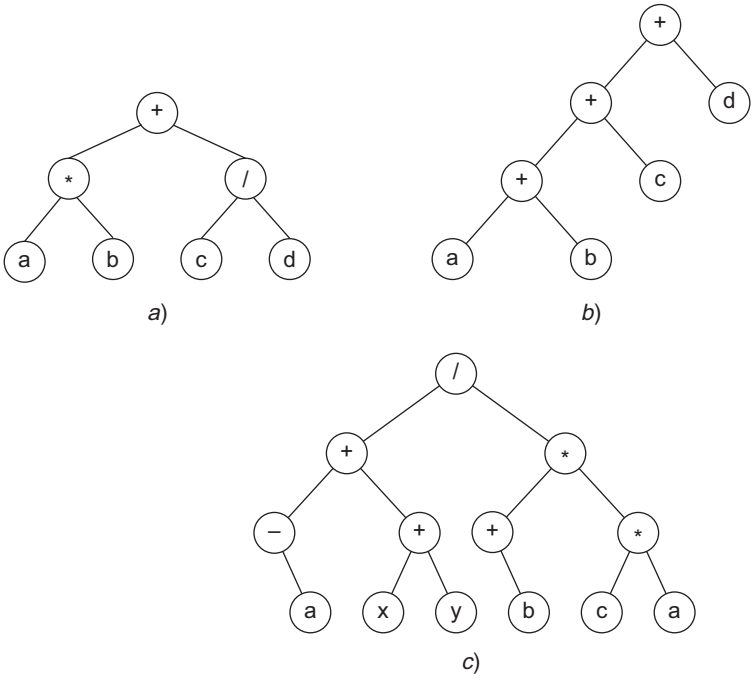


Figura 16.24. Árboles de expresión.

Las soluciones correspondientes son:

a. ((a * b) + (c / d))
b. (((a + b) + c) + d)
c. (((-a) + (x + y)) / ((+b) * (c * a)))

16.5. RECORRIDO DE UN ÁRBOL

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario que las listas enlazadas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el nodo raíz viene el primero, pero ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoría de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

El **recorrido de un árbol binario** requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, profundidad y anchura.

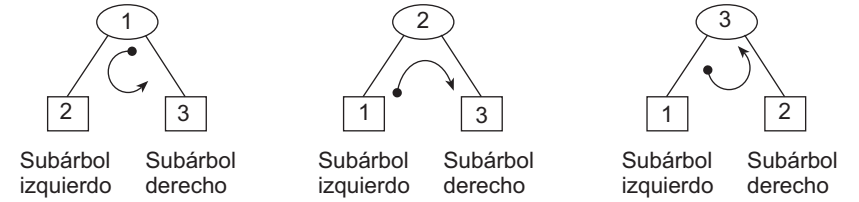
En el **recorrido en profundidad**, el proceso exige un camino desde el raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo.

En el **recorrido en anchura**, el proceso se realiza horizontalmente desde el raíz a todos sus hijos, a continuación a los hijos de sus hijos y así sucesivamente hasta que todos los nodos han sido procesados. En el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel.

Definición

El **recorrido** de un árbol supone visitar cada nodo sólo una v ez.

Dado un árbol binario que consta de raíz, un subárbol izquierdo y un subárbol derecho se pueden definir tres tipos de secuencia de recorrido en profundidad. Estos recorridos estándar se muestran en la Figura 16.25.



a) **Recorrido *preorden*** *b*) **Recorrido *enorden*** *c*) **Recorrido *postorden***

Figura 16.25. Recorridos de árboles binarios.

La designación tradicional de los recorridos utiliza un nombre para el nodo raíz (**N**), para el subárbol izquierdo (**I**) y para el subárbol derecho (**D**).

16.5.1. Recorrido *preorden*

El recorrido *preorden*[2] (**NID**) conlleva los siguientes pasos, en los que el nodo raíz va antes que los subárboles:

1. Visitar el nodo raíz (**N**).
2. Recorrer el subárbol izquierdo (**I**) en preorden.
3. Recorrer el subárbol derecho (**D**) en preorden.

Dado las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero, se procesa la raíz, a continuación el subárbol izquierdo y a continuación el subárbol derecho. Para procesar el subárbol izquierdo se siguen los mismos pasos: raíz, subárbol izquierdo y subárbol derecho (proceso recursivo). Luego se hace lo mismo con el subárbol derecho.

> ***Regla***
>
> En el recorrido preorden, el raíz se procesa antes que los subárboles izquierdo y derecho.

Si utilizamos el recorrido *preorden* del árbol de la Figura 16.26 se visita primero el raíz (nodo `A`); a continuación, se visita el subárbol izquierdo de `A`, que consta de los nodos `B`, `D` y `E`. Dado que el subárbol es a su vez un árbol, se visitan los nodos utilizando el mismo orden (`NID`). Por consiguiente, se visita primero el nodo `B`, después `D` (izquierdo) y por último `E` (derecho).

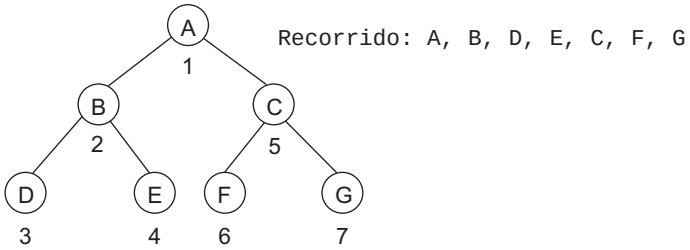


Figura 16.26. Recorrido *preorden* de un árbol binario.

A continuación, se visita subárbol derecho de `A`, que es un árbol que contiene los nodos `C`, `F` y `G`. De nuevo, siguiendo el mismo orden (`NID`), se visita primero el nodo `C`, a continuación `F` (izquierdo) y por último `G` (derecho). En consecuencia, el orden del recorrido *preorden* del árbol de la Figura 16.26 es `A-B-D-E-C-F-G`.

[2] El nombre *preorden*, viene del prefijo latino *pre* que significa "ir antes".

16.5.2. Recorrido *enorden*

El recorrido **enorden** (*inorder*) procesa primero el subárbol izquierdo, después el raíz y a continuación el subárbol derecho. El significado de *en* (*in*) es que la raíz se procesa entre los subárboles.

Si el árbol no está vacío, el método implica los siguientes pasos:

1. Recorrer el subárbol izquierdo (**I**) en enorden.
2. Visitar el nodo raíz (**N**).
3. Recorrer el subárbol derecho (**D**) en enorden.

En el árbol de la Figura 16.27, los nodos se han numerado en el orden en que son visitados durante el recorrido *enorden*. El primer subárbol recorrido es el subárbol izquierdo del nodo raíz (árbol cuyo nodo contiene la letra `B`). Este subárbol es, a su vez, otro árbol con el nodo `B` como raíz, por lo que siguiendo el orden `IND`, se visita primero `D`, a continuación `B` (nodo raíz) y por último `E` (derecha). Después, se visita el nodo raíz, `A`. Por último, se visita el subárbol derecho de `A`, siguiendo el orden `IND` se visita primero `F`, después `C` (nodo raíz) y por último `G`. Por consiguiente, el orden del recorrido enorden del árbol de la Figura 16.27 es `D-B-E-A-F-C-G`.

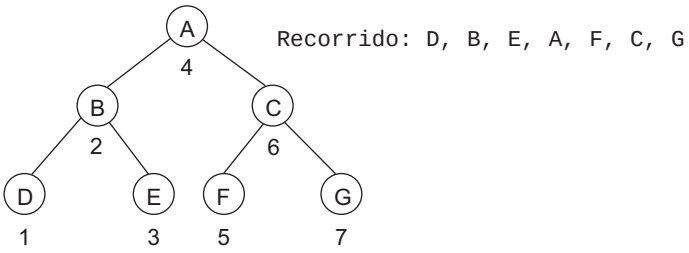


Figura 16.27. Recorrido *enorden* de un árbol binario.

16.5.3. Recorrido *postorden*

El recorrido *postorden* (**IDN**) procesa el nodo raíz (*post*) después de que los subárboles izquierdo y derecho se han procesado. Se comienza situándose en la hoja más a la izquierda y se procesa. A continuación se procesa su subárbol derecho. Por último, se procesa el nodo raíz. Las etapas del algoritmo, si el árbol no está vacío, son:

1. Recorrer el subárbol izquierdo (**I**) en postorden.
2. Recorrer el subárbol derecho (**D**) en postorden.
3. Visitar el nodo raíz (**N**).

Si se utiliza el recorrido *postorden* del árbol de la Figura 16.27 se visita primero el subárbol izquierdo de `A`. Este subárbol consta de los nodos `B`, `D` y `E`, y siguiendo el orden `IDN`, se visitará primero `D` (izquierdo), luego `E` (derecho) y, por último, `B` (nodo). A continuación, se visita el subárbol derecho de `A` que consta de los nodos `C`, `F` y `G`. Siguiendo el orden `IDN` para este árbol, se visita primero `F` (izquierdo), después `G` (derecho) y, por último, `C` (nodo). Finalmente se visita el nodo raíz `A`. Resumiendo, el orden del recorrido *postorden* del árbol de la Figura 16.27 es `D-E-B-F-G-C-A`.

EJERCICIO 16.1. Deducir el orden de los elementos en cada uno de los tres recorr idos fundamentales de los árboles binar ios siguientes.

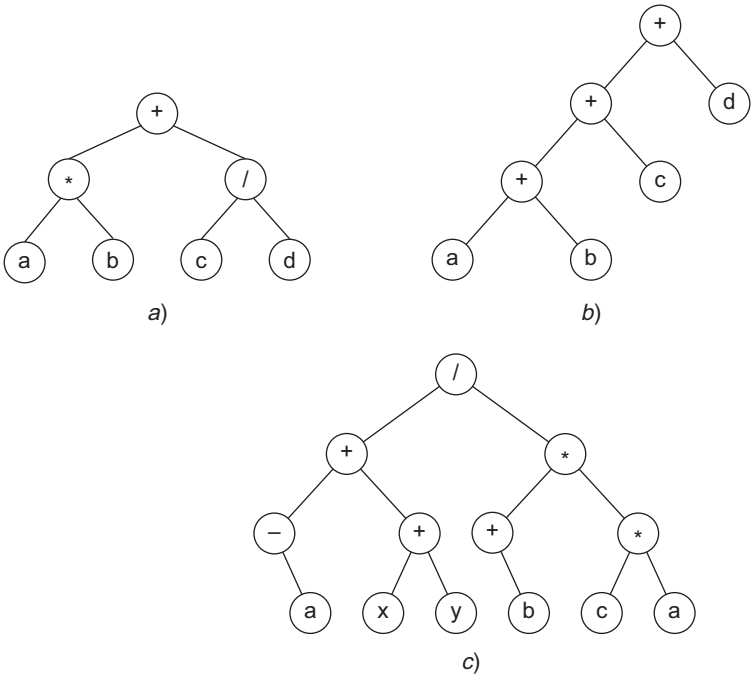


Figura 16.28. Árboles de expresión.

Los elementos de los árboles binarios listados en preorden, enorden y postorden.

	Árbol a	***Árbol b***	***Árbol c***
preorden	`+*ab/cd`	`+++abcd`	`/+-a+xy*+b*cd`
enorden	`a*b+c/d`	`a+b+c+d`	`-a+x+y/+b*c*d`
postorden	`ab*cd/+`	`ab+c+d+`	`a-xy++b+cd**/`

16.5.4. Implementación

Teniendo en cuenta que los recorridos en profundidad de un árbol binario se han definido recursivamente, las funciones que lo implementan es natural que tengan naturaleza recursiva. Prácticamente, todo consiste en trasladar la definición a la codificación. Las funciones se han de declarar en la clase `ArbolBinario` de modo público. Estas funciones `preorden()`, `inorden()`, `postorden()` deben invocar a las funciones privados de la misma clase `ArbolBinario` que tienen como argumento el atributo `raiz` que es un puntero a la clase `Nodo`. El *caso*

base, para detener la recursión, de estas funciones miembro privadas es que la raíz del árbol esté vacía (`raiz == NULL`).

```
class ArbolBinario
{

protected:

  //

public:

  // recorrido en preorden

  void preorden()
  {
    preorden(raiz);
  }

  // recorrido en ineorden

  void inorden()
  {
    inorden(raiz);
  }

  // recorrido en postorden

  void postorden()
  {
    postorden(raiz);
  }

private:

      // Recosrrido de un árbol binario en preorden
  void preorden(Nodo *r)
  {
      if (r != NULL)
      {
        r->visitar();
        preorden (r->subarbolIzdo());
        preorden (r->subarbolDcho());
      }
  };

      // Recorrido de un árbol binario en inorden

      void inorden(Nodo *r)
  {
      if (r != NULL)
      {
          inorden (r->subarbolIzdo());
          r->visitar();
          inorden (r->subarbolDcho());
      }
  }
```

```
  // Recorrido de un árbol binario en postorden
  void postorden(Nodo *r)
  {
    if (r != NULL)
    {
      postorden (r->subarbolIzdo());
      postorden (r->subarbolDcho());
      r->visitar();
    }
  }

  // Recorrido de un árbol binario en preorden
  void preorden(Nodo *r)
  {
    if (r != NULL)
    {
        r->visitar();
        preorden (r->subarbolIzdo());
        preorden (r->subarbolDcho());
    }
  }
}
```

Nota de programación

La visita al nodo se representa mediante la llamada al método de `Nodo, visitar()`. ¿Qué hacer en el método?, depende de la aplicación que se esté realizando . Si simplemente se quiere listar los nodos, puede emplearse la siguiente sentencia:

```
void visitar()
{
   cout << dato << endl;
}
```

EJEMPLO 16.7. Dado un árbol binario, eliminar cada uno de los nodos de que consta.

La función `vaciar()` (clase `ArbolBinario`) utiliza un recorrido *postorden*, de tal forma que el hecho de *visitar* el nodo se convierte en liberar la memoria del nodo con el operador `delete`. Este recorrido asegura la liberación de la memoria ocupada por un nodo después de hacerlo a su rama izquierda y derecha.

La función `vaciar(Nodo*r)` es una función miembro privada de la clase `ArbolBinario`. La codificación es:

```
void ArbolBinario::vaciar(Nodo *r)
{
    if (r != NULL)
    {
      vaciar(r->subarbolIzdo());
      vaciar(r->subarbolDcho());
      cout << "\tNodo borrado. ";
      r = NULL;
    }
}
```

La codificación de `vaciar()` de la clase `ÁrbolBinario` es:

```
void ArbolBinario::vaciar()
{
    vaciar(raiz);
}
```

16.6. IMPLEMENTACIÓN DE OPERACIONES

Se implementan propiedades y operaciones de árboles binarios. Ya que un árbol binario es un tipo de dato definido recursivamente, las funciones que implementan las operaciones tienen naturaleza recursiva, aunque siempre es posible la implementación iterativa. Las funciones que se escriben son de la clase `ArbolBinario`, tienen como argumento, normalmente, el nodo raíz del árbol o subárbol actual.

Altura de un árbol binario

El caso más sencillo de cálculo de la altura es cuando el árbol está vacío, en cuyo caso la altura es `0`. Si el árbol no está vacío, cada subárbol debe tener su propia altura, por lo que se necesita evaluar cada una por separado. Las variables `alturaIz`, `alturaDr` almacenan las alturas de los subárboles izquierdo y derecho respectivamente.

El método de cálculo de la altura de los subárboles utiliza llamadas recursivas a `altura()` con referencias a los respectivos subárboles como parámetros de la misma. Devuelve la altura del subárbol más alto más `1` (la misma del raíz).

```
int altura(Nodo *raiz)
{
  if (raiz == NULL)
    return 0 ;
  else
  {
    int alturaIz = altura (raiz->subarbolIzdo());
    int alturaDr = altura (raiz->subarbolDcho());
    if (alturaIz > alturaDr)
      return alturaIz + 1;
    else
      return alturaDr + 1;
  }
}
```

La función tiene complejidad lineal, *O(n)*, para un árbol de *n* nodos.

Árbol binario lleno

Un árbol binario lleno tiene el máximo número de nodos; esto equivale a que cumpla la propiedad de que todo nodo, excepto si es hoja, tiene dos descendientes. También, un árbol está lleno si para todo nodo, la altura de su rama izquierda es igual que la altura de su rama derecha. Con recursividad la condición es fácl de determinar:

```
bool arbolLleno(Nodo *raiz)
{
  if (raiz == NULL)
```

```
        return true;
    else
    if (altura(raiz->subarbolIzdo())!=
        altura(raiz->subarbolDcho()))
        return false;

    return arbolLleno(raiz->subarbolIzdo()) &&
           arbolLleno(raiz->subarbolDcho());
}
```

La función tiene *peor* tiempo de ejecución que `altura()`. Cada llamada recursiva a `arbolLleno()` implica llamar a `altura()` desde el siguiente nivel del árbol. La complejidad es *O(n log n)*, para un árbol lleno de `n` nodos.

Número de nodos

El número de nodos es `1` (nodo raíz) más el número de nodos del subárbol izquierdo y derecho. El número de nodos de un árbol vacío es `0`, que será *el caso base* de la recursividad.

```
int numNodos(Nodo *raiz)
{
  if (raiz == NULL)
       return 0;
    else
      return 1 + numNodos(raiz->subarbolIzdo()) +
                 numNodos(raiz->subarbolDcho());
}
```

El método accede a cada nodo del árbol, por ello tiene complejidad lineal, *O(n)*.

Copia de un árbol binario

El planteamiento es sencillo, se empieza creando una copia del nodo raíz, y se enlaza con la copia de su rama izquierda y derecha respectivamente. La función que implementa la operación, de la clase `ArbolBinario`, tiene como parámetro un puntero a la clase `Nodo` y devuelve la referencia al nodo raíz. Además, se escribe la función `copiaArbol(ArbolBinario &a)` de la clase `Arbolbinario` que tiene como parámetro un árbol binario por referencia que es donde se copia el objeto actual.

```
void ArbolBinario::copiaArbol(ArbolBinario &a)
{
     a.Praiz(copiaArbol(this->raiz));
}

Nodo* ArbolBinario::copiaArbol(Nodo* raiz)
{
    Nodo *raizCopia;
    if (raiz == NULL)
      raizCopia = NULL;
    else
    {
      Nodo* izdoCopia, *dchoCopia;
      izdoCopia = copiaArbol(raiz->subarbolIzdo());
```

```
        dchoCopia = copiaArbol(raiz->subarbolDcho());
        raizCopia = new Nodo( izdoCopia,raiz->valorNodo(), dchoCopia);
      }
      return raizCopia;
  }
```

Al igual que la función `altura()`, o `numNodos()`, se accede a cada nodo del árbol, por ello es de complejidad lineal, *O(n)*.

16.6.1. Evaluación de un árbol de expresión

En la Sección 16.5 se han construido diversos árboles binarios para representar expresiones algebraicas. La evaluación consiste en, una vez dados valores numéricos a los operandos, obtener el valor resultante de la expresión.

El algoritmo evalúa expresiones con operadores algebraicos binarios: `+`, `-`, `*`, `/` y *potenciación* (`^`). Se basa en el recorrido del árbol de la expresión en *postorden*. De esta forma se evalúa primer operando (*rama izquierda*), segundo operando (*rama derecha*) y, según el operador (nodo raíz), se obtiene el resultado.

A tener en cuenta

Una cuestión importante a considerar es que los operandos siempre son nodos hoja en el árbol binario de expresiones.

Los operandos se representan mediante letras mayúsculas. Por ello, el vector `operandos[]` tiene tantos elementos como letras mayúsculas tiene el código `ASCII`, de tal forma que la posición `0` se corresponde con `'A'`, y así sucesivamente hasta la `'Z'`. El valor numérico de un operando se encuentra en la correspondiente posición del vector.

```
double operandos[26];
```

Por ejemplo, si los operandos son `A`, `D` y `G`; sus respectivos valores se guardan en las posiciones: `operandos[0]`, `operandos[3]` y `operandos[6]`, respectivamente.

El método `evaluar()` tiene como entrada la raíz del árbol y el array con los valores de los operandos. Devuelve el resultado de la evaluación.

```
double evaluar(ArbolBinario a, double operandos[])
{
  double x, y;
  char ch ;
  Nodo *raiz;
  raiz = a.Oraiz();
  if (raiz != NULL)      // no está vacío
  {
    ch = raiz->valorNodo();
    if (ch >= 'A' && ch <= 'Z')
        return operandos[ch - 'A'];
    else
    {
        x = evaluar(raiz->subarbolIzdo(), operandos);
```

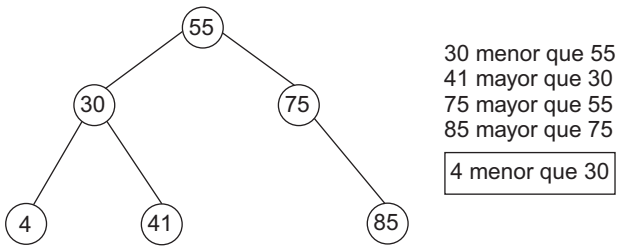
```
            y = evaluar(raiz->subarbolDcho(), operandos);
            switch (ch) {
              case '+': return x + y;
                        break;
              case '-': return x - y;
                        break;
              case '*': return x * y;
                        break;
              case '/': if (y != 0)
                            return x/y;
                        else
                            throw "Error: división por 0";
                        break;
              case '^': return pow(x, y);
            }
          }
        }
    }
```

16.7. ÁRBOL BINARIO DE BÚSQUEDA

Los árboles estudiados hasta ahora no tienen un orden definido; sin embargo, los árboles binarios ordenados tienen sentido. Estos árboles se denominan árboles binarios de búsqueda, debido a que se pueden buscar en ellos un término utilizando un algoritmo de búsqueda binaria similar al empleado en arrays.

Un **árbol binario de búsqueda** es aquel que dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos. El árbol binario del Ejemplo 16.8 es de búsqueda.

EJEMPLO 16.8. Árbol binario de búsqueda para nodos con el campo de datos de tipo int.

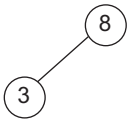


16.7.1. Creación de un árbol binario de búsqueda

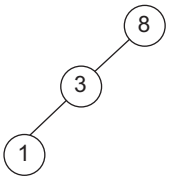
Se desea almacenar los números `8 3 1 20 10 5 4` en un árbol binario de búsqueda, siguiendo la regla: "dado un nodo en el árbol todos los datos a su izquierda deben ser menores que el dato del nodo actual, mientras que todos los datos a la derecha deben ser mayores que

el del nodo actual". Inicialmente, el árbol está vacío y se desea insertar el `8`. La única elección es almacenar el `8` en el raíz:

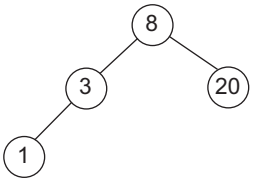
A continuación viene el `3`. Ya que `3` es menor que `8`, el `3` debe ir en el subárbol izquierdo.



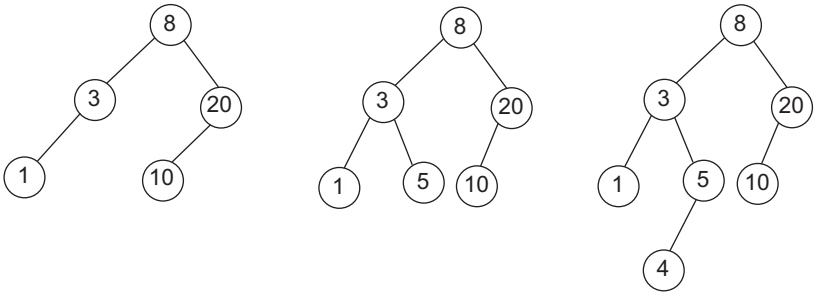
A continuación se ha de insertar `1` que es menor que `8` y que `3`, por consiguiente, irá a la izquierda y debajo de `3`.



El siguiente número es `20`, mayor que `8`, lo que implica debe ir a la derecha de `8`.



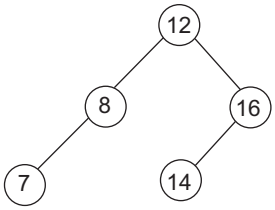
Cada nuevo elemento se inserta como una *hoja* del árbol. Los restantes elementos se pueden situar fácilmente.



Una propiedad de los árboles binarios de búsqueda es que no son únicos para los mismos datos.

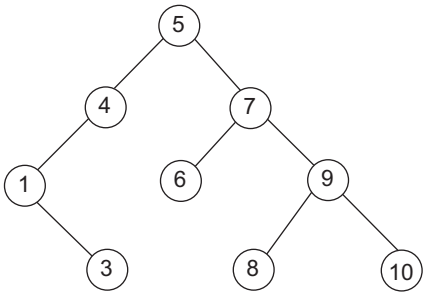
EJEMPLO 16.10. Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 14.

Solución



EJEMPLO 16.11. Construir un árbol binario de búsqueda que corresponda a un recorrido enorden cuyos elementos son: 1, 3, 4, 5, 6, 7, 8, 9 y 10.

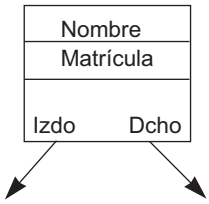
Solución



16.7.2. Nodo de un árbol binario de búsqueda

Un nodo de un árbol binario de búsqueda no difiere en nada de los nodos de un árbol binario, tiene un atributo de datos y dos enlaces a los subárboles izquierdo y derecho respectivamente. Un árbol de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. Un ejemplo de árbol binario de búsqueda es el que cada nodo contiene información relativa a un estudiante. Cada nodo almacena el nombre del estudiante, y el número de matrícula en su universidad (dato entero), que puede ser el utilizado para ordenar.

Declaración de tipos




```
class Nodo {
  protected:
```

```
        int nummat;
        char nombre[30];
        Nodo *izdo, *dcho;

  public:
  //...
};

class ArbolBinario
{
  protected:
      Nodo *raiz;
  public:
  //...
};
```

16.8. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

Los árboles binarios de búsqueda, al igual que los árboles binarios, tienen naturaleza recursiva y, en consecuencia, las operaciones sobre los árboles son recursivas, si bien siempre se tiene la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo. Devuelve la referencia al nodo del árbol, o `NULL`.
- *Inserción* de un nodo. Crea un nodo con su dato asociado y lo añade, en orden, al árbol.
- *Borrado* de un nodo. Busca el nodo del árbol que contiene un dato y lo quita del árbol. El árbol debe seguir siendo de búsqueda.
- *Recorrido* de un árbol. Los mismos recorridos de un árbol binario *preorden*, *inorden* y *postorden*.

16.8.1. Búsqueda

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

1. La clave buscada se compara con la clave del nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecha. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

A recordar

En los árboles binarios ordenados la búsqueda de una clave da lugar a un *camino de búsqueda*, de tal forma que *baja* por la rama izquierda si la clave buscada es menor que la clave del raíz, *baja* por la rama derecha si la clave es mayor.

Implementación

Si se desea encontrar un nodo en el árbol que contenga una información determinada. La función pública `buscar()` de la Clase `ArbolBinario` realiza una llamada la función privada `buscar()` de la clase `ArbolBinario` que tiene dos parámetros, un puntero a un `Nodo` y el dato que se busca. Como resultado, la función devuelve un puntero al nodo en el que se almacena la información; en el caso de que la información no se encuentre se devuelve el valor `NULL`. El algoritmo de búsqueda es el siguiente:

1. Si el nodo raíz contiene el dato buscado, la tarea es fácil: el resultado es, simplemente, su referencia y termina el algoritmo.
2. Si el árbol no está vacío, el subárbol específico por donde proseguir depende de que el dato requerido sea menor o mayor que el dato del raíz.
3. El algoritmo termina si el árbol está vacío, en cuyo caso devuelve `NULL`.

Código de la función pública `buscar()`de la clase `ArbolBinario`

```
Nodo* ArbolBinario::buscar(Tipoelemento buscado)
{
   return buscar(raiz, buscado);
}
```

Código de la función pública `buscar()`de la clase `ArbolBinario`

```
Nodo* arbolBinario::buscar(Nodo* raizSub, Tipoelemento buscado)
{
   if (raizSub == NULL)
      return NULL;
   else if (buscado == raizSub->valorNodo())
      return raizSub;
   else if (buscado < raizSub->valorNodo())
      return buscar(raizSub->subarbolIzdo(), buscado);
   else
      return buscar (raizSub->subarbolDcho(), buscado);
}
```

El Ejemplo 16.12 implementa la operación de búsqueda con un esquema iterativo.

EJEMPLO 16.12. Aplicar el algoritmo de búsqueda de un nodo en un árbol binario ordenado para implementar la operación buscar iterativamente.

La función privada de la clase `ArbolBinario`, se codifica con un bucle cuya condición de parada es que se encuentre el nodo, o bien que el *camino de búsqueda* haya finalizado (subárbol vacío, `NULL`). La función *mueve* por el árbol el puntero a la clase `Nodo raizSub`, de tal forma que baja por la rama izquierda o derecha según la clave sea menor o mayor que la clave del nodo actual.

```
Nodo* ArbolBinario:: buscarIterativo (Tipoelemento buscado)
 {
  Tipoelemento dato;
  bool encontrado = false;
```

```
    Nodo* raizSub = raiz;
    dato = buscado;
    while (!encontrado && raizSub != NULL)
    {
      if (dato == raizSub->valorNodo())
        encontrado = true;
      else if (dato < raizSub->valorNodo())
        raizSub = raizSub->subarbolIzdo();
      else
        raizSub = raizSub->subarbolDcho();
    }
    return raizSub;
  }
```

16.8.2. Insertar un nodo

Para añadir un nodo al árbol se sigue el camino de búsqueda, y al final del camimo se enlaza el nuevo nodo, por consiguiente, siempre se inserta como hoja del árbol. El árbol que resulta después de insertar sigue siendo siempre de búsqueda.

En esencia, el algoritmo de inserción se apoya en la búsqueda de un elemento, de modo que si se encuentra el elemento buscado, no es necesario hacer nada (o bien se usa una estructura de datos auxiliar para almacenar la información); en caso contrario, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en el caso de existir).

Por ejemplo, al árbol de la Figura 16.29 se le va a añadir el nodo `8`. El proceso describe un *camino de búsqueda* que comienza en el raíz `25`; el nodo `8` debe estar en el subárbol izquierdo de `25` (`8 < 25`). El nodo `10` es el raíz del subárbol actual, el nodo `8` debe estar en el subárbol izquierdo (`8 < 10`), que está actualmente vacío y, por tanto, ha terminado el *camino de búsqueda*. El nodo `8` se enlaza como hijo izquierdo del nodo `10`.

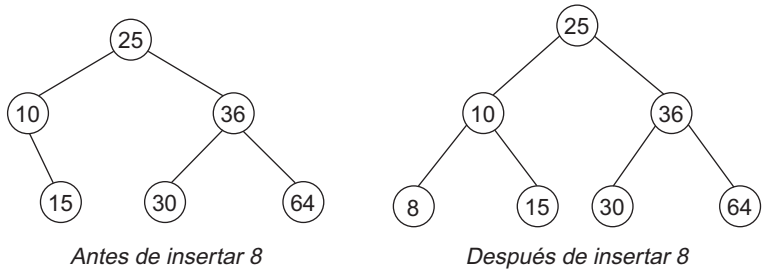
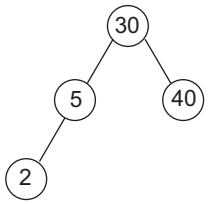


Figura 16. 29. Inserción en un árbol binario de búsqueda.

A recordar

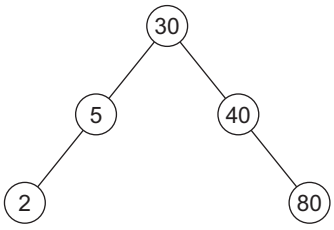
La inserción de un nuevo nodo en un árbol de búsqueda siempre se hace como nodo hoja. Para ello se *baja* por el árbol según el *camino de búsqueda.*

EJEMPLO 16.13. Insertar un elemento con cla ve 80 en el árbol binar io de búsqueda siguiente:

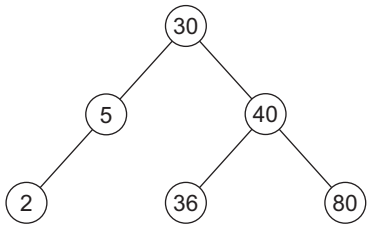


A continuación, insertar un elemento con clave 36 en el árbol binario de búsqueda resultante.

Solución



a) Inserción de 80



b) Inserción de 36

Implementación

La función `insertar()` de la clase `ArbolBinario` es el interfaz de la operación, llama la función recursiva que realiza la operación y devuelve la raíz del nuevo árbol. A esta función interna se le pasa la raíz actual, a partir de ella describe el *camino de búsqueda* y, al final, se enlaza. En un árbol binario de búsqueda no hay nodos duplicados, por ello si se encuentra un nodo igual que el que se desea insertar se lanza un excepción (o una nueva función que inserte en una estructura de datos auxiliar del propio nodo).

```
void ArbolBinario::insertar (Tipoelemento valor)
{
  raiz = insertar(raiz, valor);
}
Nodo* ArbolBinario::insertar(Nodo* raizSub, Tipoelemento dato)
{
  if (raizSub == NULL)
    raizSub = new Nodo(dato);
  else if (dato < raizSub->valorNodo())
  {
    Nodo *iz;
    iz = insertar(raizSub->subarbolIzdo(), dato);
    raizSub->ramaIzdo(iz);
  }
  else if (dato > raizSub->valorNodo())
```

```
    {
      Nodo *dr;
      dr = insertar(raizSub->subarbolDcho(), dato);
      raizSub->ramaDcho(dr);
    }
    else
      throw "Nodo duplicado"; // tratamiento de repetición
    return raizSub;
  }
```

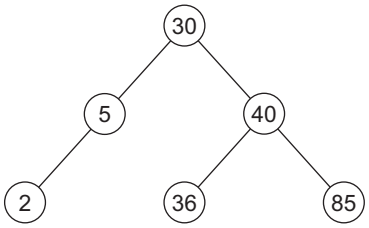
16.8.3. Eliminar un nodo

La operación de *borrado* de un nodo es también una extensión de la operación de búsqueda, si bien más compleja que la inserción, debido a que el nodo a suprimir puede ser cualquiera y la operación debe mantener la estructura de árbol binario de búsqueda después de quitar el nodo. Los pasos a seguir son:

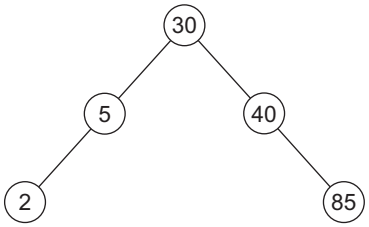
1. Buscar en el árbol la posición de "nodo a eliminar".
2. Si el nodo a suprimir tiene menos de dos hijos, reajustar los enlaces de su antecesor.
3. Si el nodo tiene dos hijos (rama izquierda y derecha), es necesario subir a la posición que éste ocupa el dato más próximo de sus subárboles (el inmediatamente superior o el inmediatamente inferior) con el fin de mantener la estructura árbol binario de búsqueda.

Los Ejemplos 16.14 y 16.15 muestran estas dos circunstancias, el primero elimina un nodo sin descendientes, el segundo elimina un nodo que, a su vez, es el raíz de un árbol con dos ramas no vacías.

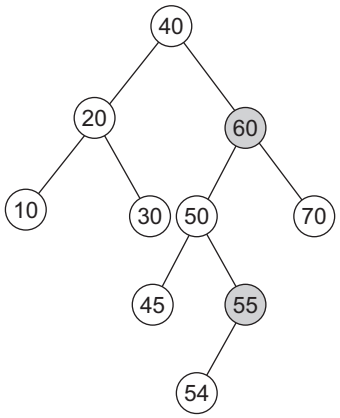
EJEMPLO 16.14. Suprimir el elemento de clave 36 del siguiente árbol binario de búsqueda:



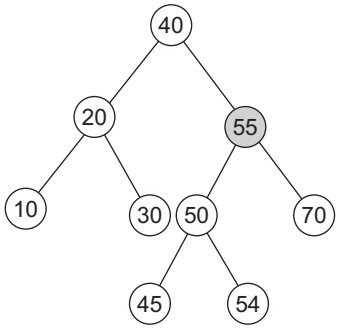
El nodo del árbol donde se encuentra la clave 36 es una hoja, por ello simplemente se reajustan los enlaces del nodo precedente en el camino de búsqueda. El árbol resultante:



EJEMPLO 16.15. Borrar el elemento de clave 60 del siguiente árbol:



Se reemplaza `60` por el elemento mayor (`55`) en su subárbol izquierdo, o por el elemento más pequeño (`70`) en su subárbol derecho. Si se opta por reemplazar por el mayor del subárbol izquierdo, se mueve el `55` al raíz del subárbol y se reajusta el árbol.

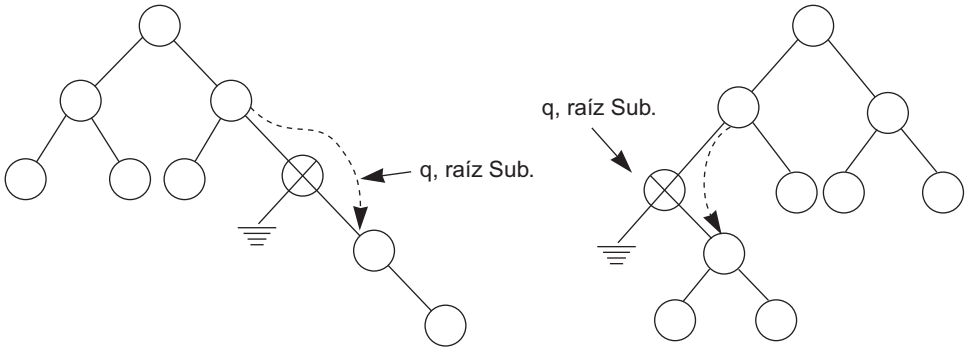


Implementación

La función `eliminar()` de la clase `ArbolBinario` es el interfaz de la operación, se le pasa el elemento que se va a buscar en el árbol para retirar su nodo; llama al método sobrecargado, privado, `eliminar()` con la raíz del árbol y el elemento.

La función lo primero que hace es buscar el nodo, siguiendo el *camino de búsqueda*. Una vez encontrado, se presentan dos casos claramente diferenciados. El primero, si el nodo a eliminar es una hoja o tiene un único descendiente, resulta una tarea fácil, ya que lo único que hay que hacer es asignar al enlace del nodo *padre* (según el *camino de búsqueda*) el descendiente del nodo a eliminar. El segundo caso, que el nodo tenga las dos ramas no vacías; esto exige, para mantener la estructura de árbol de búsqueda, reemplazar el dato del nodo por la *mayor de las claves menores* en el subárbol (otra posible alternativa: reemplazar el dato del nodo por la *menor de las claves mayores*). Como las claves menores están en la rama izquierda, se *baja* al primer nodo de la rama izquierda, y se continúa *bajando* por las ramas derecha (claves mayores) hasta alcanzar el nodo que no tiene rama derecha. Éste es el mayor de los menores, cuyo dato debe reemplazar al del nodo a eliminar. Lo que se hace es copiar el valor del dato y enlazar su padre con el hijo izquierdo. La función `reemplazar()` realiza la tarea descrita.

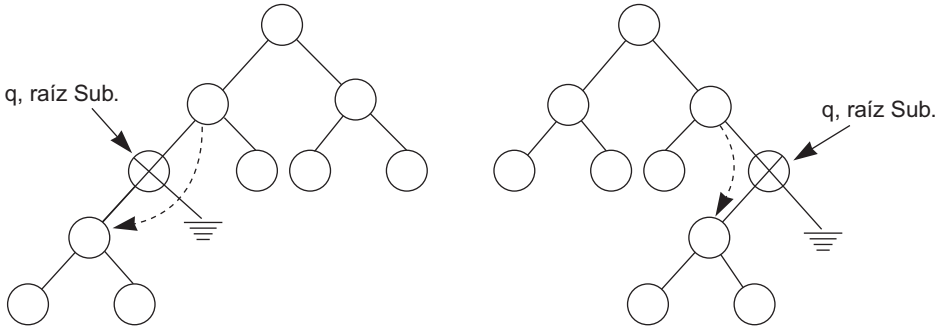
```
void ArbolBinario::eliminar (Tipoelemento valor)
{
  raiz = eliminar(raiz, valor);
}
Nodo* ArbolBinario::eliminar (Nodo *raizSub, Tipoelemento dato)
{
  if (raizSub == NULL)
    throw "No se ha encontrado el nodo con la clave";
  else if (dato < raizSub->valorNodo())
  {
    Nodo* iz;
    iz = eliminar(raizSub->subarbolIzdo(), dato);
    raizSub->ramaIzdo(iz);
  }
  else if (dato > raizSub->valorNodo())
  {
    Nodo *dr;
    dr = eliminar(raizSub->subarbolDcho(), dato);
    raizSub->ramaDcho(dr);
  }
  else     // Nodo encontrado
  {
    Nodo *q;
    q = raizSub;     // nodo a quitar del árbol
    if (q->subarbolIzdo() == NULL)
      raizSub = q->subarbolDcho();      // figura 16.30
    else if (q->subarbolDcho() == NULL)
      raizSub = q->subarbolIzdo();      // figura 16.31
    else
    {     // tiene rama izquierda y derecha
      q = reemplazar(q);      //figura 16.32
    }
    q = NULL;
  }
  return raizSub;
}
Nodo* ArbolBinario::reemplazar(Nodo* act)
{
  Nodo *a, *p;
  p = act;
  a = act->subarbolIzdo();     // rama de nodos menores
  while (a->subarbolDcho() != NULL)
  {
    p = a;
    a = a->subarbolDcho();
  }
  // copia en act el valor del nodo apuntado por a
  act->nuevoValor(a->valorNodo());
  if (p == act) // a es el hijo izquierdo de act
    p->ramaIzdo(a->subarbolIzdo()); // enlaza subarbol izquierdo. Fig. 16.32b
  else
    p->ramaDcho(a->subarbolIzdo()); // se enlaza subarbol derecho. Fig. 16.32a
     return a;
}
```



a) Es un hijo derecho

b) Es hijo derecho

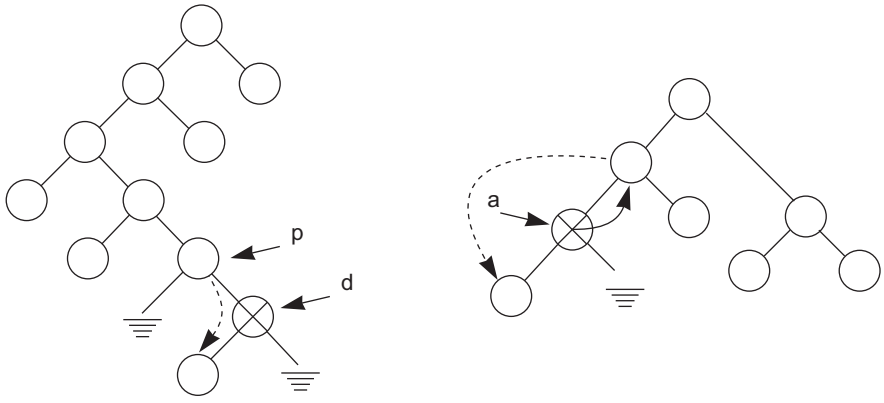
Figura 16.30. Eliminación de un nodo sin hijo izquierdo.



a) Es un hijo izquierdo

b) Es un hijo derecho

Figura 16.31. Eliminación de un nodo sin hijo derecho.



a) Se enlaza subárbol derecho

b) Se enlaza subárbol izquierdo

Figura 16.32. Eliminación de un nodo con dos hijos.

16.9. DISEÑO RECURSIVO DE UN ÁRBOL DE BÚSQUEDA

Los nodos utilizados en los árboles binarios y en los de búsqueda tienen dos enlaces, uno al nodo descendiente izquierdo y otro al derecho. Con una visión recursiva (ajustada a la propia definición de árbol) los dos enlaces pueden declararse a sendos subárboles, uno al subárbol izquierdo y el otro al subárbol derecho. La clase `ArbolBusqueda` tiene sólo un atributo `raiz` que es un puntero a la clase `Nodo`, por lo que para declararla, necesita que previamente se tenga un aviso de que la clase `Nodo` va a existir tal y como indica el siguiente código.

```
class Nodo;
class ÁrbolBusqueda
{
protected:
       Nodo* raiz;
public:
    ArbolBusqueda()
    {
      raiz = NULL;
    }
    ArbolBusqueda( Nodo *r)
    {
      raiz = r;
    }
    bool arbolVacio ()
    {
      return raiz == NULL;
    }
    void Praiz( Nodo* r){ raiz = r;}
    Nodo * Oraiz() { return raiz;}
};
```

La clase `Nodo` tiene tres atributos el dato para almacenar la información y dos atributos **arbolIzdo**, **arbolDcho** de la clase `ArbolBinario` previamente declarada.

```
typedef int Tipoelemento;

class Nodo
{
protected:
   Tipoelemento dato;
   ArbolBusqueda arbolIzdo;
   ArbolBusqueda arbolDcho;

public:
  Nodo(Tipoelemento valor)
  {
    dato = valor;
    arbolIzdo = ArbolBusqueda();// subarbol izquierdo vacío
    arbolDcho = ArbolBusqueda();// subarbol derecho vacío
  }

  Nodo(Tipoelemento valor, ArbolBusqueda ramaIzdo,
                          ArbolBusqueda ramaDcho)
```

```
  {
    dato = valor ;
    arbolIzdo = ramaIzdo;
    arbolDcho = ramaDcho;
  }
    // operaciones de acceso
  Tipoelemento valorNodo(){ return dato; }
  ArbolBusqueda subarbolIzdo(){ return arbolIzdo; }
  ArbolBusqueda subarbolDcho(){ return arbolDcho; }
  void nuevoValor(Tipoelemento d){ dato = d; }
  void ramaIzdo(ArbolBusqueda n){ arbolIzdo = n; }
  void ramaDcho(ArbolBusqueda n){ arbolIzdo = n; }
};
```

16.9.1. Implementación de las operaciones

La *búsqueda*, *inserción* y *borrado* bajan por el árbol, dando lugar al *camino de búsqueda*, con llamadas recursivas, siguiendo los detalles comentados en el Apartado 16.8 de este mismo capítulo.

Búsqueda

```
Nodo* ArbolBusqueda::buscar(Tipoelemento buscado)
{
  if (raiz == NULL)
    return NULL;
  else if (buscado == raiz->valorNodo())
    return raiz;
  else if (buscado < raiz->valorNodo())
    return raiz->subarbolIzdo().buscar(buscado);
  else
    return raiz->subarbolDcho().buscar(buscado);
}
```

Inserción

```
void ArbolBusqueda::insertar(Tipoelemento dato)
{
  if (!raiz)
    raiz = new Nodo(dato);
  else if (dato < raiz->valorNodo())
    raiz->subarbolIzdo().insertar(dato);//inserta por la izquierda
  else if (dato > raiz->valorNodo())
    raiz->subarbolDcho().insertar(dato); // inserta por la derecha
  else
    throw "Nodo duplicado";
}
```

Borrado

```
void ArbolBusqueda::eliminar (Tipoelemento dato)
{
  if (arbolVacio())
```

```
      throw "No se ha encontrado el nodo con la clave";
    else if (dato < raiz->valorNodo())
      raiz->subarbolIzdo().eliminar(dato);
    else if (dato > raiz->valorNodo())
      raiz->subarbolDcho().eliminar(dato);
    else    // nodo encontrado
    {
      Nodo *q;
      q = raiz;     // nodo a quitar del árbol
      if (q->subarbolIzdo().arbolVacio())
        raiz = q->subarbolDcho().raiz;
      else if (q->subarbolDcho().arbolVacio())
        raiz = q->subarbolIzdo().raiz;
      else
      {     // tiene rama izquierda y derecha
        q = reemplazar(q);
      }
      q->ramaIzdo(NULL);
      q->ramaDcho(NULL);
      q = NULL;
    }
  }

  Nodo* ArbolBusqueda:: reemplazar(Nodo* act)
  {
    Nodo* p;
    ArbolBusqueda a;
    p = act;
    a = act->subarbolIzdo();     // árbol con nodos menores
    while (a.raiz->subarbolDcho().arbolVacio())
    {
      p = a.raiz;
      a = a.raiz->subarbolDcho();
    }
    act->nuevoValor(a.raiz->valorNodo());
    if (p == act)
      p->ramaIzdo(a.raiz->subarbolIzdo());
    else
      p->ramaDcho(a.raiz->subarbolIzdo());
    return a.raiz;
  }
```

RESUMEN

En este capítulo se introdujo y desarrolló la estructura de datos dinámica árbol. Esta estructura, muy potente, se puede utilizar en una gran variedad de aplicaciones de programación.

La estructura árbol más utilizada normalmente es el *árbol binario*. Un **árbol binario** es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho. En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un padre.

La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana, más 1. Si el árbol A es vacío, la altura es `0`. *El nivel* o *profundidad* de un elemento es un concepto similar al de altura.

Un árbol binario no vacío está *equilibrado totalmente* si sus subárboles izquierdo y derecho tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados.

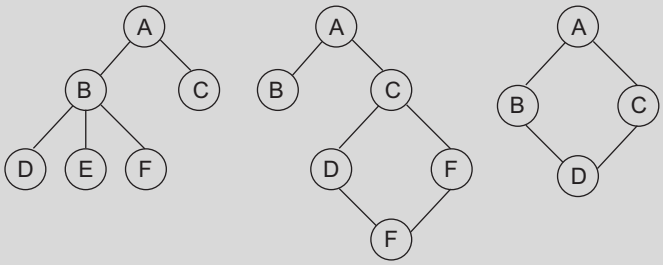
Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan, fundamentalmente, para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

BIBLIOGRAFÍA RECOMENDADA

Aho V.; Hopcroft, J., y Ullman, J.: *Estructuras de datos y algoritmos*. Addison Wesley, 1983.
Garrido, A., y Fernández, J.: *Abstracción y estructuras de datos en C++*. Delta, 2006.
Joyanes, L., y Zahonero, L.: *Algoritmos y estructuras de datos. Una perspectiva en C*. McGraw-Hill, 2004.
Joyanes, L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos en C*. Schaum, 2005.
Weis, Mark Allen: *Estructuras de datos y algoritmos*. Addison Wesley, 1992.
Wirth Niklaus: *Algoritmos + Estructuras de datos = programas*. 1986.

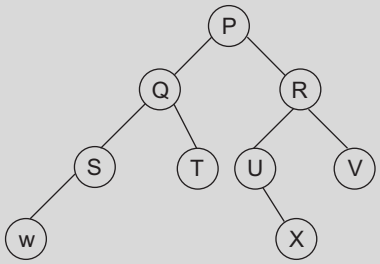
EJERCICIOS

16.1. Explicar porqué cada una de las siguientes estructuras no es un árbol binario.



16.2. Considérese el árbol siguiente:

a) ¿Cuál es su altura?
b) ¿Está el árbol equilibrado? ¿Porqué?
c) Listar todos los nodos hoja.
d) ¿Cuál es el predecesor inmediato (padre) del nodo `U`.
e) Listar los hijos del nodo `R`.
f) Listar los sucesores del nodo `R`.



16.3. Para cada una de las siguientes listas de letras.

a) Dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado.
b) Realizar recorridos enorden, preorden y postorden del árbol y mostrar la secuencia de letras que resultan en cada caso.

(i) `M, Y, T, E, R` **(iii)** `R, E, M, Y, T`
(ii) `T, Y, M, E, R` **(iv)** `C, O, R, N, F, L, A, K, E, S`

16.4. En el árbol del Ejercicio 16.2, recorrer cada árbol utilizando los órdenes siguientes: NDI, DNI, DIN.

16.5. Dibujar los árboles binarios que representan las siguientes expresiones:

a) `(A+B)/(C-D)`
b) `A+B+C/D`
c) `A-(B-(C-D)/(E+F))`
d) `(A+B)*((C+D)/(E+F))`
e) `(A-B)/((C*D)-(E/F))`

16.6. El recorrido preorden de un cierto árbol binario produce

```
ADFGHKLPQRWZ
```

y en recorrido *enorden* produce

```
GFHKDLAWRQPZ
```

Dibujar el árbol binario.

16.7. Escribir una función recursiva que cuente las hojas de un árbol binario.

16.8. Escribir una función que determine el número de nodos que se encuentran en el nivel n de un árbol binario.

16.9. Escribir una función que tome un árbol como entrada y devuelva el número de hijos del árbol.

16.10. Escribir una función *booleana* a la que se le pase un referencia a un árbol binario y devuelva verdadero (*true*) si el árbol es completo y falso (*false*) en caso contrario.

16.11. Se dispone de un árbol binario de elementos de tipo entero. Escribir métodos que calculen:

a) La suma de sus elementos.
b) La suma de sus elementos que son múltiplos de 3.

16.12. Diseñar una función iterativa que encuentre el número de nodos hoja en un árbol binario.

16.13. En un árbol de búsqueda cuyo campo clave es de tipo entero, escribir una función que devuelva el número de nodos cuya clave se encuentra en el rango [*x1*, *x2*].

16.14. Diseñar una función que visite los nodos del árbol por niveles; primero el nivel 0, después los nodos del nivel 1, nivel 2 y así hasta el último nivel.

PROBLEMAS

16.1. Se dispone de un archivo de texto en el que cada línea contiene la siguiente información

Columnas	1-20	Nombre
	21-31	Número de la Seguridad Social
	32-78	Dirección

Escribir un programa que lea cada registro de datos y lo inserte en un árbol, de modo que cuando el árbol se recorra en *inorden*, los números de la seguridad social se ordenen en orden ascendente.

16.2. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición. Hacer uso de la estructura árbol binario de búsqueda, cada nodo del árbol que tenga una palabra y su frecuencia.

16.3. Escribir un programa que procese un árbol binario cuyos nodos contengan caracteres y a partir del siguiente menú de opciones:

`I` (seguido de un carácter)	: Insertar un carácter
`B` (seguido de un carácter)	: Buscar un carácter
`RE`	: Recorrido en orden
`RP`	: Recorrido en preorden
`RT`	: Recorrido postorden
`SA`	: Salir

16.4. Escribir una función boleana `identicos()` que permita decir si dos árboles binarios son iguales.

16.5. Construir una función en la clase `ArbolBinarioBusqueda` que encuentre el nodo máximo.

16.6. Construir una función recursiva para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado (el campo clave es de tipo entero).

16.7. Escribir una función que determine la altura de un nodo. Escribir un programa que cree un árbol binario con números generados aleatoriamente y muestre por pantalla:

- La altura de cada nodo del árbol.
- La diferencia de altura entre rama izquierda y derecha de cada nodo.

16.8. Diseñar funciones no recursivas que listen los nodos de un árbol en inorden, preorden y postorden.

16.9. Dados dos árboles binarios `A` y `B` se dice que son *parecidos* si el árbol `A` puede ser transformado en el árbol B intercambiando los hijos izquierdo y derecho (de alguno de sus nodos). Escribir una función que determine dos árboles son *parecidos.*

16.10. Dado un árbol binario de búsqueda construir su árbol espejo. Árbol espejo es el que se construye a partir de uno dado, convirtiendo el subárbol izquierdo en subárbol derecho y viceversa.

16.11. Un árbol binario de búsqueda puede implementarse con un array. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición *i* del array, su hijo izquierdo se encuentra en la posición *2*i* y su hijo derecho en la posición *2*i + 1*. Diseñar a partir de esta representación las funciones con las operaciones correspondientes para gestionar interactivamente un árbol de números enteros.

16.12. Dado un árbol binario de búsqueda diseñe una función que liste los nodos del árbol ordenados descendentemente.

CAPÍTULO 17

Árboles de búsqueda equilibrados. Árboles B

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer la eficiencia de un árbol de búsqueda.
- Construir un árbol binario equilibrado conociendo el número de claves.
- Construir un árbol binario de búsqueda equilibrado.
- Describir los diversos tipos de movimientos que se hacen cuando se desequilibra un árbol.
- Diseñar y declarar la clase ArbolEquilibrado.
- Conocer las características de los árboles B.
- Utilizar la estructura de árbol B para organizar búsquedas eficientes en bases de datos.
- Implementar la operación de búsqueda de una clave en un árbol B.
- Conocer la estrategia que sigue el proceso de inserción de una clave en un árbol B.
- Implementar las operaciones del TAD árbol B en C++.

Contenido

17.1. Eficiencia de la búsqueda en un árbol ordenado.
17.2. Árbol binario equilibrado, árbol AVL.
17.3. Inserción en árboles de búsqueda equilibrados. Rotaciones.
17.4. Implementación de la operación *inserción con balanceo y rotaciones.*
17.5. Definición de un árbol B.
17.6. *TAD árbol B* y representación.
17.7. Formación de un árbol B.
17.8. Búsqueda de una clave en un árbol B.
17.9. Inserción en un árbol B.
17.10. Listado de las claves de un árbol B.

RESUMEN.
BIBLIOGRAFÍA RECOMENDADA.
EJERCICIOS.
PROBLEMAS.
ANEXOS A y B en página web del libro (lectura recomendada para profundizar en el tema).

Conceptos clave

- Altura de un árbol.
- Árbol de búsqueda.
- Árboles B.
- Camino de búsqueda.
- Complejidad logarítmica.
- Equilibrio.
- Factor de equilibrio.
- Hoja.
- Rotaciones.

INTRODUCCIÓN

En el Capítulo 16 se introdujo el concepto de árbol binario. Se utiliza un árbol binario de búsqueda para almacenar datos organizados jerárquicamente. Sin embargo, en muchas ocasiones, las inserciones y eliminaciones de elementos en el árbol no ocurren en un orden predecible; es decir, los datos no están organizados jerárquicamente.

En este capítulo se estudian tipos de árboles adicionales: los árboles equilibrados o árboles AVL, como también se les conoce, y los árbolesB.

El concepto de árbol equilibrado así como los algoritmos de manipulación son el motivo central de este capítulo. Los métodos que describen este tipo de árboles fueron descritos en 1962 por los matemáticos rusos G. M. Adelson - Velskii y E. M. Landis.

Los árboles B se utilizan para la creación de bases de datos. Así, una forma de implementar los índices de una base de datos relacional es a través de un árbol B.

Otra aplicación dada a los árboles B es la gestión del sistema de archivos de los sistemas operativos, con el fin de aumentar la eficacia en la búsqueda de archivos por los subdirectorios.

También se conocen aplicaciones de los árboles B en sistemas de comprensión de datos. Bastantes algoritmos de comprensión utilizan árboles B para la búsqueda por claves de datos comprimidos.

17.1. EFICIENCIA DE LA BÚSQUEDA EN UN ÁRBOL ORDENADO

La eficiencia de una búsqueda en un árbol binario ordenado varía entre `O(n)` y `O(log(n)),` dependiendo de la estructura que presente el árbol.

Si los elementos son añadidos en el árbol mediante el algoritmos de inserción expuesto en el capítulo anterior, la estructura resultante del árbol dependerá del orden en que sean añadidos. Así, si todos los elementos se insertan en orden creciente o decreciente, el árbol va a tener todas la ramas izquierda o derecha, respectivamente, vacías. Entonces, la búsqueda en dicho árbol será totalmente secuencial.

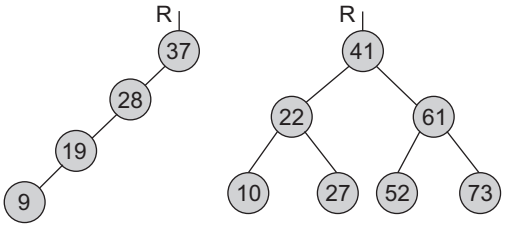


Figura 17.1. Árbol degenerado y equilibrado de búsqueda.

Sin embargo, si la mitad de los elementos insertados después de otro con clave `K` tienen claves menores de `K` y la otra mitad claves mayores de `K,` se obtiene un árbol equilibrado (también llamado balanceado), en el cual las comparaciones para obtener un elemento son como máximo $Log_2(n)$, para un árbol de n nodos.

En los árboles de búsqueda el número promedio de comparaciones que debe de realizarse para las operaciones de inserción, eliminación y búsqueda varía entre $Log_2(n)$, para el mejor de los casos, y `n` para el *peor de los casos*. Para optimizar los tiempos de búsqueda en los árboles ordenados surgen los árboles casi equilibrados, en los que la complejidad de la búsqueda es logarítmica, `O(Log (n))`.

17.2. ÁRBOL BINARIO EQUILIBRADO, ÁRBOLES AVL

Un árbol totalmente equilibrado se caracteriza por que la altura de la rama izquierda es igual que la altura de la rama derecha para cada uno de los nodos del árbol. Es un árbol ideal, no siempre se puede conseguir que el árbol esté totalmente balanceado.

La estructura de datos de árbol equilibrado que se utiliza es la árbol **AVL**. El nombre es en honor de Adelson-Velskii-Landis que fueron los primeros científicos en estudiar las propiedades de esta estructura de datos. Son árboles ordenados o de búsqueda que, además, cumplen la condición de balanceo para cada uno de los nodos.

Definición

Un árbol equilibrado o *árbol AVL* es un árbol binario de búsqueda en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren como máximo en 1.

La Figura 17.2 muestra dos árboles de búsqueda, el de la izquierda está equilibrado. El de la derecha es el resultado de insertar la clave 2 en el anterior, según el algoritmo de inserción en árboles de búsqueda. La inserción provoca que se *viole* la condición de equilibrio en el nodo raíz del árbol.

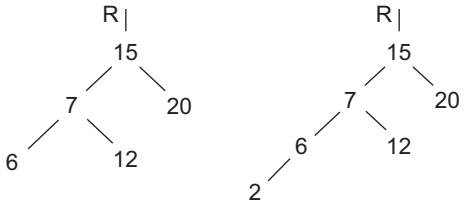


Figura 17.2. Dos árboles de búsqueda, el de la izquierda equilibr ado, el otro no.

La condición de equilibrio de cada nodo implica una restricción en las alturas de los subárboles de un árbol AVL. Si v_r es la raíz de cualquier subárbol de un árbol equilibrado, y `h` la altura de la rama izquierda entonces la altura de la rama derecha puede tomar los valores: `h-1`, `h`, `h+1`. Esto aconseja asociar a cada nodo el parámetro denominado *factor de equilibrio o balance de un nodo*. Se define como la altura del subárbol derecho menos la altura del subárbol izquierdo correspondiente. El factor de equilibrio de cada nodo en un árbol equilibrado puede tomar los valores: `1`, `-1` o `0`. La Figura 17.3 muestra un árbol balanceado con el factor de equilibrio de cada nodo.

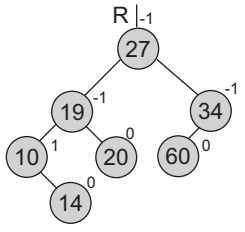


Figura 17.3. Árbol equilibrado con el factor de equilibrio de cada nodo.

A tener en cuenta

La altura o profundidad de un árbol binario es el nivel máximo de sus hojas mas uno. La altura de un árbol nulo se considera cero.

17.2.1. Altura de un árbol equilibrado, árbol AVL

No resulta fácil determinar la altura promedio de un árbol AVL, por lo que se determina la altura en el *peor de los casos*, es decir, la altura máxima que puede tener un árbol equilibrado con un número de nodos n. La altura es un parámetro importante ya que coincide con el número de iteraciones que se realizan para *bajar* desde el nodo raíz al nivel más profundo de las hojas. La eficiencia de los algoritmos de búsqueda, inserción y borrado depende de la altura del árbol AVL.

Para calcular la altura máxima de un árbol AVL de n nodos se parte del siguiente razonamiento: *¿cuál es el número mínimo de nodos que puede tener un árbol binario para que se conside reequilibrado con una altura h?* Si ese árbol es A_h, tendrá dos subárboles izquierdo y derecho respectivamente, A_i y A_d cuya altura difiera en 1, supongamos que tienen de altura h-1 y h-2 respectivamente. Al considerar que A_h es el árbol de menor número de nodos de altura h, entonces A_i y A_d también son árboles AVL de menor número de nodos, pero de altura h-1 y h-2 y son designados como A_{h-1} y A_{h-2}. Este razonamiento se sigue extendiendo a cada subárbol y se obtienes árboles equilibrados como los de la Figura 17.4.

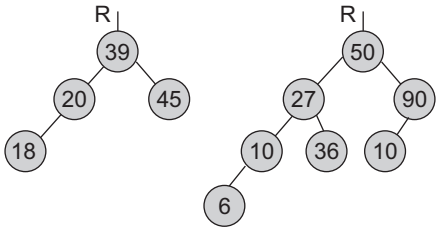


Figura 17.4. Árboles de Fibonacci.

La construcción de árboles binarios equilibrados siguiendo esta estrategia puede representarse matemáticamente:

$$A_h = A_{h-1} + A_{h-2}$$

La expresión matemática expuesta tiene una gran similitud con la ley de recurrencia que permite encontrar números de Fibonacci, $a_n = a_{n-1} + a_{n-2}$. Por esa razón, a los árboles equilibrados construidos con esta ley de formación se les conoce como *árboles de Fibonacci*.

El objetivo que se persigue es encontrar el número de nodos, n, que hace la altura máxima. El número de nodos de un árbol es la suma de los nodos de su rama izquierda, rama derecha más uno (la raíz). Si es ese número es N_h se puede escribir:

$$N_h = N_{h-1} + N_{h-2} + 1$$

donde $N_0 = 1$, $N_1 = 2$, $N_2 = 4$, $N_3 = 7$, y así sucesivamente. Se observa que los números $N_h + 1$ cumplen la definición de los números de Fibonacci. El estudio matemático de la función generadora de los números de Fibonacci permite encontrar esta relación:

$$N_h + 1 \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h$$

Tomando logaritmos se encuentra la altura h en función del número de nodos, N_h:

$h \approx 1.44 \log (N_h)$

Como conclusión, el árbol equilibrado de n nodos menos denso tiene como altura 1.44log n. Como n es el número de nodos en el *peor de los casos* de árbol AVL de altura h, se puede afirmar que la complejidad de una búsqueda es O(Log n).

A tener en cuenta

La altura de un árbol binario perfectamente equilibrado de n nodos es log n. Las operaciones que se aplican a los árboles AVL no requieren más del 44 por cien de tiempo (en el caso más desfavorable) que si se aplican a un árbol perfectamente equilibrado.

EJERCICIO 17.1. Se tienen n cla ves que se v an a organizar jerárquicamente f ormando un árbol equilibrado. Escribir un programa para formar el árbol AVL siguiendo la estrategia de los árboles de Fibonacci descr ita en la sección anter ior.

La formación de un árbol balanceado de n nodos se parece mucho a la secuencia de los números de Fibonacci:

```
a(n) = a(n-2) + a(n-1)
```

Un *árbol de Fibonacci* (árbol equilibrado) puede definirse:

1. Un árbol vacío es el árbol de Fibonacci de altura 0.
2. Un nodo único es un árbol de Fibonacci de altura 1.
3. Si A_{h-1} y A_{h-2} son árboles de Fibonacci de alturas h-1 y h-2, entonces

   $A_h = < A_{h-1}, x, A_{h-2} >$ es árbol de Fibonacci de altura h.

El número de nodos, A_h , viene dado por la sencilla relación recurrente:

$N_0 = 0$
$N_1 = 1$
$N_h = N_{h-1} + 1 + N_{h-2}$

Para conseguir un árbol AVL con un número dado, n, de nodos de mínima altura hay que distribuir equitativamente los nodos a la izquierda y a la derecha de un nodo dado. En definitiva, es seguir la relación de recurrencia anterior, que expresada recursivamente:

1. Crear nodo raíz.
2. Generar el subárbol izquierdo con $n_i = n/2$ nodos del nodo raíz utilizando la misma estrategia.

3. Generar el subárbol derecho con $n_d = n - n_i - 1$ nodos del nodo raíz utilizando la misma estrategia.

En este ejercicio, el árbol no va a ser de búsqueda. Simplemente, un árbol binario de números enteros que son leídos del teclado pero que es de mínima altura. El árbol será de búsqueda, si los números que se leen están ordenados crecientemente.

El método público `ArbolFibonacci()` de la clase `Arbolbinario` realiza una llamada al método privado `arbolFibonacci()` de la clase `Arbolbinario` que retorna un puntero a la clase `Nodo`.

```
void ArbolBinario::ArbolFibonacci(int n)
{
     raiz = arbolFibonacci(n);
}

Nodo* ArbolBinario::arbolFibonacci(int n)
{
       int nodosIz, nodosDr;
       int clave;
       Nodo *nuevoRaiz;

       if (n == 0)
         return NULL;
       else
       {
         nodosIz = n / 2;
         nodosDr = n - nodosIz - 1;
         // nodo raíz con árbol izquierdo y derecho de Fibonacci
         cin >> clave;
         nuevoRaiz = new Nodo(arbolFibonacci(nodosIz), clave,
                              arbolFibonacci(nodosDr));
       }
       return nuevoRaiz;
}
```

El método público `dibujarArbol()` de la clase `ArbolBinario`, se encarga de representar en el dispositivo estándar de salida la información almacenada en cada uno de los nodos del árbol. Usa un parámetro `h` para formatear la salida.

```
void ArbolBinario::dibujarArbol(Nodo *r, int h)
 {
      /*
        escribe las claves del arbol de fibonacci; h establece
        una separación entre nodos
      */
       int i;
       if (r != NULL)
       {
           dibujarArbol(r->subarbolIzdo(), h + 1);
           for (i = 1; i <= h; i++)
                cout << " ";
           cout <<r->valorNodo() << endl;
          dibujarArbol(r->subarbolDcho(), h + 1);
       }
}
```

El programa principal realiza las llamadas correspondientes.

```
int main()
{
   ArbolBinario arbolFib;
    int n;
    do {
       cout << "Número de nodos del árbol: ";
       cin >> n;
   } while (n <= 0);
   arbolFib.ArbolFibonacci(n);
   cout << "Árbol de Fibonacci de mínima altura:\n";
   arbolFib.dibujarArbol(arbolFib.Oraiz(), 1);
   return 0;
}
```

17.3. INSERCIÓN EN ÁRBOLES DE BÚSQUEDA EQUILIBRADOS: ROTACIONES

Los árboles equilibrados, árboles AVL, son árboles de búsqueda y, por consiguiente, para añadir un elemento se ha de seguir el mismo proceso que en los árboles de búsqueda. Se compara la nueva clave con la clave del raíz, continúa por la rama izquierda o derecha según sea menor o mayor (describe el *camino de búsqueda*), termina insertándose como nodo hoja. Esta operación, como ha quedado demostrado al determinar la altura en el peor de los casos, tiene una complejidad logarítmica. Sin embargo, la nueva inserción puede hacer que aumente la altura de una rama, de manera que cambie el factor de equilibrio del nodo raíz de dicha rama. Este hecho hace necesario que el algoritmo de inserción, *regrese* por el *camino de búsqueda* actualizando el factor de equilibrio de los nodos. La Figura 17.5 muestra un árbol equilibrado y el mismo árbol justo después de la inserción de una nueva clave que provoca que *rompa* la condición de balanceo.

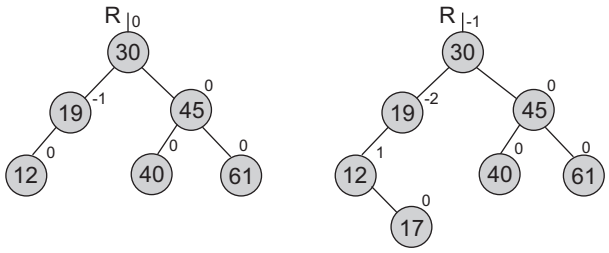


Figura 17.5. Árbol equilibrado; el mismo después de insertar la clave 17.

A recordar

Una inserción de una nueva clave, o un borrado, puede destruir el criterio de equilibrio de varios nodos del árbol. Se debe recuperar la condición de equilibrio del árbol antes de dar por finalizada la operación para que el árbol siga siendo equilibrado.

La estructura del nodo en un árbol equilibrado es una *extensión* de la declarada para un árbol binario. Para determinar si el árbol está equilibrado debe de manejarse información

relativa al balanceo o factor de equilibrio de cada nodo. Por esta razón se añade al nodo un campo más: el factor de equilibrio (`fe`). Este atributo puede tomar los valores: `-1`, `0`, `+1`.

La clase `NodoAvl` es similar a la clase `Nodo` de los árboles binarios de búsqueda. Solamente se le añade un nuevo atributo entero `fe`, a los que ya disponía la clase `Nodo`. Además, se añaden las funciones miembro encargadas de obtener el valor del nuevo atributo coma la de modificar el valor del nuevo atributo, así como los correspondientes métotos

```
class NodoAvl
{
protected:
   Tipoelemento dato;
    NodoAvl *izdo;
    NodoAvl *dcho;
    int fe;
public:
    NodoAvl(Tipoelemento valor)
    {
      dato = valor;
      izdo = dcho = NULL;
      fe = 0;
    }
    NodoAvl(Tipoelemento valor, int vfe)
    {
      dato = valor;
      izdo = dcho = NULL;
      fe = vfe;
    }
     NodoAvl(NodoAvl* ramaIzdo,Tipoelemento valor, NodoAvl* ramaDcho)
     {
       dato = valor;
       izdo = ramaIzdo;
       dcho = ramaDcho;
       fe = 0;
    }
     NodoAvl(NodoAvl* ramaIzdo, int vfe, Tipoelemento valor,
             NodoAvl* ramaDcho)
     {
       dato = valor;
       izdo = ramaIzdo;
       dcho = ramaDcho;
       fe = vfe;
    }
    // operaciones de acceso
    Tipoelemento valorNodo(){ return dato; }
    NodoAvl* subarbolIzdo(){ return izdo; }
    NodoAvl* subarbolDcho(){ return dcho; }
    void nuevoValor(Tipoelemento d){ dato = d; }
    void ramaIzdo(NodoAvl* n){ izdo = n; }
    void ramaDcho(NodoAvl* n){ dcho = n; }
    void visitar(){ cout << dato << endl;}
    void Pfe(int vfe) { fe = vfe;}
    int Ofe(){ return fe;}
};
```

Las operaciones a realizar con un árbol de búsqueda equilibrado son las mismas que con un árbol de búsqueda. Debido a los cambios del nodo, se declara la clase `ArbolAvl` sin relacionarla con `ArbolBinarioBusqueda`.

```
class ArbolAvl
{
  NodoAvl* raiz;
  ArbolAvl()
  {
    raiz = NULL;
  }
  ArbolAvl(NodoAvl * r)
  {
    raiz = r;
  }
  NodoAvl* Oraiz ()
  {
    return raiz;
  }
  void Praiz( NodoAvl *r)
  {
    raiz = r;
  }
  ...
};
```

17.3.1. Proceso de inserción de un nuevo nodo

Inicialmente, se aplica el algoritmo de inserción en un árbol de búsqueda, éste sigue el *camino de búsqueda* hasta llegar al fondo del árbol y se enlaza como nodo *hoja* y con *factor de equilibrio* `0`. Pero el proceso no puede terminar, es necesario recorrer el *camino de búsqueda* en sentido contrario, hacia la raíz, para actualizar el campo adicional *factor de equilibrio*. Después de una inserción sólo los nodos que se encuentran en el camino de búsqueda pueden haber cambiado el factor de equilibrio.

La actualización del *factor de equilibrio* (`fe`) puede hacer que éste *mejore*. Esto ocurre cuando un nodo está descompensado a la izquierda y se inserta el nuevo nodo en la rama izquierda, al crecer en altura dicha rama el `fe` se hace `0`, se ha *mejorado* el equilibrio. La Figura 17.6 muestra el árbol *a*) en el que el nodo `90` tiene `fe = 1`; en el árbol *b*), después de insertar el nodo con clave `60`, el nodo `90` tiene `fe = 0`.

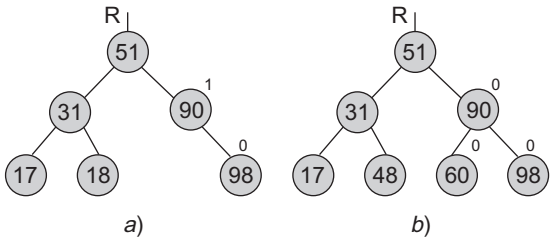


Figura 17.6. Mejora en la condición de equilibrio al insertar un nuevo nodo con clave 60.

La actualización del `fe` de un nodo del *camino de búsqueda* que, originalmente, tiene las ramas izquierda y derecha de la misma altura (`hRi = hRd`), no va a causar romper el criterio de equilibrio.

Al actualizar el nodo cuyas ramas izquierda y derecha del árbol tienen altura diferente, `|hRi-hRd|=1`, si se inserta el nodo en la rama más alta rompe el criterio de equilibrio del árbol, la diferencia de altura pasa a ser `2` y es necesario reestructurarlo.

Hay cuatro casos que se deben tener en cuenta al reestructurar un nodo `A`, según dónde se haya hecho la inserción:

1. Inserción en el subárbol izquierdo de la rama izquierda de `A`.
2. Inserción en el subárbol derecho de la rama izquierda de `A`.
3. Inserción en el subárbol derecho de la rama derecha de `A`.
4. Inserción en el subárbol izquierdo de la rama derecha de `A`.

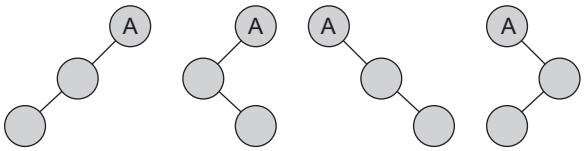


Figura 17.7. Cuatro tipos de reestructuraciones del equilibrio de un nodo.

El primer y tercer caso (*izquierda-izquierda, derecha-derecha*) se resuelven con una *rotación simple*. El segundo y cuarto caso (*izquierda-derecha, derecha-izquierda*) se resuelven con una *rotación doble*.

La rotación simple implica a dos nodos, el nodo `A` (nodo con `|fe| = 2`) y el descendiente izquierdo o derecho según el caso. En la rotación doble están implicados tres nodos, el nodo `A`, nodo descendiente izquierdo y el descendiente derecho de éste; o bien el caso simétrico, nodo `A`, descendiente derecho y el descendiente izquierdo de éste.

A recordar

Una reestructuración de los nodos implicados en la violación de *criterio de equilibrio*, ya sea una rotación simple o doble, hace que se recupere el equilibrio en todo el árbol, no siendo necesario seguir analizando los nodos del *camino de búsqueda*.

El proceso de *regresar* por el *camino de búsqueda* termina cuando se llega a la raíz del árbol, o cuando se realiza la reestructuración en un nodo del mismo. Una vez realizada una reestructuración no es necesario determinar el *factor de equilibrio* de los restantes nodos, debido a que dicho factor queda como el que tenía antes de la inserción, ya que la reestructuración hace que no aumente la altura.

17.3.2. Rotación simple

La rotación simple resuelve la *violación del equilibrio* de un nodo *izquierda-izquierda*, simétrica a la *derecha-derecha*. El árbol de la Figura 17.8*a*) tiene el nodo `C` con factor de equi-

librio `-1`; el árbol de la Figura 17.8*b*) es el resultado de insertar el nodo `A`. Resulta que ha crecido la altura de la rama izquierda, es un desequilibrio *izquierda-izquierda* que se resuelve con una rotación simple, rotación `II`. La Figura 17.9 es el árbol resultante de la rotación, el nodo `B` se ha convertido en la raíz, el nodo `C` su rama derecha y el nodo `A` continúa como rama izquierda. Con estos movimientos el árbol sigue siendo de búsqueda y se equilibra.

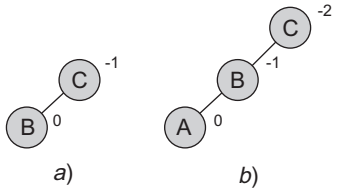


Figura 17.8. Árbol binario AVL y árbol después de inser tar nueva clave por la izquierda.

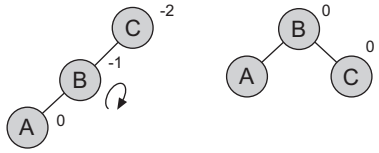


Figura 17.9. Árbol binario después de rotación simple `II`.

La Figura 17.10 muestra el otro caso de violación de la condición de equilibrio que se resuelve con una rotación simple. Inicialmente, el nodo `A` tiene como factor de equilibrio `+1`, al insertar el nodo `C` el factor de equilibrio de `A` pasa a ser `+2`, se ha insertado por la derecha y, por consiguiente, ha crecido la altura de la rama derecha. La Figura 17.11 muestra la resolución de este desequilibrio, una rotación simple que se puede denominar rotación `DD`. En el árbol resultante de la rotación, el nodo `B` se ha convertido en la raíz, el nodo `A` es su rama izquierda y el nodo `C` continúa como rama derecha. Con estos movimientos el árbol sigue siendo de búsqueda y queda equilibrado.

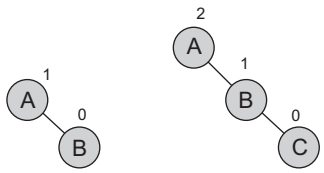


Figura 17.10. Árbol binario AVL y árbol después de inser tar nueva clave por la derecha.

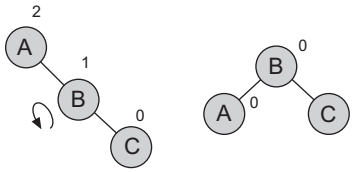


Figura 17.11. Árbol binario después de rotación simple DD.

17.3.3. Movimiento de enlaces en la rotación simple

Los cambios descritos en la rotación simple afectan a dos nodos, el tercero no se modifica, es necesario sólo una rotación. Para la rotación simple a la *izquierda*, rotación `II`, los ajustes necesarios de los enlaces, suponiendo `n` la referencia al nodo problema y `n1` la referencia al nodo de su rama izquierda:

```
n->izdo = n1->dcho;
n1->dcho = n;
n = n1;
```

Una vez realizada la rotación, los factores de equilibrio de los nodos que intervienen siempre es `0`, los subárboles izquierdo y derecho tienen la misma altura. Incluso, la altura del subárbol implicado es la misma después de la inserción que antes. La Figura 17.12 muestra estos movimientos de los enlaces.

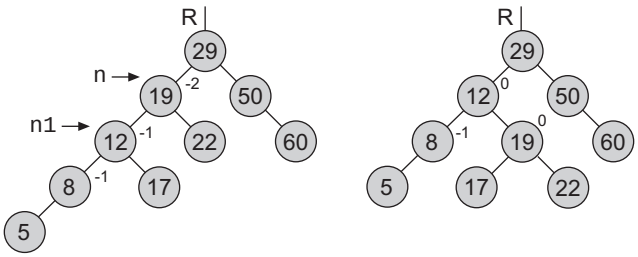


Figura 17.12. Rotación simple a izquierda en un árbol después de inser tar la clave 5.

Si la rotación simple es a *derecha*, rotación `DD`, los cambios en los enlaces del nodo `n` (con factor de equilibrio `+2`) y del nodo de su rama derecha, `n1`:

```
n->dcho = n1->izdo;
n1->izdo = n;
n = n1;
```

Realizada la rotación, los factores de equilibrio de los nodos que intervienen es 0. Se puede observar que estos ajustes son los simétricos a los realizados en la rotación `II`.

17.3.4. Rotación doble

Con la rotación simple no es posible resolver todos los casos de violación del criterio de equilibrio. El árbol de búsqueda de la Figura 17.13*a*) está desequilibrado, con factores de equilibrio `+2, -1` y `0`. La Figura 17.13*b*) aplica la rotación simple, rotación `DD`. La única solución consiste en *subir* el nodo `40` como raíz del subárbol, como rama izquierda situar al nodo `30` y como rama derecha el nodo `60`, la altura del subárbol resultante es la misma que antes de insertar. Se ha realizado una rotación doble, *derecha-izquierda*, en la que intervienen los tres nodos.

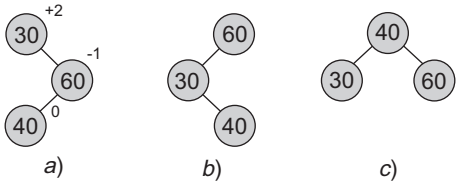


Figura 17.13. *a*) Árbol después de insertar clave 40. *b*) Rotación D *c*) Rotación doble para equilibrar.

La Figura 17.14*a*) muestra un árbol binario de búsqueda después de insertar la clave `60`. Al *volver* por el *camino de búsqueda* para actualizar los factores de equilibrio, el nodo `75` pasa a tener `fe = -1` (se ha insertado por su izquierda), el nodo `50` pasa a tener `fe = +1` y `el` nodo `80` tendrá como `fe = -2`. Es el caso simétrico al descrito en la Figura 17.13, se reestablece el equilibrio con una rotación doble, simétrica con respecto a la anterior como se muestra en la Figura 17.14*b*).

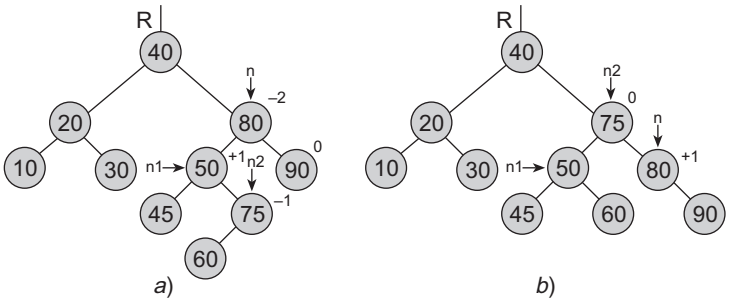


Figura 17.14. *a*) Árbol después de insertar clave 60. *b*) Rotación doble, izquierda derecha.

Recuerde

La rotación doble resuelve dos casos simétricos, se pueden denominar rotación `ID` y rotación `DI`. En la rotación doble hay que mover los enlaces de tres nodos , el nodo *padre*, el *descendiente* y el *descendiente del descendiente* por la rama contraria.

17.3.5. Movimiento de enlaces en la rotación doble

Los cambios descritos en la rotación doble afectan a tres nodos, el nodo problema `n`, el descendiente por la rama desequilibrada `n1`, y el descendiente de `n1` (por la izquierda o la derecha, según el tipo de rotación doble) apuntado por `n2`. En los dos casos simétricos de rotación doble, rotación *izquierda-derecha* (rotación `ID`) y rotación *derecha-izquierda* (rotación `DI`), el nodo `n2` pasa a ser la raíz del nuevo subárbol.

Los movimientos de los enlace para realizar la rotación `ID`:

```
n1->dcho = n2->izdo;
n2->izdo = n1;
n->izdo = n2->dcho;
n2->dcho = n;
n = n2;
```

Los factores de equilibrio de los nodos implicados en la rotación `ID` depende del factor de equilibrio, antes de la inserción, del nodo apuntado por `n2`, según esta tabla:

Si	`n2->fe = -1`	`n2->fe = 0`	`n2->fe = 1`
	`n->fe = 1`	0	0
	`n1->fe = 0`	0	-1
	`n2->fe = 0`	0	0

Los movimientos de los enlaces para realizar la rotación `DI` (observar la simetría en los movimientos de los punteros):

```
n1->izdo = n2->dcho;
n2->dcho = n1;
n->dcho = n2->izdo;
n2->izdo = n;
n = n2 ;
```

Los factores de equilibrio de los nodos implicados en la rotación `DI` también dependen del factor de equilibrio previo del nodo `n2`, según la tabla:

Si	`n2->fe = -1`	`n2->fe = 0`	`n2->fe = 1`
	`n->fe = 0`	0	-1
	`n1->fe = 1`	0	0
	`n2->fe = 0`	0	0

A recordar

La complejidad del algoritmo de inserción de una clave en un árbol de búsqueda AVL es la suma de la complejidad para bajar al nivel de las hojas (O(log n)) más la complejidad en el peor de los casos de la vuelta por el camino de búsqueda, para actualizar el factor de equilibrio de los nodos que es O(log n), más la complejidad de los movimientos de los enlaces en la rotación, que tiene complejidad constante. En definitiva, la complejidad de la inserción es O(log n), complejidad logarítmica.

17.4. IMPLEMENTACIÓN DE LA INSERCIÓN CON BALANCEO Y ROTACIONES

La realización de la fase de inserción es igual que la escrita para los árboles de búsqueda. Ahora se añade la fase de actualización de los factores de equilibrio; una vez insertado, se activa un *flag* para indicar que ha crecido en altura, de tal forma que al *regresar* por el *camino de búsqueda* calcule los nuevos factores de equilibrio de los nodos que forman el camino. Cuando la inserción se ha realizado por la rama izquierda del nodo, la altura crece por la izquierda y, por tanto, disminuye en `1` el factor de equilibrio; si se hace por la rama derecha, el factor de equilibrio aumenta en `1`. El proceso termina si la altura del subárbol no aumenta. También termina si se produce un desequilibrio, ya que cualquier rotación tiene la propiedad de que la altura del subárbol resultante es la misma que antes de la inserción.

A continuación, se escriben los métodos privados (miembros privados de la clase `ArbolAvl`) que implementan los cuatro tipos de rotaciones y de la operación de inserción. Todos devuelven la referencia al nodo raíz del subárbol implicado en la rotación.

Rotaciones

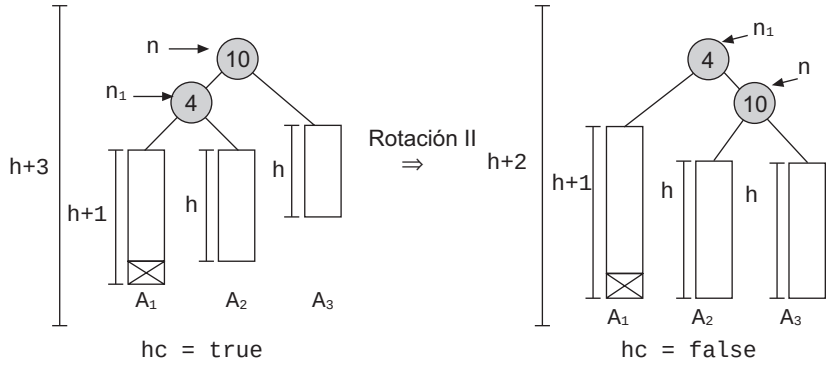


Figura 17.15.[1] Rotación II. Esquema gráfico. Árbol deja de aumentar en altura.

```
NodoAvl* ArbolAvl::rotacionII(NodoAvl* n, NodoAvl* n1)
{     //Figura 17.5
    n->ramaIzdo(n1->subarbolDcho());
    n1->ramaDcho(n);
      // actualización de los factores de equilibrio
    if (n1->Ofe() == -1) // la condición es true en la inserción
    {
      n->Pfe(0);
      n1->Pfe(0);
    }
```

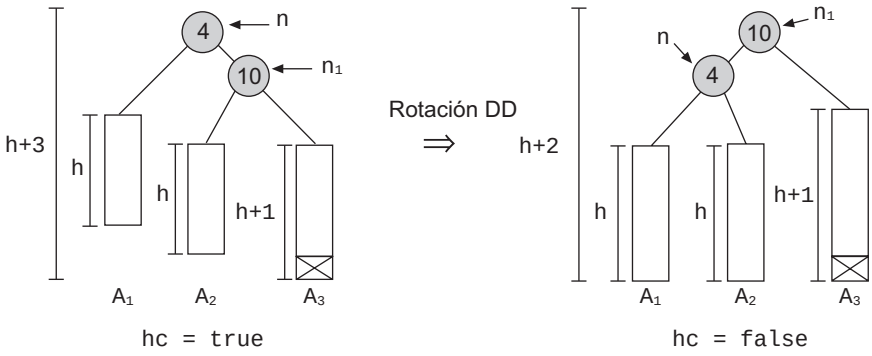


Figura 17.16. Rotación DD. Esquema gráfico. Árbol deja de aumentar en altura.

```
    else
    {
      n->Pfe(-1);
```

[1,2] En [KRUSE 94] el lector podrá encontrar una excelente referencia para profundizar en temas avanzados de árboles AVL, B y *tries* (árboles de búsqueda lexicográficos. Los códigos de las funciones, procedimientos y programas están escritos) en lenguaje Pascal, pero sus conceptos teóricos y prácticos sirven para la implantación en cualquier lenguaje.

```
      n1->Pfe(1);
    }
    return n1;
}

NodoAvl* ArbolAvl::rotacionDD(NodoAvl* n, NodoAvl* n1)
{   //Figura 17.16
    n->ramaDcho(n1->subarbolIzdo());
    n1->ramaIzdo(n);
                // actualización de los factores de equilibrio
    if (n1->Ofe() == +1) // la condición es true en la inserción
    {
      n->Pfe(0);
      n1->Pfe(0);
    }
    else
    {
      n->Pfe(+1);
      n1->Pfe(-1);
    }
    return n1;
}
```

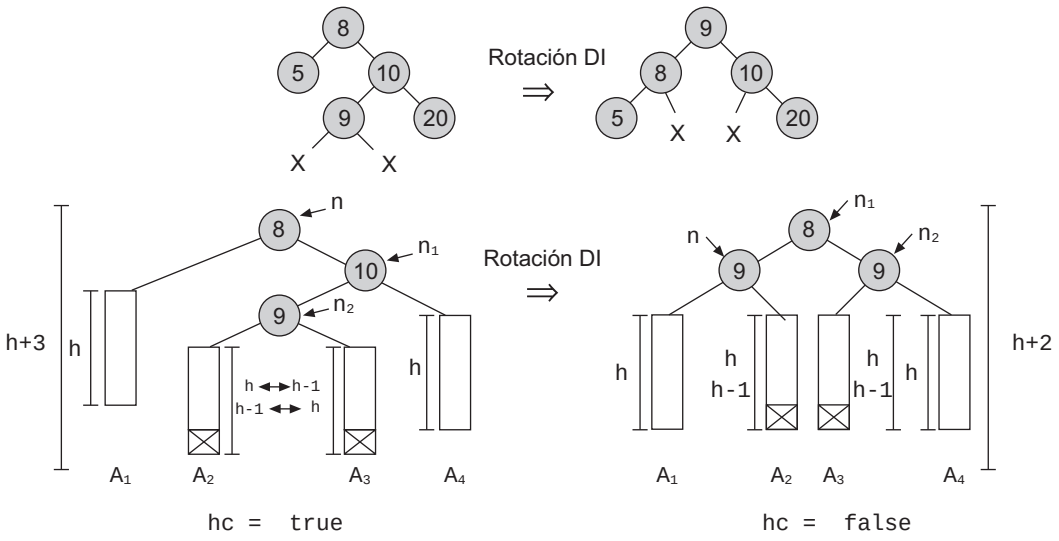


Figura 17.17. Rotación DI. Esquema gráfico. Árbol deja de aumentar en altura[3,4].

```
NodoAvl* ArbolAvl::rotacionDI(NodoAvl* n, NodoAvl* n1)
{//Figura 17.17
   NodoAvl* n2;

    n2 = n1->subarbolIzdo();
    n->ramaDcho(n2->subarbolIzdo());
```

[3] Si A_2 tiene altura h, A_3 tiene altura h–1; si A_2 tiene altura *h–1*, A_3 tiene altura h. No puede darse en las inserciones que A_2 tenga altura h y A_3 tenga altura *h*, ya que en este caso no hay desequilibrio en el nodo n_2.

[4] Ibid, [KRUSE 84].

```
    n2->ramaIzdo(n);
    n1->ramaIzdo(n2->subarbolDcho());
    n2->ramaDcho(n1);
            // actualización de los factores de equilibrio
    if (n2->Ofe() == +1)
      n->Pfe(-1);
    else
      n->Pfe(0);

    if (n2->Ofe() == -1)
      n1->Pfe(+1);
    else
      n1->Pfe(0);
    n2->Pfe(0);
    return n2;
}
```

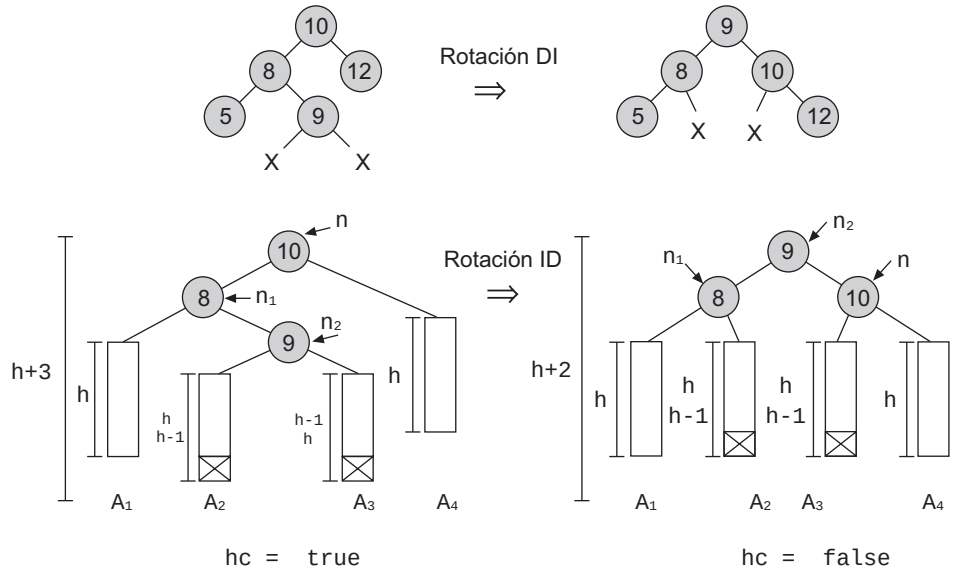


Figura 17.18. Rotación ID. Esquema gráfico. Árbol deja de aumentar en altura[5].

```
NodoAvl* ArbolAvl::rotacionID(NodoAvl* n, NodoAvl* n1)
{     //Figura 17.18
   NodoAvl* n2;
   n2 = n1->subarbolDcho();
   n->ramaIzdo(n2->subarbolDcho());
   n2->ramaDcho(n);
   n1->ramaDcho(n2->subarbolIzdo());
   n2->ramaIzdo(n1);
   // actualización de los factores de equilibrio
   if (n2->Ofe() == +1)
     n1->Pfe(-1);
```

[5] Ibid, [KRUSE 94].

```
    else
      n1->Pfe(0);
    if (n2->Ofe() == -1)
      n->Pfe(1);
    else
      n->Pfe(0);
    n2->Pfe(0);
    return n2;
}
```

Inserción con balanceo

El método público `insertarAvl()` es el interfaz de la operación, llamada a la función interna privada, recursiva, que realiza la operación y devuelve la raíz del nuevo árbol.

```
NodoAvl* ArbolAvl::insertarAvl(NodoAvl* raiz, Tipoelemento dt, bool &hc)
{
    NodoAvl *n1;

    if (raiz == NULL)
    {
      raiz = new NodoAvl(dt);
      hc = true;
    }
     else if (dt < raiz->valorNodo())
    {
       NodoAvl *iz;
       iz = insertarAvl(raiz->subarbolIzdo(), dt, hc);
       raiz->ramaIzdo(iz);
       // regreso por los nodos del camino de búsqueda
       if (hc)            // siempre se comprueba si creció en altura
       {
         // decrementa el fe por aumentar la altura de rama izquierda
         switch (raiz->Ofe())
         {
             case 1:         // tenía +1 y creció su izquierda
                raiz->Pfe(0);
                hc = false;  // árbol deja de crecer
                break;
             case 0:         // tenía 0 y creció su izquierda
                raiz->Pfe(-1);  // árbol sigue creciendo
                break;
             case -1:        // aplicar rotación a la izquierda
                n1 = raiz->subarbolIzdo();
                if (n1->Ofe() == -1)
                  raiz = rotacionII(raiz, n1);
                else
                  raiz = rotacionID(raiz, n1);
                hc = false;  // árbol deja de crecer en ambas rotaciones
         }
       }
    }
    else if (dt > raiz->valorNodo())
    {
      NodoAvl *dr;
```

```
        dr = insertarAvl(raiz->subarbolDcho(), dt, hc);
        raiz->ramaDcho(dr);
          // regreso por los nodos del camino de búsqueda
        if (hc)                // siempre se comprueba si creció en altura
        {
         // incrementa el fe por aumentar la altura de rama izquierda
         switch (raiz->Ofe())
         {
           case 1:             // aplicar rotación a la derecha
             n1 = raiz->subarbolDcho();
             if (n1->Ofe() == +1)
               raiz = rotacionDD(raiz, n1);
             else
               raiz = rotacionDI(raiz,n1);
             hc = false; // árbol deja de crecer en ambas rotaciones
             break;
           case                0: // tenía 0 y creció su derecha
             raiz->Pfe(+1);    // árbol sigue creciendo
             break;
           case -1:            // tenía -1 y creció su derecha
             raiz->Pfe(0);
             hc = false;       // árbol deja de crecer
         }
        }
      }
      else
        throw "No puede haber claves repetidas " ;
      return raiz;
}
```

EJERCICIO 17.2. Se quiere formar un árbol binario de búsqueda equilibrado de altura 5. El campo dato de cada nodo que sea una referencia a un objeto que guarda un número entero, que será la clave de búsqueda. Una vez formado el árbol, mostrar las claves en orden creciente y el número de nodos de que consta el árbol.

La formación del árbol equilibrado se puede hacer con repetidas llamadas a la función miembro `insertarAvl()`, de la clase `ArbolAvl` que realiza las llamadas, cuando es necesario, a las *rotaciones* implementadas como métodos privados de la clase. La condición para terminar la formación del árbol está expuesta en el enunciado: la altura del árbol sea igual a 5. Por ello, se escribe la función miembro publica `altura()` de la clase `ArbolAvl` para determinar dicho parámetro. También se escribe la función miembro pública `visualizar()`, que es un recorrido en inorden, para mostrar las claves en orden creciente y a la vez contar los nodos visitados. Se escribe, además, la función miembro que encuentra el número de nodos de una árbol, así como un programa principal que realiza las correspondientes llamadas

```
int mayor (int x, int y)
{  // calcula el mayor de los números que recibe como parámetro
  return (x > y ? x : y);
}
int ArbolAvl::altura(NodoAvl* r)
{ // método público que decide la altura de un árbol apuntado por r
   if (r != NULL)
     return mayor(altura(r->subarbolIzdo()),
```

```
      altura(r->subarbolDcho())) + 1;
    else
      return false;
}
int ArbolAvl::cuantos()
{ // método público que calcula el número de nodos de un árbol
  return cuantos(raiz);
}
int ArbolAvl::cuantos (NodoAvl* r)
{  // método privado que calcula el número de nodos apuntado por r.
   if (r)
   {
      int cuantosIzquierda, cuantosDerecha;
      cuantosIzquierda = cuantos(r -> subarbolIzdo());
      cuantosDerecha = cuantos(r -> subarbolDcho());
      return cuantosIzquierda + cuantosDerecha + 1;
   }
   else
      return 0;
}
void ArbolAvl::dibujarArbol(NodoAvl *r, int h)
 {
 // escribe las claves del árbol estableciendo separación entre nodos
      int i;
      if (r != NULL)
      {
        dibujarArbol(r->subarbolIzdo(), h + 1);
        for (i = 1; i <= h; i++)
        cout << " ";
        cout << r->valorNodo() << endl;
        dibujarArbol(r->subarbolDcho(), h + 1);
      }
 }

int main
{
   ArbolAvl a; const int TOPE = 999;
   int numNodos;
   randomize();
   while (a.altura(a.Oraiz()) < 5)
   {
     a.insertarAvl(random(TOPE)+1);
   }
   numNodos = a.cuantos();
   cout << "\n Número de nodos: " << numNodos << endl;
   a.dibujarArbol(a.Oraiz(), 1);
   return 0;
}
```

17.5. DEFINICIÓN DE UN ÁRBOL B

Cuando se tiene un conjunto de datos masivo, por ejemplo el conjunto `1.000.000` de clientes de un banco, los registros no pueden estar en memoria principal, y se ubican en memoria auxiliar, normalmente en disco. Los accesos a disco son *críticos*, consumen recursos y necesitan

notablemente más tiempo que las instrucciones en memoria, se necesita reducir al mínimo el número de accesos a disco. Para conseguir esto se emplean árboles de búsqueda *m-arios*, que ya no tienen dos ramas como los binarios, sino que pueden tener hasta *m* ramas o subárboles descendientes, además las claves se organizan a *la manera de los árboles de búsqueda*; el objetivo es que la altura del árbol sea lo suficientemente pequeña ya que el numero de iteraciones, y, por tanto, de acceso a disco, de la operación de búsqueda depende directamente de la altura. Un tipo particular de estos árboles son los árboles *B*, también los denominados *B*+ y *B** que proceden de pequeñas modificaciones del anterior.

Los árboles *B* son árboles *m*-arios, cada nodo tiene como máximo *m* ramas, no tienen subárboles vacíos y siempre están *perfectamente equilibrados*. Cada nodo (en árboles *B* se acostumbra a denominar *página*) es una unidad a la que se accede en bloque. Las estructura de datos que representa un árbol *B* de orden *m* tiene las siguientes características:

- Todas las páginas hoja están en el mismo nivel.
- Todos las páginas internas, menos la raíz, tienen a lo sumo *m* ramas (no vacías) y como mínimo *m/2* ramas.
- El número de claves en cada página interna es uno menos que el número de sus ramas, y estas claves dividen las de las ramas a manera de un árbol de búsqueda.
- La raíz tiene como máximo *m* ramas, puede llegar a tener hasta 2 y ninguna si el árbol consta de la raíz solamente.

Los árboles *B* más utilizados son los de orden 5, un orden mayor aumenta considerablemente la complejidad de los algoritmos de inserción y de borrado, un orden menor disminuye la eficacia de la localización de claves. Según la definición dada, el máximo número de claves de una página o nodo es 4 y el máximo de ramas o subárboles es 5.

La Figura 17.19 muestra un árbol *B* de orden 5, las claves de búsqueda son valores enteros. El árbol tiene 3 niveles, todas las páginas contienen 2, 3 o 4 claves. La raíz es la excepción, sólo tiene 1 clave. Todas las páginas, que son *hoja* del árbol, están en el nivel más bajo de éste, en la figura el nivel 3. Las claves mantienen una ordenación de izquierda a derecha dentro de cada página. Estas claves dividen a los nodos descendientes a *la manera de un árbol de búsqueda*, claves de nodo izquierdo menores, claves de nodo derecho mayores. Esta organización supone una extensión natural de los árboles binarios de búsqueda. La forma de localizar una clave en el árbol *B* consiste en seguir un camino de búsqueda, que se determina de igual manera que en los árboles binarios, con una adaptación a la característica de que cada nodo tiene *m–1* claves.

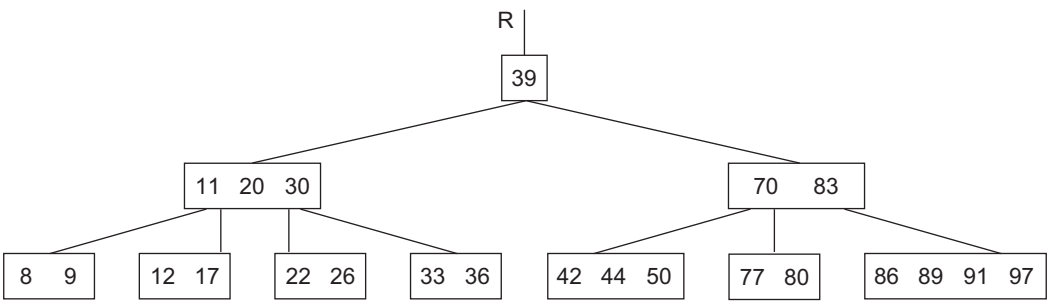


Figura 17.19. Árbol B de orden 5 de claves enteras.

17.6. *TAD ÁRBOL B* Y REPRESENTACIÓN

A la forma de organizar claves en la estructura árbol *B*, se le asocian una serie de operaciones básicas para poder buscar, añadir y eliminar claves. En definitiva, un árbol *B* es un tipo abstracto de datos con las siguientes operaciones básicas:

Crear	Inicializa el árbol *B* como árbol vacío.
Buscar	Dada una clave realiza la búsqueda de la clave. Devuelve la dirección del nodo y la posición en éste.
Insertar	Añade una nueva clave al árbol *B*. El árbol resultante sigue teniendo las características de árbol *B*.
Eliminar	Borra una clave del árbol B. El árbol resultante mantiene las características de árbol *B*.

17.6.1. Representación de una página

Los elementos de un árbol B son los nodos o *páginas* del árbol. Una *página* guarda las claves y las direcciones de las ramas de *páginas descendientes*. De manera natural surge la utilización de dos arrays dinámicos, y un atributo adicional `cuenta` que en todo momento contenga el número de claves de la *página*. El atributo privado `m` contiene siempre el numero de páginas descendientes máximo que puede contener la página. La clase `Pagina` declara los dos arrays dinámicos de claves y punteros a páginas con visibilidad *protegida*, para que las operaciones tengan restricciones de acceso. El constructor crea los arrays, ajustados al orden del árbol que recibe como parámetro. Además, el método `nodoLLeno()`, devuelve *verdadero* si el número de claves es `m-1` (máximo de claves de una página). El método `nodoSemiVacio()`, devuelve *verdadero* si el número de claves es menor que `m/2` (mínimo de claves que puede haber en una página). Se incluyen, además, los métodos para obtener y poner las claves y ramas de cada uno de los atributos de la página. Por su parte las funciones miembro `Ocuenta()` y `Pcuenta()`, se encargan de permitir el acceso al atributo `cuenta` y la modificación del propio atributo.

```
typedef int tipoClave;
class Pagina;
typedef Pagina * PPagina;
class Pagina
{
protected:
    tipoClave *claves; // puntero array de claves variables
    PPagina *ramas;    // puntero array de punteros a páginas variable
    int cuenta;        // número de claves que almacena la página
private:
    int m;      //máximo número de claves que puede almacenar la página
public:
    // crea una página vacía de un cierto orden dado
    Pagina (int orden)
    {
       m = orden;
       claves = new tipoClave[orden];
       ramas = new PPagina[orden];
       for (int k = 0; k <= orden; k++)
       ramas[k] = NULL;
     }
```

```
     // decide si un nodo está lleno
     bool nodoLLeno()
     {
       return (cuenta == m - 1);
     }
     // decide si una página tiene menos de la mitad de claves
     bool nodoSemiVacio()
     {
        return (cuenta < m / 2);
     }
      // obtener la clave que ocupa la posición i en el array de claves
     tipoClave Oclave(int i){ return claves[i];}
      // cambiar la clave que ocupa la posición i en el array de claves
     void Pclave(int i, tipoClave clave){ claves[i] = clave;}
      // obtener la rama que ocupa la posición i en el array de ramas
     Pagina* Orama(int i){ return ramas[i];}
      // cambiar la rama que ocupa la posición i en el array de ramas
     void Prama(int i, Pagina * p) { ramas[i] = p;}
      // obtener el valor de cuenta
     int Ocuenta(){ return cuenta;}
      // cambiar el valor de cuenta
     void Pcuenta( int valor) { cuenta = valor;}
};
```

Nota de programación

En un *árbol B de orden m*, el número de ramas de una página *no hoja* es uno más que el de sus claves. Por ello, las ramas se indexan de 0 a `m-1`, y las claves de `1` a `m-1`. El subárbol de `ramas[0]` se corresponde con las cla ves descendientes menores que `claves[1]`.

17.6.2. Tipo abstracto árbol B, clase ÁrbolB

El interfaz de la clase *ÁrbolB* se corresponde con las operaciones fundamentales del tipo abstracto: *buscar, insertar, eliminar*. Los métodos internos de la clase son los encargados del desarrollo de las operaciones del *tipo abstracto*. La clase tiene el atributo *orden* y la referencia al raíz del árbol como un puntero a la clase `Pagina`. Los constructores de la clase establece el orden del árbol e inicializa la raíz a `NULL` (*árbol vacío*). Se incluyen, además, los métodos públicos encargados de acceder y modificar el orden del árbol y la raíz del árbol.

```
class ArbolB
{
protected:
    int orden;
    Pagina *raiz;
public:
    ArbolB()
    {
        orden = 0;
        raiz = NULL;
    };
    ArbolB(int m)
```

```
    {
        orden = m;
        raiz = NULL;
    }
    bool arbolBvacio()
    {
        return raiz == NULL;
    }
    Pagina * Oraiz(){ return raiz;}
    void Praiz( Pagina * r){ raiz = r;}
    int Oorden(){ return orden;}
    void Porden(int ord){ orden = ord;}
    void crear() { orden = 0; raiz = NULL;}
    Pagina* buscar(tipoClave cl, int &n);
    void insertar(tipoClave cl);
    void eliminar(tipoClave cl);
};
```

17.7. FORMACIÓN DE UN ÁRBOL B

Las claves que se añaden a un árbol B siempre se insertan a partir de un nodo hoja, como ocurre en los árboles binarios. Además, al estar un árbol B perfectamente equilibrado, todas la hojas de un árbol B se encuentren en el mismo nivel, esto impone el comportamiento característico de los árboles B: *crecen "hacia arriba", crecen en la raíz*. Los pasos a seguir para añadir un nueva clave en un árbol B:

- Búsqueda de la clave a insertar en el árbol. Se sigue el *camino de búsqueda* que determina las claves de los nodos.
- En el caso de que la clave no esté en el árbol, la búsqueda termina en un nodo *hoja*. Entonces, empieza el proceso de inserción de la clave.
- De no estar lleno el nodo hoja, la inserción es posible en ese nodo y termina la inserción.
- Si está llena la *hoja*, la inserción en ella no es posible. Ahora se pone de manifiesto el comportamiento característico de los árboles B, se divide el nodo en dos en el mismo nivel del árbol, excepto la clave *mediana* que no se incluye en ninguno de los dos nodos, sino que *sube* en el árbol por el *camino de búsqueda* para, a su vez, repetir el proceso de inserción en el nodo *antecedente*. Por esto ***un árbol B crece hacia arriba***; esta *ascensión* de la clave *mediana* puede propagarse y llegar al nodo raíz, entonces éste se divide en dos nodos y la clave enviada hacia arriba se convierte en una nueva raíz. Ésta es la forma que tiene el *árbol B* de crecer en altura.

17.7.1. Creación de un árbol B de orden 5

A continuación se va a seguir los pasos de formación de un *árbol B* de orden `5`, para facilitar la comprensión las claves son números enteros. Al ser de orden `5` el número máximo de claves de un nodo es `4` y el máximo de ramas `5`.

Supóngase que las claves que se insertan:

```
6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27
```

- Con las cuatro primeras se completa el primer nodo, en orden creciente a la manera de árbol de búsqueda. La Figura 17.20(*a*) muestra este nodo.

- La clave siguiente, 8,encuentra el nodo ya lleno. La clave *mediana* de las cinco claves es 6. El nodo lleno se divide en dos, la clave mediana *"sube"* y como no hay nodo antecedente se crea otro nodo con la clave *mediana*, es la nueva raíz del árbol. El árbol ha crecido en altura como puede observarse en la Figura 17.20(*b*).

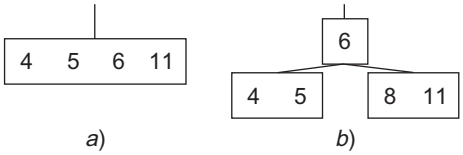


Figura 17.20. Formación de un árbol B de orden 5.

- Las siguientes claves 9, 12 se insertan en el nodo derecha de la raíz, según el criterio de búsqueda, ambas claves son mayores que 6. La Figura 17.21*a*) muestra el árbol después de esta inserción.
- La clave 21 *baja* por el *camino de búsqueda*, rama derecha del nodo raíz. El nodo hoja que le corresponde está lleno; por consiguiente, el nodo se parte en dos y la clave mediana, 11, asciende por el *camino de búsqueda* para ser insertada en el nodo antecedente, en este caso la raíz. El proceso se repite, ahora si hay *hueco* para que la clave 11 sea insertada. La Figura 17.21*b*) muestra el árbol resultante después de la creación del nuevo nodo.

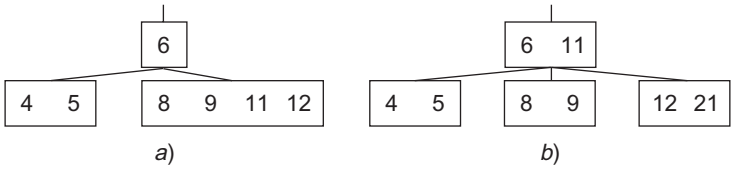


Figura 17.21. Inserción de nuevas claves en un árbol B de orden 5.

- Las siguientes claves 14, 10, 19 se insertan en los nodos *hoja* que les corresponde, según el *camino de búsqueda*.
- Al insertar la siguiente clave, 28, el *camino de búsqueda* determina que se inserte en un nodo que está lleno (nodo derecho del último nivel). De nuevo se produce el proceso de división del nodo y ascensión, por el *camino de búsqueda*, de la clave *mediana* que ahora es 19. El antecedente, nodo raíz, tiene dos claves y, por consiguiente, se inserta 19.
- Las claves que vienen a continuación, 3, 17, 32, 15 son insertadas en los nodos *hoja* que determina el *camino de búsqueda*. Cada uno de estos nodos tiene un número de claves menor que el máximo, m-1, por ello cada clave se inserta directamente en la hoja, en la posición que le corresponde según su valor. La Figura 17.22 representa al árbol después de estas inserciones.

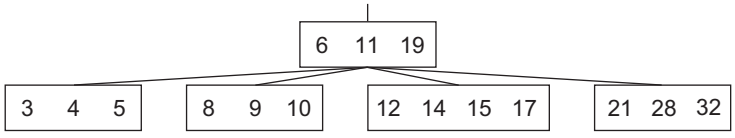


Figura 17.22. Representación de árbol B de orden 5.

- Ahora hay que añadir la clave `16`; *"baja"* por el *camino de búsqueda* (rama derecha de clave `11`). El nodo donde va a ser insertado está lleno, se produce la división del nodo y la clave mediana, `15`, *"sube"* por el *camino de búsqueda* para que sea añadida en el nodo antecedente. Como éste no está lleno, se inserta la clave `15`; el nodo raíz ha quedado completo (Figura 17.23).

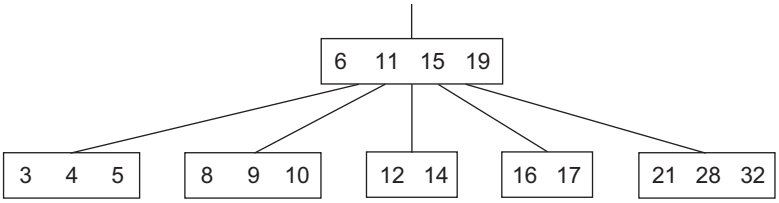


Figura 17.23. Árbol B de orden 5 después de una división de nodo .

- Al insertar la clave `26` se completa el nodo más a la derecha del último nivel. Por último, la clave `27` *"baja"* por el mismo *camino de búsqueda*, el nodo *hoja* que le corresponde está completo, entonces se divide en dos y *"sube"* la clave *mediana*, `27`, al nodo *padre*. Ocurre que este también está completo, se repite el proceso de división en dos nodos, y *"sube"* la nueva clave *mediana*, `15`. Como el nodo dividido es el raíz, con esta clave *mediana* se forma un nuevo nodo raíz: el árbol ha crecido en altura. El *árbol crece hacia arriba*, hacia la raíz.

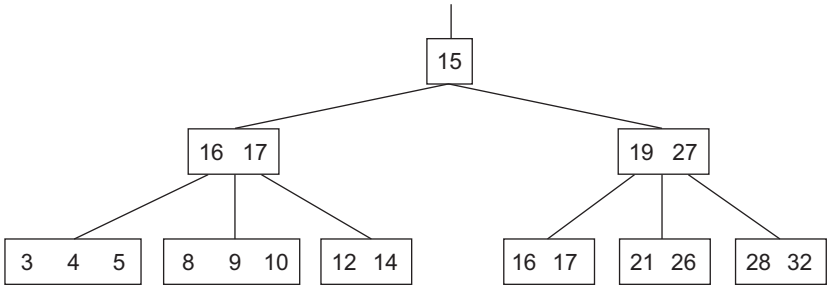


Figura 17.24. Árbol B de orden 5 con todas las claves.

A tener en cuenta

En el proceso de formación de un *árbol B* hay que destacar dos hechos relevantes. El primero, la división de un nodo prepara la estructura para nuevas inserciones, ya que deja *huecos* en los nodos implicados . En segundo lugar , la clave que *"sube"* al nodo antecedente es la clave *mediana* del nodo lleno, no tiene por qué coincidir con la clave que se está insertando; se puede afirmar que no importa el orden en que lleguen las claves en el balanceo del árbol.

17.8. BÚSQUEDA DE UNA CLAVE EN UN ÁRBOL B

Los elementos de un nodo en un árbol B son *claves de búsqueda*, dividen a las ramas inmediatas, rama izquierda y derecha, a la manera de un árbol de búsqueda. Por consiguiente, el algoritmo es similar al de búsqueda en un árbol binario ordenado, salvo que en los árboles B cuando se está analizando una *página* hay que inspeccionar las claves de que consta. La inspección da como resultado la posición de la clave, o bien la rama por donde seguir buscando (*camino de búsqueda*).

El método que implementa la operación, si encuentra la clave, devuelve la dirección del nodo que contiene a la clave y la posición que ocupa. Si la clave no está en el árbol, devuelve `NULL`. La búsqueda en cada nodo la realiza el método auxiliar `buscarNodo()`.

17.8.1. `buscarNodo()`

El método devuelve `true` si encuentra la clave en el árbol. Además, en el argumento *k* se obtiene la posición que ocupa la clave en la *página*, o bien la rama por donde continuar el proceso de búsqueda. Aprovecha la ordenación de las claves en el nodo, de tal forma que inspecciona las claves de la *página* en orden descendente y termina cuando `claves[index] <= cl`, (`actual->Oclave(index) <= cl`) así se asegura que *index* es la posición que ocupa la clave o el índice de la rama del árbol en la que puede ser encontrada.

```
bool ArbolB::buscarNodo(Pagina* actual, tipoClave cl, int & k)
{
     int index;
     bool encontrado;
     if (cl < actual->Oclave(1))
     {
       encontrado = false;
       index = 0;
     }
     else
     {      // orden descendente
       index = actual->Ocuenta();
       while (cl < actual->Oclave(index) && (index > 1))
         index--;
       encontrado = cl == actual->Oclave(index);
     }
     k = index;
     return encontrado;
}
```

17.8.2. `buscar()`

Este método privado de la clase `ArbolB` controla el proceso de búsqueda. El método *baja* por las ramas del árbol hasta encontrar la clave. El método se implementa recursivamente, la condición para terminar de hacer llamadas recursivas es que se haya localizado la clave, o bien el puntero a la página `actual es` a `NULL`, por tanto, no hay más nodos en los que buscar. Devuelve la referencia al nodo donde se encuentra la clave, o bien `NULL` . La función miembro está sobrecargada; la función interna privada realiza la búsqueda desde la raíz.

```
Pagina* ArbolB::buscar( tipoClave cl, int &n)
{
   return buscar( raiz, cl, n);
}

Pagina* ArbolB::buscar(Pagina* actual, tipoClave cl, int &n)
{
   if (actual == NULL)
   {
      return NULL;
   }
   else
   {
      bool esta = buscarNodo(actual, cl, n);
      if (esta) // la clave se encuentra en el nodo actual
        return actual;
      else
        return buscar(actual->Orama(n), cl, n);  //llamada recursiva
   }
}
```

17.9. INSERCIÓN EN UN ÁRBOL B

Debido a que las claves de un árbol B están organizadas como un árbol *m-ario de búsqueda*, la operación que inserta una clave en un árbol B sigue la misma estrategia que la inserción en un árbol binario de búsqueda. La nueva clave siempre se inserta en una hoja, para lo cual *baja* por el *camino de búsqueda*, hasta alcanzar el nodo hoja. Esta primera parte del algoritmo utiliza el método `buscarNodo()para` determinar la rama por donde *bajar*.

Una vez en el nodo hoja, si no está lleno, se inserta directamente en la posición que devuelve `buscarNodo()`, de tal forma que la clave queda ordenada. El método `meterPagina()`realiza esta parte de la operación.

Sólo si el nodo está lleno la inserción afecta a la estructura del árbol, ya que se ha de crear un nuevo nodo y, además, para que el árbol mantenga las características de *árbol B*, *asciende* la clave mediana al nodo antecedente. El método `dividirNodo()` es donde se implementan las acciones de esta parte del algoritmo.

La formulación recursiva es la más adecuada para reflejar el proceso de propagación, *hacia arriba*, en la *división de los nodos*, debido a que al retornar la llamada recursiva se *regresa por el camino de búsqueda*, se pasa por los nodos en sentido inverso a como se *bajó*.

El método interno `insertar()` tiene dos argumentos, la clave y el nodo raíz; ésta puede cambiar, si el árbol crece en altura y por ello devuelve la referencia al raíz. Pasa control a `empujar()` que, con una estrategia recursiva, realiza el proceso de inserción. Los métodos `meterPagina()` y `dividirNodo()` son llamados desde `empujar()`, según que la inserción se realice en un nodo con posiciones vacías o lleno, respectivamente.

17.9.1. Método `insertar()`

Se encarga de crear una nueva raíz si la *propagación, hacia arriba,* del proceso de división llega al actual raíz (el árbol aumenta su altura). El método está sobrecargado. En el primer caso,

el método es público y realiza la llamada al método `insertar()` pero teniendo como parámetro la raíz del árbol. Véase Figura 17.25.

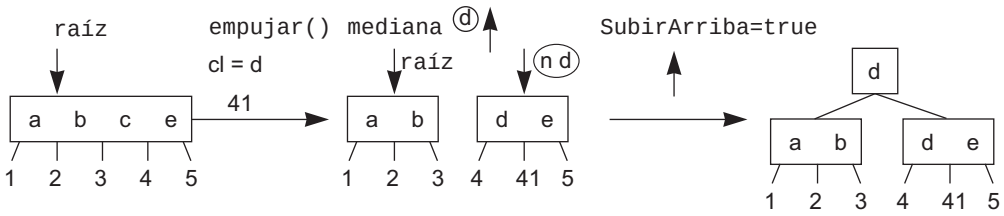


Figura 17.25. Crecimiento en altura de un árbol.

```
 // método público
void ArbolB::insertar(tipoClave cl)
{
    raiz = insertar(raiz, cl);
}
//método privado
Pagina* ArbolB::insertar(Pagina* raiz, tipoClave cl)
{
    bool subeArriba;
    int mediana;
    Pagina* nd;
    subeArriba = empujar(raiz, cl, mediana, nd);
    if (subeArriba)
    {
      // El árbol crece en altura por la raíz.
      // sube una nueva clave mediana y un nuevo hijo derecho nd
      // en la implementación se mantiene que las claves que son
      // menores que mediana se encuentran en raiz y las mayores en nd
       Pagina* p;
       p = new Pagina(orden);   // nuevo nodo
       p->Pcuenta(1);           // tiene una sola clave
       p->Pclave(1,mediana);
       p->Prama(0,raiz);        // claves menores
       p->Prama(1,nd);          // claves mayores
       raiz = p;
    }
    return raiz;
}
```

17.9.2. Método `empujar()`

Es el método más importante, controla la realización de las tareas de búsqueda y posterior inserción. Lo primero que hace es *"bajar"* por el *camino de búsqueda* hasta llegar a una rama vacía. A continuación, se prepara para *"subir"* (activa el indicador `subeArriba`) por las páginas del camino y realizar la inserción. En esta *vuelta atrás*, primero se encuentra con la *Página* hoja, si hay *hueco* mete la clave y el proceso termina. De estar completa, llama a `divi-`

`dirNodo()` que crea una nueva *Página* para repartir las claves; la clave *mediana* sigue *"subiendo"* por el *camino de búsqueda*.

La simulación de *"bajar"* y luego *"subir"* por el *camino de búsqueda* se implementa fácilmente mediante las llamadas recursivas de `empujar()`, de tal forma que cuando las llamadas retornan, se está volviendo (*"subiendo"*) a las de la *Páginas* por los que pasó anteriormente: *regresa* por el *camino de búsqueda*.

La inserción de la clave mediana en la *Página* antecesor, debido a la división de la *Página*, puede, a su vez, causar el desbordamiento de la misma, dando como resultado la propagación del proceso de partición *hacia arriba*, pudiendo llegar a la raíz. Ésta es la única forma de que aumente la altura del árbol B. En el caso de que la altura del árbol aumente es cuando el método `insertar()` crea un nuevo nodo que almacena una sola clave y dos punteros que apuntan a Páginas con claves menores y mayores respectivamente.

El método `empujar()` es privado y retorna el valor verdadero si hay clave que sube hacia arriba en cuyo caso se retorna en mediana, y además las claves más pequeñas están apuntadas por el puntero a la página actual y las mayores por el puntero a página nuevo.

```
bool ArbolB::empujar(Pagina* actual, tipoClave cl,
                     tipoClave &mediana, Pagina *& nuevo)
{
   int k ;
   bool subeArriba = false;
   if (actual == NULL)
   { // envía hacia arriba la clave cl y su rama derecha NULL
     // para que se inserte en la Página padre
      subeArriba = true;
      mediana = cl;
      nuevo = NULL;
       // el dato Página de nuevo está a NULL
   }
   else
   {
      bool esta;
      esta = buscarNodo(actual, cl, k);
      if (esta)
        throw "\nClave duplicada";
        // siempre se ejecuta
      subeArriba = empujar(actual->Orama(k), cl, mediana, nuevo);
        // devuelve control; vuelve por el camino de búsqueda
      if (subeArriba)
      {
         if (actual->nodoLLeno()) // hay que dividir la página
            dividirNodo(actual, mediana, nuevo, k);
      }
      else
      { //cabe en la página, se inserta la mediana y su rama derecha
        subeArriba = false;
        meterPagina(actual, mediana, nuevo, k);
      }
   }
   return subeArriba;
}
```

Nota de programación

En un árbol B los elementos son claves de búsqueda, por ello no se contempla que haya elementos repetidos. Ésa es la razón para que el método *empujar* levante una excepción cuando el método `buscarNodo()` encuentra la clave que se quiere insertar.

18.5.3. Método `meterPagina()`

Este método privado de la clase `ArbolB` inserta una clave en una *Página* que tiene un número de claves menor que el máximo. El método es invocado una vez que `empujar()` ha comprobado que hay *hueco* para añadir a la *Página* una nueva clave. Se le pasa, como argumentos, la dirección de la *Página*, la clave, la dirección de la rama con la *Página* sucesor, y la posición a partir de la cual se inserta. Véase Figura 17.26

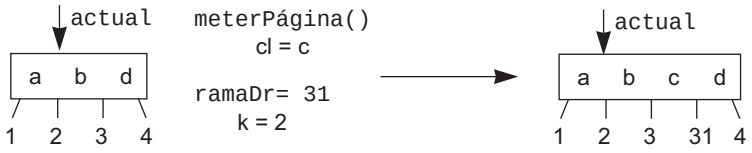


Figura 17.26. Proceso de Meter en una página.

```
void ArbolB::meterPagina(Pagina* actual, tipoClave cl,
                         Pagina *ramaDr, int k)
{
     // desplaza a la derecha los elementos para hacer un hueco
    for (int i = actual->Ocuenta(); i >= k + 1; i--)
     {
         actual->Pclave(i + 1, actual->Oclave(i));
         actual->Prama(i + 1, actual->Orama(i));
    }
     // pone la clave y la rama derecha en la posición k+1
    actual->Pclave(k + 1, cl);
    actual->Prama(k + 1, ramaDr);
     // incrementa el contador de claves almacenadas
    actual->Pcuenta(actual->Ocuenta()+1) ;
}
```

17.9.4. Método `dividirNodo()`

Este método resuelve el problema de que la *Página* donde se debe insertar la clave esté llena. Virtualmente, la *Página* se divide en dos y la clave mediana es enviada *hacia arriba*, para una *re-inserción* posterior en una *Página* padre o bien en una nueva raíz en el caso de que el árbol deba crecer en altura. Para ello, se crea una nueva *Página* a la que se desplazan las claves mayores de la mediana y sus correspondientes ramas, dejando en la *Página* original las claves menores.

El algoritmo, en primer lugar, encuentra la posición que ocupa la clave mediana; después mueve claves y ramas a la *Página* nueva y, según la posición k de inserción, mete la clave en la Página original o en la nueva. Por último, *"extrae"* la clave mediana que siempre se deja en el nodo original. Con la técnica de llamadas recursiva, el método `empujar()`insertará, posteriormente, la clave mediana en la *Página* antecedente. El argumento `mediana` recibe la clave que se inserta y se actualiza con la nueva mediana; de igual forma, el argumento `nuevo` recibe la rama derecha de la clave a insertar y se actualiza con la *Página* creada.

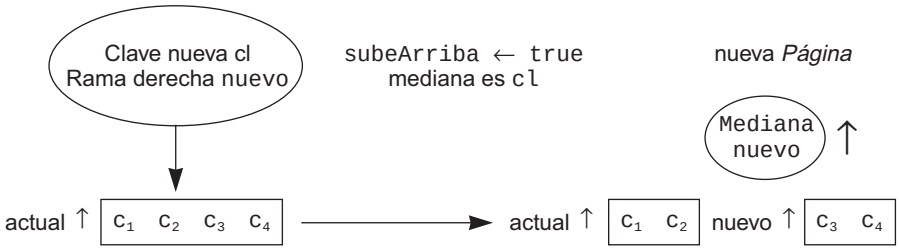


Figura 17.27. Proceso de partición de una *Página*.

```
void ArbolB::dividirNodo(Pagina* actual, tipoClave &mediana,
                         Pagina * &nuevo, int pos)
{
    int i, posMdna, k;
    Pagina *nuevaPag;
    k = pos;
     // posición de clave mediana
    posMdna = (k <= orden/2) ? orden/2 : orden/2 + 1;
    nuevaPag = new Pagina(orden);
    for (i = posMdna + 1; i < orden; i++)
    {
      /* desplazada la mitad derecha a la nueva Página, la clave
        mediana se queda en Página actual */
      nuevaPag->Pclave(i - posMdna, actual->Oclave(i));
      nuevaPag->Prama(i - posMdna, actual->Orama(i));
    }
    nuevaPag->Pcuenta ((orden - 1) - posMdna); // claves de nueva Página
    actual->Pcuenta(posMdna);   // claves en Página origen
     // inserta la clave y rama en la Página que le corresponde
    if (k <= orden / 2)
        meterPagina(actual, mediana, nuevo, pos); // en Página origen
    else
    {
        pos = k - posMdna;
        meterPagina(nuevaPag, mediana, nuevo, pos); // en Página nueva
    }
     // extrae clave mediana de la Página origen
    mediana = actual->Oclave(actual->Ocuenta());
     // Rama0 del nuevo nodo es la rama de la mediana
    nuevaPag->Prama(0, actual->Orama(actual->Ocuenta()));
    actual->Pcuenta(actual->Ocuenta() -1); // se quita la mediana
    nuevo = nuevaPag; // devuelve la nueva Página
}
```

17.9.5. Método `escribir()`

El método público `escribir()` realiza la tarea de visualizar en pantalla un `Arbolb` pequeño. Al igual que otros métodos está sobrecargado, el primero de ellos realiza la llamada al segundo con un argumento más que es la raíz del árbol. El parámetro h de escribir se usa para formatear la salida y que se pueda visualizar en la pantalla el árbol.

```
void ArbolB::escribir( )
{
     escribir (raiz, 1);
}
void ArbolB::escribir( Pagina * r, int h)
{
   int i;
   if (r != NULL)
   {
       escribir( r->Orama(0),h + 5);
       for (i = 1; i <= r->Ocuenta()/2;i++)
       { // llamadas recursivas a la mitad de los subárboles
          escribir(r->Orama(i),h + 5);
          cout << endl;
       }
        // visualización de las claves de la página apuntada por r
       for (i = 1; i<= r->Ocuenta();i++)
       {
           for (int j = 0; j <= h; j++)
              cout << " ";
           cout << r->Oclave(i) << endl;
       }
       // llamadas recursivas a la otra mitad de los subárboles
       for (i = r->Ocuenta() / 2 + 1 ; i<= r->Ocuenta();i++)
           escribir(r->Orama(i),h + 5);
       cout << endl;
   }
}
```

17.10. LISTADO DE LAS CLAVES DE UN ÁRBOL B

Recorrer una estructura de datos implica *visitar* cada uno de los elementos de que consta, así ocurre con los vectores, con las listas lineales y circulares, ... En los árboles se distinguen distintos tipos de recorrido, en profundidad y en anchura. El recorrido en profundidad se puede hacer en *preorden*, *inorden* y *postorden*. En los árboles B se puede aplicar los mismos criterios para visitar sus claves, con la particularidad de que un nodo (*Página*) de un árbol B hay, normalmente, más de una clave y más de dos subárboles.

Un recorrido que resulta interesante en un árbol B es el que *visita* las claves del árbol en orden creciente. Como cada clave de la *Página* del árbol B divide a las de sus ramas adyacentes a la manera de un árbol de búsqueda, se aplica el recorrido en *inorden* de un árbol binario, con una pequeña adaptación, para *visitar* en orden creciente las claves de un árbol B. La implementación se realiza con dos métodos: `listaCreciente()` que es el interfaz público de la clase `ArbolB`, y `inOrden()` de la clase `ArbolB` como método privado que realiza el recorrido por las claves.

```
void ArbolB::listarCreciente()
{
    inOrden(raiz) ;
}
void ArbolB::inOrden(Pagina *r)
{
    if (r)
    {
        inOrden(r->Orama(0));
        for (int k = 1; k <= r->Ocuenta(); k++)
        {
            cout << r->Oclave(k) << " ";
            inOrden(r->Orama(k));
        }
    }
}
```

Nota de programación

El recorrido en *inorden* del árbol B se puede implementar iterativamente. Para ello se utiliza una *Pila* en la que se guardan las referencias a las Páginas del árbol.

EJERCICIO 17.3. Escribir una aplicación para gestionar un árbol B de orden 5, utilizando las operaciones del TAD ya implementadas en la clase ArbolB . Deben realizarse estas acciones:

- *Dar de alta claves.*
- *Buscar una clave dada.*
- *Eliminar una clave dada.*
- *Listar las claves en orden creciente.*

El programa crea un `ArbolB` de con claves de tipo entero. Las operaciones se presentan en forma de menú; la opción *"dar de alta claves"* genera n claves aleatoriamente, en el rango de `1` a `9999`, que se añaden al árbol llamando al método `insertar()`. Las acciones de *buscar, eliminar y listar* se corresponden con las operaciones implementadas en la clase.

Codificación. Véase página web del libro

Lecturas recomendadas para profundizar en el tema

En la lectura recomendada Anexo A puede consultar una ampliación sobre "Eliminación en árboles AVL".
En la lectura recomendada Anexo B puede consultar una ampliación sobre "Eliminación en árboles B".

RESUMEN

Los árboles binarios de búsqueda son estructuras de datos importantes que permiten localizar una *clave de búsqueda* con un coste logarítmico. Sin embargo, el tiempo de búsqueda puede crecer hasta hacerse lineal si el árbol está degenerado; en ese sentido, la eficiencia de las operaciones dependen de lo aleatorias que sean las claves de entrada. Para evitar este problema surgen los árboles de búsqueda equilibrados, llamados árboles AVL. La eficiencia de la *búsqueda* en los árboles equilibrados es mayor que en los árboles de búsqueda no equilibrados.

Un árbol binario equilibrado es un árbol de búsqueda caracterizado por diferir las alturas de las ramas derecha e izquierda del nodo raíz a lo sumo en uno, y que los subárboles izquierdo y derecho son, a su vez, árboles equilibrados.

Las operaciones de los árboles de búsqueda son también operaciones de los árboles equilibrados. Las operaciones de inserción y de borrado añaden o eliminan, respectivamente, un nodo en el árbol de igual forma que en los árboles de búsqueda no equilibrados. Además, es necesario incorporar una segunda parte de reestructuración del árbol para asegurar el equilibrio de cada nodo. La solución que se aplica ante un desequilibrio de un nodo viene dada por las *rotaciones*. Se pueden aplican dos tipos de rotaciones, *rotación simple y rotación doble*; el que se aplique una u otra depende del nodo desequilibrado y del *equilibrio* del hijo que se encuentra en la rama más alta.

Las operaciones de inserción y de borrado en árboles binarios equilibrados son más *costosas* que en los no equilibrados, debido a que el algoritmo debe *retroceder* por el *camino de búsqueda*, para actualizar el factor de equilibrio de los nodos que lo forman y al *coste* de la rotación cuando se viola la condición de equilibrio. Por el contrario, la operación de búsqueda es, de promedio, menos costosa que en los árboles binarios no equilibrados.

Los *árboles B* tienen una serie de características que permiten que la búsqueda de una clave sea muy eficiente, por ello, si las claves se encuentran en un disco, el número de accesos al disco disminuye notablemente. En aplicaciones que acceden a archivos que están en disco, el mayor tiempo de proceso es el de las operaciones de *entrada/salida*. Reducir el número de pasos para buscar una clave, o para insertar, supone reducir el número de operaciones de entrada y salida y, por consiguiente, el tiempo de proceso. Entonces, un árbol-B es una estructura de datos utilizada para almacenar claves (*de búsqueda*) que residen en memoria externa, tal como un disco.

El número de claves de una *Página* (nodo) en un árbol B *de orden m*, que no sea la raíz, es como mínimo `m/2` y como máximo `m-1`. Además, las claves de una *Página* divide a las claves de sus ramas a la manera de árbol de búsqueda. El número de ramas descendientes de una *Página* es una más que el número de claves, excepto las *Páginas* que son hoja.

Un árbol B siempre está perfectamente equilibrado. La inserción de una nueva clave se realiza en un hoja, si está llena *asciende* la clave *mediana* a la *Página padre*, si éste también está lleno sigue *ascendiendo*, pudiendo llegar a la raíz, si también se encuentra llena se crea una nueva *Página* con la clave *mediana*. Se dice que los árboles B crecen hacia arriba, hacia la raíz. Los árboles B más utilizados son los de orden 5.

BIBLIOGRAFÍA RECOMENDADA

Aho V.; Hopcroft, J., y Ullman, J.: *Estructuras de datos y algoritmos*. Addison Wesley, 1983.

Garrido, A., y Fernández, J.: *Abstracción y estructuras de datos en C++*. Delta, 2006.

Joyanes, L., y Zahonero, L.: *Algoritmos y estructuras de datos. Una perspectiva en C*. McGraw-Hill, 2004.

Joyanes, L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos en C*. Schaum, 2005.

Kruse Robert, L.: *Estructura de datos y diseño de programa*. Prentice-Hall, 1998.
Lipschutz, S.: *Estructura de datos*. McGraw-Hill, 1987.
Nyhoff, L.: *TADs, Estructuras de datos y resolución de problemas con C++*. Pearson, Prentice-Hall, 2005.
Weis, Mark Allen: *Estructuras de datos y algoritmos*. Addison Wesley, 1992.
Wirth Niklaus: *Algoritmos + Estructuras de datos = programas*. 1986.

EJERCICIOS

17.1. Dibujar el árbol binario de búsqueda equilibrado que resulta las claves: 14, 6, 24, 35, 59, 17, 21, 32, 4, 7, 15 y 22.

17.2. Dada la secuencia de claves enteras: 100, 29, 71, 82, 48, 39, 101, 22, 46, 17, 3, 20, 25, 10. Dibujar el árbol AVL correspondiente. Eliminar claves consecutivamente hasta encontrar un nodo que viole la condición de equilibrio y cuya restauración sea con una rotación doble.

17.3. En el árbol construido en el Ejercicio 17.1 eliminar el nodo raíz. Hacerlo tantas veces como sea necesario hasta que se desequilibre un nodo y haya que aplicar una rotación simple.

17.4. Encontrar una secuencia de n claves que al ser insertadas en un árbol binario de búsqueda vacío se apliquen las cuatro rutinas de rotación: `II`, `ID`, `DD`, `DI`.

17.5. Dibujar el árbol equilibrado después de insertar en orden creciente 31 ($2^5 - 1$) elementos del 11 al 46.

17.6. ¿Cuál es el número mínimo de nodos de un árbol binario de búsqueda equilibrado de altura 10?

17.7. Escribir el método recursivo `buscarMin()` que devuelva el nodo de clave mínima de un árbol de búsqueda equilibrado.

17.8. Dibujar un árbol AVL de altura 6 con el criterio del peor de los casos, es decir, en el que cada nodo tenga como factor de equilibrio ± 1.

17.9. En el árbol equilibrado formado en el Ejercicio 17.8, eliminar una de las hojas menos profundas. Representar las operaciones necesarias para restablecer el equilibrio.

17.10. Escribir el método recursivo `buscarMax()` que devuelva el nodo de clave máxima de un árbol de búsqueda equilibrado.

17.11. Escribir los métodos `buscarMin()` y `buscarMax()` en un árbol de búsqueda equilibrado de manera iterativa.

17.12. Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80; dibuja el árbol B de orden 5 formado con dichas claves.

17.13. La operación de inserción de una clave en una *Pagina* llena se ha realizado partiendo ésta y elevando la clave mediana. Analizar, si es posible, esta otra estrategia: buscar el número de claves del hermano izquierdo y derecho, si alguno no está lleno mover claves para realizar la inserción.

17.14. Un árbol B de orden 5 se insertan las claves de manera secuencial: 1, 2, 3, ... n. ¿Qué claves dan origen a la división de una *Pagina*? ¿Qué claves hacen que la altura del árbol crezca? Las claves son eliminadas en el mismo orden en que fueron insertadas. ¿Qué claves hacen que las *Paginas* se queden con un número de claves menor que 2 y den lugar a la unión de dos? ¿Qué claves hacen que la altura del árbol disminuya?

17.15. La búsqueda de una clave en un árbol B se ha implementado siguiendo una estrategia recursiva, implementar de nuevo la operación con una estrategia iterativa.

17.16. Se asume que la unidad de disco contiene un número de pistas, a su vez cada pista está dividida en un número de bloques de tamaño *m*. En cuanto a tiempo de proceso, las operaciones que realiza un programa de acceso a los archivos ubicados en disco prevalecen frente a las operaciones en memoria; el tiempo de acceso a disco es del orden de decenas de milisegundo (cada vez se mejora este tiempo). Con el fin de minimizar el tiempo de acceso al disco, el orden de árbol B debe ser tal que permita que una hoja de tamaño máximo quepa en un bloque del disco. Determinar el número máximo de accesos al disco necesario para encontrar un elemento en un árbol B de *n* elementos.

17.17. Se estima que un archivo va a tener 1 millón de registros, cada registro ocupa 50 bytes de memoria. En el supuesto de que cada bloque tenga una capacidad de 1.000 bytes y que la referencia al bloque ocupa 4 bytes, diseñar una organización con árboles B para este archivo.

17.18. Un árbol B* se considera un tipo de árbol B con la característica adicional de que cada nodo está, al menos, lleno en las dos terceras partes, en lugar de la mitad como ocurre a un árbol B, excepto quizá el nodo raíz. La inserción de una nueva clave en el árbol B* sigue los mismos pasos que la inserción en un árbol B, salvo que si el nodo donde se ha de insertar está lleno, se mueven claves entre el nodo padre, el hermano (con un número de claves menor que el máximo), y el nodo donde se inserta para dejar hueco y realizar la operación. De esta forma, se pospone la división del nodo hasta que los hermanos estén completos, entonces éstos pueden dividirse en tres nodos, cada uno estará con una ocupación de las dos terceras partes. Codificar los cambios que necesitan los métodos que implementan la operación de inserción para aplicarla a un árbol B*.

17.19. Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80; dibuja el árbol B* de orden 5 que se construye con esas claves.

PROBLEMAS

17.1. Dado un archivo de texto construir un árbol AVL con todas sus palabras y frecuencia. El archivo se denomina *carta.dat.*

17.2. Añadir al programa escrito en el Problema 17.1 una función que devuelva el número de veces que aparece en el texto, una palabra dada.

17.3. En el archivo *alumnos.txt* se encuentran los nombres completos de los alumnos de las escuelas taller de la Comunidad Alcarreña. Escribir un programa para leer el archivo y formar, inicialmente, un árbol de búsqueda con respecto a la clave *apellido*. Una vez formado el árbol, construir con sus nodos un árbol de Fibonacci.

17.4. La implementación de la operación insertar en un árbol equilibrado se realiza de manera natural en forma recursiva. Escribir de nuevo la codificación aplicando una estrategia iterativa.

17.5. Un archivo F contiene las claves, enteros positivos, que forman un árbol binario de búsqueda equilibrado R. El archivo se grabó en el recorrido por niveles del árbol R. Escribir un programa que realice las siguientes tareas: *a*) Leer el archivo F para volver a construir el árbol equilibrado. *b*) Buscar una clave, requerida al usuario, y en el caso de que esté en el árbol mostrar las claves que se encuentran en el mismo nivel del árbol.

17.6. La implementación de la operación *eliminar* en un árbol binario equilibrado se ha realizado en forma recursiva. Escribir de nuevo la codificación aplicando una estrategia iterativa.

17.7. En un archivo se ha almacenado los habitantes de n pueblos de la comarca natural *Peñas Rubias*. Cada registro del archivo tiene el nombre del pueblo y el número de habitantes. Se quiere asociar los nombres de cada habitante a cada pueblo, para lo que se ha pensado en una estructura de datos que consta de un array de n elementos. Cada elemento tiene el nombre del pueblo y la raíz de un árbol AVL con los nombres de los habitantes del pueblo.
Escribir un programa que cree la estructura. Como entrada de datos, los nombres de los habitantes que se insertarán en el árbol AVL del pueblo que le corresponde.

17.8. La operación de borrar una clave en un árbol AVL se ha realizado de tal forma que cuando el nodo de la clave a eliminar tiene las dos ramas se reemplaza por la clave mayor del subárbol izquierdo. Implementar la operación de borrado de tal forma que cuando el nodo a eliminar tenga dos ramas reemplace, aleatoriamente, por la clave mayor de la rama izquierda o por la clave menor de la rama derecha.

17.9. Al problema descrito en 17.7 se desea añadir la posibilidad de realizar operaciones sobre la estructura. Así, añadir la posibilidad de cambiar el nombre de una persona de un determinado pueblo. Esta operación debe de mantener el árbol como árbol de búsqueda, al cambiar el nombre y ser la clave de búsqueda el nombre puede ocurrir que se rompa la condición. Otra opción que debe de permitir es dar de baja un pueblo entero de tal forma que todos sus habitantes se añadan a otro pueblo de la estructura. Por último, una vez que se vaya a terminar la ejecución del programa grabar en un archivo cada pueblo con sus respectivos habitantes.

17.10. Una empresa de servicios tiene tres departamentos: *comercial*(*1*), *explotación*(*2*) y *marketing*(*3*). Cada empleado está adscrito a uno de ellos. Se ha realizado una redistribución del personal entre ambos departamentos, los cambios están guardados en el archivo *laboral.txt*. El archivo contiene en cada registro los campos: *identificador*, *origen*, *destino*. El campo origen puede tomar los valores 1, 2, 3 dependiendo del departamento origen del empleado, cuya identificación es una secuencia de 5 dígitos que es el primer campo del registro. El campo destino también puede tomar los valores 1, 2, 3 según el departamento al que es destinado.

Escribir un programa que guarde los registros del archivo en tres árboles AVL, uno por cada departamento origen y realice los intercambios de registros en los árboles, según el campo *destino*.

17.11. Escribir la codificación del método: `void listadoEnRango(int c1, int c2);` que muestre las claves de un árbol B mayores que c_1 y menores que c_2.

17.12. Escribir la codificación de el método `listadoDecreciente()`, que recorra y muestre las claves de un árbol B en orden decreciente.

17.13. Cada uno de los centros de enseñanza del estado consta de una biblioteca escolar. Cada centro de enseñanza está asociado con número de orden (valor entero), los centros de cada provincia tienen números consecutivos, en el rango de las unidades de 1.000. Así por ejemplo, a Madrid le corresponde del 1 al 1.000, a Toledo del 1.001 al 2.000 ... Escribir un programa que permita gestionar la información indicada, formando una estructura en memoria de árbol B con un máximo de 4 claves por página. La clave de búsqueda del árbol B es el número de orden del centro, además, tiene asociado el nombre del centro. El programa debe permitir añadir centros, eliminar, buscar la existencia de un centro por la clave y listar los centros existentes.

17.14. En el Problema 17.13 cuando se termina la ejecución se pierde toda la información. Modificar el programa para que al terminar la información se grabe la estructura en un archivo de nombre `centros.txt`. Escribir un programa que permita leer el archivo `centros.txt` para generar, a partir de el, la estructura árbol B. La estructura puede experimentar modificaciones: nuevos centros, eliminación de alguno existente, por consiguiente, al terminar la ejecución debe escribirse de nuevo el árbol en el archivo.

17.15. Se quiere dar más contenido a la información tratada en el Problema 17.13. Ya se ha especificado que la clave de búsqueda del árbol B es el número de orden del centro de enseñanza. Además, cada clave tiene que llevar asociada la raíz de un árbol binario de búsqueda que representa a los títulos de la biblioteca del centro. El árbol de búsqueda biblioteca tiene como campo clave el *título* del libro (tiene más campos, como *autor* …). Escribir un programa que partiendo de la información guardada en el archivo *centros.txt* cree un nuevo árbol B con los centros y el árbol binario de títulos de la biblioteca de cada centro.

17.16. A la estructura ya creada del Problema 17.13, añadir los métodos necesarios para realizar las siguientes operaciones:

- Dada una provincia cuyo rango de centros es conocido, por ejemplo de 3.001 a 3.780, eliminar en todos sus centros escolares los libros que estén repetidos en cada centro, y que informe del total de libros liberados.
- Dado un centro n, en su biblioteca se desea que de ciertos libros haya m ejemplares.

17.17. Implemente el TAD árbol B en memoria externa, en disco. Al hacerlo en memoria externa los enlaces deben ser números de registro en los que se almacena el nodo o página descendiente. Para crear un nuevo nodo hay que buscar un "*hueco*" en el archivo, representado por el número de registro, y escribir el nodo. Cada vez que se modifica un nodo, por una inserción de una clave o un borrado hay que escribir el registro.