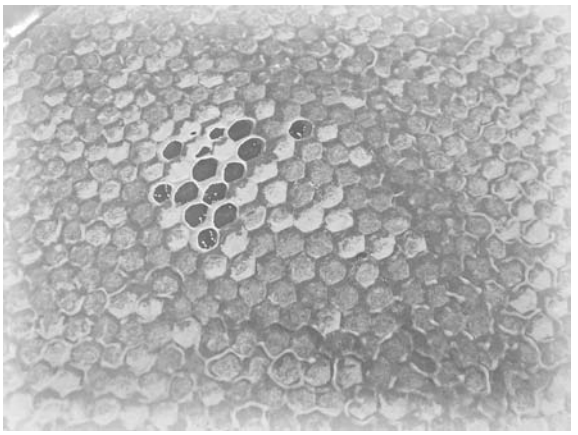


CAPÍTULO 7

Árboles

7.1 Introducción

Este capítulo estudia la estructura de datos conocida con el nombre de ***árbol***. Presenta sus principales características, cómo se relacionan sus componentes y analiza las operaciones que pueden aplicárseles.

Los árboles son estructuras de datos no lineales. Cada elemento, conocido con el nombre de **`nodo`**, puede tener varios sucesores. En términos generales, un árbol se define como una colección de nodos donde cada uno, además de almacenar información, guarda la dirección de sus sucesores. Se conoce la dirección de uno de los nodos, llamado **`raíz`**, y a partir de él se tiene acceso a todos los otros miembros de la estructura.

Existen diversas maneras de representar un árbol, las más comunes son: grafos, anidación de paréntesis y diagramas de Venn. La figura

7.1 muestra un árbol representado por medio de un grafo, en el cual cada nodo está indicado por un círculo y la relación entre ellos por un arco.

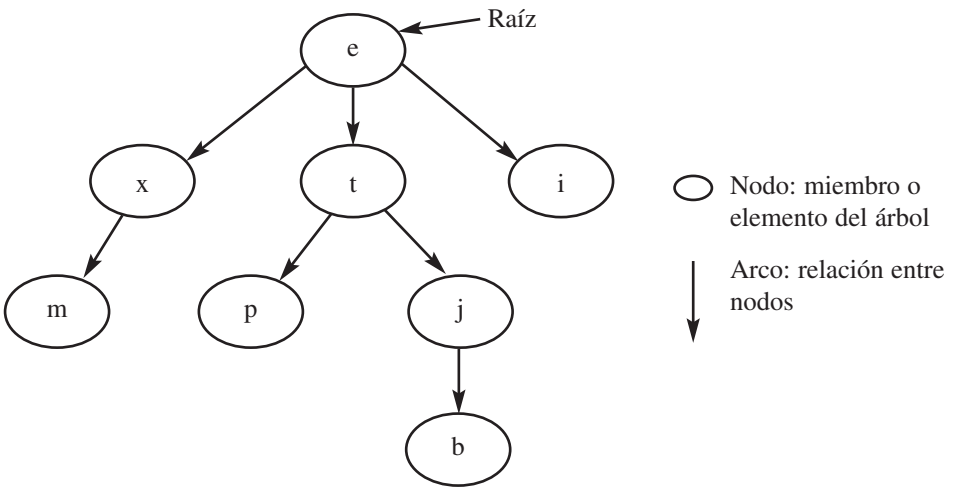


FIGURA 7.1 *Estructura* Árbol *representada con un grafo*

Al observar la representación del árbol por medio de una gráfica puede observarse a cada nodo como un árbol. Por consiguiente se dice que un árbol está formado por 0 o más subárboles, llegando así a una definición recursiva de esta estructura de datos.

En una estructura tipo árbol se definen relaciones entre sus miembros. A continuación se presentan las relaciones más importantes y se ejemplifican usando el árbol de la figura 7.1.

- ***Hijo***. Se dice que un nodo es hijo (o descendiente) de otro si este último apunta al primero. El nodo que almacena el valor *t* es hijo del nodo que almacena el valor *e*. Los nodos *p* y *j* son hijos de *t*.
- ***Padre***. Se dice que un nodo es padre de otro si este último es apuntado por el primero. El nodo que almacena el valor *j* es padre del nodo que almacena el valor *b*. El nodo *e* es padre de *x*, *t* e *i*.
- El origen de cada arco está en el nodo padre y la flecha llega al nodo hijo.
- ***Hermano***. Dos nodos son hermanos si son apuntados por el mismo nodo, es decir si tienen el mismo padre. Los nodos *x*, *t* e *i* son hermanos.

Además, los nodos pertenecen a una de las siguientes categorías según su ubicación en la estructura.

- ***Raíz***. Se dice que un nodo es raíz si a partir de él se relacionan todos los otros nodos. Si un árbol no es vacío, entonces tiene un único nodo raíz. En la figura 7.1, el nodo raíz es el que almacena el valor *e*.
- ***Hoja*** o ***terminal***. Se dice que un nodo es una hoja del árbol (o terminal) si no tiene hijos. En la figura 7.1, los nodos que almacenan los valores *m*, *p*, *b* e *i* son hojas o nodos terminales.
- ***Interior***. Se dice que un nodo es interior si no es raíz ni hoja. En la figura 7.1, los nodos que almacenan los valores *x*, *t*, y *j* son nodos interiores.

Se define el nivel y grado de cada nodo y la altura y el grado del árbol de la siguiente manera:

- ***Nivel de un nodo***. Se dice que el nivel de un nodo es el número de arcos que deben ser recorridos, partiendo de la raíz, para llegar hasta él. La raíz tiene nivel 1. En el árbol de la figura 7.1, el nivel de los nodos que almacenan los valores *x*, *t* e *i* es 2 y el nivel del nodo *b* es 4.
- ***Altura del árbol***. Se dice que la altura de un árbol es el máximo de los niveles, considerando todos sus nodos. El árbol de la figura 7.1 tiene una altura igual a 4.
- ***Grado de un nodo***. Se dice que el grado de un nodo es el número de hijos que tiene dicho nodo. En la figura 7.1, el grado del nodo que almacena el valor *t* es 2 y el grado del nodo *x* es 1.
- ***Grado del árbol***. Se dice que el grado de un árbol es el máximo de los grados, considerando todos sus nodos. El árbol de la figura 7.1, es de grado 3.

7.2 Árboles binarios

Un ***árbol binario*** es un árbol de grado 2 en el cual sus hijos se identifican como subárbol izquierdo y subárbol derecho. Por lo tanto, cada nodo almacena información y las direcciones de sus descendientes (máximo 2). Es un tipo de árbol muy usado, ya que saber el número máximo de hijos que puede tener cada nodo facilita las operaciones sobre ellos. La figura 7.2 presenta el esquema de un nodo de un árbol binario.

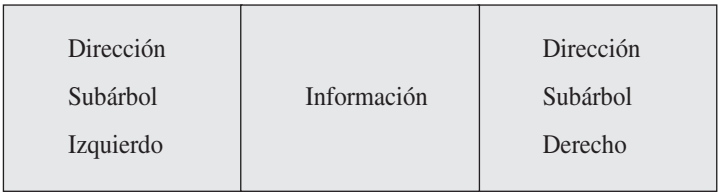


FIGURA 7.2 *Estructura de un nodo de un árbol binario*

La figura 7.3 presenta un ejemplo de uso de un árbol binario. En este caso, la estructura se emplea para almacenar el árbol genealógico de *María*. En cada nodo se guarda la información de los ancestros de *María* y los arcos indican la relación entre ellos.

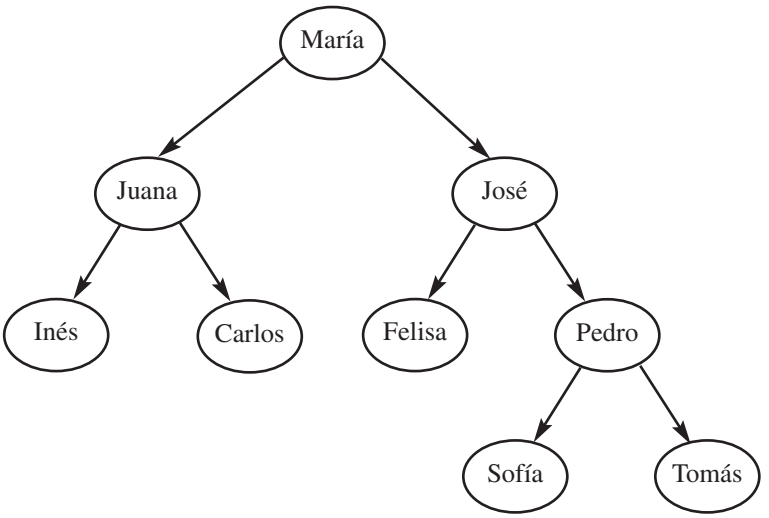


FIGURA 7.3 *Ejemplo de árbol binario*

La característica de este tipo de árbol (cada nodo tiene máximo 2 hijos) se puede aprovechar para organizar la información. Retomando el ejemplo de la figura 7.3, se puede establecer que los hijos izquierdos representen los ascendientes femeninos de *María*, mientras que los hijos derechos los ascendientes masculinos. Así, la mamá de *María* es *Juana* y su abuela materna es *Inés*. El papá de *María* es *José*. A su vez, la mamá de *José* es *Felisa* y su papá es *Pedro*.

La implementación más efectiva de los árboles binarios es por medio de memoria dinámica, obteniendo así una estructura dinámica. Las figuras 7.4 y 7.5

presentan las plantillas de la clase `NodoArbol` y de la clase `ArbolBinario` respectivamente. Se usan plantillas para dar mayor generalidad a la solución. La clase `NodoArbol` tiene tres atributos, uno que representa la información a almacenar por lo que se define de tipo **`T`**, y otros dos que representan la dirección del hijo izquierdo y del hijo derecho respectivamente, por lo que se declaran como punteros a objetos de la misma clase. Por su parte, la clase `ArbolBinario` tiene un único atributo que representa la dirección del primer elemento del árbol (la raíz) por lo cual es de tipo puntero a un objeto de tipo `NodoArbol`.

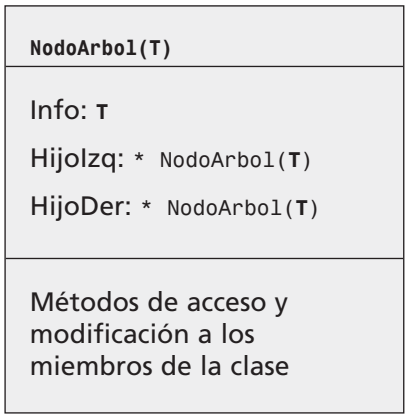


FIGURA 7.4 *Clase* `NodoArbol`

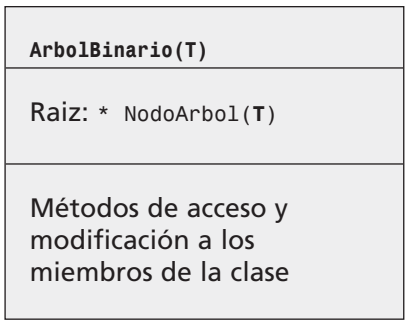


FIGURA 7.5 *Clase* `ArbolBinario`

A continuación se presenta el código en lenguaje ***C++*** correspondiente a la definición de las plantillas de las clases `NodoArbol` y `ArbolBinario`.

```
/* Prototipo de la plantilla de la clase ArbolBinario. De esta manera,
➥en la clase NodoArbol se podrá hacer referencia a ella. */
template <class T>
class ArbolBinario;

/* Declaración de la clase NodoArbol. Cada nodo almacena la información
➥(que es la razón de ser de la estructura tipo árbol) y las direcciones
➥de sus hijos izquierdo y derecho. En la sección pública se establece la
➥relación de amistad entre esta clase y la clase ArbolBinario para que los
➥métodos de esta última puedan tener acceso a sus miembros privados. */
template <class T>
class NodoArbol
{
   private:
       T Info;
       NodoArbol<T> *HijoIzq;
       NodoArbol<T> *HijoDer;
   public:
       NodoArbol();
       T RegresaInfo();
       friend class ArbolBinario<T>;
};

/* Declaración del método constructor por omisión. Inicializa las ligas
➥a los subárboles con el valor de NULL, indicando que están vacías. */
template <class T>
NodoArbol<T>::NodoArbol()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que permite conocer la información almacenada en el nodo. */
template <class T>
T NodoArbol<T>::RegresaInfo()
{
   return Info ;
}


/* Declaración de la clase ArbolBinario. Su atributo es un puntero al
➥nodo raíz. */
template <class T>
class ArbolBinario
{
   private:
       NodoArbol<T> *Raiz;
```

```
    public:
        ArbolBinario ();
        /* En esta sección se declaran los métodos de acceso y
        ➥modificación a los miembros de la clase. */
};

/* Declaración del método constructor. Inicializa el puntero a la raíz con
➥el valor NULL, indicando que el árbol está vacío (no tiene nodos). */
template <class T>
ArbolBinario<T>::ArbolBinario()
{
    Raiz= NULL;
}
```

La clase `NodoArbol` se utiliza para representar un nodo de un árbol binario, por lo tanto se incluyen tres atributos: uno para almacenar información de cualquier tipo (tipo `T`) y los otros dos para almacenar la dirección de los subárboles izquierdo y derecho respectivamente, los cuales son punteros a objetos de la misma clase. La sección pública contiene tres miembros (podría tener más o menos), dependiendo de la definición de la clase que se haga. Estos elementos son: el método constructor, un método que facilita (a usuarios externos a la clase) conocer la información guardada, y la declaración de amistad con la clase `ArbolBinario`. Esta última declaración permite que los métodos de la clase amiga tengan acceso a sus miembros privados y protegidos.

A partir de la clase `NodoArbol` se define la clase `ArbolBinario`, la cual está formada por un atributo único (tipo puntero a un objeto `NodoArbol`) que representa el puntero al nodo raíz del árbol binario. Este puntero permite el acceso a todos los elementos del árbol ya que la raíz tiene la dirección de sus dos hijos, éstos, la dirección de sus respectivos hijos y así hasta llegar a nodos terminales. En la sección pública se declaran los métodos necesarios para tener acceso a los atributos, y de esta manera manipular la información almacenada.

7.2.1 Operaciones en árboles binarios

En esta sección se estudian las operaciones de creación y recorrido de un árbol binario. La primera hace referencia a crear una estructura que responda a las características analizadas e ir almacenando información en cada uno de los nodos.

La segunda permite visitar todos los nodos de un árbol sin repetir ninguno, aprovechando el conocimiento que se tiene acerca de la estructura.

La ***creación*** de un árbol binario se lleva a cabo a partir de la raíz. Se crea un nodo y se almacena su información. Posteriormente se pregunta si dicho nodo tiene hijo izquierdo, si la respuesta es afirmativa, entonces se invoca nuevamente el método pero ahora con el subárbol izquierdo. El proceso se repite con cada nodo hasta llegar a las hojas. Luego, se hace lo mismo para crear cada uno de los subárboles derechos. Se utiliza la instrucción **`new()`** para asignar un espacio de memoria de manera dinámica.

A continuación se presenta el método para llevar a cabo la secuencia de pasos descrita.

```
/* Plantilla del método que crea un árbol binario. Recibe como parámetro
➥un apuntador a un subárbol. La primera vez es la raíz del árbol la cual
➥se inicializó con el valor NULL, indicando que el árbol está vacío. */
template <class T>
void ArbolBinario<T>::CreaArbol(NodoArbol<T> *Apunt)
{
   char Resp;
   /* Se crea un nodo. */
   Apunt= new  NodoArbol<T>;
   cout<<"\n\nIngrese la información a almacenar:";
   cin>>Apunt->Info;
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo izquierdo (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
   {
        /* Se invoca al método con el subárbol izquierdo. Se usa la
        ➥definición recursiva de un árbol. */
        CreaArbol(Apunt->HijoIzq);
        Apunt->HijoIzq= Raiz;
   }
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo derecho (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
   {
        /* Se invoca al método con el subárbol derecho. Se usa la
        ➥definición recursiva de un árbol. */
        CreaArbol(Apunt->HijoDer);
        Apunt->HijoDer= Raiz;
   }
   Raiz= Apunt;
}
```

El ***recorrido*** de un árbol binario consiste en visitar todos sus nodos una sola vez. Por lo tanto, podrá hacerse (aprovechando las características de la estructura del árbol) de tres maneras diferentes: visitando la **raíz**, el **hijo izquierdo** y el **hijo derecho**, o visitando el **hijo izquierdo**, la **raíz** y el **hijo derecho**, o bien, visitando el **hijo izquierdo**, el **hijo derecho** y la **raíz**. En los tres casos, la regla se aplica hasta llegar a las hojas. Estos métodos se conocen con el nombre de ***preorden***, ***inorden*** y ***postorden*** respectivamente.

Preorden	Inorden	Postorden
1. Visita la raíz	**1.** Recorre el subárbol izquierdo	**1.** Recorre el subárbol izquierdo
2. Recorre el subárbol izquierdo	**2.** Visita la raíz	**2.** Recorre el subárbol derecho
3. Recorre el subárbol derecho	**3.** Recorre el subárbol derecho	**3.** Visita la raíz

Considerando el árbol binario de la figura 7.6, el resultado de los tres recorridos es el siguiente.

- **Preorden**: 304 – 550 – 143 – 2020 – 1995 – 876 – 609 – 300
- **Inorden**: 143 – 550 – 2020 – 304 – 876 – 1995 – 609 – 300
- **Postorden**: 143 – 2020 – 550 – 876 – 300 – 609 – 1995 – 304

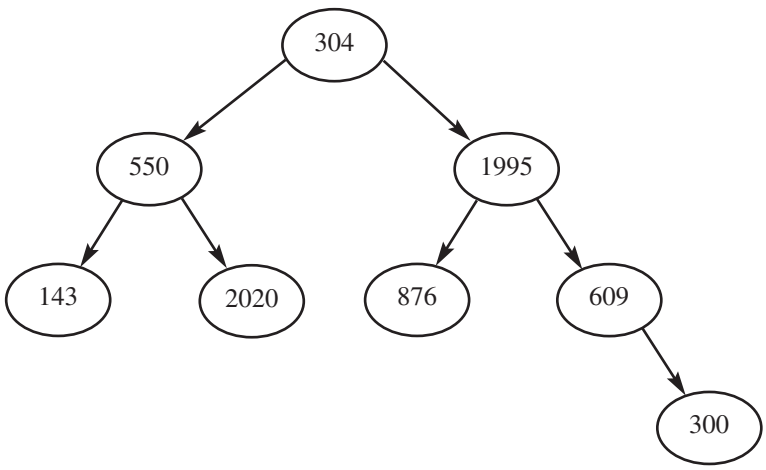


FIGURA 7.6 *Recorrido de árboles binarios*

Los métodos para llevar a cabo esta operación se presentan a continuación. En los tres casos la operación de visitar la raíz se consideró como la impresión de su contenido, aunque podría ser cualquier operación válida según el tipo de información almacenada en el nodo.

```
/* Método que realiza el recorrido preorden de un árbol binario. Se usa
➥el recorrido para imprimir la información almacenada en cada uno de sus
➥nodos. Recibe como parámetro el nodo a visitar. La primera vez es la
➥raíz del árbol, luego será la raíz del subárbol izquierdo y la raíz del
➥subárbol derecho y así hasta llegar a las hojas. */
template <class T>
void ArbolBinario<T>::Preorden (NodoArbol<T> *Apunt)
{
   if (Apunt)
   {
        cout<< Apunt->Info << " ";
        Preorden(Apunt->HijoIzq);
        Preorden(Apunt->HijoDer);
   }
}

/* Método que realiza el recorrido inorden de un árbol binario. Se usa
➥el recorrido para imprimir la información almacenada en cada uno de sus
➥nodos. Recibe como parámetro el nodo a visitar. La primera vez es la
➥raíz del árbol, luego será la raíz del subárbol izquierdo y la raíz del
➥subárbol derecho y así hasta llegar a las hojas. */
template <class T>
void ArbolBinario<T>::Inorden (NodoArbol<T> *Apunt)
{
   if (Apunt)
   {
        Inorden(Apunt->HijoIzq);
        cout<< Apunt->Info << " ";
        Inorden(Apunt->HijoDer);
   }
}

/* Método que realiza el recorrido postorden de un árbol binario. Se usa
➥el recorrido para imprimir la información almacenada en cada uno de sus
➥nodos. Recibe como parámetro el nodo a visitar. La primera vez es la
➥raíz del árbol, luego será la raíz del subárbol izquierdo y la raíz del
➥subárbol derecho y así hasta llegar a las hojas. */
template <class T>
void ArbolBinario<T>::Postorden (NodoArbol<T> *Apunt)
```

```
{
   if (Apunt)
   {
        Postorden(Apunt->HijoIzq);
        Postorden(Apunt->HijoDer);
        cout<< Apunt->Info << " ";
   }
}
```

En los tres métodos, la instrucción de imprimir se da sobre el contenido de la raíz. La naturaleza recursiva de los métodos permite lograr la impresión de todos los nodos. Las instrucciones que forman cada uno de los métodos son las mismas, lo único que cambia es el orden en el cual se ejecutan.

Si se analiza el recorrido preorden con el árbol de la figura 7.6, se puede observar que primero se imprime el número 304, luego se invoca el método con el subárbol izquierdo, quedando pendiente el recorrido con el subárbol derecho (internamente se guardan en una pila las instrucciones pendientes de ejecutar). Lo mismo sucede cuando llega con el subárbol izquierdo, imprime el valor 550 e invoca el método con su subárbol izquierdo y deja pendiente el recorrido con su subárbol derecho. Una vez agotado el lado izquierdo, pasa al lado derecho del último nodo visitado. Se van tomando de la pila todos los subárboles derechos que quedaron pendientes de visitar y se van recorriendo. Como consecuencia, el primer número (correspondiente a un subárbol derecho) que se imprime es el 2020, luego el 1995 y como este último tiene subárbol izquierdo, entonces se invoca al método con éste (876). Se repite el proceso hasta que ya no queden nodos a visitar.

En un árbol binario también pueden realizarse otras operaciones como buscar, insertar o eliminar un dato en un árbol ya generado. Estas operaciones serán analizadas en un tipo especial de árboles binarios, los cuales se tratan en la siguiente sección.

El programa 7.1 presenta una aplicación de árboles binarios. El programa imprime los datos de todos los ascendientes femeninos de un individuo, tanto de la rama materna como de la paterna. Se utiliza un objeto tipo árbol binario para almacenar los datos de los ascendientes de una persona, es decir, su árbol genealógico. En la raíz de cada subárbol izquierdo se almacena la información de un ascendiente femenino, mientras que en la raíz de cada subárbol derecho se guarda la información de un ascendiente masculino. Considerando el árbol de la figura 7.7, algunas de las relaciones familiares representadas son:

Anahí es mamá de Juan
José es papá de Juan
Inés es mamá de Anahí
Pedro es papá de Anahí
Ana es mamá de José
Luis es papá de José

y el programa imprimirá que los ascendientes femeninos de Juan son Anahí, Inés y Ana. El programa 7.1 incluye las clases `ArbolBinario` y `Personas` sólo con los métodos requeridos para la aplicación.

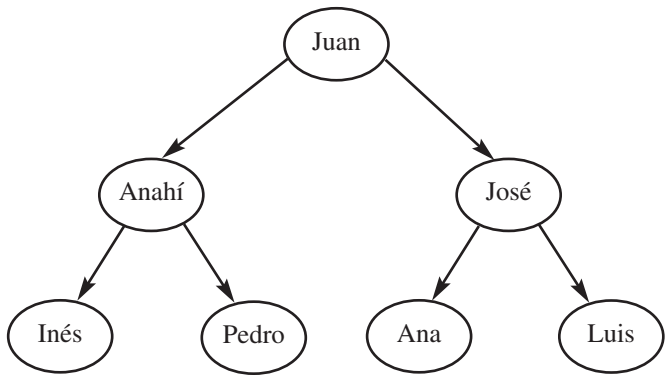


FIGURA 7.7 *Árbol genealógico de* Juan

Programa 7.1

```
/* Programa que imprime los datos de los ascendientes femeninos de un
➥individuo. Primero forma el árbol genealógico y posteriormente genera
➥el reporte. */

/* Definición de la clase Persona. */
class Persona
{
   private:
       int AnioNac, Vive;
       char NomPers[64], LugNac[64];
```

```
   public:
       Persona();
       Persona(int, int, char[], char[]);
       friend  istream & operator>> (istream & , Persona & );
       friend  ostream & operator<< (ostream & , Persona & );
};

/* Declaración del método constructor por omisión. */
Persona::Persona()
{}

/* Declaración del método constructor con parámetros. */
Persona::Persona(int ANac, int Vi, char NomP[], char LugN[])
{
   AnioNac= ANac;
   Vive= Vi;
   strcpy(NomPers, NomP);
   strcpy(LugNac, LugN);
}

/* Sobrecarga del operador >> para permitir la lectura de objetos tipo
➥Persona de manera directa. */
istream & operator>>(istream & Lee,  Persona & ObjPers)
{
   cout<<"\n\nIngrese nombre de la Persona:";
   Lee>> ObjPers.NomPers;
   cout<<"\n\nIngrese año de nacimiento:";
   Lee>> ObjPers.AnioNac;
   cout<<"\n\nIngrese lugar de nacimiento:";
   Lee>> ObjPers.LugNac;
   cout<<"\n\n¿Está viva?:";
   Lee>> ObjPers.Vive;
   return  Lee;
}

/* Sobrecarga del operador << para permitir la escritura de objetos tipo
➥Persona de manera directa. */
ostream & operator<< (ostream & Escribe, Persona & ObjPers)
{
   Escribe<<"\n\nDatos de la Persona\n";
   Escribe<<"\nNombre:  "<<ObjPers.NomPers;
   Escribe<<"\nLugar de nacimiento:  "<<ObjPers.LugNac;
   Escribe<<"\nAño de nacimiento:  "<<ObjPers.AnioNac;
   if (ObjPers.Vive == 1)
       Escribe<<"\nEstá viva.\n";
   else
       Escribe<<"\nNo está viva.\n";
   return  Escribe;
}
```

```
/* Prototipo de la plantilla de la clase ArbolBinario. Así, en la clase
➥NodoArbol se podrá hacer referencia a ella. */
template <class T>
class ArbolBinario;

/* Declaración de la clase NodoArbol. Cada nodo almacena la información
➥que es la razón de ser de la estructura tipo árbol y las direcciones de
➥su hijo izquierdo y de su hijo derecho. */
template <class T>
class NodoArbol
{
   private:
       T Info;
       NodoArbol<T> *HijoIzq;
       NodoArbol<T> *HijoDer;
   public:
       NodoArbol();
       T RegresaInfo();
       void ActualizaInfo(T);
       friend class ArbolBinario<T>;
};

/* Declaración del método constructor por omisión. Inicializa
➥las ligas a los subárboles con el valor de NULL. Indica nodo sin
➥descendientes. */
template <class T>
NodoArbol<T>::NodoArbol()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que regresa la información almacenada en el nodo. */
template <class T>
T NodoArbol<T>::RegresaInfo()
{
   return Info;
}

/* Método para actualizar la información almacenada en el nodo. */
template <class T>
void NodoArbol<T>::ActualizaInfo(T Dato)
{
   Info= Dato ;
}
```

```
/* Declaración de la clase ArbolBinario. Tiene un puntero al nodo
➥raíz. */
template <class T>
class ArbolBinario

{
   private:
       NodoArbol<T> *Raiz;
   public:
       ArbolBinario ();
       NodoArbol<T> *RegresaRaiz();
       void CreaArbol(NodoArbol<T> *);
       void ImprimeIzq(NodoArbol<T> *);
};

/* Declaración del método constructor. Inicializa el puntero a la raíz
➥con el valor NULL. Indica que el árbol está vacío. */
template <class T>
ArbolBinario<T>::ArbolBinario()
{
   Raiz= NULL;
}

/* Método que regresa el valor del apuntador a la raíz del árbol. */
template <class T>
NodoArbol<T> *ArbolBinario<T>::RegresaRaiz()
{
   return Raiz;
}

/* Método que crea un árbol binario. */
template <class T>
void ArbolBinario<T>::CreaArbol(NodoArbol<T> *Apunt)
{
   char Resp;
   Apunt= new  NodoArbol<T>;
   cout<<"\n\nIngrese la información a almacenar:";
   cin>>Apunt->Info;
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo izquierdo (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
   {
       CreaArbol(Apunt->HijoIzq);
       Apunt->HijoIzq= Raiz;
   }
   cout<<"\n\n"<<Apunt->Info<<" ¿Tiene hijo derecho (S/N)? ";
   cin>>Resp;
   if (Resp  ==  's')
```

```
    {
        CreaArbol(Apunt->HijoDer);
        Apunt->HijoDer= Raiz;
    }
    Raiz= Apunt;
}

/* Método que imprime la información almacenada en las raíces de todos
➥los subárboles izquierdos. La primera vez recibe como dato la raíz del
➥árbol. */
template <class T>
void ArbolBinario<T>::ImprimeIzq(NodoArbol<T> *Apunt)
{
    if (Apunt)
    {
        if (Apunt->HijoIzq)
        {
             cout<<Apunt->HijoIzq->Info;
             ImprimeIzq(Apunt->HijoIzq);
        }
        ImprimeIzq(Apunt->HijoDer);
    }
}

/* Función principal. Crea el árbol genealógico de un individuo y
➥posteriormente imprime los datos de todos sus ascendientes femeninos. */
void main()
{
    ArbolBinario<Persona> Genealogico;
    Persona Individuo;
    NodoArbol<Persona> *Ap;

    Ap= Genealogico.RegresaRaiz();
    /* Se invoca el método que crea el árbol genealógico. */
    Genealogico.CreaArbol(Ap);
    Ap= Genealogico.RegresaRaiz();

    /* Se recupera la información del individuo. */
    Individuo= Ap->RegresaInfo();
    cout<<"\n\n\n_____\n\n";
    cout<<"Los ascendientes femeninos de: \n"<<Individuo;
    cout<<"\n\n_____\n";

    /* Se invoca el método que imprime los datos de los ascendientes
    ➥femeninos. */
    Genealogico.ImprimeIzq(Ap);
}
```

7.2.2 Árboles binarios de búsqueda

Un ***árbol binario de búsqueda*** se caracteriza porque la información de cada nodo es mayor que la información de cada uno de los nodos que están en su subárbol izquierdo y menor que la almacenada en los nodos que están en su subárbol derecho. La figura 7.8 presenta un ejemplo de árbol binario de búsqueda. Observe que todos los valores que están a la izquierda del 710 son menores que él. A su vez, los que están a su derecha son mayores. La misma regla se aplica en todos los nodos.

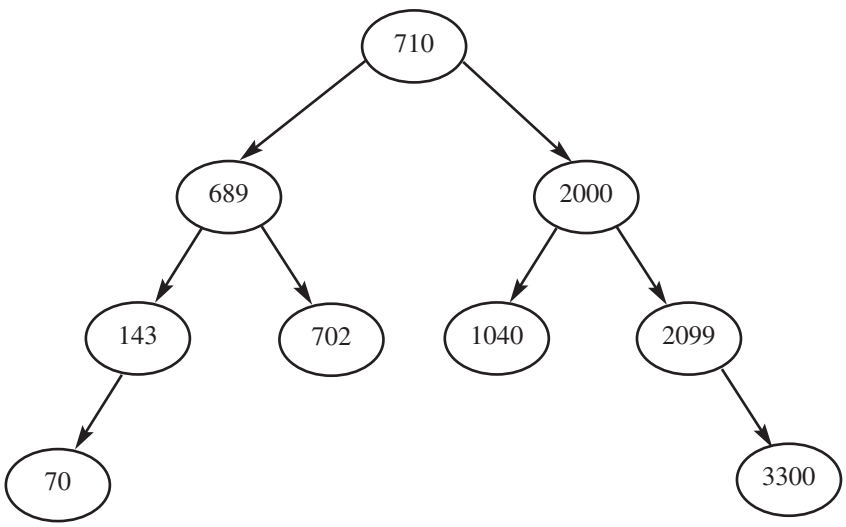


FIGURA 7.8 *Ejemplo de árbol binario de búsqueda*

El recorrido inorden de un árbol binario de búsqueda genera una lista ordenada de manera creciente de todos sus elementos. Tomando el árbol de la figura 7.8, este recorrido proporcionaría los elementos en el siguiente orden:

70 – 143 – 689 – 702 – 710 – 1040 – 2000 – 2099 – 3300

El orden que existe entre la información almacenada en el árbol facilita la operación de búsqueda de cualquiera de sus elementos. A continuación se analizarán las operaciones de búsqueda, inserción y eliminación en árboles binarios de búsqueda.

Operación de búsqueda

Para llevar a cabo la **búsqueda** de un elemento en un árbol binario de búsqueda se procede de la siguiente manera:

1. Se evalúa si el nodo visitado (la primera vez es la raíz) está definido.
2. Si la respuesta es afirmativa, se pregunta si el dato buscado es menor que el dato visitado.
   2.1. Si la respuesta es afirmativa, se procede a buscar el dato en el subárbol izquierdo.
   2.2. Si la respuesta es negativa, se pregunta si el dato buscado es mayor que el dato visitado.
      2.2.1. Si la respuesta es afirmativa, se procede a buscar el dato en el subárbol derecho.
      2.2.2. Si la respuesta es negativa, la búsqueda termina exitosamente. El dato fue encontrado.
3. Si la respuesta a la pregunta 1 es negativa, entonces termina la búsqueda con un fracaso.

A continuación se presenta un ejemplo de aplicación del algoritmo visto. En el árbol de la figura 7.9 se desea encontrar el valor 705. Con las líneas punteadas se señalan los nodos que se van visitando hasta llegar al buscado. En la tabla 7.1 se muestra la secuencia de pasos requeridos, usando el algoritmo, para realizar esta operación.

TABLA 7.1 Operación de búsqueda en un *árbol binario de búsqueda*

Operación	*Descripción*
1	Se evalúa si el nodo visitado (710) está definido. En este caso sí lo está.
2	Se evalúa si el dato buscado (705) es menor que la información almacenada (710) en el nodo visitado. En este caso sí lo es.
3	Se invoca el método con el subárbol izquierdo.
4	Se evalúa si el nodo visitado (689) está definido. En este caso sí lo está.
5	Se evalúa si el dato buscado (705) es menor que la información almacenada (689) en el nodo visitado. En este caso no lo es.

continúa

TABLA 7.1 Continuación

Operación	*Descripción*
6	Se evalúa si el dato buscado (705) es mayor que la información almacenada (689) en el nodo visitado. En este caso sí lo es.
7	Se invoca el método con el subárbol derecho.
8	Se evalúa si el nodo visitado (702) está definido. En este caso sí lo está.
9	Se evalúa si el dato buscado (705) es menor que la información almacenada (702) en el nodo visitado. En este caso no lo es.
10	Se evalúa si el dato buscado (705) es mayor que la información almacenada (702) en el nodo visitado. En este caso sí lo es.
11	Se invoca el método con el subárbol derecho.
12	Se evalúa si el nodo visitado (705) está definido. En este caso sí lo está.
13	Se evalúa si el dato buscado (705) es menor que la información almacenada (705) en el nodo visitado. En este caso no lo es.
14	Se evalúa si el dato buscado (705) es mayor que la información almacenada (705) en el nodo visitado. En este caso no lo es.
15	La búsqueda termina con éxito. El dato fue encontrado.

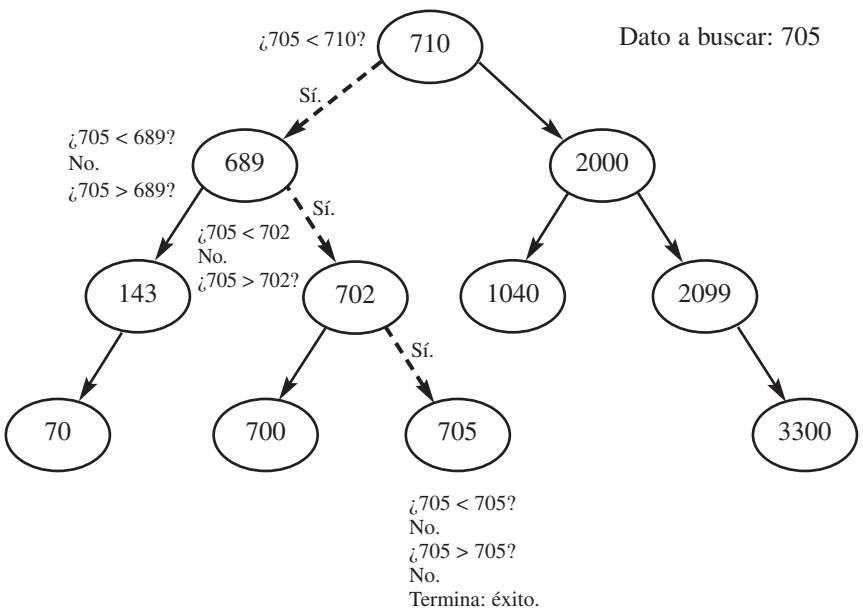


FIGURA 7.9 *Ejemplo de operación de búsqueda*

Como se puede deducir de los pasos presentados, el número de comparaciones se reduce a la mitad en cada nodo visitado. No se requiere visitar todos los nodos. El método que implementa esta operación es el siguiente:

```
/* Método que busca un dato en un árbol binario de búsqueda. Recibe como
➥parámetros un apuntador, que es la dirección del nodo a visitar (la
➥primera vez es el apuntador a la raíz) y el dato a buscar. Regresa como
➥resultado la dirección del nodo encontrado o el valor NULL, si la bús-
➥queda termina con fracaso. */
template <class T>
NodoArbol<T> * ArbolBinBus<T>::Busqueda (NodoArbol<T> *Apunt, T Dato)
{
   if (Apunt)
        if (Dato < Apunt->Info)
            return Busqueda(Apunt->HijoIzq, Dato);
        else
            if (Dato > Apunt->Info)
                return Busqueda(Apunt->HijoDer, Dato);
            else
                return Apunt;
   else
        return NULL;
}
```

Operación de inserción

Insertar un nuevo elemento en un árbol binario de búsqueda requiere buscar la posición que debe ocupar el nuevo nodo de tal manera que no altere el orden del árbol. En la solución que aquí se propone no se aceptan elementos repetidos. Pueden existir aplicaciones en las cuales sí se permita. Sin embargo, en esta solución, si se detecta que en el árbol ya está almacenado un valor igual al que se pretende insertar, la operación se interrumpe. Los principales pasos para llevar a cabo esta operación son:

1. Se evalúa si el nodo (la primera vez es la raíz) visitado está definido.
2. Si la respuesta es afirmativa, se pregunta si el dato a insertar es menor que el dato visitado.
   2.1. Si la respuesta es afirmativa, se invoca el proceso de inserción con el subárbol izquierdo.
   2.2. Si la respuesta es negativa, se pregunta si el dato a insertar es mayor que el dato visitado.

2.2.1. Si la respuesta es afirmativa, entonces se invoca el proceso de inserción con el subárbol derecho.

2.2.2. Si la respuesta es negativa, entonces el proceso de inserción termina sin haberse realizado, ya que no se permiten elementos repetidos.

3. Si la respuesta al paso 1 es negativa, se crea el nuevo nodo, se almacena la información y se establecen las ligas entre el nuevo nodo y su padre.

Suponga que en el árbol de la figura 7.10 se quiere insertar el valor 1500. Las líneas punteadas indican los nodos visitados hasta encontrar el adecuado (1040) para proceder a la inserción. El nodo en negritas es el nuevo elemento agregado al árbol y la línea (también en negritas) es la liga entre éste y su padre. Aplicando el algoritmo dado, se realiza la secuencia de operaciones que se muestra en la tabla 7.2.

TABLA 7.2 Operación de inserción en un *árbol binario de búsqueda*

Operación	*Descripción*
1	Se evalúa si el nodo visitado (710) está definido. En este caso sí lo está.
2	Se evalúa si el dato a insertar (1500) es menor que la información almacenada (710) en el nodo visitado. En este caso no lo es.
3	Se evalúa si el dato a insertar (1500) es mayor que la información almacenada (710) en el nodo visitado. En este caso sí lo es.
4	Se invoca el método con el subárbol derecho.
5	Se evalúa si el nodo visitado (2000) está definido. En este caso sí lo está.
6	Se evalúa si el dato a insertar (1500) es menor que la información almacenada (2000) en el nodo visitado. En este caso sí lo es.
7	Se invoca el método con el subárbol izquierdo.
8	Se evalúa si el nodo visitado (1040) está definido. En este caso sí lo está.
9	Se evalúa si el dato a insertar (1500) es menor que la información almacenada (1040) en el nodo visitado. En este caso no lo es.
10	Se evalúa si el dato a insertar (1500) es mayor que la información almacenada (1040) en el nodo visitado. En este caso sí lo es.
11	Se invoca el método con el subárbol derecho.
12	Se evalúa si el nodo visitado (NULL) está definido. En este caso no lo está.
13	Se crea un nuevo nodo, se le asigna el valor 1500 y se establece la liga entre él y su padre (el nodo que almacena el número 1040). El proceso termina.

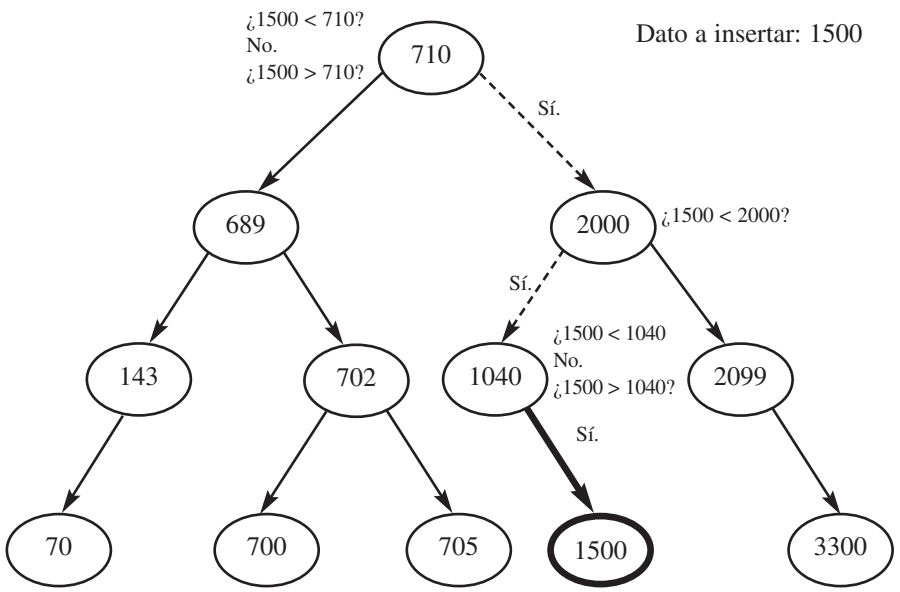


FIGURA 7.10 *Ejemplo de operación de inserción*

A continuación se presenta el método definido para insertar un nuevo nodo a un árbol binario de búsqueda.

```
/* Método que inserta un nodo en un árbol binario de búsqueda. Recibe
➥como parámetros un apuntador (la primera vez es la raíz del árbol) y
➥la información que se quiere almacenar en el nuevo nodo. En esta
➥implementación no se permite que haya información duplicada en el
➥árbol. */
template <class T>
void ArbolBinBus<T>::InsertaNodoSinRep(NodoArbol<T> *Apunt, T Dato)
{
   NodoArbol<T> *ApAux;
   if (Apunt)
   {
        if (Dato < Apunt->Info)
        {
            InsertaNodoSinRep(Apunt->HijoIzq, Dato);
            Apunt->HijoIzq= Raiz;
        }
```

```
            else
                if (Dato > Apunt->Info)
                {
                     InsertaNodoSinRep(Apunt->HijoDer, Dato);
                     Apunt->HijoDer= Raiz;
                }
            Raiz= Apunt;
        }
        else
        {
            /* Se crea un nuevo nodo, se le asigna la información y se
            ➥establecen las ligas entre los nodos correspondientes. */
            ApAux= new  NodoArbol<T>();
            ApAux->Info= Dato;
            Raiz= ApAux;
        }
    }
```

Operación de eliminación

Para **eliminar** un elemento en un árbol binario de búsqueda se requiere buscar el valor deseado y quitar el nodo que lo contiene. Este último paso se lleva a cabo de maneras diferentes dependiendo si el nodo eliminado es terminal o no. Si se trata de una hoja, entonces se quita directamente. En otro caso, para no perder las ligas a sus descendientes, se debe reemplazar por el nodo que se encuentra más a la derecha del subárbol izquierdo o por el que se encuentra más a la izquierda del subárbol derecho. En la solución que se da en este libro se usa el elemento que está más a la derecha del subárbol izquierdo. Los principales pasos para llevar a cabo esta operación son:

1. Se evalúa si el nodo visitado (la primera vez es la raíz) está definido.
2. Si la respuesta es afirmativa, entonces se pregunta si el dato a eliminar es menor que el dato visitado.
   2.1. Si la respuesta es afirmativa, se invoca el proceso de eliminación con el subárbol izquierdo.
   2.2. Si la respuesta es negativa, se pregunta si el dato a eliminar es mayor que el dato visitado.

2.2.1. Si la respuesta es afirmativa, se invoca el proceso de eliminación con el subárbol derecho.

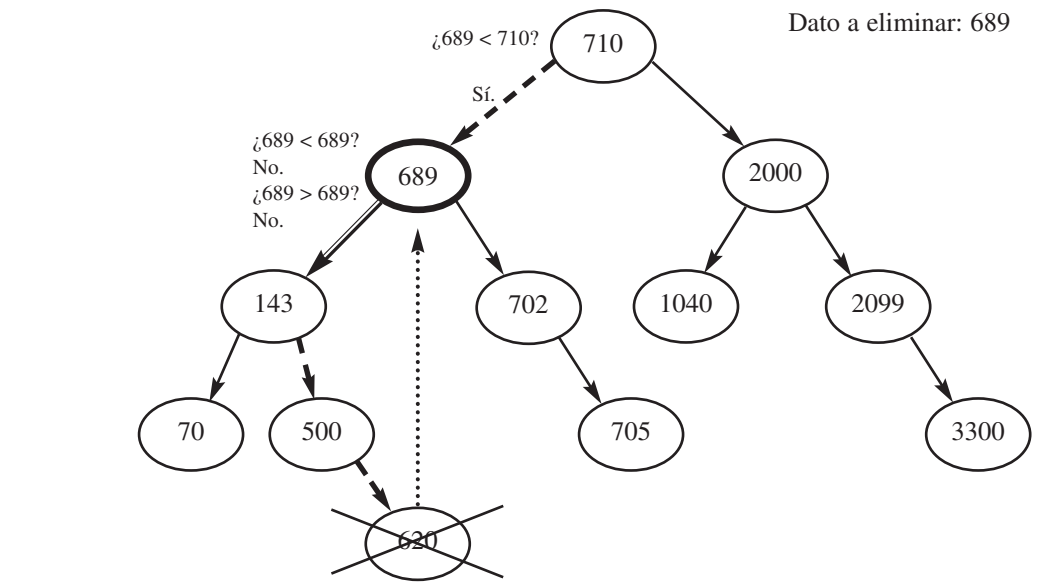
2.2.2. Si la respuesta es negativa, se elimina el nodo. Si es hoja, la eliminación es directa. Si tiene sólo un hijo se reemplaza por éste y si tiene dos se reemplaza por el que está más a la derecha del subárbol izquierdo. En estos dos últimos casos se libera el espacio de memoria del hijo, mientras que en el primero el correspondiente al nodo en cuestión.

3. Si la respuesta al paso 1 es negativa, entonces el dato no está en el árbol. El proceso de eliminación termina con fracaso.

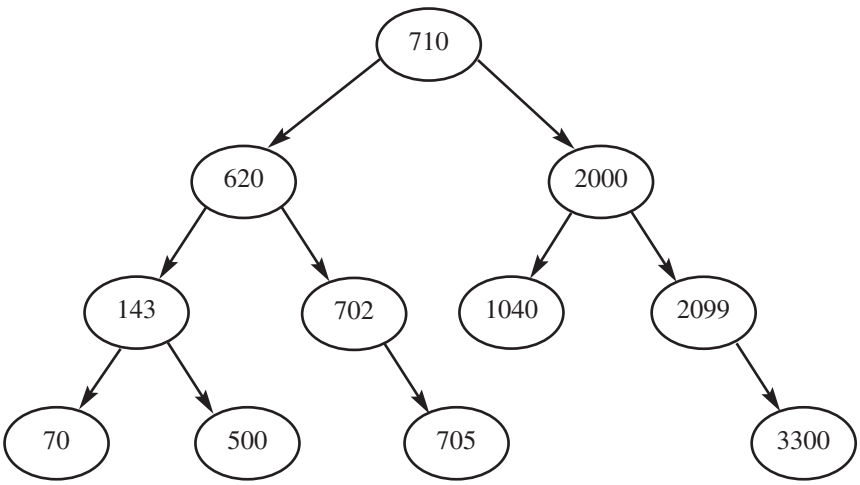
Suponga que en el árbol binario de búsqueda de la figura 7.11 se quiere eliminar el valor 689. En (a) se muestra el camino que se sigue para llegar al nodo deseado y al nodo cuyo contenido reemplazará al 689. En (b) se presenta el árbol una vez que el nodo fue eliminado. La tabla 7.3 presenta las operaciones realizadas, siguiendo el algoritmo dado, para eliminar el 689.

TABLA 7.3 Operación de eliminación en un *árbol binario de búsqueda*

Operación	*Descripción*
1	Se evalúa si el nodo visitado (710) está definido. En este caso sí lo está.
2	Se evalúa si el dato a eliminar (689) es menor que la información almacenada (710) en el nodo visitado. En este caso sí lo es.
3	Se invoca el método con el subárbol izquierdo.
4	Se evalúa si el nodo visitado (689) está definido. En este caso sí lo está.
5	Se evalúa si el dato a eliminar (689) es menor que la información almacenada (689) en el nodo visitado. En este caso no lo es.
6	Se evalúa si el dato a eliminar (689) es mayor que la información almacenada (689) en el nodo visitado. En este caso no lo es.
7	Se llegó al nodo que se quiere quitar. Como tiene dos hijos se reemplaza su contenido con el del nodo que está más a la derecha del subárbol izquierdo y se libera el espacio de memoria correspondiente al hijo. El proceso termina.



a)



b)

FIGURA 7.11 *Ejemplo de la operación de eliminación a) Antes de eliminar el valor 689, b) después de eliminarlo y reemplazarlo por el 620*

A continuación se presenta el método, escrito en ***C++***, que permite eliminar un nodo de un árbol binario de búsqueda.

```
/* Método que elimina un nodo de un árbol binario de búsqueda. Recibe
➥como parámetro un apuntador (la primera vez es la raíz) y el dato a
➥eliminar. */
template <class T>
void ArbolBinBus<T>::EliminaNodo(NodoArbol<T> *Apunt, T Dato)
{
   if (Apunt)
        if (Dato < Apunt->Info)
        {
            EliminaNodo(Apunt->HijoIzq, Dato);
            Apunt->HijoIzq= Raiz;
        }
        else
            if (Dato > Apunt->Info)
            {
                 EliminaNodo(Apunt->HijoDer, Dato);
                 Apunt->HijoDer= Raiz;
            }
            else
            {
                 NodoArbol<T> *ApAux1,*ApAux2,*ApAux3;
                 ApAux3= Apunt;
                 /* Encuentra el nodo que contiene el dato a eliminar.
                 ➥Verifica si tiene hijos. */
                 if (!ApAux3->HijoDer)
                      if (!ApAux3->HijoIzq)
                      /* Si no tiene hijo derecho ni izquierdo, entonces
                      ➥se redefine como vacío. */
                           Apunt= NULL;
                      else
                           /* Si sólo tiene hijo izquierdo, el nodo
                           ➥eliminado se reemplaza con éste.*/
                           Apunt= ApAux3->HijoIzq;
                 else
                      if (!ApAux3->HijoIzq)
                           /* Si sólo tiene hijo derecho, el nodo
                           eliminado se reemplaza con éste. */
                           Apunt= ApAux3->HijoDer;
                      else
                      {
                           /* Si tiene ambos hijos, entonces se reempla-
                           ➥za (en esta solución) por el nodo que está
                           ➥más a la derecha del subárbol izquierdo. */
```

```
                        ApAux1= ApAux3->HijoIzq;
                        ApAux2= ApAux3;
                        while (ApAux1->HijoDer)
                        {
                            ApAux2= ApAux1;
                            ApAux1= ApAux1->HijoDer;
                        }
                        ApAux3->Info= ApAux1->Info;
                        if (ApAux3 == ApAux2)
                            ApAux3->HijoIzq= NULL;
                        else
                            if (!ApAux1->HijoIzq)
                                ApAux2->HijoDer= NULL;
                            else
                                ApAux2->HijoDer= ApAux1->HijoIzq;
                        ApAux3= ApAux1;
                    }
                delete(ApAux3);
            }
            Raiz= Apunt;
}
```

El método dado, como el correspondiente a la inserción, se definió de tipo **`void`**. Sin embargo, ambos pueden modificarse y declararse enteros de tal manera que regresen un valor que indique si la operación se llevó a cabo o no con éxito.

El programa 7.2 presenta la plantilla de la clase árbol binario de búsqueda con los métodos analizados. También incluye la plantilla correspondiente a la clase que define al nodo del árbol. Por razones de espacio, de algunos métodos ya explicados sólo se escribe el prototipo y el encabezado.

Programa 7.2

```
/* Prototipo de la plantilla de la clase ArbolBinBus. Así, en la clase
➥NodoArbol se podrá hacer referencia a ella. */
template <class T>
class ArbolBinBus;

/* Declaración de la clase NodoArbol. Cada nodo almacena la información
➥(razón de ser de la estructura tipo árbol) y las direcciones de sus hijos
➥izquierdo y derecho. Se incluye una relación de amistad con la clase
➥ArbolBinBus para que éste pueda tener acceso a sus miembros privados. */
```

```
template <class T>
class NodoArbol
{
   private:
       T Info;
       NodoArbol<T> *HijoIzq;
       NodoArbol<T> *HijoDer;
   public:
       NodoArbol();
       T RegresaInfo() ;
       void ActualizaInfo(T);
       friend class ArbolBinBus<T>;
};

/* Declaración del método constructor por omisión. Inicializa
las ligas a los subárboles con el valor NULL, indicando que no tiene
➥hijos. */
template <class T>
NodoArbol<T>::NodoArbol()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que regresa la información almacenada en el nodo. */
template <class T>
NodoArbol<T>::RegresaInfo()
{
   return Info ;
}

/* Método para actualizar la información almacenada en el nodo. */
template <class T>
void NodoArbol<T>::ActualizaInfo(T Dato)
{
   Info= Dato ;
}

/* Declaración de la clase ArbolBinBus. Su atributo es un puntero al
➥nodo raíz. */
template <class T>
class ArbolBinBus
{
   private:
       NodoArbol<T> *Raiz;
```

```
   public:
       ArbolBinBus ();
       NodoArbol<T> *RegresaRaiz();
       void Preorden (NodoArbol<T> *);
       void Inorden (NodoArbol<T> *);
       void Postorden (NodoArbol<T> *);
       NodoArbol<T> * Busqueda (NodoArbol<T> *, T);
       void InsertaNodoSinRep (NodoArbol<T> *, T);
       void EliminaNodo (NodoArbol<T> *, T);
};

/* Declaración del método constructor. Inicializa el puntero a la raíz
➥con el valor NULL, indicando árbol vacío (no tiene nodos). */
template <class T>
ArbolBinBus<T>::ArbolBinBus()
{
   Raiz= NULL;
}

/* Método que regresa el valor del apuntador a la raíz del árbol. */
template <class T>
NodoArbol<T> *ArbolBinBus<T>::RegresaRaiz()
{
   return Raiz;
}

/* Método que realiza el recorrido preorden de un árbol binario de bús-
➥queda. Recibe como parámetro el nodo a visitar (la primera vez es la
➥raíz). */
template <class T>
void ArbolBinBus<T>::Preorden (NodoArbol<T> *Apunt)
{
   /* Ya analizado, razón por la que se omite. */
}

/* Método que realiza el recorrido inorden de un árbol binario de bús-
➥queda. Recibe como parámetro el nodo a visitar (la primera vez es la
➥raíz). */
template <class T>
void ArbolBinBus<T>::Inorden (NodoArbol<T> *Apunt)
{
   /* Ya analizado, razón por la que se omite. */
}

/* Método que realiza el recorrido postorden de un árbol binario de bús-
➥queda. Recibe como parámetro el nodo a visitar (la primera vez es la
➥raíz). */
template <class T>
void ArbolBinBus<T>::Postorden (NodoArbol<T> *Apunt)
```

```
{
   /* Ya analizado, razón por la que se omite. */
}

/* Método que busca un dato en un árbol binario de búsqueda. Recibe como
➥parámetros la dirección del nodo a visitar (la primera vez es la raíz)
➥y el dato a buscar. Regresa como resultado la dirección del nodo
➥encontrado o el valor NULL, si la búsqueda termina con fracaso. */
template <class T>
NodoArbol<T> * ArbolBinBus<T>::Busqueda (NodoArbol<T> *Apunt, T Dato)
{
   if (Apunt)
       if (Dato < Apunt->Info)
           return Busqueda(Apunt->HijoIzq, Dato);
       else
           if (Dato > Apunt->Info)
                return Busqueda(Apunt->HijoDer, Dato);
           else
                return Apunt;
   else
       return NULL;
}

/* Método que inserta un nodo en un árbol binario de búsqueda. Recibe como
➥parámetros la dirección del nodo a visitar (la primera vez es la raíz) y
➥la información que se quiere almacenar en el nuevo nodo. En esta imple-
➥mentación no se permite que haya información duplicada en el árbol. */
template <class T>
void ArbolBinBus<T>::InsertaNodoSinRep(NodoArbol<T> *Apunt, T Dato)
{
   /* Ya analizado, razón por la que se omite. */
}

  /* Método que elimina un nodo del árbol binario de búsqueda. Recibe
➥como parámetro la dirección del nodo a visitar (la primera vez es la
➥raíz) y el dato a eliminar. */
template <class T>
void ArbolBinBus<T>::EliminaNodo(NodoArbol<T> *Apunt, T Dato)
{
   /* Ya analizado, razón por la que se omite. */
}
```

El programa 7.3 presenta un ejemplo de aplicación de la estructura árbol binario de búsqueda. Utiliza la clase `Producto` definida en el programa 6.2 del capítulo anterior y la clase `ArbolBinBus` del programa 7.2. Ambas se incluyen por medio de bibliotecas. El programa permite crear un árbol cuyos nodos guardarán la información de los productos; esta información se almacena ordenadamente según su clave. Además, ofrece la opción de dar de baja o buscar un producto y generar un reporte de todos los productos ordenados por claves.

Programa 7.3

```
/* Este programa es para almacenar un conjunto de productos (ordenados
➥por clave), utilizando un árbol binario de búsqueda. Además, se pueden
➥eliminar y buscar productos ya registrados y generar un reporte con la
➥información de todos los productos. La biblioteca "Productos.h" tiene
➥la clase Producto utilizada en el programa 6.2. Por su parte, la
➥biblioteca "ArbolBinBusqueda.h" contiene la plantilla de la clase
➥ArbolBinBus del programa 7.2. */

#include "Productos.h"
#include "ArbolBinBusqueda.h"

/* Función que despliega al usuario las opciones de trabajo. Regresa la
➥opción seleccionada. */
int Menu()
{
   int Opcion;
   do {
       cout<<"\n\n\n\tOpciones de trabajo:\n";
       cout<<"\t1.Ingresar nuevo producto.\n";
       cout<<"\t2.Dar de baja un producto.\n";
       cout<<"\t3.Reporte de todos los productos ordenados por
       ➥clave.\n";
       cout<<"\t4.Buscar un producto por clave.\n";
       cout<<"\t5.Terminar el proceso.\n\n";
       cout<<"\tIngrese opción seleccionada:  ";
       cin>>Opcion;
     } while (Opcion <1 || Opcion > 5);
     return Opcion;
}


/* Función principal desde la cual se controla la ejecución de las
➥operaciones seleccionadas por el usuario. */
void main()
{
   ArbolBinBus<Producto> Inventario;
   NodoArbol<Producto> *Ap1, *Ap2;
   Producto Prod;
   int Opc, Cla;

   do {
       Opc= Menu();
       switch (Opc)
       {
         /* Se registra un nuevo producto. No se aceptan productos con
         ➥claves repetidas. */
```

```
            case 1:{
                    cin>>Prod;
                    Ap1= Inventario.RegresaRaiz();
                    Inventario.InsertaNodoSinRep(Ap1, Prod);
                    break;
                    }
            /* Se elimina un producto ya registrado. */
            case 2:{
                    cout<<"\n\nIngrese la clave del producto a eliminar:";
                    cin>>Cla;
                    Producto Prod(Cla, "", 0);
                    Ap1= Inventario.RegresaRaiz();
                    Inventario.EliminaNodo(Ap1, Prod);
                    break;
                    }
            /* Con el método Inorden se genera un reporte de todos los
            ➥productos ordenados por clave. */
            case 3:{
                    Ap1= Inventario.RegresaRaiz();
                    cout<<"\n\n\n————————————————————————\n\n";
                    cout<<"PRODUCTOS EN INVENTARIO\n\n";
                    cout<<"————————————————————————\n\n";
                    Inventario.Inorden(Ap1);
                    break;
                    }
            /* Se busca un elemento por su clave. Si ya está registrado
            ➥entonces se despliegan todos sus datos. En caso contrario,
            ➥sólo un mensaje informativo. */
            case 4: {

                    cout<<"\n\nIngrese la clave del producto a buscar:";
                    cin>>Cla;
                    Producto Prod(Cla, "", 0);
                    Ap1= Inventario.RegresaRaiz();
                    Ap2= Inventario.Busqueda(Ap1, Prod);
                    if (Ap2)
                    {
                        cout<<"\n\n\nExiste un producto registrado con esa
                        ➥clave.\n";
                        cout<<Ap2->RegresaInfo();
                    }
                    else
                        cout<<"\n\nNo se ha registrado ningún producto con
                              ➥esa clave. \n";
                    break;
                    }
            case 5: cout<<"\n\n\nFIN DEL PROCESO.\n\n\n";
                    break;
            }
    } while (Opc >=1 & &  Opc < 5);
}
```

7.3 Árboles balanceados

Un ***árbol balanceado*** es un árbol binario de búsqueda en el cual la diferencia entre la altura de su subárbol derecho y la altura de su subárbol izquierdo *es menor o igual a 1*. De esta manera se controla el crecimiento del árbol y se garantiza mantener la eficiencia en la operación de búsqueda. La diferencia entre las alturas de los subárboles se conoce como *factor de equilibrio (FE)*, el cual se expresa como se muestra a continuación:

FE = altura hijo derecho – altura hijo izquierdo

La figura 7.12 muestra un árbol binario de búsqueda en el que cada nodo tiene un factor de equilibrio asociado. La raíz tiene un *FE* igual a **1** ya que el subárbol derecho tiene una altura de 3 y el izquierdo de 2. El nodo que almacena el valor 99 tiene un *FE* igual a **–1** porque su subárbol derecho tiene altura 0 y el izquierdo 1. En cambio, el nodo que guarda el 508 tiene un *FE* igual a **0** porque sus dos subárboles tienen la misma altura. Al observar los *FE* de todos los nodos se puede afirmar que dicho árbol está balanceado, porque éstos son, en valor absoluto, menores o iguales a uno.

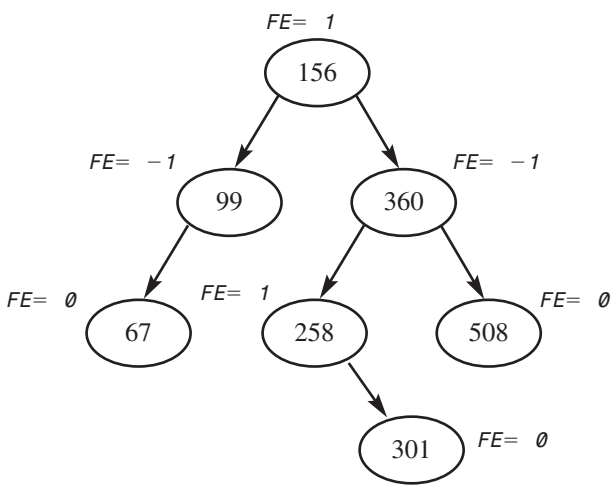


FIGURA 7.12 *Árbol binario de búsqueda con factores de equilibrio*

Considere que los elementos: 24 – 31 – 87 – 99 – 105 y 126 se almacenan, en ese orden, en un árbol binario de búsqueda. Luego de insertar los 6 valores, se obtiene un árbol como el de la figura 7.13. Si se tuviera que buscar un dato en este árbol se tendrían que hacer tantas comparaciones como nodos haya antes de llegar al deseado. En estructuras semejantes se pierden todas las ventajas que ofrecen los árboles. Para evitar que esto suceda surgen los árboles balanceados en los que se realizan balanceos o ajustes de los nodos luego de efectuar inserciones o eliminaciones que hayan provocado la pérdida del equilibrio.

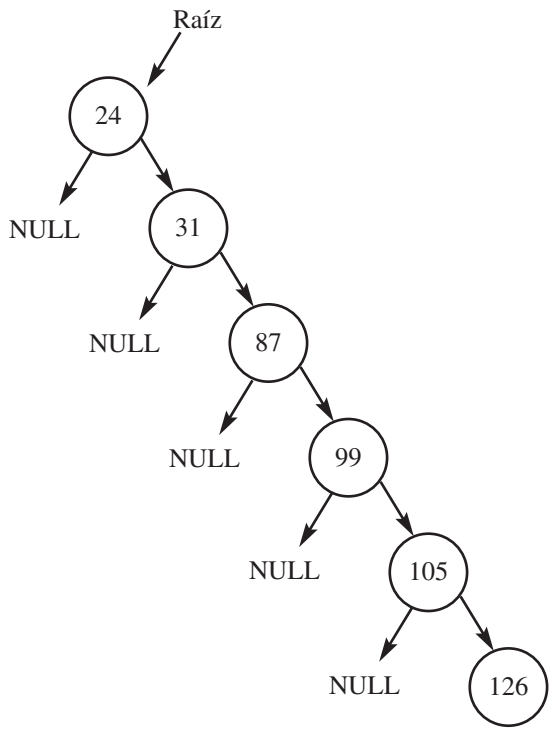


FIGURA 7.13 *Inserción provocando desequilibrio en un árbol binario de búsqueda*

Reacomodo del árbol

Para lograr que un árbol siga estando balanceado, luego de cada inserción o eliminación, se debe determinar si su factor de equilibrio es mayor a 1. En caso afirmativo, se deben reacomodar los nodos de tal manera que se vuelva a tener un valor menor o igual a 1. Este movimiento se denomina **rotación**. La rotación puede ser simple o compuesta dependiendo del número de nodos que participen.

La **rotación simple** se presenta cuando están involucrados dos hijos derechos (HD-HD) o dos hijos izquierdos (HI-HI), y afecta sólo las ligas de dos nodos. La figura 7.14 presenta gráficamente las dos variantes de este tipo de rotación.

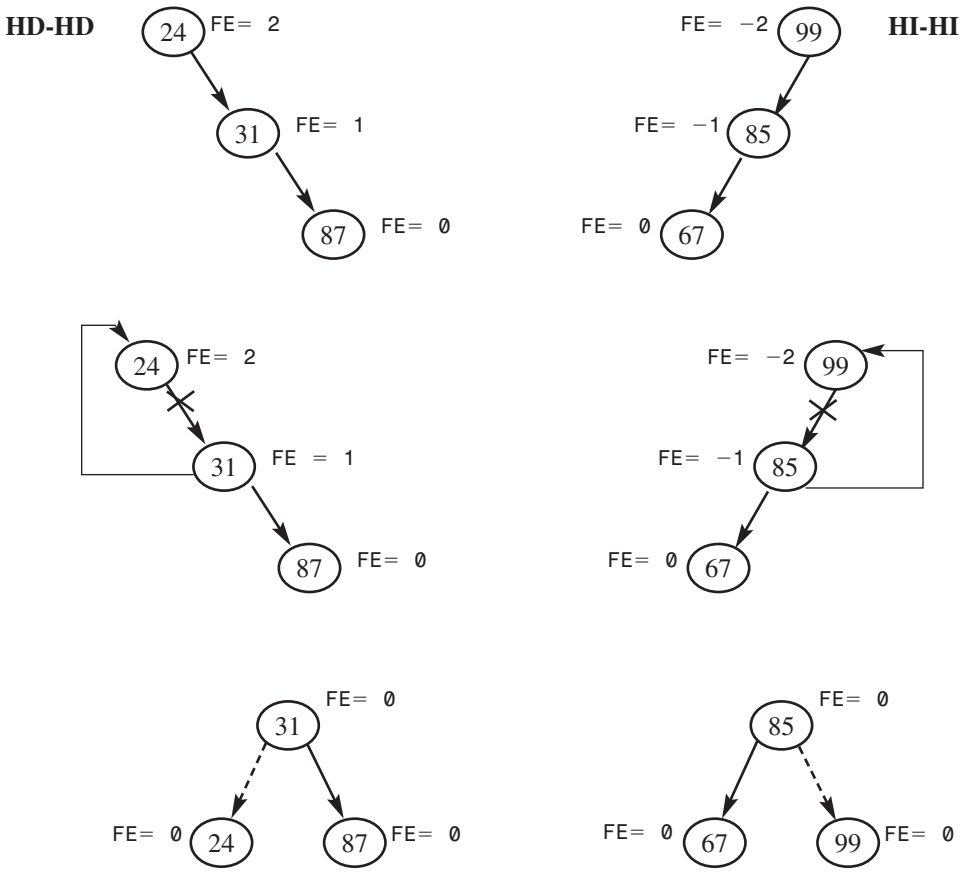


FIGURA 7.14 *Rotación simple*

Analizando el caso de la rotación HD-HD se observa que el nodo con información 24 tiene un *FE* igual a 2, lo cual indica que se ha perdido el equilibrio en su rama derecha. A su vez, en el nodo que guarda el 31 se tiene un *FE* igual a 1, con lo cual se puede determinar que la rotación debe involucrar también a la rama derecha de éste. Luego de efectuar la rotación, este nodo tiene como hijo izquierdo a su padre recuperando así el equilibrio.

En el caso de la rotación HI-HI, se tiene el nodo que almacena el valor 99 con un *FE* igual a –2, lo cual indica que se ha perdido el equilibrio en su rama izquierda. Además, el nodo con información 85 tiene un *FE* igual a –1, con lo cual se puede determinar que la rotación también debe involucrar a la rama izquierda de éste. Luego de efectuar la rotación, este último nodo tiene como hijo derecho a su padre.

Más adelante se explica cómo deben actualizarse los punteros a los nodos afectados durante la rotación simple.

La **rotación compuesta** se presenta cuando están involucrados un hijo derecho y un hijo izquierdo (HD-HI) o un hijo izquierdo y uno derecho (HI-HD), y afecta las ligas de tres nodos. La figura 7.15 presenta gráficamente las dos variantes de este tipo de rotación.

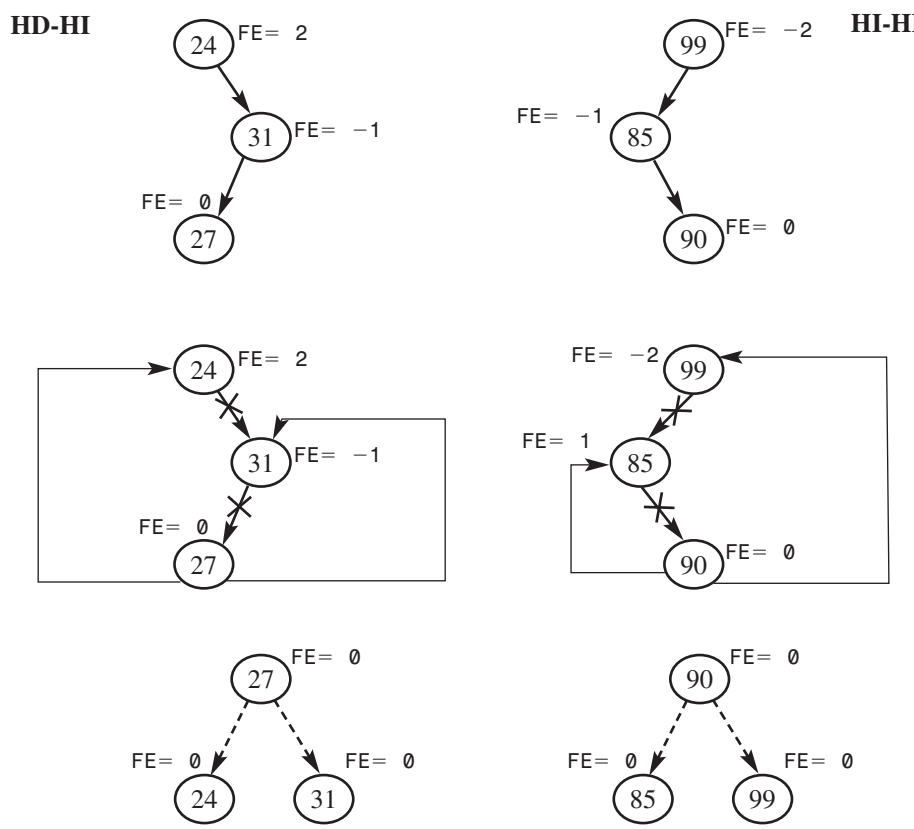


FIGURA 7.15 *Rotación compuesta*

Analizando el caso de la rotación HD-HI se observa que el nodo con información 24 tiene un *FE* igual a 2, lo cual indica que se ha perdido el equilibrio en su rama derecha. A su vez, en el nodo que guarda el 31 se tiene un *FE* igual a –1, lo que implica que su rama izquierda tiene mayor altura que su rama derecha. Por tanto, la rotación que se debe aplicar es la de hijo derecho-hijo izquierdo. Luego de efectuar la rotación, el nodo 27 queda como padre del 24 (antes su *abuelo*) y del 31 (antes su padre).

En el caso de la rotación HI-HD se observa que el nodo con información 99 tiene un *FE* igual a –2, lo cual indica que se ha perdido el equilibrio en su rama izquierda. Por otra parte, su hijo izquierdo (el nodo que guarda el 85) tiene un *FE* igual a 1, lo que implica que su rama derecha tiene mayor altura que su rama izquierda. Por lo tanto, la rotación que se debe aplicar es la de hijo izquierdo-hijo derecho. Luego de efectuar la rotación, el nodo que almacena el 90 queda como padre del 99 (antes su *abuelo*) y del 85 (antes su padre).

Más adelante se explica cómo deben actualizarse los punteros a los nodos afectados durante la rotación compuesta.

En los árboles balanceados, cada nodo debe almacenar (además de los elementos ya vistos) su factor de equilibrio. Luego de insertar o eliminar un nodo se calculan los factores de equilibrio de todos los nodos involucrados en la operación y, dependiendo del valor resultante, se procede a realizar la rotación que corresponda y a hacer la reasignación. La plantilla de la clase nodo se define como se muestra a continuación.

```
/* Plantilla de la clase nodo de un árbol balanceado. Se incluye un
➥nuevo atributo, llamado FE, para almacenar el factor de equilibrio del
➥nodo. Se establece una relación de amistad con la clase ArbolBalanceado
➥para que ésta pueda tener acceso a sus miembros privados. */
template <class T>
class NodoArbolBal
{
   private:
       NodoArbolBal<T> *HijoIzq;
       NodoArbolBal<T> *HijoDer;
       T Info;
       int FE;
   public:
       NodoArbolBal();
       T RegresaInfo();
       void ActualizaInfo(T) ;
       friend class ArbolBalanceado<T>;
};
```

Inserción en árboles balanceados

La **inserción** de un nuevo nodo se lleva a cabo atendiendo las características de los árboles binarios de búsqueda, pero teniendo en cuenta además la altura de los subárboles, de tal manera que no se viole lo mencionado sobre el factor de equilibrio. Los casos que pueden presentarse son:

1. La altura del subárbol derecho es igual a la altura del subárbol izquierdo, por lo tanto, sin importar dónde se realice la inserción, el equilibrio no se pierde.
2. La altura del subárbol derecho es mayor que la altura del subárbol izquierdo, por lo tanto, si la inserción no afecta la altura del subárbol derecho no se requiere rotación, en caso contrario sí.
3. La altura del subárbol izquierdo es mayor que la altura del subárbol derecho, por lo tanto, si la inserción no afecta la altura del subárbol izquierdo no se requiere rotación, en caso contrario sí.

Cualquiera que sea la situación, se procede a insertar el nuevo nodo y luego se actualizan los factores de equilibrio procediendo a la rotación de los nodos si correspondiera. El método para realizar esta operación se presenta a continuación. Las rotaciones simples y compuestas se escribieron como métodos independientes.

```
/* Método que realiza la rotación simple HI-HI en un árbol balanceado.
➥Además, reasigna el FE del nodo involucrado en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHI_HI(NodoArbolBal<T>
                ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   Apunt->HijoIzq= ApAux1->HijoDer;
   ApAux1->HijoDer= Apunt;
   Apunt->FE= 0;
   return ApAux1;
}

/* Método que realiza la rotación simple HD-HD en un árbol balanceado.
➥Además, reasigna el FE del nodo involucrado en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHD_HD(NodoArbolBal<T>
                ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   Apunt->HijoDer= ApAux1->HijoIzq;
   ApAux1->HijoIzq= Apunt;
   Apunt->FE= 0;
   return ApAux1;
}
```

```
/* Método que realiza la rotación compuesta HI-HD en un árbol balanceado.
➥Además, reasigna los FE de los nodos involucrados en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHI_HD(NodoArbolBal<T>
               ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   NodoArbolBal<T> *ApAux2;
   ApAux2= ApAux1->HijoDer;
   Apunt->HijoIzq= ApAux2->HijoDer;
   ApAux2->HijoDer= Apunt;
   ApAux1->HijoDer= ApAux2->HijoIzq;
   ApAux2->HijoIzq= ApAux1;
   if (ApAux2->FE == -1)
        Apunt->FE= 1;
   else
        Apunt->FE= 0;
   if (ApAux2->FE == 1)
        ApAux1->FE= -1;
   else
        ApAux1->FE= 0;
   return ApAux2;
}

/* Método que realiza la rotación compuesta HD-HI en un árbol balanceado.
➥Además, reasigna los FE de los nodos involucrados en la rotación. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RotacionHD_HI(NodoArbolBal<T>
               ➥*Apunt, NodoArbolBal<T> *ApAux1)
{
   NodoArbolBal<T> *ApAux2;
   ApAux2= ApAux1->HijoIzq;
   Apunt->HijoDer= ApAux2->HijoIzq;
   ApAux2->HijoIzq= Apunt;
   ApAux1->HijoIzq= ApAux2->HijoDer;
   ApAux2->HijoDer= ApAux1;
   if (ApAux2->FE == 1)
        Apunt->FE= -1;
   else
        Apunt->FE= 0;
   if (ApAux2->FE == -1)
        ApAux1->FE= 1;
   else
        ApAux1->FE= 0;
   return ApAux2;
}

/* Método que inserta un nuevo elemento en un árbol balanceado. Recibe
➥como parámetros el dato a insertar, un puntero al nodo a visitar (la
➥primera vez es la raíz) y un entero (Band) que la primera vez trae el
➥valor 0.*/
```

```
template <class T>
void ArbolBalanceado<T>::InsertaBalanceado(T Dato,NodoArbolBal<T>
➥*Apunt,int *Band)
{
   NodoArbolBal<T> *ApAux1, *ApAux2;
   if (Apunt != NULL)
   {
        if (Dato < Apunt->Info)
        {
            /* Se invoca el método con el subárbol izquierdo. */
            InsertaBalanceado(Dato, Apunt->HijoIzq, Band);
            Apunt->HijoIzq= Raiz;
            if (0 < *Band)    /* Verifica si creció el subárbol
                              ➥izquierdo. */
                switch ( Apunt->FE )
                {
                   case 1: {
                            Apunt->FE = 0;
                            *Band= 0;
                            break;
                            }
                   case 0: {
                            Apunt->FE = -1;
                            break;
                           }
                   case -1: {
                            ApAux1 = Apunt->HijoIzq;
                            if (ApAux1->FE <= 0)
                                Apunt= RotacionHI_HI(Apunt, ApAux1);
                            else
                                Apunt= RotacionHI_HD(Apunt, ApAux1);
                            Apunt->FE = 0;
                            *Band = 0;
                           }
                }
        }
        else
            if (Dato > Apunt->Info)
            {
                 /* Invoca el método con el subárbol derecho. */
                 InsertaBalanceado(Dato, Apunt->HijoDer,Band);
                 Apunt->HijoDer= Raiz;
                 if (0 < *Band)      /* Verifica si creció el
                                     ➥subárbol derecho. */
                     switch ( Apunt->FE )
                     {
                         case -1: {
                                  Apunt->FE = 0;
                                  *Band= 0;
                                   break;
                                   }
```

```
                                case 0: {
                                           Apunt->FE = 1;
                                           break;
                                         }
                                case 1: {
                                           ApAux1= Apunt->HijoDer;
                                        if (ApAux1->FE >= 0)
                                            Apunt= RotacionHD_HD(Apunt,
                                            ➥ApAux1);
                                        else
                                             Apunt= RotacionHD_HI(Apunt,
                                             ➥ApAux1);
                                        Apunt->FE = 0;
                                        *Band= 0;
                                        }
                                }
                    }
                    Raiz= Apunt;
          }
          else
          {
               /* Inserta un nuevo nodo y actualiza el valor de Band indicando
               ➥que el árbol creció. */
               ApAux2= new  NodoArbolBal<T>();
               ApAux2->Info= Dato;
               ApAux2->FE= 0;
               *Band = 1;
               Raiz= ApAux2;
          }
     }
```

Eliminación en árboles balanceados

La **eliminación** de un nodo se lleva a cabo atendiendo las características de los árboles binarios de búsqueda, pero teniendo en cuenta además la altura de los subárboles, de tal manera que no se viole lo mencionado sobre el factor de equilibrio. Los casos que pueden presentarse son:

1. La altura del subárbol derecho es igual a la altura del subárbol izquierdo, por lo tanto, sin importar dónde se realice la eliminación, el equilibrio no se pierde.
2. La altura del subárbol derecho es mayor que la altura del subárbol izquierdo, por lo tanto, si la eliminación no afecta la altura del subárbol izquierdo no se requiere rotación, en caso contrario sí.

3. La altura del subárbol izquierdo es mayor que la altura del subárbol derecho, por lo tanto, si la eliminación no afecta la altura del subárbol derecho no se requiere rotación, en caso contrario sí.

Cualquiera que sea la situación, se elimina el nodo y luego se actualizan los factores de equilibrio de todos los nodos involucrados, procediendo a su rotación, si correspondiera. Este proceso termina cuando se llega a la raíz.

A continuación se presenta el método para realizar esta operación. Se reutilizan los métodos vistos para las rotaciones compuestas (los cuales no se vuelven a presentar). En el caso de los correspondientes a las rotaciones simples, dado que no se ajustan totalmente a la eliminación, se dan con los cambios requeridos. Se definieron dos métodos auxiliares que ayudan a la reestructuración del árbol si éste pierde el equilibrio.

```
/* Método auxiliar del método EliminaBalanceado que reestructura el
➥árbol cuando la altura de la rama izquierda ha disminuido. */
template <class T>
NodoArbolBal<T>* ArbolBalanceado<T>::RestructuraI(NodoArbolBal<T> *Nodo,
                                                  int *Aviso)
{
   NodoArbolBal<T> *ApAux;
   if ( *Aviso > 0)
   {
         switch (Nodo->FE)
         {
            case -1: Nodo->FE= 0;
                     break;
            case 0:  Nodo->FE= 1;
                     *Aviso= 0;
                     break;
            case 1:  ApAux= Nodo->HijoDer;
                     if (ApAux->FE >= 0)   //Rotación HD-HD
                     {
                          Nodo->HijoDer= ApAux->HijoIzq;
                          ApAux->HijoIzq= Nodo;
                          switch (ApAux->FE)
                          {
                             case 0:  Nodo->FE= 1;
                                      ApAux->FE= -1;
                                      *Aviso= 0;
                                      break;
                             case 1:  Nodo->FE= 0;
                                      ApAux->FE= 0;
                                      break;
```

```
                        }
                        Nodo= ApAux;
                    }
                    else     //Rotación HD-HI
                    {
                        Nodo= RotacionHD_HI(Nodo, ApAux);
                        Nodo->FE= 0;
                    }
                    break;
                }
    }
    return Nodo;
}

/* Método auxiliar del método EliminaBalanceado que reestructura el
➥árbol cuando la altura de la rama derecha ha disminuido. */
template <class T>
NodoArbolBal<T>* ArbolBalanceado<T>::RestructuraD(NodoArbolBal<T> *Nodo,
                                                  int *Aviso)
{
   NodoArbolBal<T> *ApAux;
     if (*Aviso > 0)
     {
          switch (Nodo->FE)
          {
             case 1:  Nodo->FE= 0;
                      break;
             case 0:  Nodo->FE= -1;
                      *Aviso= 0;
                      break;
             case -1:  ApAux= Nodo->HijoIzq;
                      if (ApAux->FE <= 0)  //Rotación HI-HI
                      {
                          Nodo->HijoIzq= ApAux->HijoDer;
                          ApAux->HijoDer= Nodo;
                          switch (ApAux->FE)
                          {
                             case 0:  Nodo->FE= -1;
                                      ApAux->FE= 1;
                                      *Aviso= 0;
                                      break;
                             case -1: Nodo->FE= 0;
                                      ApAux->FE= 0;
                                      break;
                          }
                           Nodo= ApAux;
                      }
```

```
                    else       //Rotación HI-HD
                    {
                        Nodo= RotacionHI_HD(Nodo, ApAux);
                        Nodo->FE= 0;
                    }
                    break;
            }
    }
    return Nodo;
}

/* Método auxiliar del método EliminaBalanceado que sustituye el
➥elemento que se desea eliminar por el que se encuentra más a la derecha
➥del subárbol izquierdo. */
template <class T>
void ArbolBalanceado<T>::Sustituye(NodoArbolBal<T> *Nodo,
                        ➥NodoArbolBal<T> *ApAux, int *Avisa)
{
    if (Nodo->HijoDer != NULL)
    {
        Sustituye (Nodo->HijoDer, ApAux, Avisa);
        if (ApAux->HijoIzq == NULL)
            Nodo->HijoDer= NULL;
        else
            Nodo->HijoDer= ApAux->HijoIzq;
        Nodo= RestructuraD(Nodo, Avisa);
    }
    else
    {
        ApAux->Info= Nodo->Info;
        Nodo= Nodo->HijoIzq;
        *Avisa= 1;
    }
    ApAux->HijoIzq= Nodo;
}

/* Método que elimina un elemento en un árbol balanceado. Luego de la
➥eliminación se actualizan los factores de equilibrio de cada nodo hasta
➥la raíz y se reestructura el árbol si fuera necesario. */
template <class T>
void ArbolBalanceado<T>::EliminaBalanceado(NodoArbolBal<T> *Apunt,
                        ➥NodoArbolBal<T> *ApAnt, int *Avisa, T Dato)
{
    NodoArbolBal<T> *ApAux;
    int Bandera;
    if (Apunt != NULL)
        if (Dato < Apunt->Info)
```

```
{
     if (*Avisa > 0)
          Bandera= 1;
     else
         if (*Avisa != 0)
              Bandera= -1;
     *Avisa= -1;
     EliminaBalanceado(Apunt->HijoIzq, Apunt, Avisa, Dato);
     Apunt= RestructuraI(Apunt, Avisa);
     if (ApAnt != NULL)
         switch (Bandera)
         {
             case -1:  ApAnt->HijoIzq= Apunt;
                       break;
             case 1:  ApAnt->HijoDer= Apunt;
                       break;
             default:  break;
         }
     else
         Raiz= Apunt;
}
else
{
     if  (Dato > Apunt->Info)
     {
          if (*Avisa < 0)
               Bandera= -1;
          else
               if (*Avisa != 0)
                    Bandera=1;
          *Avisa= 1;
          EliminaBalanceado(Apunt->HijoDer, Apunt, Avisa, Dato);
          Apunt= RestructuraD(Apunt,Avisa);
          if (ApAnt != NULL)
             switch (Bandera)
             {
                case -1: ApAnt->HijoIzq= Apunt;
                         break;
                case 1:  ApAnt->HijoDer= Apunt;
                         break;
                default: break;
             }
         else
             Raiz= Apunt;
     }
     else
     {
         ApAux= Apunt;
         if (ApAux->HijoDer == NULL)
```

```
                    {
                        Apunt= ApAux->HijoIzq;
                        if (*Avisa != 0)
                             if (*Avisa < 0)
                                  ApAnt->HijoIzq= Apunt;
                             else
                                  ApAnt->HijoDer= Apunt;
                        else
                             if (Apunt == NULL)
                                 Raiz= NULL;
                             else
                                 Raiz= Apunt;
                             *Avisa= 1;
                    }
                    else
                         if (ApAux->HijoIzq == NULL)
                         {
                             Apunt= ApAux->HijoDer;
                             if (*Avisa != 0)
                                  if (*Avisa < 0)
                                       ApAnt->HijoIzq= Apunt;
                                  else
                                       ApAnt->HijoDer= Apunt;
                             else
                                 if (Apunt == NULL)
                                     Raiz= NULL;
                                 else
                                      Raiz= Apunt;
                             *Avisa= 1;
                    }
                    else
                    {
                         Sustituye (ApAux->HijoIzq, ApAux, Avisa);
                         Apunt= RestructuraI(Apunt, Avisa);
                         if (ApAnt != NULL)
                              if (*Avisa <= 0)
                                   ApAnt->HijoIzq= Apunt;
                              else
                                  ApAnt->HijoDer= Apunt;
                         else
                              Raiz= Apunt;
                    }
               }
    else
         cout<<"\n\nEl dato a eliminar no se encuentra registrado.\n\n";
}
```

El programa 7.4 presenta la plantilla de la clase `ArbolBalanceado` con los encabezados de los métodos analizados, los cuales no se repiten. Se incluye un método para la impresión de todos los nodos junto con su factor de equilibrio.

Programa 7.4

```
/* Prototipo de la plantilla de la clase ArbolBalanceado. De esta
➥manera, en la clase NodoArbolBal se podrá hacer referencia a ella. */

template <class T>
class ArbolBalanceado;

/* Declaración de la clase de un nodo de un árbol balanceado. Además de
➥almacenar la información, las direcciones de los hijos izquierdo y
➥derecho, guarda el factor de equilibrio. */
template <class T>
class NodoArbolBal
{
   private:
       NodoArbolBal<T> *HijoIzq;
       NodoArbolBal<T> *HijoDer;
       T Info;
       int FE;
   public:
       NodoArbolBal();
       T RegresaInfo();
       void ActualizaInfo(T);
       friend class ArbolBalanceado<T>;
};

/* Declaración del método constructor. Inicializa los apuntadores a
➥ambos hijos con el valor de NULL, indicando vacío. */
template <class T>
NodoArbolBal<T>::NodoArbolBal()
{
   HijoIzq= NULL;
   HijoDer= NULL;
}

/* Método que regresa la información almacenada en el nodo. */
template <class T>
T NodoArbolBal<T>::RegresaInfo()
{
   return Info;
}
```

```
/* Método que permite actualizar la información almacenada en el nodo. */
template <class T>
void NodoArbolBal<T>::ActualizaInfo(T Dato)
{
   Info= Dato;
}

/* Declaración de la clase ArbolBalanceado. Se incluyen sólo los proto-
➥tipos de los métodos presentados más arriba. */
template <class T>
class ArbolBalanceado
{
   private:
       NodoArbolBal<T> *Raiz;
   public:
       ArbolBalanceado ();
       NodoArbolBal<T> * RegresaRaiz();
       NodoArbolBal<T> * Busca (NodoArbolBal<T> *, T) ;
       void InsertaBalanceado (T, NodoArbolBal<T> *, int *);
       NodoArbolBal<T> * RotacionHI_HD (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RotacionHD_HI (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RotacionHI_HI (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RotacionHD_HD (NodoArbolBal<T> *,
                                        ➥NodoArbolBal<T> *);
       NodoArbolBal<T> * RestructuraI (NodoArbolBal<T> *, int *);
       NodoArbolBal<T> * RestructuraD (NodoArbolBal<T> *, int *);
       void EliminaBalanceado (NodoArbolBal<T> *, NodoArbolBal<T> *,
                               ➥int *, T);
       void Sustituye (NodoArbolBal<T> *,NodoArbolBal<T> *,  int *);
       void Imprime (NodoArbolBal<T> *);
};

/* Declaración del método constructor. Inicializa el puntero a la raíz
➥con el valor NULL, indicando que el árbol está vacío. */
template <class T>
ArbolBalanceado<T>::ArbolBalanceado()
{
   Raiz= NULL;
}

/* Método que regresa el apuntador a la raíz del árbol.*/
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::RegresaRaiz()
{
   return Raiz;
}
```

```
/* Método que busca un valor dado como parámetro en el árbol. Recibe
➥como parámetros el puntero del nodo a visitar (la primera vez es la
➥raíz) y el dato a buscar. Regresa el puntero al nodo donde lo encontró
➥o NULL si no está en el árbol. */
template <class T>
NodoArbolBal<T> * ArbolBalanceado<T>::Busca (NodoArbolBal<T> *Apunt, T
Dato)
{
   if (Apunt != NULL)
        if (Apunt->Info == Dato)
            return Apunt;
        else
            if (Apunt->Info > Dato)
                return Busca(Apunt->HijoIzq, Dato);
            else
                return Busca(Apunt->HijoDer, Dato);
   else
        return NULL;
}

/* Método que imprime el contenido del árbol. Recibe como parámetro el
➥apuntador al nodo a visitar (la primera vez es la raíz del árbol).
➥Utiliza recorrido inorden para que el contenido se imprima en orden
➥creciente. */
template <class T>
void ArbolBalanceado<T>::Imprime(NodoArbolBal<T> *Apunt)
{
   if (Apunt != NULL)
   {
        Imprime(Apunt->HijoIzq);
        cout<<Apunt->Info <<"\n\n";
        Imprime(Apunt->HijoDer);
   }
}
```

A continuación se presenta una aplicación de árboles balanceados. Se define una clase `Fabrica` para almacenar los datos más importantes de una fábrica y las operaciones que pueden realizarse sobre ellos. Esta clase se almacena en la biblioteca "*Fabricas.h*". El programa 7.5 muestra esta clase y el programa 7.6, la aplicación de árboles en la cual se incluyen la biblioteca mencionada y "*ArbolBalanceado.h*" que corresponde a la plantilla de árboles balanceados del programa 7.4.

Programa 7.5

```
/* Definición de la clase Fabrica. Se incluyen varios operadores
➥sobrecargados para que puedan ser utilizados por los métodos de la
➥clase ArbolBalanceado. Asimismo, se declaran como amigos los operadores
➥de entrada (>>) y de salida (<<) para que objetos de este tipo puedan
➥leerse e imprimirse directamente con cin y cout respectivamente. */
class Fabrica
{
   private:
       int Clave;
       char Nombre[MAX], Domicilio[MAX], Telefono[MAX];
   public:
       Fabrica();
       Fabrica(int, char [], char[], char[]);
       void CambiaDomic(char[]);
       void CambiaTelef(char[]);
       int operator > (Fabrica);
       int operator < (Fabrica);
       int operator == (Fabrica);
       friend istream & operator>> (istream & , Fabrica & );
       friend ostream & operator<< (ostream & , Fabrica & );
};

/* Declaración del método constructor por omisión. */
Fabrica::Fabrica()
{}

/* Declaración del método constructor con parámetros. */
Fabrica::Fabrica(int Cla, char Nom[], char Domic[], char Tel[])
{
   Clave= Cla;
   strcpy(Nombre, Nom);
   strcpy(Domicilio, Domic);
   strcpy(Telefono, Tel);
}

/* Método que actualiza el domicilio de una fábrica. */
void Fabrica::CambiaDomic(char NuevoDom[])
{
   strcpy(Domicilio, NuevoDom);
}

/* Método que actualiza el teléfono de una fábrica. */
void Fabrica::CambiaTelef(char NuevoTel[])
{
   strcpy(Telefono, NuevoTel);
}
```

```
/* Sobrecarga del operador > lo cual permite comparar dos objetos tipo
➥Fabrica. La comparación se hace teniendo en cuenta solamente la clave. */
int Fabrica::operator > (Fabrica ObjFab)
{
   if (Clave > ObjFab.Clave)
        return 1;
   else
        return 0;
}

/* Sobrecarga del operador < lo cual permite comparar dos objetos tipo
➥Fabrica. La comparación se hace teniendo en cuenta solamente la clave. */
int Fabrica::operator < (Fabrica ObjFab)
{
   if (Clave < ObjFab.Clave)
        return 1;
   else
        return 0;
}

/* Sobrecarga del operador == lo cual permite comparar dos objetos tipo
➥Fabrica. La comparación se hace teniendo en cuenta solamente la clave. */
int Fabrica::operator == (Fabrica ObjFab)
{
   if (Clave == ObjFab.Clave)
        return 1;
   else
        return 0;
}

/* Sobrecarga del operador >> para permitir la lectura de objetos de
➥tipo Fabrica de manera directa con el cin. */
istream & operator>> (istream & Lee, Fabrica & ObjFab)
{
   cout<<"\n\nIngrese nombre de la fábrica:";
   Lee>>ObjFab.Nombre;
   cout<<"\n\nIngrese clave de la fábrica:";
   Lee>>ObjFab.Clave;
   cout<<"\n\nIngrese domicilio de la fábrica:";
   Lee>>ObjFab.Domicilio;
   cout<<"\n\nIngrese teléfono de la fábrica:";
   Lee>>ObjFab.Telefono;
   return Lee;
}

/* Sobrecarga del operador << para permitir la impresión de objetos de
➥tipo Fabrica de manera directa con el cout. */
ostream & operator<<(ostream & Escribe, Fabrica & ObjFab)
```

```
{
   cout<<"\n\nDatos de la fábrica\n";
   Escribe<<"Nombre:  "<<ObjFab.Nombre<<endl;
   Escribe<<"Clave:  "<<ObjFab.Clave<<endl;
   Escribe<<"Domicilio:  "<<ObjFab.Domicilio<<endl;
   Escribe<<"Teléfono:    "<<ObjFab.Telefono<<endl;
   return Escribe;
}
```

Programa 7.6

```
/* Programa que utiliza un árbol balanceado para almacenar ordenadamente
➥los datos de ciertas fábricas. El usuario puede dar de alta nuevas
➥fábricas, eliminar alguna ya registrada, obtener un reporte de todas
➥(ordenadas según su clave) y actualizar sus direcciones y teléfonos. Se
➥incluyen dos bibliotecas, una con la plantilla de la clase ArbolBalan-
➥ceado presentada en el programa 7.4 y la otra con la clase Fabrica del
➥programa 7.5. */

#include "ArbolBalanceado.h"
#include "Fabricas.h"

/* Función que despliega en pantalla las opciones de trabajo para el
➥usuario. */
int Menu()
{
   int Opc;
   do {
       cout<<"\n\n\t\tOpciones de trabajo.\n\n\n";
       cout<<"(1) Capturar los datos de una fábrica.\n";
       cout<<"(2) Dar de baja una fábrica.\n";
       cout<<"(3) Imprimir los datos de todas las fábricas, ordenadas
       ➥por clave.\n";
       cout<<"(4) Cambiar el domicilio de una fábrica.\n";
       cout<<"(5) Cambiar el teléfono de una fábrica.\n";
       cout<<"(6) Terminar la sesión de trabajo.\n\n";
       cout<<"Ingrese la opción seleccionada:";
       cin>>Opc;
   } while (Opc > 6 || Opc < 1);
   return Opc;
}
```

```
/* Función principal. De acuerdo a la opción de trabajo seleccionada por
➥el usuario invoca los métodos que correspondan. */
void main()
{
   ArbolBalanceado<Fabrica> Proveedores;
   NodoArbolBal<Fabrica> *Apunt1, *Apunt2;
     Fabrica Prov;
     int Operac, Band, Clave;
     char NuevoDom[MAX], NuevoTel[MAX];

   do {
     Operac= Menu();
     switch (Operac)
     {
        /* Se registra una nueva fábrica siempre que la clave dada por
        ➥el usuario no se encuentre en el árbol. */
        case 1: {
                 cin>>Prov;
                 Band= 0;
                 Apunt1= Proveedores.RegresaRaiz();
                 Proveedores.InsertaBalanceado(Prov, Apunt1, &Band);
                 break;
                 }
        /* En caso de dar de baja una fábrica registrada, se solicita sólo
        ➥la clave ya que es el dato que identifica a cada elemento. */
        case 2: {
                 cout<<"\n\nIngrese la clave de la fábrica a eliminar:";
                 cin>>Clave;
                 Fabrica Prov(Clave, "","","");
                 Band= 0;
                 Apunt1= Proveedores.RegresaRaiz();
                 Proveedores.EliminaBalanceado(Apunt1, NULL,
                 ➥&Band, Prov);
                 break;
                 }
        /* Se imprimen los datos de todas las fábricas, ordenadas de
        ➥menor a mayor por clave. */
        case 3: {
                 Apunt1= Proveedores.RegresaRaiz();
                 Proveedores.Imprime(Apunt1);
                 break;
                 }
        /* Se actualiza la dirección de una fábrica. Para llevar a cabo
        ➥esta operación, primero se debe encontrar la fábrica de interés,
        ➥luego recuperar todo el objeto, actualizar el domicilio y pos-
        ➥teriormente redefinir el contenido del nodo con el objeto ya
        ➥modificado. */
```

```
        case 4: {
                cout<<"\n\nIngrese la clave de la fábrica:";
                cin>>Clave;
                cout<<"\n\nIngrese nuevo domicilio:";
                cin>>NuevoDom;
                Fabrica Prov(Clave, "","","");
                Apunt1= Proveedores.RegresaRaiz();
                Apunt2= Proveedores.Busca(Apunt1, Prov);
                if (Apunt2)
                {
                    Prov= Apunt2->RegresaInfo();
                    Prov.CambiaDomic(NuevoDom);
                    Apunt2->ActualizaInfo(Prov);
                }
                else
                    cout<<"\n\nEsa fábrica no está registrada. \n";
                break;
                }
        /* Se actualiza el teléfono de una fábrica. Para llevar a cabo
        ➥esta operación, primero se debe encontrar la fábrica de interés,
        ➥luego recuperar todo el objeto, actualizar el teléfono y poste-
        ➥riormente redefinir el contenido del nodo con el objeto ya
        ➥modificado. */
        case 5: {
                cout<<"\n\nIngrese la clave de la fábrica:  ";
                cin>>Clave;
                cout<<"\n\nIngrese nuevo teléfono:  ";
                cin>>NuevoTel;
                Fabrica Prov(Clave, "", "", "");
                Apunt1= Proveedores.RegresaRaiz();
                Apunt2= Proveedores.Busca(Apunt1,  Prov);
                if (Apunt2)
                {
                    Prov= Apunt2->RegresaInfo();
                    Prov.CambiaTelef(NuevoTel);
                    Apunt2->ActualizaInfo(Prov);
                }
                else
                    cout<<"\n\nEsa fábrica no está registrada. \n";
                break;
                }
        }
    } while (Operac < 6);
}
```

7.4 Árboles-B

Las estructuras tipo árboles estudiadas hasta aquí son utilizadas para almacenar información en la memoria principal de la computadora. Sin embargo, en prácticamente todas las aplicaciones se requiere que los datos a procesar se guarden en dispositivos secundarios, de tal manera que permanezcan aún después de terminado el procesamiento. Además, el volumen de información manejado exige el uso de medios externos de almacenamiento. Por lo tanto, resulta necesario contar con estructuras que permitan organizar la información guardada en archivos. Los ***árboles-B*** son una variante de los árboles balanceados y cubren esa necesidad. En estas estructuras, a cada nodo se le conoce con el nombre de página y las páginas se guardan en algún dispositivo de almacenamiento secundario.

Las principales características de un árbol-B de grado n son:

- La página raíz almacena como mínimo 1 dato y como máximo 2n datos.
- La página raíz tiene como mínimo 2 descendientes.
- Las páginas intermedias y hojas almacenan entre n y 2n datos.
- Las páginas intermedias tienen entre (n+1) y (2n+1) páginas descendientes.
- Todas las páginas hojas tienen la misma altura.
- La información guardada en las páginas se encuentra ordenada.

La figura 7.16 presenta un ejemplo de un árbol-B, de grado 2. En la raíz se almacenan dos datos, lo que origina que tenga tres descendientes. La página hoja que está más a la izquierda guarda todos los datos que son menores al primer dato (105) de la página raíz, la segunda hoja contiene los datos mayores a 105 y menores a 320, mientras que la tercera hoja almacena los datos mayores al segundo dato de la página raíz (320). Cada una de las páginas hojas tiene entre 2 y 4 elementos. Si no fueran hojas, tendrían: la primera 3, la segunda 5 y la tercera 4 páginas descendientes respectivamente.

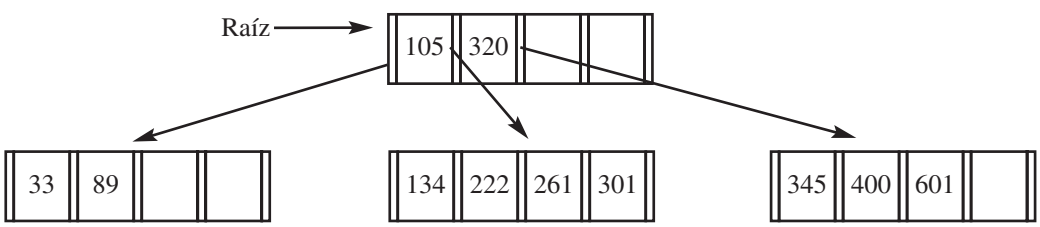


FIGURA 7.16 *Ejemplo de un árbol-B*

Las operaciones que pueden realizarse sobre la información almacenada en árboles-B son: búsqueda, inserción y eliminación. Estas operaciones están implementadas en herramientas diseñadas especialmente para el manejo de archivos, por lo que generalmente no se requiere su programación. En esta sección se analizan para que usted pueda entenderlas y cuando haga uso de algún manejador de archivos sepa qué está pasando internamente con los datos.

Búsqueda en árboles-B

La operación de **búsqueda** es similar a la estudiada en los árboles binarios de búsqueda, por ser los árboles-B una generalización de los primeros. En este tipo de árboles se recuperan del medio secundario de almacenamiento páginas completas de información y se procede a buscar en ellas el dato deseado. Si se encuentra, termina la búsqueda; en caso contrario se procede con la página que corresponda según el dato con el cual se compara, ya sea menor o mayor. Si se requiere recuperar una nueva página pero no existe, entonces se tiene un caso de fracaso. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.
2. Se evalúa si la página está vacía.
   2.1. Si la respuesta es afirmativa, entonces la búsqueda termina con fracaso.
   2.2. Si la respuesta es negativa, entonces ir al paso 3.
3. Se compara el dato buscado con cada elemento almacenado en la página.
   3.1. Si es igual, la búsqueda termina con éxito.
   3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.
   3.3. Si es mayor, se avanza al siguiente dato de la misma página.
      3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
      3.3.2. Si no es el último, se regresa al paso 3.

Considere el árbol-B de la figura 7.17. Si se quisiera encontrar el valor 319 se procedería, siguiendo el algoritmo dado, tal como se muestra en la tabla 7.4.

TABLA 7.4 Operación de búsqueda en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los datos: 105 – 320
2	Se evalúa si está vacía. En este caso la respuesta es negativa.
3	Se compara el dato buscado (319) con el primer elemento de la página (105). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato buscado (319) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los datos: 134 – 222 – 261 – 301.
6	Se evalúa si está vacía. En este caso la respuesta es negativa.
7	Se compara el dato buscado (319) con el primer elemento de la página (134). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (222).
8	Se compara el dato buscado (319) con el valor 222. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (261).
9	Se compara el dato buscado (319) con el 261. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (301).
10	Se compara el dato buscado (319) con el valor 301. Es mayor y ya no hay más elementos en la misma página, entonces se toma la dirección de la página que está a la derecha del 301.
11	Se recupera la página con los datos: 310 – 319.
12	Se evalúa si está vacía. En este caso la respuesta es negativa.
13	Se compara el dato buscado (319) con el primer elemento de la página (310). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (319).
14	Se compara el dato buscado (319) con el valor 319. Es igual, por lo tanto la búsqueda termina con éxito.

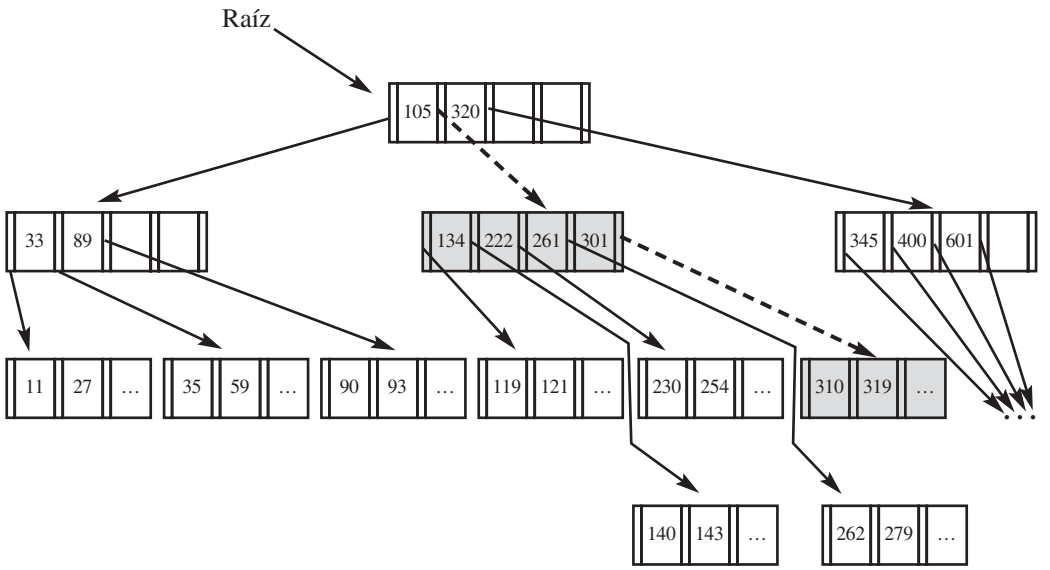


FIGURA 7.17 *Búsqueda en árboles-B*

En la figura 7.17 las líneas punteadas indican las páginas que se van visitando hasta llegar al dato buscado. Por su parte, las páginas sombreadas son las que se recuperan para hacer la comparación del elemento buscado con los elementos almacenados en el árbol.

Inserción en árboles-B

La operación de **inserción** se caracteriza porque los nuevos elementos siempre se guardan a nivel de las hojas, y puede originar la reestructuración del árbol incluso hasta la raíz provocando esto último que la altura aumente en uno. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz).
2. Se evalúa si es una página hoja.
   2.1. Si la respuesta es afirmativa, se evalúa si la cantidad de elementos almacenados en ella es menor a 2n, siendo n el grado del árbol.
      2.1.1. Si la respuesta es afirmativa, entonces se procede a insertar el nuevo valor en el lugar correspondiente.

2.1.2. Si la respuesta es negativa, entonces se divide la página en dos y los (2n + 1) datos se distribuyen entre las páginas resultantes (n en cada página) y el valor del medio sube a la página padre. Si la página padre no tuviera espacio para este valor, entonces se procede de la misma manera, se divide en dos y el elemento del medio sube a la siguiente página, pudiendo repetirse este proceso hasta llegar a la raíz, en cuyo caso se aumenta la altura del árbol.

2.2. Si no es una hoja, se compara el elemento a insertar con cada uno de los valores almacenados para encontrar la página descendiente donde proseguir la búsqueda. Se sigue con el paso 1.

Considere el árbol-B, de grado 2, de la figura 7.18 en el cual se quiere insertar el valor 60. La línea punteada señala la página que se recupera y donde se agrega el 60. En la tabla 7.5 se presenta la secuencia de pasos necesarios para realizar esta operación.

TABLA 7.5 Operación de inserción en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si es una página hoja. No lo es.
3	Se compara el dato a insertar (60) con el valor 105. Es menor, entonces se toma la dirección de la página que está a la izquierda del 105.
4	Se recupera la página con los valores: 33 – 89.
5	Se evalúa si es una página hoja. Sí lo es.
6	Se evalúa si el total de elementos almacenados (2) es menor a 2n. Sí lo es.
7	Se inserta el valor 60 entre el 33 y 89 de tal manera que no altere el orden.

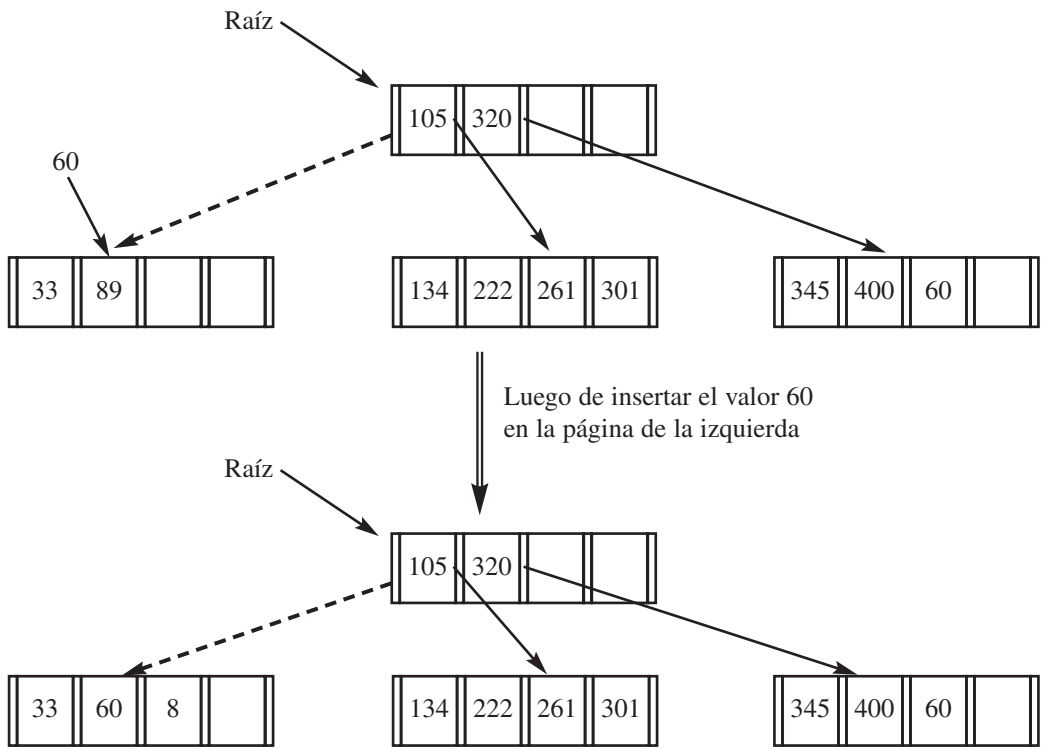


FIGURA 7.18 *Inserción del valor 60*

En el ejemplo anterior no hubo cambios en la estructura del árbol. Considere ahora el árbol-B, de grado 2, de la figura 7.19 en el cual se quiere insertar el valor 120. En la tabla 7.6 se presenta la secuencia de operaciones realizadas al aplicar el algoritmo visto.

TABLA 7.6 Operación de inserción en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si es página hoja. No lo es.
3	Se compara el dato a insertar (120) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).

continúa

TABLA 7.6 Continuación

Operación	*Descripción*
4	Se compara el dato a insertar (120) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los valores: 134 – 222 – 261 – 301.
6	Se evalúa si es página hoja. Sí lo es.
7	Se evalúa si el total de elementos almacenados (4) es menor a 2n. No lo es.
8	Se divide la página en dos y el valor del medio (222) sube a la página padre, en la cual hay espacio. Los demás valores se distribuyen entre las dos nuevas páginas.

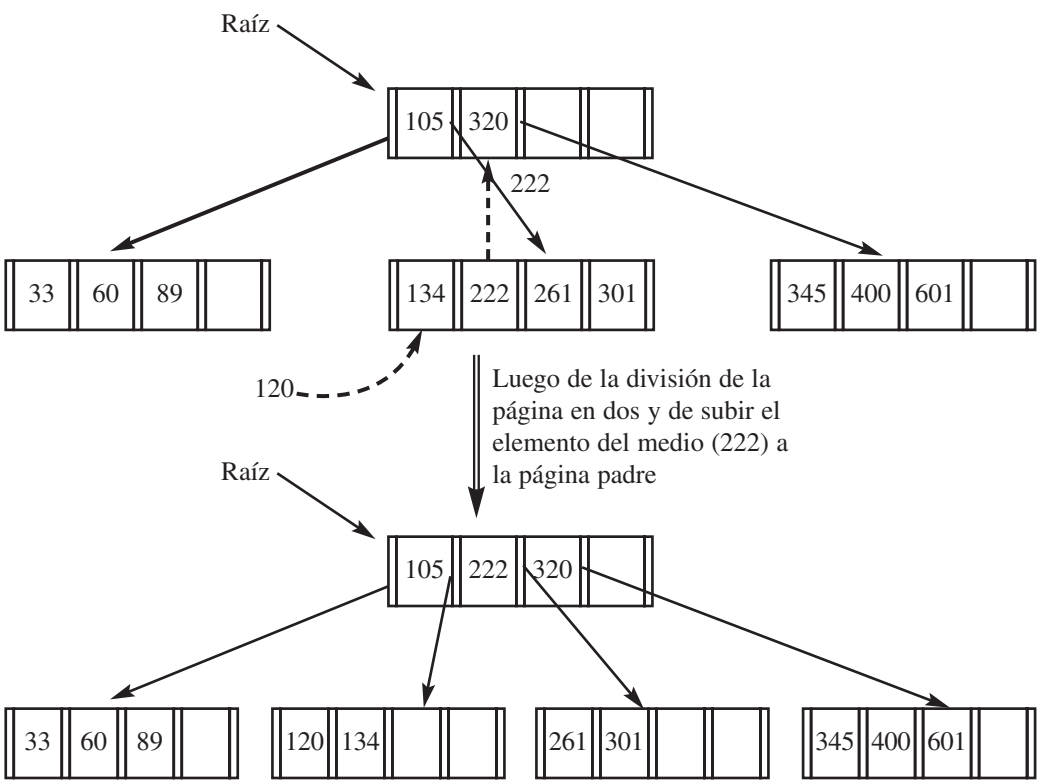


FIGURA 7.19 *Inserción del valor 120*

Como se puede apreciar en la figura 7.19, luego de insertar el valor 120 se modifica la estructura del árbol-B. La página en la que debía insertarse el nuevo dato estaba completa, por lo que se dividió en dos y subió el dato central a la página padre. Finalmente la página padre quedó con tres elementos y por lo tanto con cuatro descendientes.

A continuación se presenta un ejemplo en el cual la estructura del árbol se modifica en cuanto al número de páginas y a la altura. En el árbol-B, de grado 2, de la figura 7.20 se quiere insertar el valor 850. La tabla 7.7 muestra las operaciones que se realizaron al aplicar el algoritmo de inserción.

TABLA 7.7 Operación de inserción en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320 – 505 – 720
2	Se evalúa si es página hoja. No lo es.
3	Se compara el dato a insertar (850) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato a insertar (850) con el valor 320. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (505).
5	Se compara el dato a insertar (850) con el valor 505. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (720).
6	Se compara el dato a insertar (850) con el valor 720. Es mayor y ya no hay más elementos en la misma página, entonces se toma la dirección de la página que está a la derecha del 720.
7	Se recupera la página con los valores: 765 – 800 – 801 – 976.
8	Se evalúa si es página hoja. Sí lo es.
9	Se evalúa si el total de elementos almacenados (4) es menor a 2n. No lo es.
10	Se divide la página en dos y el valor del medio (801) sube a la página padre. Los demás valores se distribuyen entre las dos nuevas páginas.
11	En la página padre no hay espacio, el número de elementos almacenados es igual a 2n. Por lo tanto, se debe dividir en dos y el dato del medio (505) debe guardarse en una página que será la nueva raíz del árbol. La altura del árbol crece en una unidad.

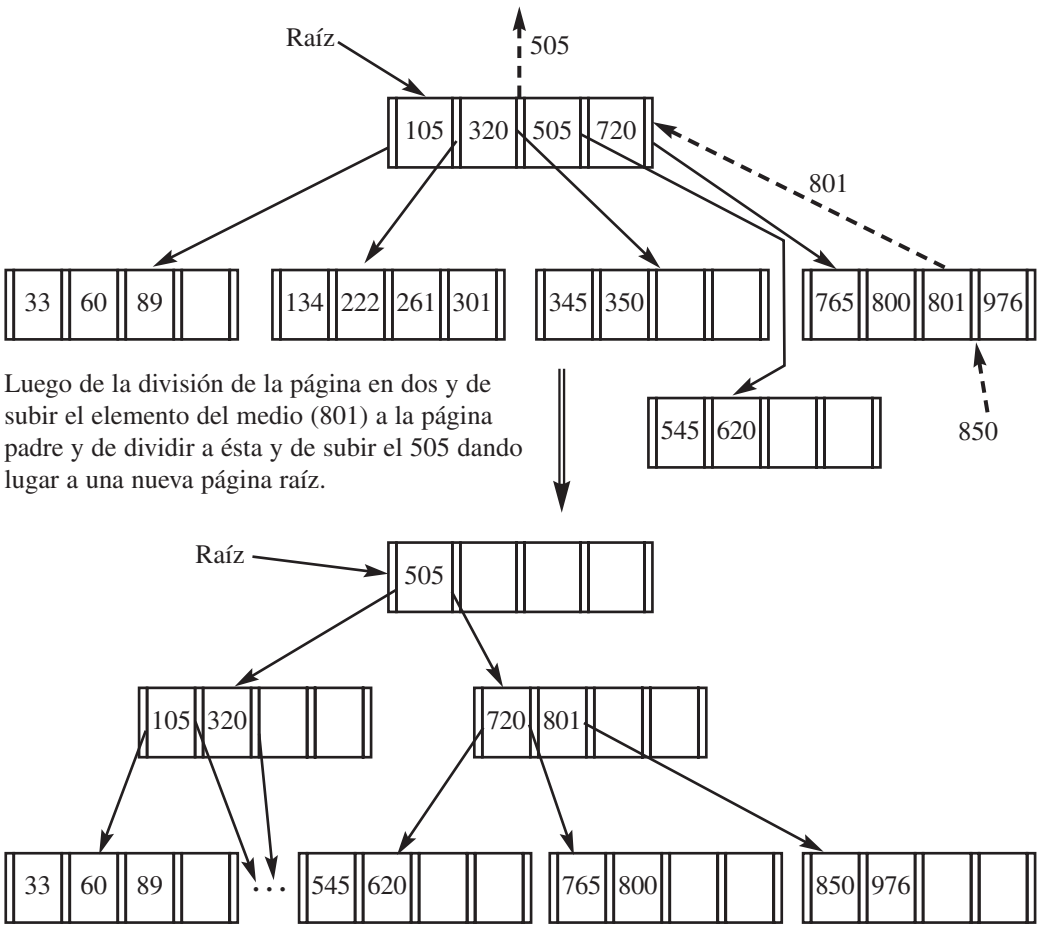


FIGURA 7.20 *Inserción del valor 850*

Eliminación en árboles-B

La operación de **eliminación** consiste en quitar un elemento del árbol-B cuidando que mantenga las propiedades vistas. Es decir, el número de datos en cada página debe ser mayor o igual a n y menor o igual a 2n. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.
2. Se evalúa si la página está vacía.

2.1. Si la respuesta es afirmativa, entonces la operación de eliminación termina con fracaso.

2.2. Si la respuesta es negativa, entonces ir al paso 3.

3. Se compara el dato a eliminar con cada elemento almacenado en la página.

3.1. Si es igual, entonces se elimina y se procede de la siguiente manera:

3.1.1. Si el dato estaba en una página hoja y el número de elementos de ésta sigue siendo un valor comprendido entre n y 2n, entonces la operación de eliminación termina.

3.1.2. Si el dato estaba en una página y el número de elementos de ésta queda menor que n, entonces se debe bajar el dato más cercano de la página padre y sustituirlo por el que se encuentre más a la izquierda del subárbol derecho o por el que se encuentre más a la derecha del subárbol izquierdo, siempre que esta página no pierda la condición y se fusionan.

3.1.3. Si el dato estaba en la página raíz o en una página intermedia, entonces se debe sustituir por el que se encuentre más a la izquierda del subárbol derecho o por el que se encuentre más a la derecha del subárbol izquierdo, siempre que esta página no pierda la condición. Si es así, termina la eliminación con éxito. En caso contrario, se debe bajar el dato más cercano de la página padre y fusionar las páginas que son hijas de éste.

3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.

3.3. Si es mayor, se avanza al siguiente dato de la misma página.

3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.

3.3.2. Si no es el último, se regresa al paso 3.

El proceso de fusión de páginas puede llegar hasta la raíz, en cuyo caso la altura del árbol disminuye en uno.

Analice el siguiente ejemplo. Se tiene un árbol-B, de grado 2 (ver figura 7.21) en el cual se quiere eliminar el valor 222. Aplicando el algoritmo dado, se realizan las operaciones que aparecen en la tabla 7.8.

TABLA 7.8 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (222) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato a eliminar (222) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los valores: 134 – 222 – 261 – 301.
6	Se compara el dato a eliminar (222) con el valor 134. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (222).
7	Se compara el dato a eliminar (222) con el valor 222. Es igual, entonces se elimina.
8	Se evalúa si el dato eliminado estaba en una página hoja. Sí lo estaba.
9	Se evalúa si el número de elementos (3) es $\geq$ n y $\leq$ 2n. Sí lo es. La operación termina con éxito.

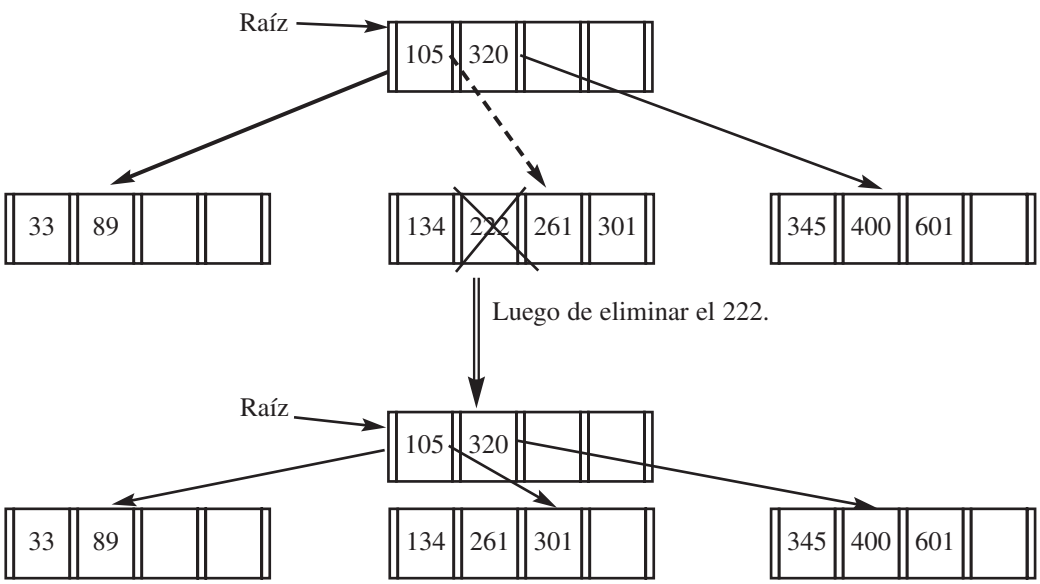


FIGURA 7.21 *Eliminación del valor 222*

En el ejemplo anterior se presentó el caso más simple de eliminación. El dato buscado estaba en una página hoja que a su vez tenía más de n elementos. Observe ahora el siguiente caso. En la figura 7.22 se tiene un árbol-B, de grado 2, del cual se quiere eliminar el valor 89. Aplicando el algoritmo dado, se llevan a cabo las operaciones mostradas en la tabla 7.9.

TABLA 7.9 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (89) con el valor 105. Es menor, entonces se toma la dirección de la página que está a la izquierda del 105.
4	Se recupera la página con los valores: 33 – 89.
5	Se compara el dato a eliminar (89) con el valor 33. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (89).
6	Se compara el dato a eliminar (89) con el valor 89. Es igual, entonces se elimina.
7	Se evalúa si el dato eliminado estaba en una página hoja. Sí lo estaba.
8	Se evalúa si el número de elementos (1) es $\geq$ n y $\leq$ 2n. No lo es, entonces se debe bajar el dato más cercano de la página padre (105) y sustituir a éste por el que se encuentre más a la izquierda del subárbol derecho (134).
9	Se evalúa si el número de elementos de la página del subárbol derecho (de la cual se quitó el 134) cumple con la condición. En este caso (2) es $\geq$ n y $\leq$ 2n. La operación termina con éxito.

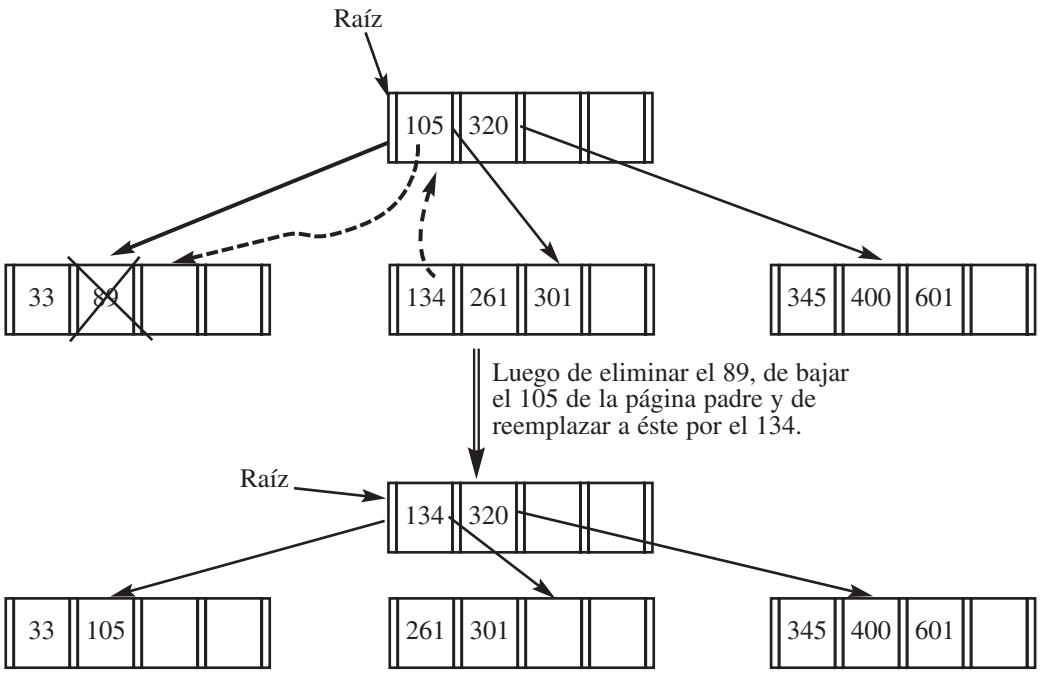


FIGURA 7.22 *Eliminación del valor 89*

Por último, analice el siguiente ejemplo. En el árbol-B, de grado 2, de la figura 7.23 se quiere eliminar el valor 48. En este caso la altura del árbol, luego de efectuar todas las reestructuraciones que corresponden, disminuye en uno La tabla 7.10 presenta las operaciones que se realizan al aplicar el algoritmo.

TABLA 7.10 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con los valores: 48.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (48) con el valor 48. Es igual, entonces se elimina.
4	Se evalúa si el dato eliminado estaba en una página hoja. No lo estaba.
5	Se debe sustituir el elemento eliminado por el que se encuentre más a la derecha del subárbol izquierdo (44).

continúa

TABLA 7.10 Continuación

Operación	*Descripción*
6	Se evalúa si el número de elementos (1) de esa página es ≥ n y ≤ 2n. No lo es, entonces se debe bajar el dato más cercano (41) de la página padre y fusionar las páginas que son hijas de éste (22 – 30 – 41 – 42).
7	Se evalúa si el número de elementos de la página de la que se bajó el valor 41 es ≥ n y ≤ 2n. Es (1), por lo tanto se debe bajar el dato más cercano (44) de la página padre y fusionar las páginas que son hijas de éste (20 – 44 – 59 – 72). La fusión afectó la raíz, disminuyendo en 1 la altura del árbol. La operación termina con éxito

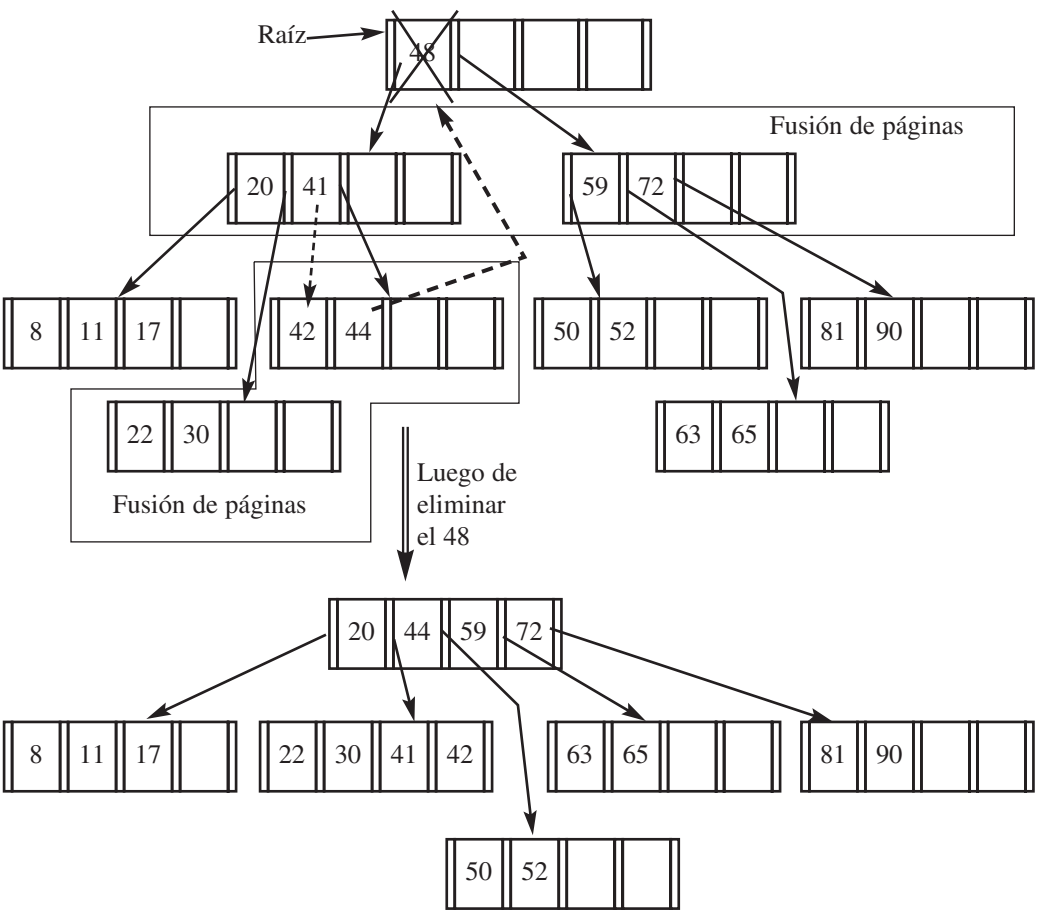


FIGURA 7.23 *Eliminación del valor 48*

7.5 Árboles-B+

Los ***árboles-B+*** son una variante de los árboles-B, diferenciándose de estos últimos por el hecho de que toda la información se encuentra almacenada en las hojas. En la raíz y en los nodos intermedios se guardan solamente las claves o índices que permiten llegar a un cierto dato.

Las principales características de un árbol-B+ de grado n son:

- La página raíz almacena como mínimo 1 dato y como máximo 2n datos.
- La página raíz tiene como mínimo 2 descendientes.
- Las páginas intermedias y hojas almacenan entre n y 2n datos.
- Las páginas intermedias tienen entre (n+1) y (2n+1) páginas descendientes.
- Todas las páginas hojas tienen la misma altura.
- La información se encuentra ordenada.
- Toda la información se encuentra en las páginas hojas, por lo que la clave guardada en la raíz o páginas intermedias se duplica.
- La información guardada en la raíz o en páginas intermedias cumple la función de índices que facilitan el acceso a un cierto dato.

La figura 7.24 presenta un ejemplo de un árbol-B+, de grado 2. En la raíz se almacenan valores que funcionan como índices para llegar a los datos que están en las hojas (es de suponer que el árbol almacena datos más complejos que simples números enteros).

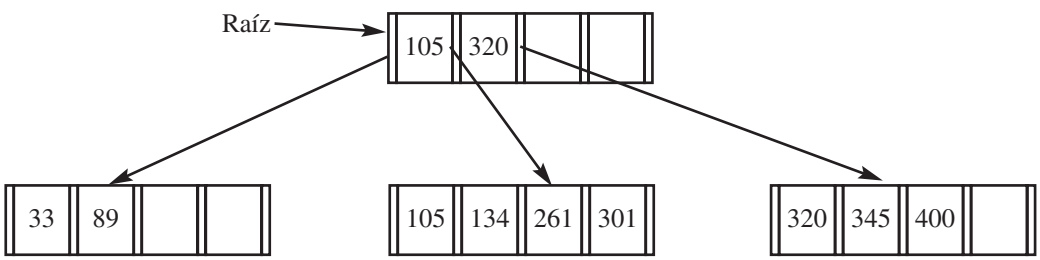


FIGURA 7.24 *Ejemplo de un árbol-B+*

Las operaciones que pueden realizarse sobre la información almacenada en árboles-B+ son: búsqueda, inserción y eliminación. Estas operaciones están implementadas en herramientas diseñadas especialmente para el manejo de archivos, por lo que generalmente no se requiere su programación en las aplicaciones.

Como en el caso de los árboles-B, se explican para que pueda entenderlas y cuando haga uso de algún manejador de archivos sepa qué está pasando internamente con los datos.

Búsqueda en árboles-B+

La operación de **búsqueda** es similar a la estudiada en los árboles-B. La diferencia es que en estos árboles la búsqueda termina siempre en las páginas hojas (donde está la información completa). Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.
2. Se evalúa si la página está vacía.
   - 2.1. Si la respuesta es afirmativa, entonces la búsqueda termina con fracaso.
   - 2.2. Si la respuesta es negativa, entonces ir al paso 3.
3. Se compara el dato buscado con cada elemento almacenado en la página.
   - 3.1. Si es igual, entonces se evalúa si es una página hoja.
     - 3.1.1. Si la respuesta es afirmativa, entonces la búsqueda termina con éxito.
     - 3.1.2. Si la respuesta es negativa, entonces se debe recuperar la página descendiente por el lado derecho y se regresa al paso 1.
   - 3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.
   - 3.3. Si es mayor, se avanza al siguiente dato de la misma página.
     - 3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
     - 3.3.2. Si no es el último, se regresa al paso 3.

La tabla 7.11 presenta los pasos requeridos para buscar el valor 320 en el árbol-B+ de la figura 7.24.

TABLA 7.11 Operación de búsqueda en un *árbol-B+*

Operación	*Descripción*
1	Se recupera la página con los datos: 105 – 320.
2	Se evalúa si está vacía. En este caso la respuesta es negativa.
3	Se compara el dato buscado (320) con el primer elemento de la página (105). Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato buscado (320) con el valor 320. Es igual.
5	Se evalúa si la página donde fue encontrado el 320 es una página hoja. No lo es. Se debe revisar la página descendiente por el lado derecho.
6	Se recupera la página con los datos: 320 – 345 – 400.
7	Se compara el dato buscado (320) con el primer elemento de la página (320). Es igual.
8	Se evalúa si la página donde fue encontrado el 320 es una página hoja. Sí lo es. La búsqueda termina con éxito.

Inserción en árboles-B+

La operación de **inserción** es similar a la estudiada en los árboles-B. La diferencia consiste en que cuando se produce la división de una página en dos (por dejar de cumplir la condición de que el número de elementos debe ser $\geq$ n y $\leq$ 2n) se debe subir una copia de la clave (o índice) del elemento del medio. Sólo se duplica información cuando la clave que sube es de una página hoja. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz).
2. Se evalúa si es una página hoja.
   2.1. Si la respuesta es afirmativa, se evalúa si la cantidad de elementos almacenados en ella es menor a 2n.
      2.1.1. Si la respuesta es afirmativa, entonces se procede a insertar el nuevo valor en el lugar correspondiente.
      2.1.2. Si la respuesta es negativa, se divide la página en dos y los (2n + 1) datos se distribuyen entre las páginas resultantes y una copia del valor del medio (o de una clave del mismo) sube a la página

padre. Si la página padre no tuviera espacio para este valor, entonces se procede de la misma manera, se divide en dos y el elemento del medio sube a la siguiente página, pudiendo repetirse este proceso hasta llegar a la raíz, en cuyo caso se aumenta la altura del árbol.

2.2. Si no es una hoja, se compara el elemento a insertar con cada uno de los valores almacenados para encontrar la página descendiente donde proseguir la búsqueda. Se regresa al paso 1.

Considere el siguiente ejemplo. En el árbol-B^+, de grado 2, de la figura 7.25 se quiere insertar el valor 287. Aplicando el algoritmo dado, se realizan las operaciones que se muestran en la tabla 7.12.

TABLA 7.12 Operación de inserción en un *árbol-B^+*

Operación	*Descripción*
1	Se recupera la página con los valores: 105 – 320.
2	Se evalúa si es una página hoja. No lo es.
3	Se compara el dato a insertar (287) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (320).
4	Se compara el dato a insertar (287) con el valor 320. Es menor, entonces se toma la dirección de la página que está a la izquierda del 320.
5	Se recupera la página con los valores: 105 – 222 – 261 – 301.
6	Se evalúa si es una página hoja. Sí lo es.
7	Se evalúa si el total de elementos almacenados (4) es menor a 2n. No lo es.
8	Se divide la página en dos y los elementos se distribuyen entre ellas, subiendo una copia del valor del medio (261) a la página padre.
9	Se evalúa (antes de la inserción) si el número de elementos en la página padre (2) es menor a 2n. Sí lo es. La operación termina con éxito.

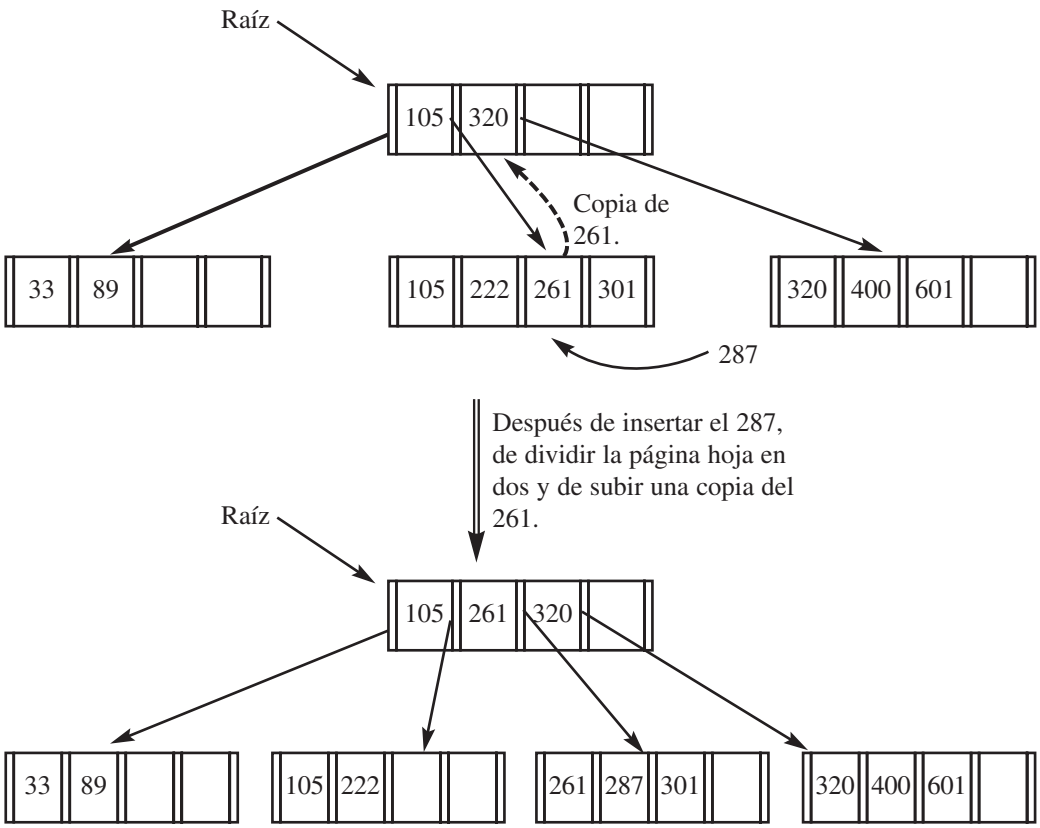


FIGURA 7.25 *Inserción del valor 287*

Eliminación en árboles-B+

La operación de **eliminación** consiste en quitar un elemento del árbol-B+ cuidando que mantenga las propiedades vistas. Es decir, el número de datos en cada página debe ser mayor o igual a n y menor o igual a 2n. Como los datos siempre están en las páginas hojas, cuando se encuentran se quitan (sólo de la hoja, sin importar si además están en un nodo intermedio o raíz) y sólo se reestructura el árbol si la página quedó con menos de n elementos. Los pasos para llevar a cabo esta operación son los siguientes:

1. Se recupera una página (la primera vez es la página raíz) y se la lleva a memoria.

2. Se evalúa si la página está vacía.
   2.1. Si la respuesta es afirmativa, entonces la operación de eliminación termina con fracaso.
   2.2. Si la respuesta es negativa, entonces ir al paso 3.
3. Se compara el dato a eliminar con cada elemento almacenado en la página.
   3.1. Si es igual, se evalúa si está en una página hoja.
      3.1.1. Si la respuesta es negativa entonces se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
      3.1.2. Si la respuesta es afirmativa, entonces se elimina y se evalúa si el número de elementos que quedó sigue siendo mayor o igual a n.
         3.1.2.1 Si la respuesta es afirmativa, entonces la operación termina con éxito. Las páginas intermedias no se modifican aunque almacenen una copia del elemento eliminado.
         3.1.2.2 Si la respuesta es negativa, entonces se debe bajar el dato más cercano de la página padre y sustituir a éste por el que se encuentre más a la izquierda del subárbol derecho o por el que se encuentre más a la derecha del subárbol izquierdo, siempre que esta página no pierda la condición. En caso contrario, se debe bajar el dato más cercano de la página padre y fusionar las páginas que son hijas de éste.
   3.2. Si es menor se toma la dirección de sus descendientes por el lado izquierdo y se regresa al paso 1.
   3.3. Si es mayor, se avanza al siguiente dato de la misma página.
      3.3.1. Si es el último, se toma la dirección de sus descendientes por el lado derecho y se regresa al paso 1.
      3.3.2. Si no es el último, se regresa al paso 3.

El proceso de fusión puede llegar hasta la raíz, en cuyo caso la altura del árbol disminuye en uno. Cuando se llevan a cabo las fusiones, se deben quitar todas aquellas claves copias de elementos eliminados en las páginas hojas.

La tabla 7.13 presenta la secuencia de operaciones requeridas para llevar a cabo la eliminación del valor 261 del árbol-B^+, de grado 2, de la figura 7.26.

TABLA 7.13 Operación de eliminación en un *árbol-B*

Operación	*Descripción*
1	Se recupera la página con el valor: 105.
2	Se evalúa si la página está vacía. No lo está.
3	Se compara el dato a eliminar (261) con el valor 105. Es mayor y no hay más elementos en la misma página, entonces se toma la dirección de la página que está a la derecha del 105.
4	Se recupera la página con los valores: 105 – 261.
5	Se compara el dato a eliminar (261) con el valor 105. Es mayor y hay más elementos en la misma página, entonces se avanza al siguiente valor (261).
6	Se compara el dato a eliminar (261) con el valor 261. Es igual.
7	Se evalúa si está en una página hoja. Sí lo está, entonces se elimina.
8	Se evalúa si el número de elementos que quedó en la página (1) es $\geq$ n y $\leq$ 2n. No lo es.
9	Se baja el dato más cercano de la página padre (105) y éste no se puede sustituir, ya que el número de elementos de su otro hijo es 2. Por lo tanto, se baja y se fusionan sus páginas descendientes. En este caso, la altura del árbol disminuye en uno. El proceso termina con éxito.

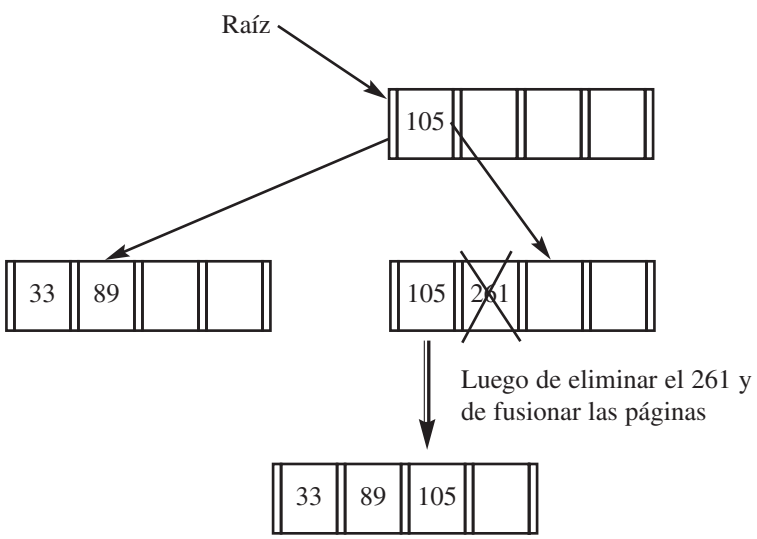
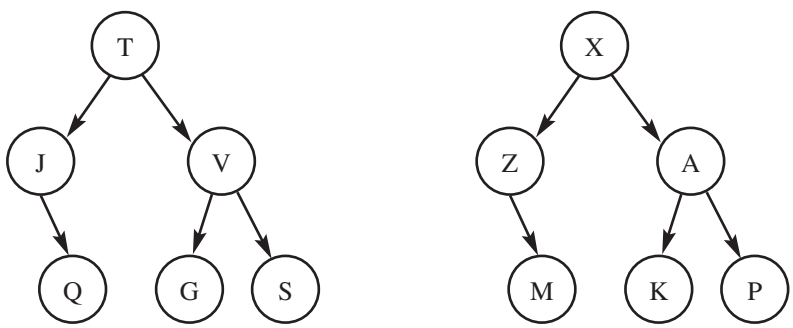


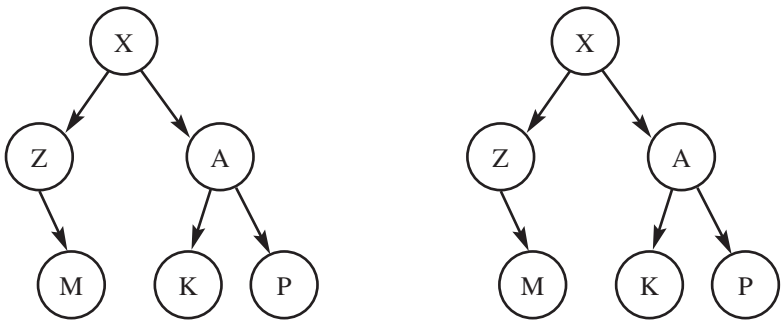
FIGURA 7.26 *Eliminación del valor 261*

Ejercicios

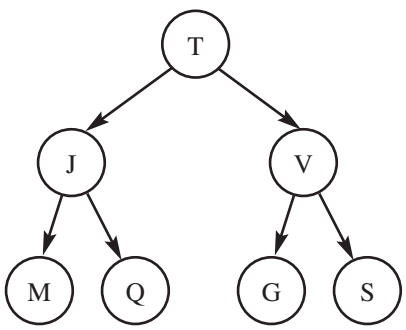
1. Defina una plantilla para la clase `ArbolMultiple`. Decida qué atributos y métodos incluir. ¿Puede implementar esta clase en ***C++***? Justifique su respuesta.
2. Escriba un método que cuente el número de hojas de un árbol binario. ¿Podría resolverlo por medio de una función? Justifique su respuesta.
3. Escriba un método que cuente el número de nodos intermedios de un árbol binario. ¿Podría resolverlo por medio de una función? Justifique su respuesta.
4. Escriba un método que calcule la altura de un árbol binario. ¿Podría resolverlo por medio de una función? Justifique su respuesta.
5. Escriba un método que imprima todos los ascendientes masculinos de un individuo cuyos datos genealógicos fueron almacenados en un árbol binario.
6. Escriba un método que imprima todos los datos de los ascendientes que estén vivos de un individuo cuyos datos genealógicos fueron almacenados en un árbol binario.
7. Se dice que dos árboles son *similares* cuando sus estructuras son iguales. Escriba una función en ***C++*** que determine si dos árboles (dados como parámetros) son *similares*. En la siguiente figura se presenta un ejemplo de dos árboles que tienen esta característica.



8. Se dice que dos árboles son *equivalentes* cuando sus estructuras son iguales y además tienen el mismo contenido. Escriba una función en ***C++*** que determine si dos árboles (dados como parámetros) son *equivalentes*. En la siguiente figura se presenta un ejemplo de dos árboles que tienen esta característica.



9. Se dice que un árbol binario es *completo* si todos sus nodos, excepto las hojas, tienen dos hijos. Escriba una función en ***C++*** que determine si un árbol (dado como parámetro) es *completo*. En la siguiente figura se presenta un ejemplo de un árbol que tiene esta característica.



10. Retome la plantilla de la clase `ArbolBinario` presentada en este libro, e incluya un método que imprima por niveles toda la información de un objeto tipo árbol. Si el árbol fuera el que aparece en el problema 8, la impresión sería:

X – Z – A – M – K – P

11. Considere que no puede manejar memoria dinámica para representar una estructura tipo árbol binario. Sin embargo, dadas las características de la información, usted decide que la mejor estructura para su almacenamiento y posterior uso es un árbol. Utilice un arreglo unidimensional, guardando en cada casilla la información correspondiente a un nodo, de tal manera que se mantengan las relaciones (padre-de y/o hijo-de) entre ellos. Diseñe e implemente las operaciones de búsqueda, inserción y eliminación que se ajuste a esta nueva representación.

12. Utilice un árbol binario de búsqueda para almacenar datos de tipo clientes bancarios. Para ello defina una clase `ClienteBanco`, con los atributos y los métodos que considere necesarios, atendiendo lo que se pide más abajo. El número de cliente será el atributo según el que se ordenará la información en el árbol. Escriba un programa en ***C++***, que mediante un menú de opciones, permita:

    a) Generar un reporte de todos los clientes de un banco, ordenados por su número de cliente.

    b) Generar un reporte de todos los clientes que tengan una antigüedad mayor a los 5 años. Puede darle generalidad a su solución, dejando el número de años como un dato a ingresar por el usuario.

    c) Generar un reporte de todos los clientes que tengan como mínimo dos cuentas diferentes en el banco.

    d) Dar de alta un nuevo cliente. El usuario proporcionará todos los datos del cliente a registrar.

    e) Dar de baja un cliente registrado. El usuario dará como dato el número del cliente.

    f) Actualizar el saldo de alguna de las cuentas de un cliente. El usuario dará como datos el número del cliente, el número de la cuenta a actualizar y el nuevo saldo.

    g) Actualizar los datos personales (por ejemplo domicilio, teléfono casa, teléfono oficina, etcétera) de un cliente. Su programa debe permitir que en la misma opción se pueda modificar uno o todos los datos personales.

13. Utilice un árbol binario balanceado para almacenar datos relacionados a insectos. Para ello defina una clase `Insecto`, con los atributos y los métodos que considere necesarios, atendiendo lo que se pide más abajo. Cada insecto tendrá una clave, que será el atributo que permita que la información esté ordenada en el árbol. Escriba un programa en ***C++***, que mediante un menú de opciones, pueda:

    a) Registrar un nuevo insecto. El usuario dará todos los datos necesarios.

    b) Dar de baja un insecto registrado. El usuario dará la clave del insecto a eliminar.

    c) Generar un reporte de todos los insectos, ordenados por clave.

    d) Generar un reporte de todos los insectos que viven en el área del Mediterráneo europeo.

e) Generar un reporte de todos los insectos que viven sólo en el desierto de Rub al-Jali.

f) Generar un reporte de todos los insectos que se alimentan de madera en estado de descomposición.

14. Inserte los siguientes datos en un árbol-B, de grado 2. Los números que se dan como datos pueden representar datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 95 – 10 – 34 – 87 – 56 – 99 – 12 – 23 – 50 – 40 – 60 – 54 – 33 – 20 – 91 – 17 – 18 – 94

15. En el árbol-B generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

    Eliminar: 99 – 60 – 23 – 12 – 95 – 40

16. Inserte los siguientes datos en un árbol-B, de grado 2. Los números que se dan como datos pueden representar datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 105 – 99 – 104 – 80 – 16 – 74 – 112 – 230 – 71 – 33 – 86 – 399 – 33 – 120 – 51 – 67 – 90 – 84 – 45 – 405 – 257 – 110

17. En el árbol-B generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

    Eliminar: 399 – 80 – 105 – 84 – 86 – 51 – 67 – 33 – 112 – 104

18. Inserte los siguientes datos en un árbol-B^+, de grado 2. Los números que se dan como datos pueden representar a datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 120 – 100 – 240 – 817 – 356 – 199 – 249 – 326 – 500 – 170 – 360 – 257 – 358 – 104 – 921 – 590 – 328 – 140

19. En el árbol-B^+, de grado 2, generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

    Eliminar: 328 – 356 – 100 – 817 – 921 – 500 – 358 – 328 – 590 – 104 – 249

20. Inserte los siguientes datos en un árbol-B^+, de grado 2. Los números que se dan como datos pueden representar a datos más complejos (objetos). Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la inserción.

    Insertar: 350 – 180 – 420 – 700 – 390 – 200 – 150 – 400 – 300 – 100 – 500 – 310 – 660 – 580 – 880 – 670 – 370 – 140 – 230 – 490 – 510

21. En el árbol-B^+, de grado 2, generado en el problema anterior elimine los datos que se señalan a continuación. Dibuje el árbol a medida que sufra cambios en su estructura como consecuencia de la eliminación.

    Eliminar: 880 – 420 – 100 – 580 – 180 – 230 – 400 – 700 – 660 – 670 – 490 – 140 – 350 – 370