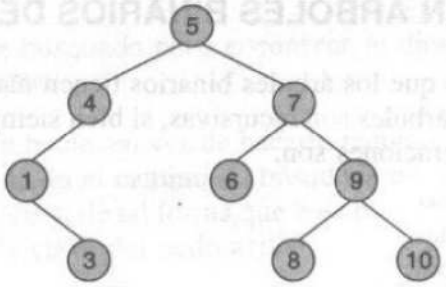
Ejemplo 14.10

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

Solución



14.7.2. Nodo de un árbol binario de búsqueda

Un nodo de un árbol binario de búsqueda no difiere en nada de los nodos de un árbol binario, tiene un campo de datos y dos punteros a los subárboles izquierdo y derecho respectivamente.

Un árbol de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. Un ejemplo de árbol binario de búsqueda es el que cada nodo contiene información relativa a una persona. Cada nodo almacena un nombre de un alumno (dato de tipo cadena) y el número de matrícula en su universidad (dato entero).

Declaración de tipos


```
struct nodo {
   int nummat;          /* número de matrícula */
   char nombre[30];     /* nombre del alumno */
   struct nodo *izdo, *dcho;
};
typedef struct nodo Nodo;
```

Creación de un nodo

La función tiene como entrada un dato entero, que representa un número de matrícula y el nombre. Devuelve un puntero al nodo creado.

```
Nodo* crearNodo(int id, const char* n)
{
   Nodo* t;
   t = (Nodo*) malloc(sizeof(Nodo));
```

```
    t -> nummat = id;
    strcpy(t -> nombre, n);
    t -> izdo = t -> dcho = NULL;
    return t;
}
```

14.8. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

De lo expuesto se deduce que los árboles binarios tienen naturaleza recursiva y, en consecuencia, las operaciones sobre los árboles son recursivas, si bien siempre se tiene la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo.
- *Inserción* de un nodo.
- *Recorrido* de un árbol.
- *Borrado* (eliminación) de un nodo.

14.8.1. Búsqueda

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

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

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

Buscar una información específica

Si se desea encontrar un nodo en el árbol que contenga una información determinada, la función `buscar()` tiene dos parámetros, un puntero al árbol y el dato que se busca. Como resultado, la función devuelve un puntero al nodo en el que se almacena la información; en el caso de que la información no se encuentre se devuelve el valor 0. El algoritmo de búsqueda es el siguiente:

1. Comprobar si el árbol está vacío. En caso afirmativo se devuelve 0. Si la raíz contiene el dato buscado, la tarea es fácil: el resultado es, simplemente, un puntero a la raíz.
2. Si el árbol no está vacío, el subárbol específico depende de que dato requerido sea más pequeño o mayor que el dato del nodo raíz.
3. La función de búsqueda se consigue llamando recursivamente a la función `buscar()` con un puntero al subárbol izquierdo o derecho como parámetro.

El código C de la función `buscar()`:

```
Nodo* buscar (Nodo* raiz, TipoElemento buscado)
{
  if (!raiz)
    return 0;
  else if (buscado == raiz -> dato)
    return raiz;
```

```
    else if (buscado < raiz -> dato)
      return buscar (raiz -> izdo, buscado);
    else
      return buscar (raiz -> dcho, buscado);
}
```

Ejemplo 14.11

Se desea aplicar la función de búsqueda para encontrar la dirección del nodo que contenga el número de matrícula de un alumno.

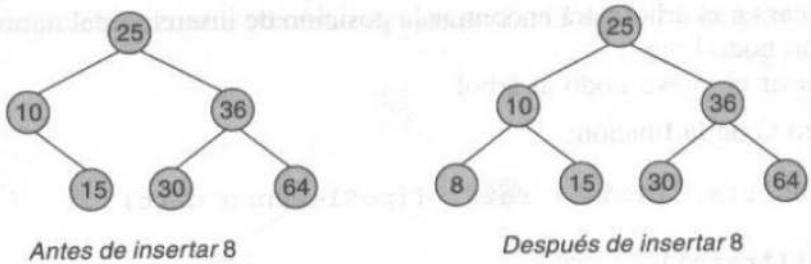
La función se escribe con un bucle, en vez de hacerlo recursivamente. El bucle termina cuando se encuentra el nodo, o bien cuando el camino de búsqueda ha finalizado.

La función se *mueve* por el árbol, de tal forma que baja por la rama izquierda o derecha según la clave sea menor o mayor que la clave del nodo actual.

```
Nodo* buscar (Nodo* raiz, int matricula)
{
  int encontrado = 0;

  while (!encontrado && raiz != NULL)
  {
    if (matricula == raiz -> nummat)
      encontrado = 1;
    else if (matricula < raiz -> nummat)
      raiz = raiz -> izdo;
    else if (matricula > raiz -> nummat)
      raiz = raiz -> dcho;
  }
  return raiz;
}
```

14.8.2. Insertar un nodo

Una característica fundamental que debe poseer el algoritmo de inserción es que el árbol resultante de una inserción en un árbol de búsqueda ha de ser también de búsqueda. En esencia, el algoritmo de inserción se apoya en la localización de un elemento, de modo que si se encuentra el elemento (*clave*) buscado, no es necesario hacer nada; en caso contrario, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en el caso de existir).



Antes de insertar 8 *Después de insertar* 8

Figura 14.29. Inserción en un árbol binario de búsqueda.

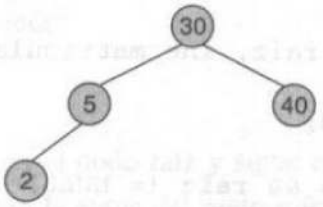
Por ejemplo, considérese el caso de añadir el nodo `8` al árbol de la Figura 14.29. Se comienza el recorrido en el nodo raíz `25`; la posición `8` debe estar en el subárbol izquierdo de `25 (8 < 25)`. En el nodo `10`, la posición de `8` debe estar en el subárbol izquierdo de `10`, que está actualmente vacío. El nodo `8` se introduce como un hijo izquierdo del nodo `10`.

A recordar

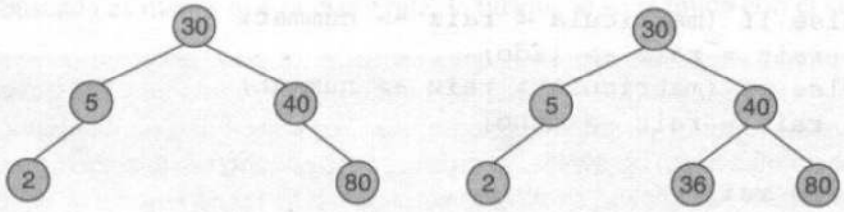
La inserción de un nuevo nodo en un árbol de búsqueda siempre se hace como nodo hoja. Para ello se *baja* por el árbol según el *camino de búsqueda*.

Ejemplo 14.12

Insertar un elemento con clave 80 en el árbol binario de búsqueda siguiente:



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



a) Inserción de 80 *b*) Inserción de 36

14.8.3. Función `insertar()`

La función `insertar()` que inserta nuevos nodos es sencilla. Se deben declarar dos argumentos, un puntero al raíz del árbol y el dato que tendrá el nodo. La función creará un nuevo nodo y lo inserta en el lugar correcto en el árbol de modo que el árbol permanezca como binario de búsqueda.

La operación de *inserción* de un nodo es una extensión de la operación de búsqueda. Los pasos a seguir son:

1. Asignar memoria para una nueva estructura nodo.
2. Buscar en el árbol para encontrar la posición de inserción del nuevo nodo, que se colocará como nodo hoja.
3. Enlazar el nuevo nodo al árbol.

El código C de la función:

```
void insertar (Nodo** raiz, TipoElemento dato)
{
  if (!(*raiz))
    *raiz = crearNodo(dato);
```

```
    else if (dato < (*raiz) -> dato)
      insertar (&((*raiz) -> izdo), dato);
    else
      insertar (&((*raiz) -> dcho), dato);
}
```

Si el árbol está vacío, es fácil insertar la entrada en el lugar correcto. El nuevo nodo es la raíz del árbol y el puntero `raiz` se pone apuntando a ese nodo. El parámetro `raiz` debe ser un parámetro referencia ya que debe ser leído y actualizado, por esa razón se declara puntero a puntero (`Nodo**`) . Si el árbol no está vacío, se debe elegir entre insertar el nuevo nodo en el subárbol izquierdo o derecho, dependiendo de que el *dato* sea más pequeño o mayor que el *dato* en la raíz del árbol.

La función `crearNodo()` reserva memoria para el nuevo nodo y asigna el nuevo dato. Esta función se debe adaptar a cada aplicación, así en el Apartado 14.7.2, se crea un nodo para guardar el número de matrícula y el nombre de una persona.

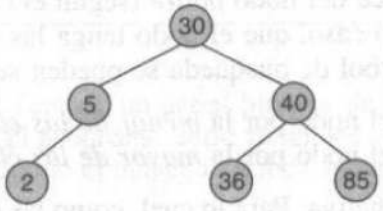
14.8.4. Eliminación

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

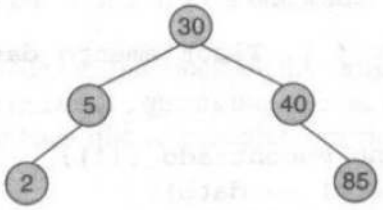
1. Buscar en el árbol para encontrar la posición de nodo a eliminar.
2. Reajustar los punteros de sus antecesores si el nodo a suprimir tiene menos de 2 hijos, o subir a la posición que éste ocupa el nodo más próximo en valor del campo clave (inmediatamente superior o inmediatamente inferior) con objeto de mantener la estructura de árbol binario.

Ejemplo 14.13

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

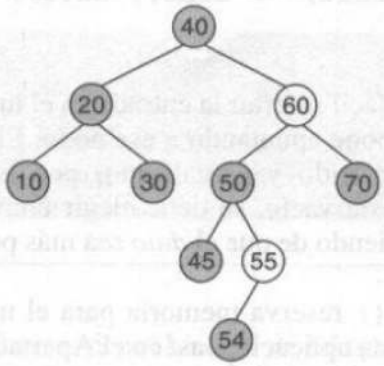


El nodo del árbol donde se encuentra la clave `36` es una hoja, por ello simplemente se reajustan los enlaces del nodo precedente en el camino de búsqueda. El árbol resultante:

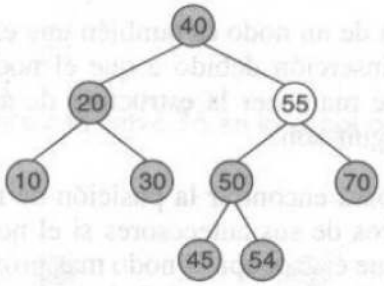


Ejemplo 14.14

Borrar el elemento de clave `60` del siguiente árbol:



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



14.8.5. Función `eliminar()`

La función `eliminar()` presenta dos casos claramente diferenciados. El primero, si el nodo a eliminar es una hoja o tiene un único descendiente, resulta una tarea fácil, ya que lo único que hay que hacer es asignar al enlace del nodo *padre* (según el camino de búsqueda) el descendiente del nodo a eliminar. El segundo caso, que el nodo tenga las dos ramas no vacías; en cuyo caso para mantener la estructura de árbol de búsqueda se pueden seguir dos alternativas:

- Reemplazar el dato del nodo por la *menor de las claves mayores* en su subárbol derecho.
- Reemplazar el dato del nodo por la *mayor de las claves menores* en su subárbol izquierdo.

Se elige la segunda alternativa. Para lo cual, como las claves menores están en la rama izquierda, se *baja* al primer nodo de la rama izquierda, y como la clave mayor está en la rama derecha, se continúa *bajando* por la rama derecha hasta alcanzar el nodo hoja. Éste es el mayor de los menores que reemplaza a la clave del nodo a eliminar. La función `reemplazar()` realiza la tarea descrita. El código en C:

```
void eliminar (Nodo** r, TipoElemento dato)
{
  if (!(*r))
    puts("!! Nodo no encontrado !!");
  else if (dato < (*r) -> dato)
    eliminar(&(*r) -> izdo, dato);
```

```
  else if (dato > (*r) -> dato)
    eliminar(&(*r) -> dcho,dato);
  else                                              /* Nodo encontrado */
  {
    Nodo* q;                                /* puntero al nodo a suprimir */
    q = (*r);
    if (q -> izdo == NULL)
      (*r) = q -> dcho;
    else if (q -> dcho == NULL)
      (*r) = q -> izdo;
    else
    {                                    /* tiene rama izquierda y derecha */
      reemplazar(&q);
    }
    free(q);
  }
}

void reemplazar(Nodo** act)
{
  Nodo* a, *p;

  p = *act;
  a = (*act) -> izdo;                                /* rama de menores */
  while (a -> dcho)
  {
    p = a;
    a = a -> dcho;
  }
  (*act) -> dato = a -> dato;
  if (p == (*act))
    p -> izdo = a -> izdo;
  else
    p -> dcho = a -> izdo;
  (*act) = a;
}
```

Ejercicio 14.2

Con los registros de estudiantes, formar un árbol binario de búsqueda, ordenado respecto al campo clave número de matrícula. El programa debe tener las opciones de mostrar los registros ordenados y eliminar un registro dando el número de matrícula.

Cada registro tiene sólo dos campos de información: `nombre` y `nummat`. Además, los campos de enlace con el subárbol izquierdo y derecho.

Las operaciones utilizadas son `insertar`, `eliminar`, `buscar` y `visualizar` el árbol. Los algoritmos de las tres primeras operaciones ya están descritos anteriormente. La operación de `visualizar` consiste en recorrer en `inorden` el árbol, cada vez que se visita el nodo raíz se escriben los datos del estudiante.

No se vuelve a escribir el código de las funciones; dicho código puede consultarse en los apartados anteriores. Si se escribe `visualizar()`, que es una aplicación de recorrer en *inorden* el árbol de búsqueda, este tipo de recorrido hace que se muestren por pantalla los registros.

```
#include <stdio.h>
#include <string.h>
```

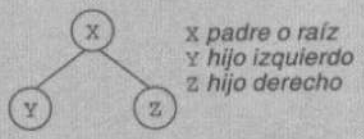
```
#include <stdlib.h>
#include <conio.h>
struct nodo {
  int nummat;
  char nombre[30];
  struct nodo *izdo, *dcho;
};
typedef struct nodo Nodo;

Nodo* crearNodo(int id, char* n);
Nodo* buscar (Nodo* p, int buscado);
void insertar (Nodo** raiz, int matricula, char* nombre);
void reemplazar(Nodo** act);
void eliminar (Nodo** raiz, int matricula);
void visualizar (Nodo* r);

void main()
{
  int nm;
  char nom[30];
  Nodo* raiz = NULL;
            /* Crea el árbol */
  do{
    printf("Numero de matricula(0 -> Fin): "); scanf("%d%*c",&nm);
    if (nm)
    {
      printf("Nombre: "); gets(nom);
      insertar(&raiz,nm,nom);
    }
  }while (nm);
    /* Opciones de escribir el árbol o borrar un registro */
  clrscr();
  do{
    puts("     1. Mostrar el árbol\n");
    puts("     2. Eliminar un registro\n");
    puts("     3. Salir\n ");
    do scanf("%d%*c", &nm); while(nm<1 || nm>3);
    if (nm == 1) {
      printf("\n\t Registros ordenados por número de matrícula:\n");
      visualizar(raiz);
    }
    else if (nm == 2){
      int cl;
      printf("Clave: "); scanf("%d",&cl);
      eliminar(&raiz,cl);
    }
  }while (nm != 3);
}
void visualizar (Nodo* r)
{
  if (r)
  {
```

```
        visualizar(r -> izdo);
        printf("Matricula %d \t %s \n",r->nummat,r->nombre);
        visualizar(r -> dcho);
    }
}
```

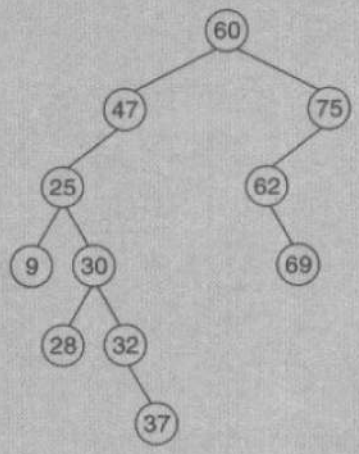
RESUMEN

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

La estructura árbol más utilizada normalmente es el *árbol binario*. Un árbol binario es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho.

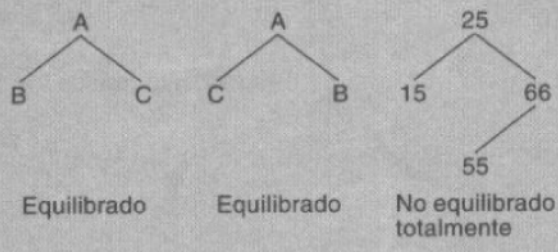


En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un padre. X es un *antecesor* o *ascendente* del elemento Y.



La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana, más 1. Si el árbol A es vacío, la altura es 0. La altura del árbol anterior es 6. *El nivel* o *profundidad* de un elemento es un concepto similar al de altura. En el árbol anterior el nivel de 30 es 3 y el nivel de 37 es 5. Un nivel de un elemento se conoce también como *profundidad*.

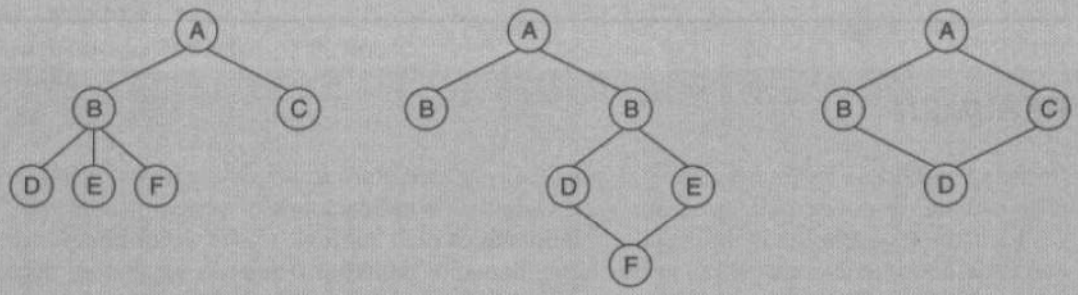
Un árbol binario no vacío está *equilibrado totalmente* si sus subárboles izquierdo y derecho tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados.



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

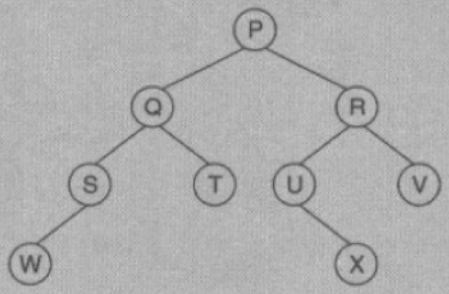
EJERCICIOS

14.1. Explicar por qué cada una de las siguientes estructuras no es un árbol binario.



14.2. Considérese el árbol siguiente:

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



14.3. Para cada una de las siguientes listas de letras:

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

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

14.4. En el árbol del Ejercicio 14.2, recorrer dicho árbol utilizando los órdenes siguientes: NDI, DNI, DIN.

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

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

14.6. El recorrido *preorden* de un cierto árbol binario produce:

```
ADFGHKLPQRWZ
```

y en recorrido *enorden* produce:

```
GFHKDLAWRQPZ
```

Dibujar el árbol binario.

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

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

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

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

14.11. Se dispone de un árbol binario de elementos de tipo entero. Escribir funciones que calculen:

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

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

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

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

PROBLEMAS

14.1. Crear un archivo de datos en el que cada línea contenga la siguiente información:

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

Escribir un programa que lea cada registro de datos y lo inserte en un árbol, de modo que cuando el árbol se recorra utilizando recorrido en orden, los números de la seguridad social se ordenen en orden ascendente. Imprimir una cabecera «DATOS DE EMPLEADOS ORDENADOS POR NÚMERO SEGURIDAD SOCIAL». A continuación se han de imprimir los tres datos utilizando el siguiente formato de salida:

Columnas	1-11	Número de la Seguridad Social
	25-44	Nombre
	58-104	Dirección