# CAPíTULO 16

# ÁRBOLES

## CONTENIDO

## INTRODUCCIÓN

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

Los árboles son muy utilizados en informática para representar fórmulas algebraicas como un método eficiente para búsquedas grandes y complejas, listas dinámicas y aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de las aplicaciones citadas, los árboles se utilizan en diseño de compiladores, proceso de texto y algoritmos de búsqueda.

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

## CONCEPTOS CLAVE

- Árbol.
- Árbol binario.
- Árbol binario de búsqueda.
- Conceptos teóricos *(nivel, profundidad,* raíz, *hoja,* rama, ...).
- *Enorden.*
- Nodo.
- *Preorden.*
- *Postorden.*
- Recorrido de un árbol.
- *Subárbol.*

## 16.1. ÁRBOLES GENERALES

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general. La Figura 16.1 representa dos ejemplos de árboles generales.

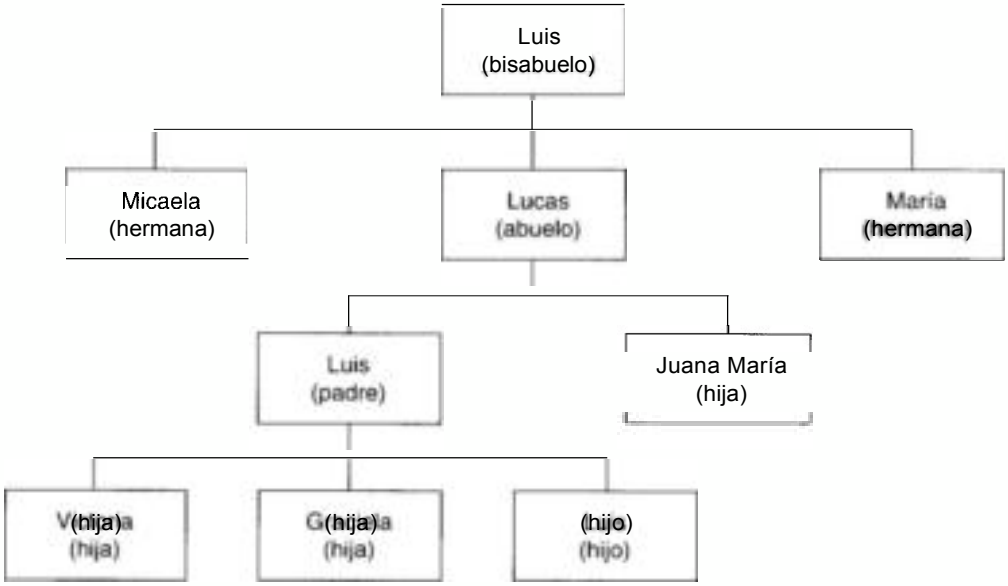



**Figura 16.1.** Árbol genealógico (bisabuelo–bisnietos).

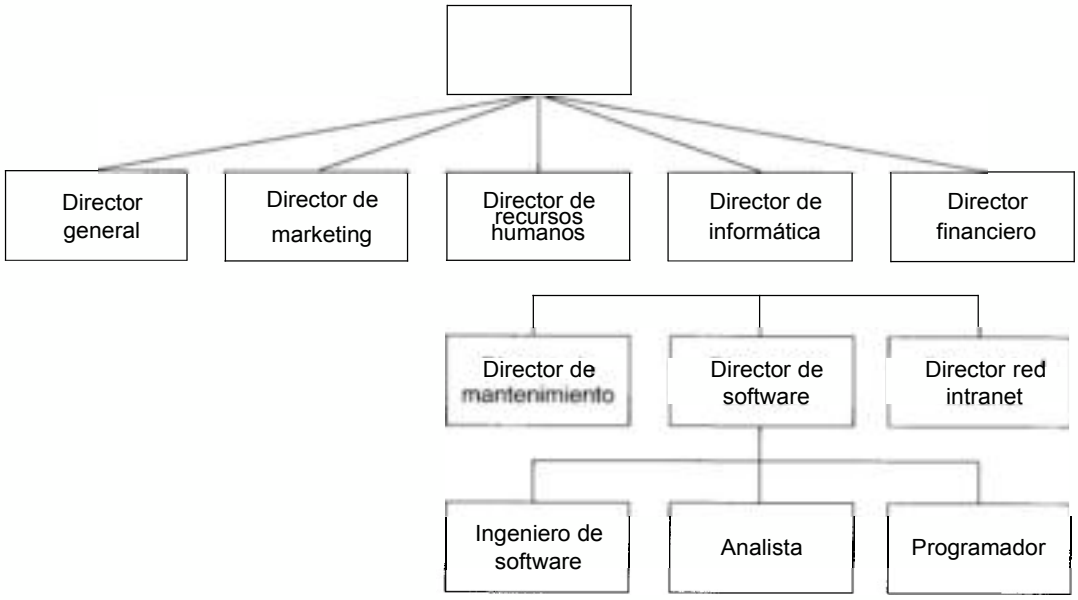



**Figura 16.2.** Estructura jerárquica tipo árbol.

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

**Definición 1:** Un **árbol** consta de un conjunto finito de elementos, llamados nodos **y** un conjunto finito de líneas dirigidas, llamadas ramas, que conectan los nodos.

**Definición 2:** Un **árbol** es un conjunto de uno o más nodos tales que:

1. Hay un nodo diseñado especialmente llamado **raíz.**
2. Los nodos restantes se dividen en **n≥o** conjuntos disjuntos tales que $T_1...T_n$, en donde cada uno de estos conjuntos es un árbol. A $T_1, T_2,...T_n$ se les denomina *subárboles* del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz.** Obsérvese en la definición 2 que el árbol ha sido definido de modo recursivo ya que los subárboles se definen como árboles. La Figura **16.3** muestra un árbol.

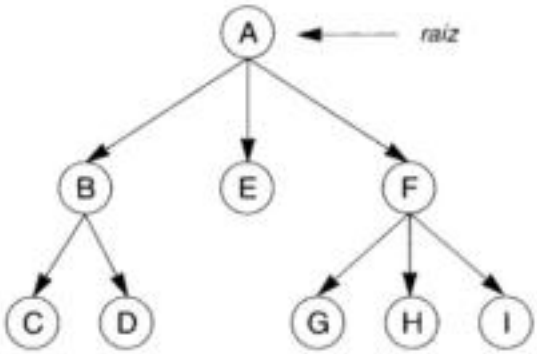



**Figura 16.3.** Árbol.

## Terminología

Además del raíz existen muchos términos utilizados en la descripción de los atributos de un árbol. En la Figura 16.4, el nodo **A** es el raíz. Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores.

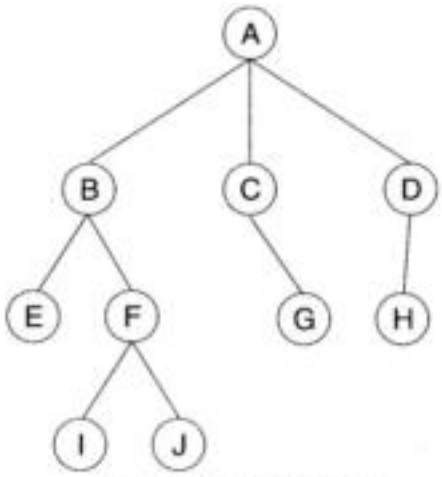
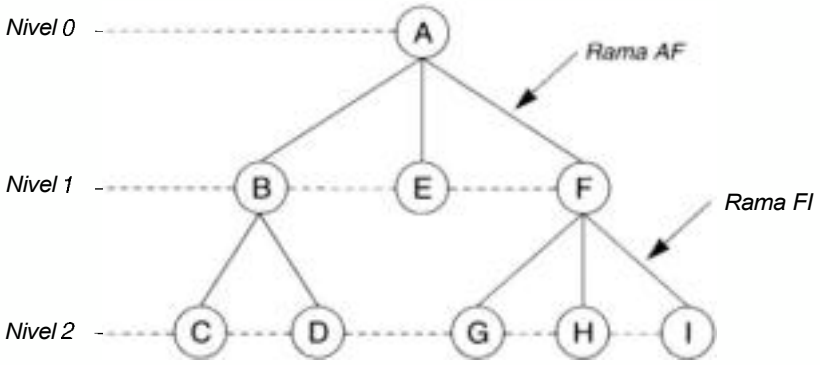



**Figura 16.4.** Árbol general.

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

El **nivel** de un nodo es su distancia al raíz. El raíz tiene una distancia cero de sí misma, por lo que se dice que el raíz está en el nivel 0. Los hijos del raíz están en el nivel 1, sus hijos están en el nivel 2 y así sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la relación entre niveles y hermanos. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos. Por ejemplo, en el nivel 2 (Fig. 16.5), C y D son hermanos, al igual que lo son G, H e I, pero D y G no son hermanos ya que ellos tienen diferentes padres.




| | |
|---|---|
| *padre:* **A,** B, F | *hermanos:* {B, E, F), *{C,* D}, {G, H, I} |
| *hijos: B,* E, F, *C,* D, G, **H,** I | *hojas: C,* D, E, G, H, I |

**Figura 16.5.** Terminología **de** árboles.

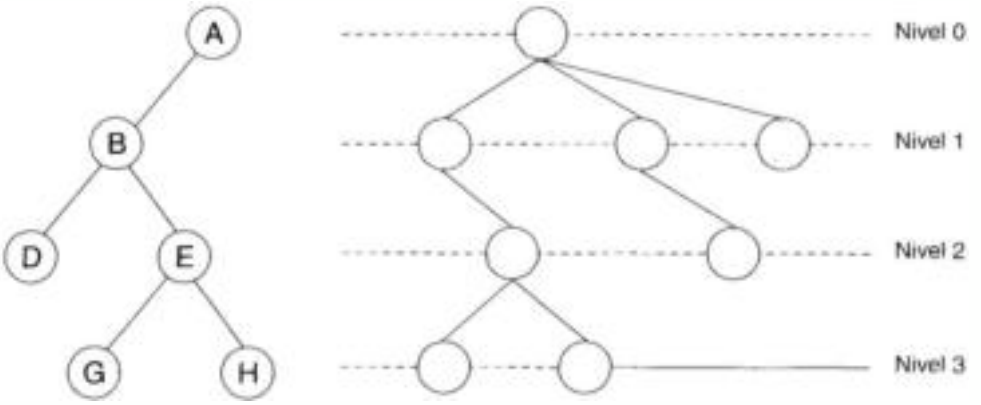
Existen varias formas de dibujar los atributos de los árboles y sus nodos. Un **camino** es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el raíz. **En** la Figura 16.5, el camino desde el raíz a la hoja **I,** se representa por AFI. Incluye dos ramas distintas AF y FI.

La altura o profundidad de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición' la altura de un árbol vacío es 0. La Figura 16.5 contiene nodos en tres niveles : 0, 1 y 2. Su altura es *3.*

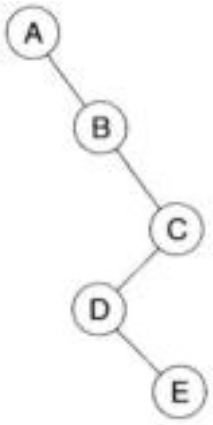
## *Definición*

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

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




*(a) Profundidad 4* *(b) Profundidad 4*




*(c) Profundidad 5*



**Figura 16.6.** Árboles de profundidades diferentes.

Un árbol se divide en subárboles. Un **subárbol** es cualquier estructura conectada por debajo del raíz. Cada nodo de un árbol es la raíz de **un** subárbol que se define por el nodo **y** todos los descendientes del nodo. El primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En la Figura 16.5, BCD es un subárbol al igual que E y FGHI. Obsérvese que por esta definición, un nodo simple es un subárbol. Por consiguiente, el subárbol B se puede dividir en subárboles C y D mientras que el subárbol F contiene los subárboles G, H e I. Se dice que G, H, I, C y D son subárboles sin descendientes. El concepto de subárbol conduce a una ***definición recursiva*** de un árbol. Un árbol es un conjunto de nodos que:

1. O bien es vacío, o bien
2. Tiene un nodo determinado llamado *raíz* del que jerárquicamente descienden cero o más subárboles, **que** son también árboles.

Un árbol **está equilibrado** cuando, dado un número máximo de ***k*** hijos para cada nodo y la **altura del árbol** *h,* cada nodo de nivel $l < h - 1$ tiene exactamente ***k*** hijos. El árbol está **equilibrado perfectamente** cuando cada nodo de nivel $l < h$ tiene exactamente ***k*** hijos.

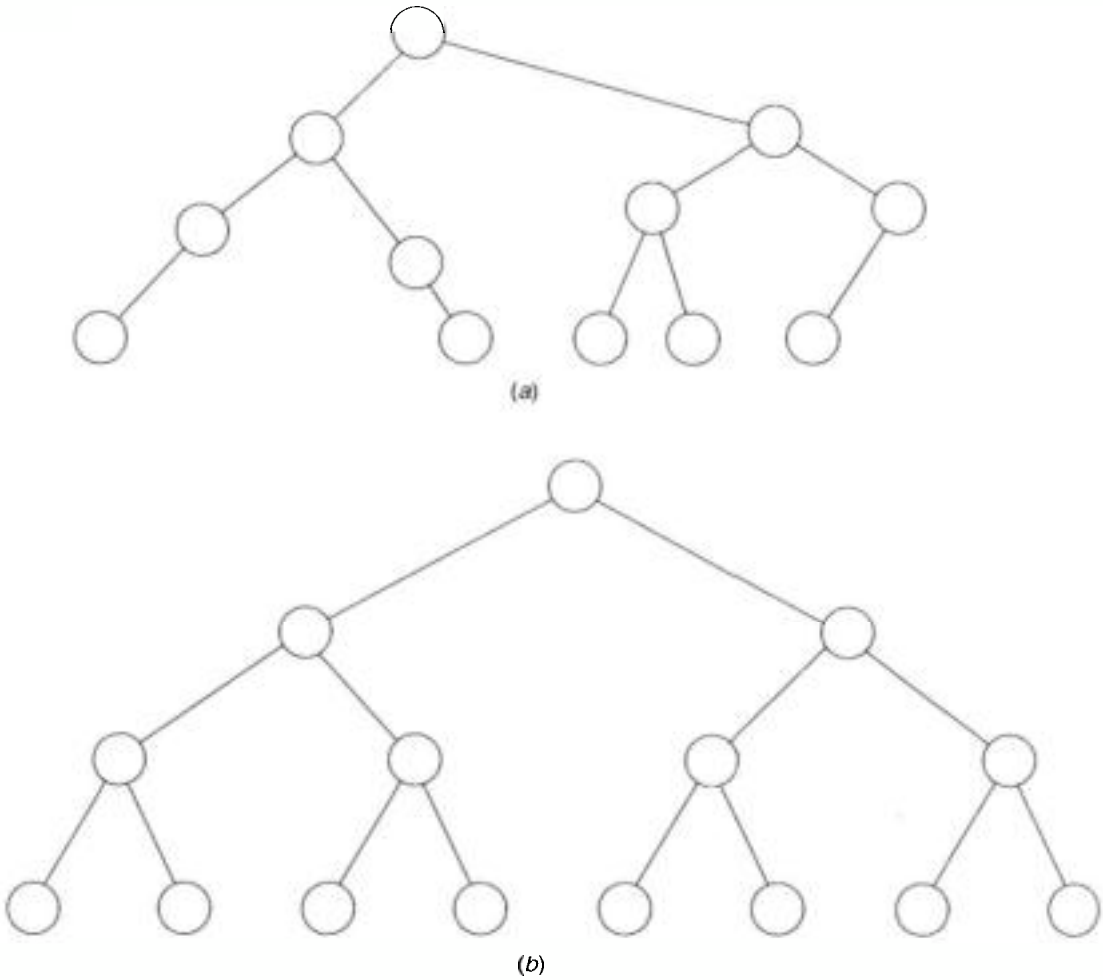



**Figura 16.7.** (*a*) Un árbol equilibrado; (*b*) Un árbol perfectamente equilibrado.

## 16.1.1. Representación de un árbol

Aunque un árbol se implementa en un lenguaje de programación como C mediante punteros, cuando se ha de representar en papel, existen tres formas diferentes de representación. La primera es el diagrama o carta de organización utilizada hasta ahora en las diferentes figuras. El término que se utiliza para esta notación es el de árbol general.

***Representación en niveles de profundidad***

Este tipo de representación es el utilizado para representar sistemas jerárquicos en modo texto o número en situaciones tales como facturación, gestión de ***stocks*** en almacenes, etc.

Por ejemplo, en las Figuras 16.8 y 16.9 se aprecia una descomposición de una computadora en sus diversos componentes en una estructura árbol. Otro ejemplo podría ser una distribución en árbol de las piezas de una tienda de recambios de automóviles distribuidas en niveles de profundidad según los números de parte o códigos de cada repuesto (motor, bujía, batería, piloto, faro, embellecedor, etc.).

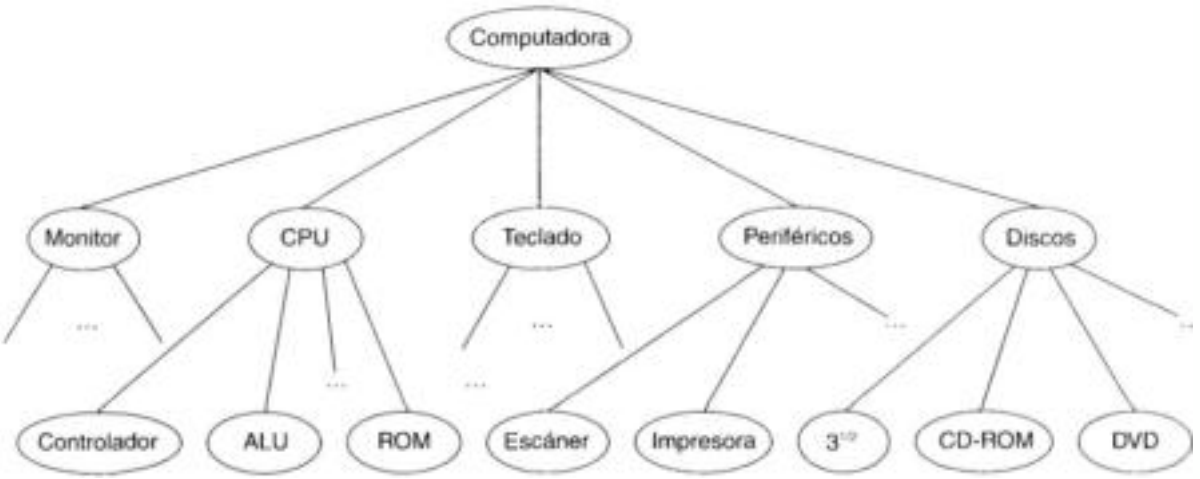



**Figura 16.8.** Árbol general (computadora).

| Número código | Descripción |
|---|---|
| 501 | Computadora |
| 501-11 | Monitor |
| ... | |
| 501-21 | CPU |
| 501-211 | Controlador |
| 501-212 | ALU |
| ... | ... |
| 501-219 | ROM |
| 501-31 | Teclado |
| ... | ... |
| 501-41 | Periféricos |
| 501-411 | Escáner |
| 501-412 | impresora |
| 501-51 | Discos |
| 501-511 | CD-ROM |
| 501-512 | CD-RW |
| 501-513 | DVD |

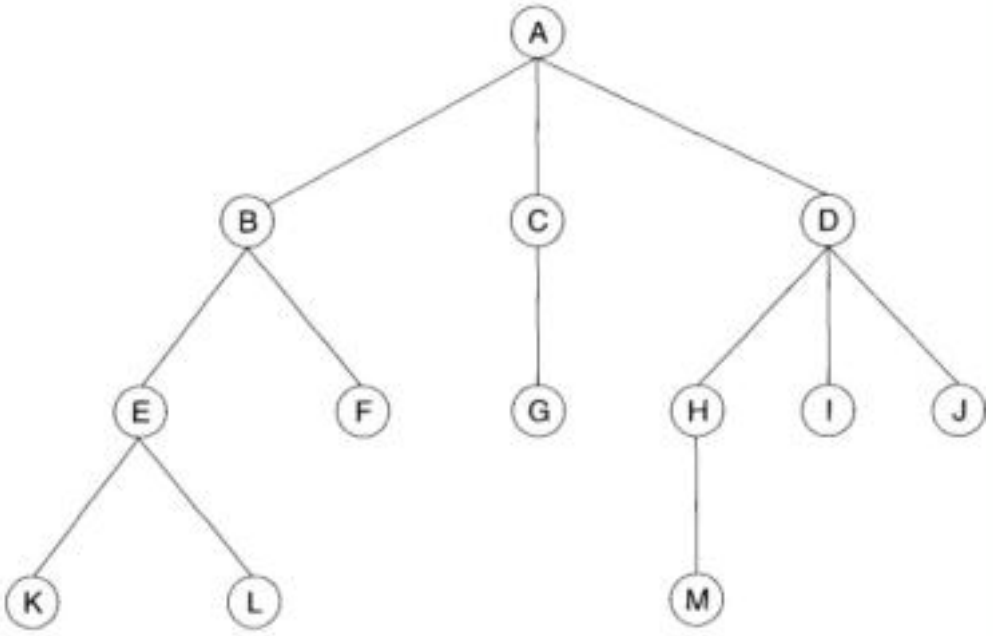
**Figura 16.9.** Árbol en nivel de profundidad (computadora).

Representación *de* lista

Otro formato utilizado para representar un árbol es la lista entre paréntesis. Ésta es la notación utilizada con expresiones algebraicas. En esta representación, cada paréntesis abierto indica el comienzo de un nuevo nivel; cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol. La notación en paréntesis de la Figura 16.3 es: `A(B (C, D), E, F, (G, H, I))`.

## Ejemplo 16.1

*Convertir el árbol general siguiente en representación en lista.*




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

## 16.2. RESUMEN DE DEFINICIONES

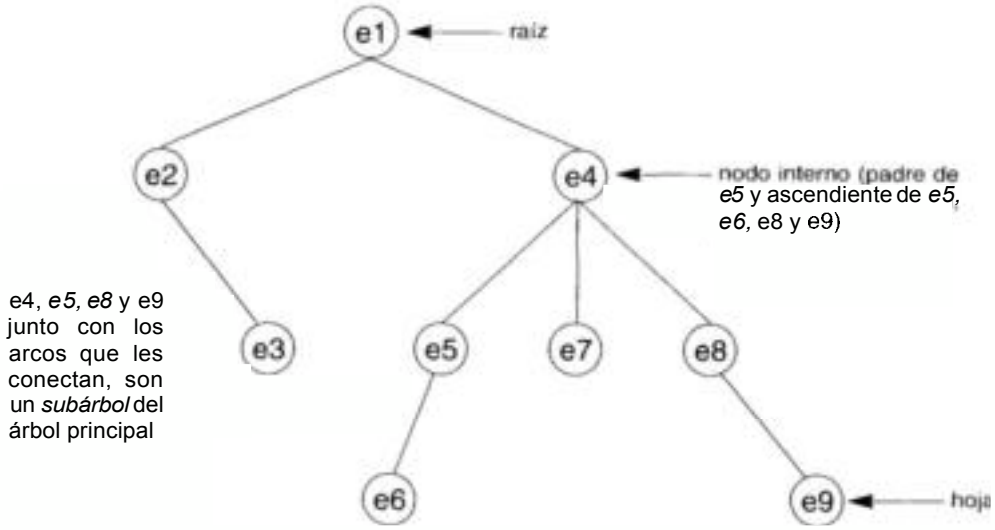
**1.** Dado un conjunto E de elementos:

- Un árbol puede estar *vacio;* es decir, no contiene ningún elemento,
- Un árbol *no vacío* puede constar de un único elemento $e \in E$ denominado un **nodo,** o bien
- Un árbol consta de un nodo $e \in E$, conectado por arcos directos a un número finito de otros árboles.

**2.** Definiciones:

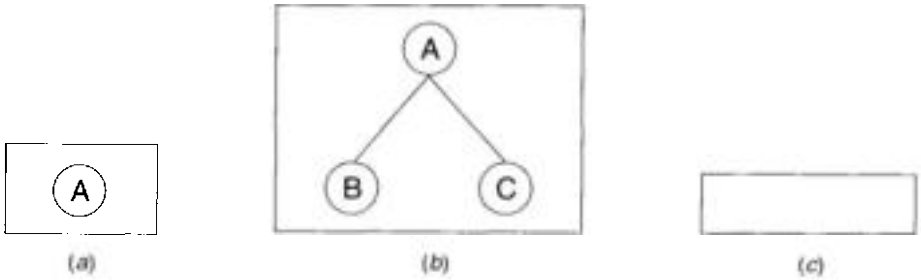
- El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
- Las flechas que conectan un nodo a otro se llaman **arcos** o **ramas.**
- Los **nodos terminales,** esto es, nodos de los cuales no se deduce ningún nodo, se denominan **hojas.**
- Los nodos que no son hojas se denominan **nodos internos** o **nodos no terminales.**
- En un árbol una rama va de un nodo $n_1$ **a** un nodo $n_2$, se dice que $n_1$ es el padre de $n_2$ **y** que $n_2$ es un **hijo** de $n_1$.
- $n_1$ se llama **ascendiente** de $n_2$ si $n_1$ es el padre de $n_2$ o si $n_1$ es el padre de un ascendiente de $n_2$.
- $n_2$ se llama **descendiente** de $n_1$ si $n_1$ es un ascendiente de $n_2$.
- Un **camino** de $n_1$ a $n_2$ es una secuencia de arcos contiguos que van de $n_1$ a $n_2$.
- La **longitud** de un camino es el número de arcos que contiene (en otras palabras el número de nodos – 1).
- El **nivel** de un nodo es la longitud del camino que lo conecta al raíz.
- La **profundidad** o **altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja.

- Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es a su vez un árbol.
- Sea S un subárbol de un árbol **A:** si para cada nodo *n* de **SA, SA** contiene también todos los descendientes de *n* en **A. SA** se llama un **subárbol completo** de **A.**
- Un árbol está **equilibrado** cuando, dado un número máximo *K* de hijos de cada nodo y la **altura del árbol** *h,* cada nodo de nivel $k < h-1$ tiene exactamente *K* hijos. El árbol está equilibrado perfectamente entre cada nodo de nivel $l<h$ tiene exactamente *K* hijos.




## 16.3. ÁRBOLES BINARIOS

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



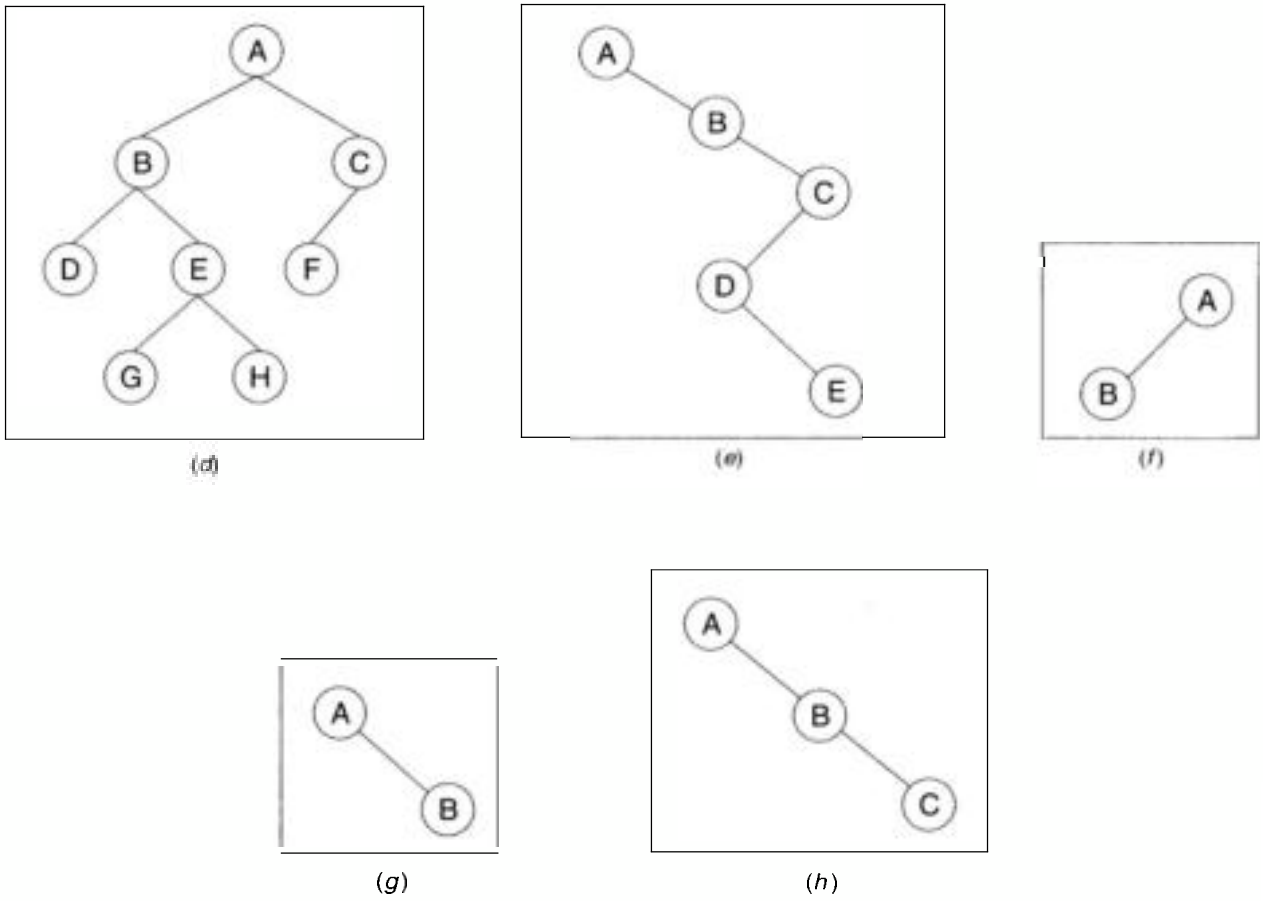



**Figura 16.10.** Árboles binarios.

## *Nota*

Un árbol binario no puede tener más de dos subárboles.

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

| | |
|---|---|
| {R} | *Nodo raíz* |
| {$I_1$, $I_2$, ...$I_n$} | *Subárbol izquierdo de R* |
| {$D_1$, $D_2$, ...$D_n$} | *Subárbol derecho de R* |

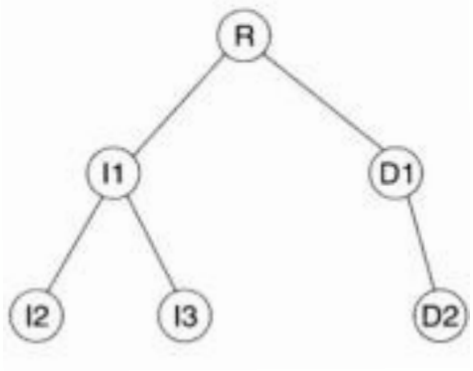



**Figura 16.11.** Árbol binario.

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

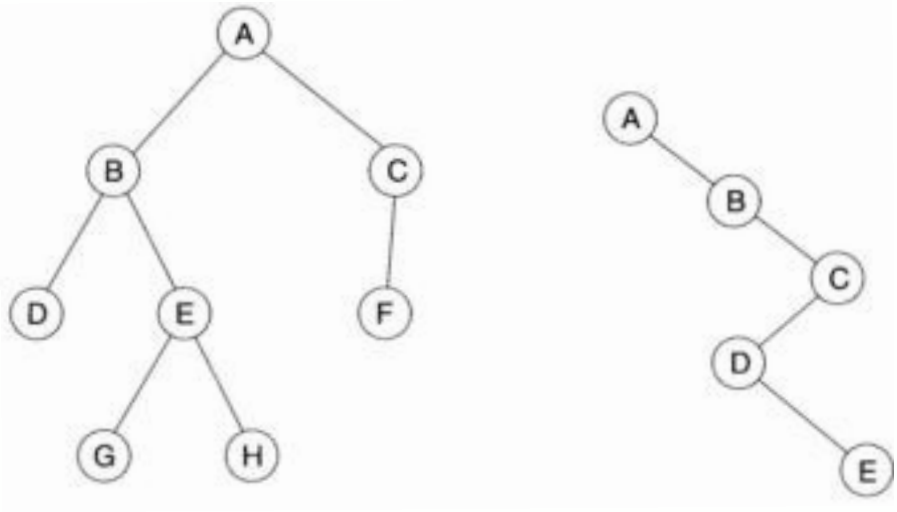



**Figura 16.12.** Árboles binarios: *(a)* profundidad **4;** *(b)* profundidad 5.

En la Figura **16.**I2 *(u)* el árbol A contiene 8 nodos en una profundidad de **4,** mientras que el árbol 16.I2 *(h)* contiene *5* nodos y una profundidad *5*. Este último caso es una forma especial, denominado **árbol degenerado,** en el que existe un solo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

### 16.3.1. Equilibrio

La distancia de un nodo al raíz determina la eficiencia con la que puede ser localizado. Por ejemplo, dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo sólo un camino de bifurcación

o de ramas, el que conduce al nodo deseado. De modo similar, los nodos a nivel 2 de un árbol sólo pueden ser accedidos siguiendo sólo dos ramas del árbol.

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

Utilizando esta fórmula el equilibrio del nodo raíz los ocho árboles de la Figura 16.10 son *(u)* 0 *(h)* 0, *(c)* 0 por definición, (*d*) –1, ***(e)*** **4,** (*f*) –1, (*g*) 1, (*h*) 2.

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también perfectamente equilibrados. Dado que esta definición ocurre raramente se aplica una definición alternativa. Un árbol binario está equilibrado si la altura de sus subárboles difiere en no más de uno (su factor de equilibrio es –1, 0, +1) y sus subárboles son también equilibrados.

## 16.3.2. Árboles binarios completos

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

Un árbol binario completo que contiene 2" nodos a nivel *n* es un **árbol lleno.** Un árbol lleno es un árbol binario que tiene el máximo número de entradas para su altura. Esto sucede cuando el Último nivel está lleno. La Figura 16.**I3** muestra un árbol binario completo; el árbol de la Figura 16.14 (*b*) se corresponde con uno lleno.

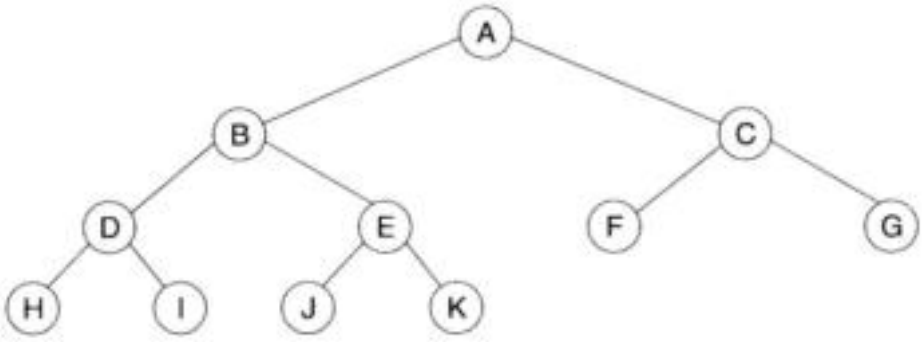
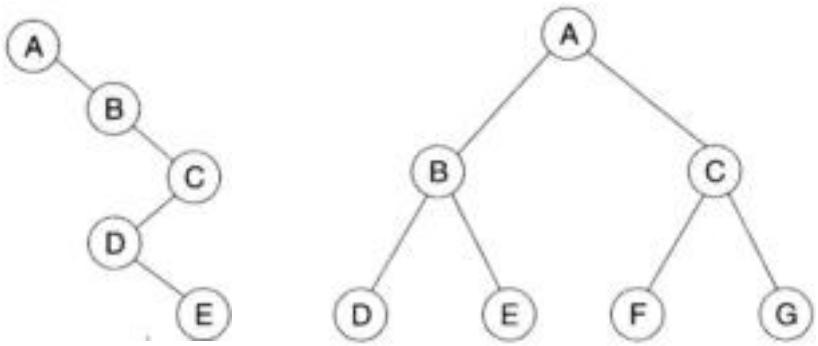



**Figura 16.13.** Árbol completo (profundidad **4).**




Árbol degenerado (profundidad 5) (*a*)

Árbol lleno (profundidad 3) (*b*)

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

El Último caso de árbol es un tipo especial denominado **árbol degenerado** en el que hay un solo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada. En la Figura 16.15 se muestran árboles llenos y completos.

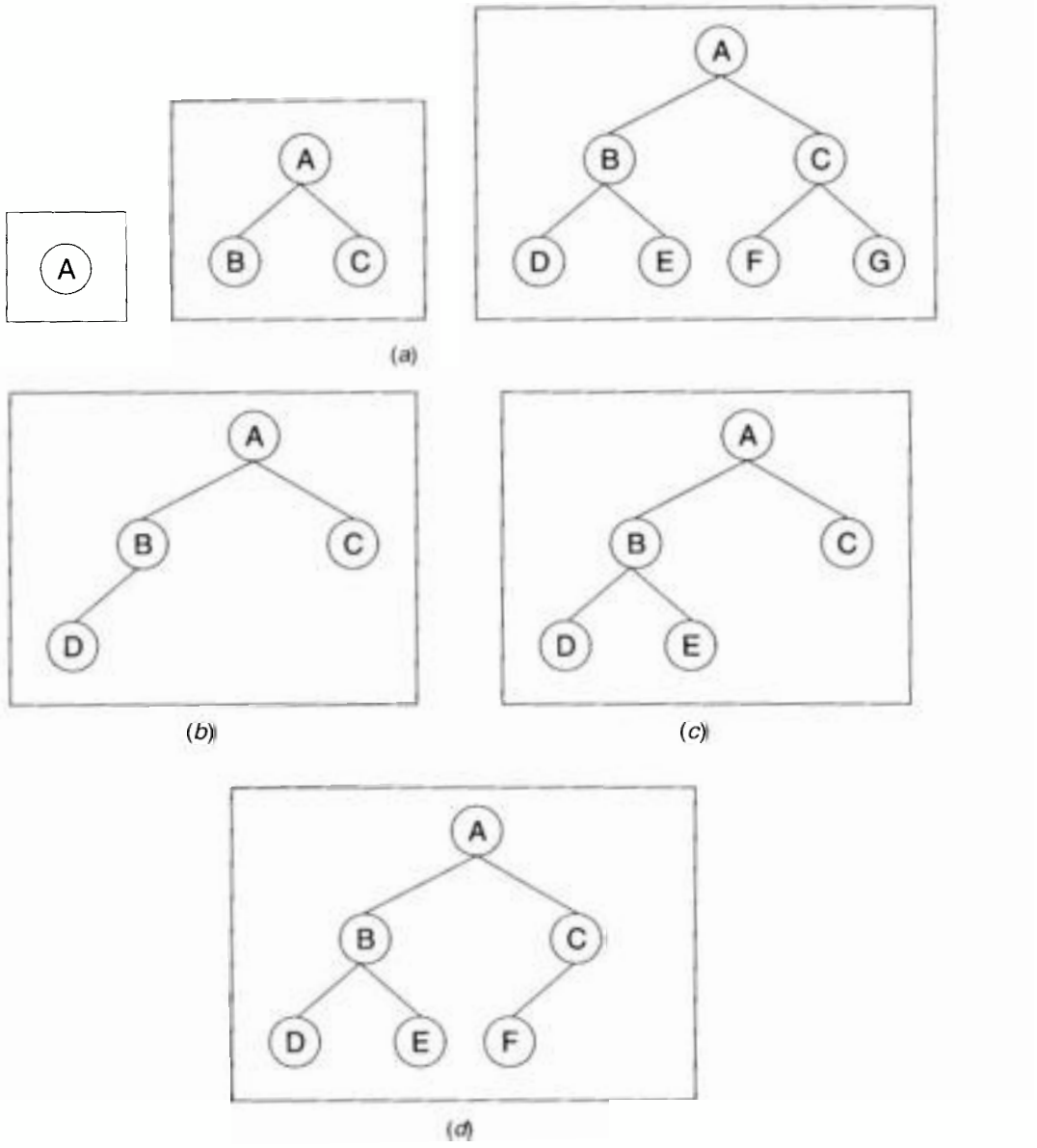



**Figura 16.15.** (a) Árboles llenos (en niveles 0, 1 y 2); (b),(c) y (d) árboles completos (en nivel 2).

Los árboles binarios y llenos de profundidad $k+1$ proporcionan algunos datos matemáticos que es necesario comentar. En cada caso, existe un nodo $(2^n)$ al nivel 0 (raíz), dos nodos $(2^1)$ a nivel 1, cuatro nodos $(2^1)$ a nivel 2, etc. A través de los primeros $k-1$ niveles hay $2^1-1$ nodos.

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

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

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

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

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

Aplicando logaritmos a la ecuación con desigualdad anterior

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

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

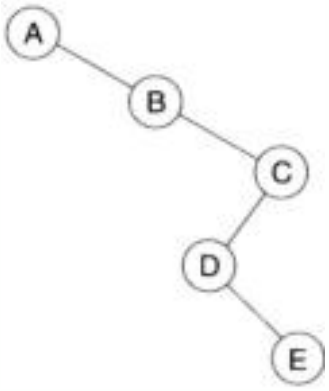
$h = /log, n + 1/$ *(parte entera de* $log_2 n + 1$*)*

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

### Ejemplo 16.2

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

La profundidad máxima de un árbol con *5* nodos es *5*




La profundidad mínima $n$ (número de niveles más uno) de un árbol con *5* nodos es

$k \leq log_2 (5) < k + 1$
$log_2 (5) = 2.32$ **y** *la profundidad* $n = 3$

### Ejemplo 16.3

*La profundidad de un árbol degenerado con n nodos es n, dudo que es la longitud del camino más largo (raíz a nodo)* **más** *1.*

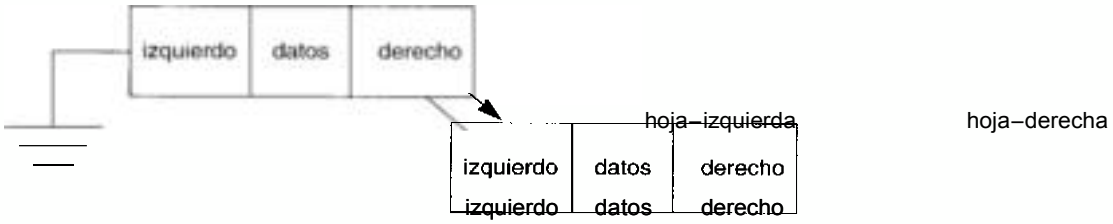
En el árbol binario completo con $n$ nodos, la profundidad del árbol es el valor entero de *log,* n + 1, que es **a** su vez la distancia del camino más largo desde el raíz a un nodo más uno.

Suponiendo que el árbol tiene $n = 10.000$ elementos, el camino **más** largo es

```
int (log 10000) + 1 = int (13.28) + 1 = 14
```

## 16.4. ESTRUCTURA DE UN ÁRBOL BINARIO

La estructura de un árbol binario se construye con nodos. Cada nodo debe contener el campo dato (datos a almacenar) y dos campos punteros, uno ai subárbol izquierdo y otro al subárbol derecho, que se conocen como **puntero izquierdo (izquierdo, izdo)** y **puntero derecho (derecho, dcho)** respectivamente. Un valor NULL indica un árbol vacío.




El algoritmo correspondiente a la estructura de un árbol es el siguiente:

```
Nodo
  subarbolIzquierdo          < puntero a Nodo>
  datos                      < Tipodato >
  subarbolDerecho            < puntero a Nodo>
Fin Nodo
```

La Figura 16.16 muestra un árbol binario y su estructura en nodos:

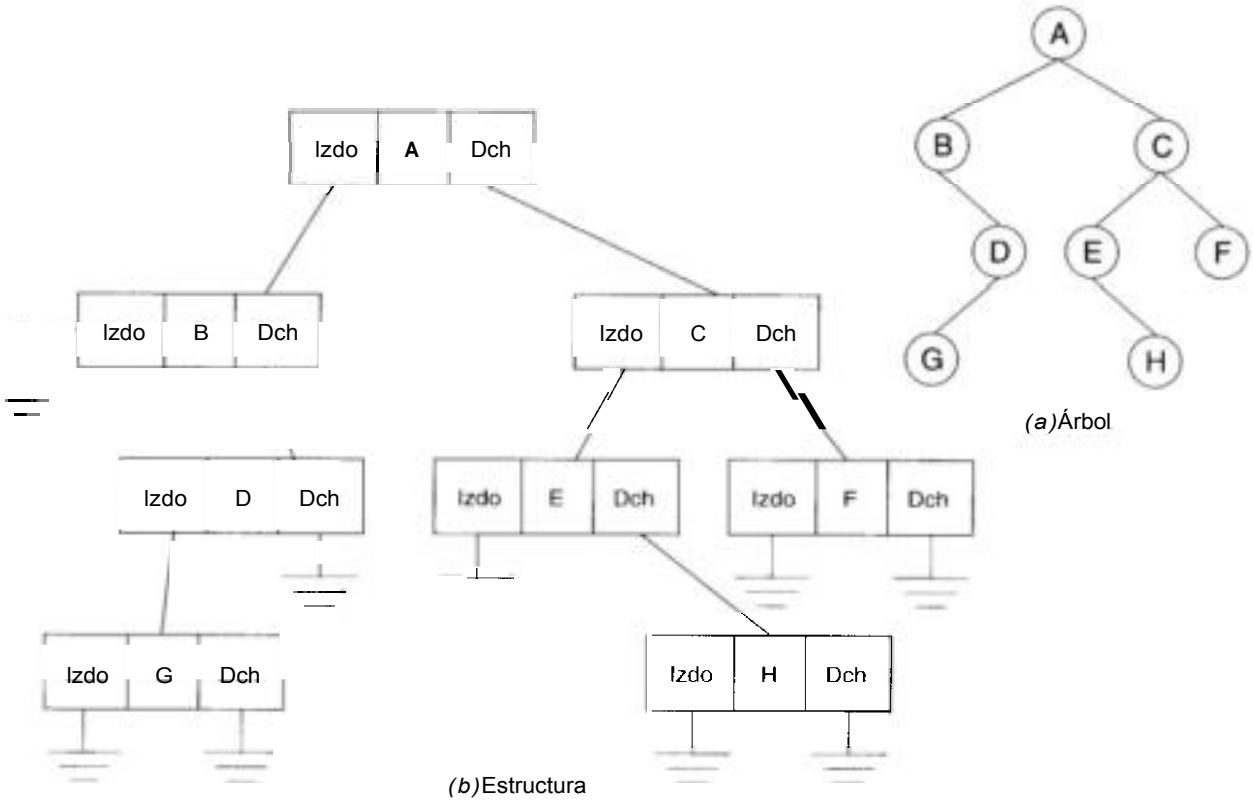
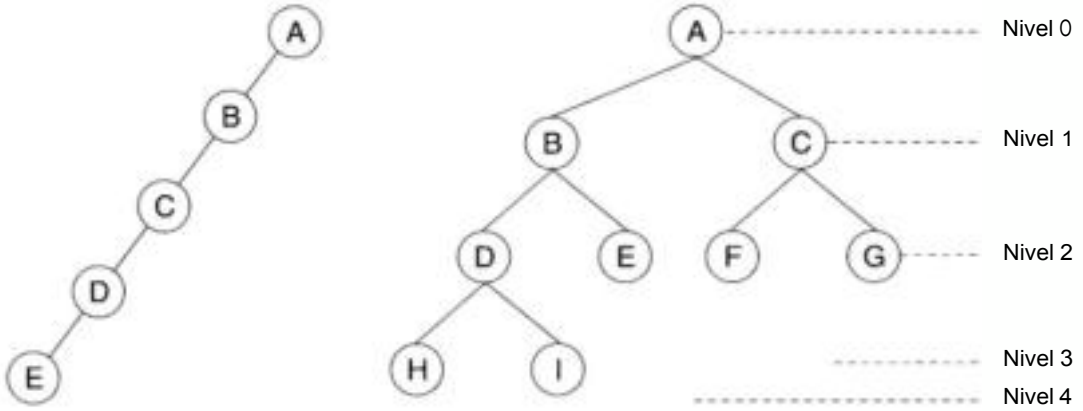


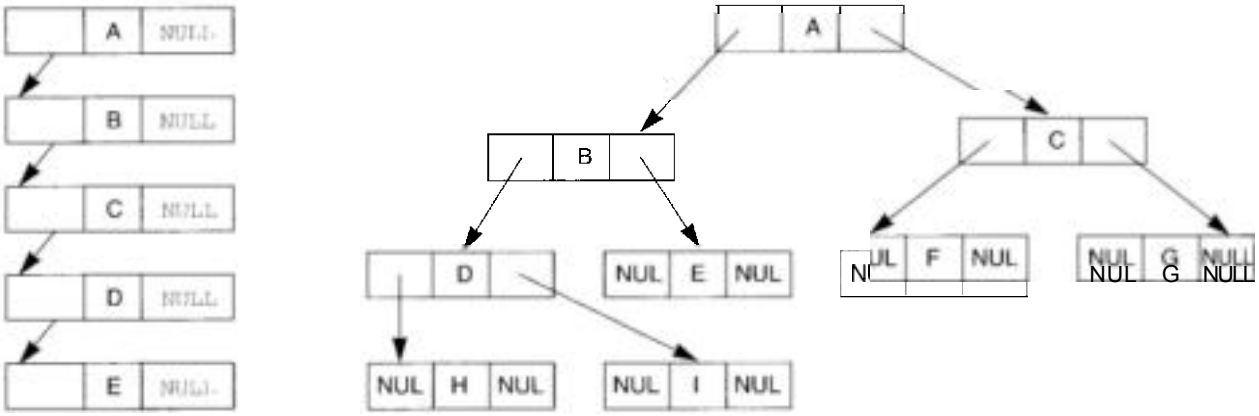

**Figura 16.16.** Árbol binario **y su** estructura en nodos

**Ejemplo 16.4**

Representar la estructura en nodos de los dos árboles binarios de raíz **A:**




La representación enlazada de estos dos árboles binarios es:

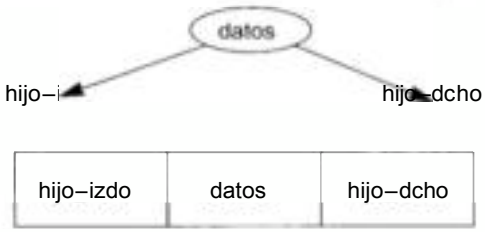



## 16.4.1. Diferentes tipos de representaciones en C

Los nodos pueden ser representados con la estructura `struct`. Suponiendo que el nodo tiene los campo `Datos, Izquierdo y Derecho`.

*Representación 1*

```
typedef struct nodo "puntero-arbol;
typedef struct nodo {
  int datos;
  puntero-arbol hijo-izdo, hijo-dcho;
};
```




*Representación 2*

```
typedef int TipoElemento; /* Puede ser cualquier tipo */

struct Nodo {
  TipoElemento Info;
  struct Nodo *hijo_izdo, *hijo_dcho;
};

typedef struct Nodo ElementoDeArbolBin;
typedef ElementoDeArbolBin *ArbolBinario;
```

Para crear un nodo de un árbol binario, con la representación 2, se reserva memoria para el nodo, se asigna el dato al campo `info` y se inicializa los punteros `hijo_izdo`, `hijo_dcho` a `NULL`.

```
ArbolBinario CrearNodo(TipoElemento x)
{
  ArbolBinario a;
  a = (ArbolBinario) malloc(sizeof(ElementoDeArbolBin ));
  a -> Info = x;
  a-> hijo-dcho = a -> hijo-izdo = NULL;
  return a;
}
```

Si por ejemplo se desea crear un árbol binario de raíz 9, rama izquierda 7 y rama derecha 11:

```
ArbolBinario raiz;
raiz = CrearNodo(9);
raiz -> hijo-izdo = CrearNodo(7);
raiz -> hijo-dcho = CrearNodo(11);
```

## 16.5. OPERACIONES EN ÁRBOLES BINARIOS

Una vez que se tiene creado un árbol binario, se pueden realizar diversas operaciones sobre él. El hacer uso de una operación u otra dependerá de la aplicación que se le quiera dar al árbol. Algunas de las operaciones típicas que se realizan en árboles binarios son:

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

Todas estas operaciones se pueden realizar recorriendo el árbol binario de un modo sistemático. El

[2] En el apartado siguiente se estudia el importante concepto de *árbol de expresión*.

recorrido de un árbol es la operación de visita al árbol, o lo que es lo mismo, la visita a cada nodo del árbol una vez y sólo una. La visita de un árbol es necesaria en muchas ocasiones, por ejemplo, si se desea imprimir la información contenida en cada nodo. Existen diferentes formas de visitar o recorrer un árbol que se estudiarán más tarde.

## 16.6. ÁRBOLES DE EXPRESIÓN

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

La Figura 16.17 representa la expresión infija `a*(b+c)td` y su árbol de expresión. En una primera

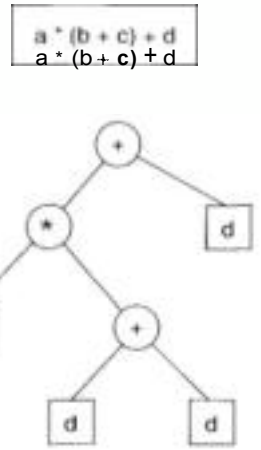



**Figura 16.17.** Una expresión infija y su árbol de expresión.

observación vemos que los paréntesis no aparecen en el árbol.

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

1. Cada hoja es un operando.
2. Los nodos raíz e internos son operadores.
3. Los subárboles son subexpresiones en las que el nodo raíz es un operador.

Los árboles binarios se utilizan para representar expresiones en memoria; esencialmente, en compiladores de lenguaje de programación. La Figura 16.18 muestra un árbol binario de expresiones para la expresión aritmética `(a + b)* c`.

Obsérvese que los paréntesis no se almacenan en el árbol pero están implicados en la forma del

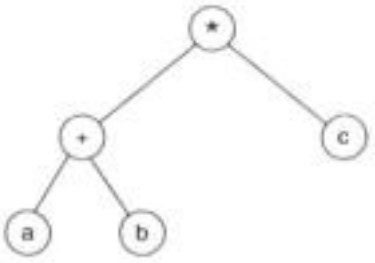



**Figura 16.18.** Árbol binario de expresiones que representa `(a+b) *c`.

árbol. Si se supone que todos los operadores tienen dos operandos, se puede representar una expresión por un árbol binario cuya raíz contiene un operador y cuyos subárboles izquierdo y derecho son los operandos izquierdo y derecho respectivamente. Cada operando puede ser una letra (`x`, `y`, `a`, `b`, etc.) o una subexpresión representada como un subárbol. En la Figura 16.19 se puede ver como el operador que está en la raíz es `*`, su subárbol izquierdo representa la subexpresión `(x + y)` y su subárbol derecho representa la subexpresión `(a-b)`. El nodo raíz del subárbol izquierdo contiene el operador **(+)** de la subexpresión izquierda y el nodo raíz del subárbol derecho contiene el operador **(–)** de la subexpresión derecha. Todos los operandos letras se almacenan en nodos hojas.

Utilizando el razonamiento anterior, se puede escribir la expresión almacenada en la Figura 16.20 como

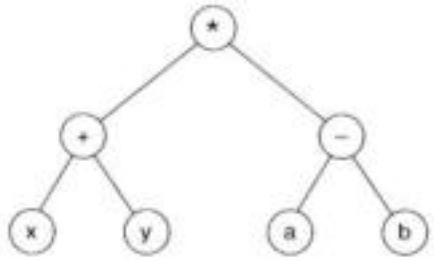



**Figura 16.19.** Árbol de expresión `(x+y)*(a-b)`

```
(x*(y-z))+(a-b)
```

en donde se han insertado paréntesis alrededor de subexpresiones del árbol (la operación $y - Z$, subexpresión inás interna, tiene el nivel de prioridad mayor).

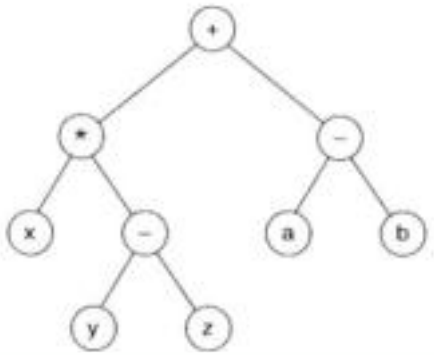
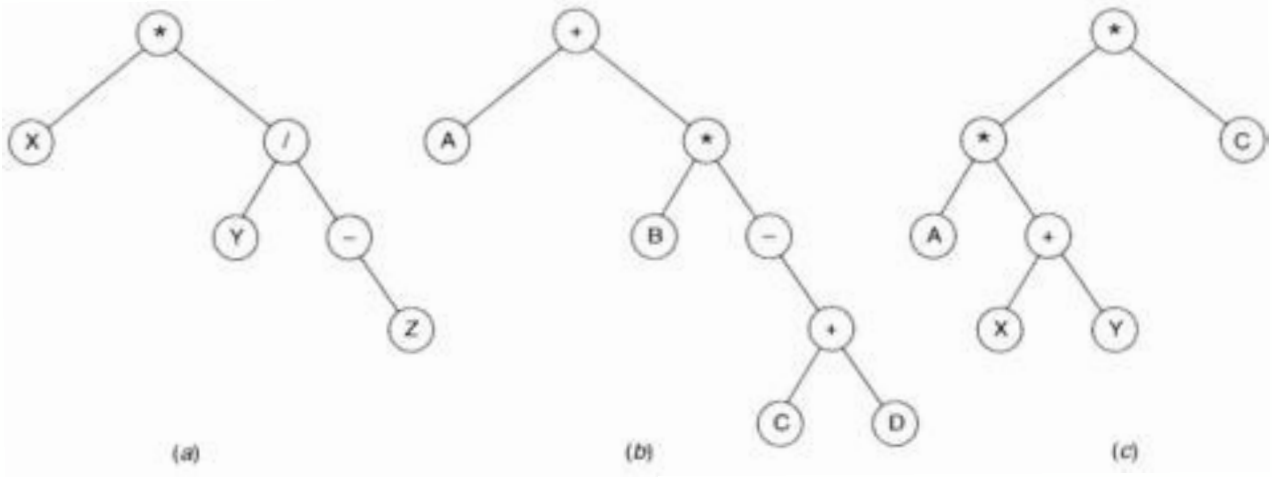



**Figura 16.20.** Árbol de expresión `(x*(y-z))+(a-b)`.

### Ejemplo 16.5

*Deducir las expresiones que representan los siguientes árboles binarios.*




***Soluciones***

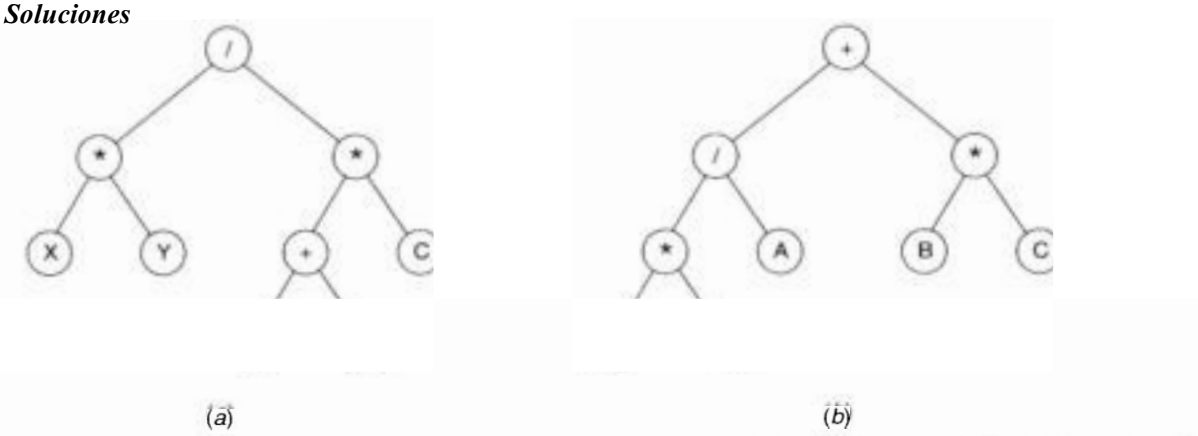
(a) `X*(Y/-Z)`
(b) `A +(B*-(C+D))`
(c) `(A*(X+Y))*C`

### Ejemplo 16.6

*Dibujar la representación en árbol binario de cada una de las siguientes expresiones.*

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

***Soluciones***




## 16.6.1. Reglas para la construcción de árboles de expresión

Los árboles de expresiones se utilizan en las coinputadoras para evaluar expresiones usadas en programas. El algoritmo más sencillo para construir un árbol de expresión es uno que lee una expresión completa que contiene paréiitesis en la misma. Una expresión con paréntesis es aquella en que

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

Por consiguiente `(4+(5*6))` es un ejemplo de una expresión completa entre paréntesis. Su valor es 34. Si se desean cambiar las prioridades, se escribe `((4+5)*6)`, su valor es 54. A fin de ver la prioridad en las expresiones, considérese la expresión

```
(4*5) + 6/7 - (8+9)
```

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

```
(4*5) + (6/7) - (8+9)
```

El orden de los operadores aquí es + y –. Por consiguiente, se puede escribir

```
((4*5) + (6/7)) - (8+9)
```

Por último la expresión completa entre paréntesis será

```
(((4*5) + (6/7)) - (8+9))
```

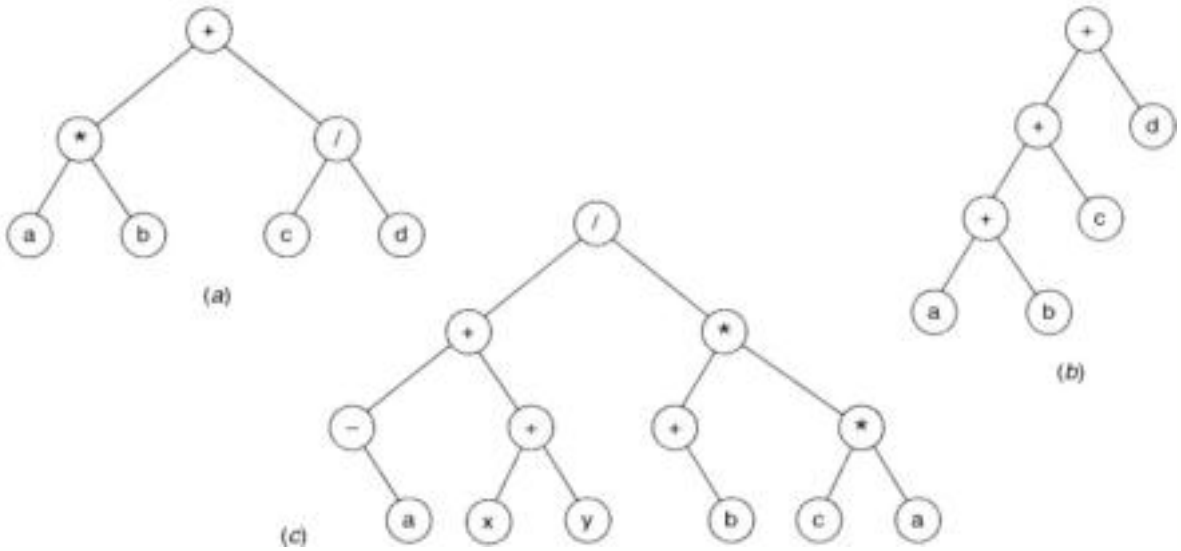
El algoritmo para la construcción de un árbol de expresión es:

1. La primera vez que se encuentra un paréntesis a izquierda, crea un nodo y lo hace en el raíz. Se llama a éste, el *nodo actual* y se sitúa su puntero en una pila.
2. Cada vez que se encuentre un nuevo paréntesis a izquierda, crear un nuevo nodo. Si el nodo actual no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho. Hacer el nuevo nodo el nodo actual y situar su puntero en una pila.
3. Cuando se encuentra un operando, crear un nuevo nodo y asignar el operando a su campo de datos. Si el nodo actual no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho.
4. Cuando se encuentra un operador, sacar un puntero de la pila y situar el operador en el campo datos del nodo del puntero.
5. Ignorar paréntesis derecho y blancos.

---

### Ejemplo 16.7

*Calcular las expresiones correspondientes de* ***los*** *árboles de expresión.*



Las soluciones correspondientes son:

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

**Ejercicio 16.1** (a realizar por el lector)

***Dibujar** los **árboles binarios de expresión** correspondiente **a cada una de las siguientes expresiones:***

*(a)* `(a + b) / (c - d * e) + e + 9 * h/a`
*(b)* `-x -y * z + (a + b + c / d * e)`
*(c)* `((a + b) > (c - e)) || a < f && (x < y || y > z)`

## 16.7. RECORRIDO DE UN ÁRBOL

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

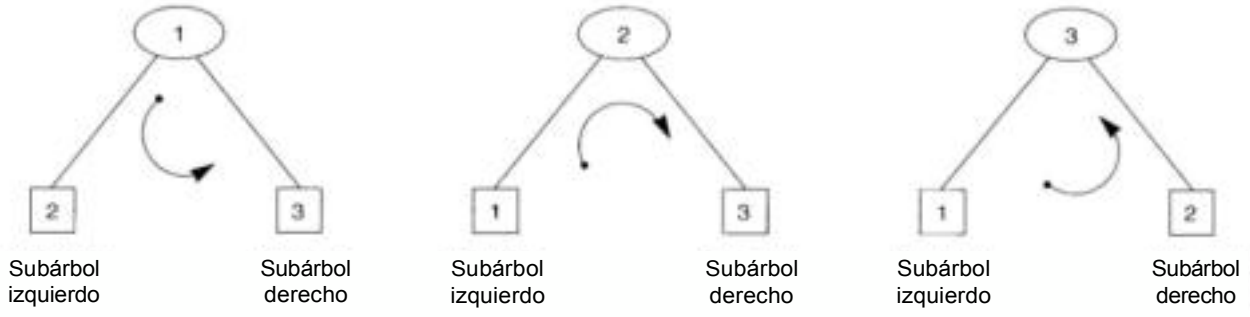
Un **recorrido de un árbol binario** requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, profundidad y anchura.

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

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

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

Dado un árbol binario que consta de un raíz, un subárbol izquierdo y un subárbol derecho se pueden definir tres tipos de secuencia de recorrido en profundidad. Estos recorridos estándar se muestran en la Figura 16.21.




*(a) Recorrido preorden* *(b) Recorrido en orden* *(c) Recorrido postorden*

**Figura 16.21.** Recorridos de árboles binarios

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

Según sea la estrategia a seguir, los recorridos se conocen como **enorden** ***(inorder),*** **preorden** (*preorder*) y **postorden** ***(postorder)***

| | |
|---|---|
| **Preorden** | (nodo-izquierdo-derecho) **(NID)** |
| **Enorden** | (izquierdo-nodo-derecho) **(IND)** |
| **Postorden** | (izquierdo-derecho-nodo) **(IDN)** |

## 16.7.1. Recorrido *preorden*

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

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

Dado las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero, se procesa la raíz, a continuación el subárbol izquierdo y a continuación el subárbol derecho. Para procesar el subárbol izquierdo, se hace una llamada recursiva al procedimiento ***preorden*** y luego se hace lo mismo con el subárbol derecho. El algoritmo recursivo correspondiente para un árbol T es:

```
si T no es vacio entonces
  inicio
    ver los datos en el raiz de T
    Preorden (subarbol izquierdo del raiz de T)
    Preorden (subarbol derecho del raiz de T)
  fin
```

### *Regla*

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

Si utilizamos el recorrido preorden del árbol de la Figura 16.22 se visita primero el raíz (nodo A). A continuación se visita el subárbol izquierdo de A, que consta de los nodos B, D y E. Dado que el subárbol es a su vez un árbol, se visitan los nodos utilizando el orden NID. Por consiguiente, se visita primero el nodo B, después D (izquierdo) y, por último, E (derecho).

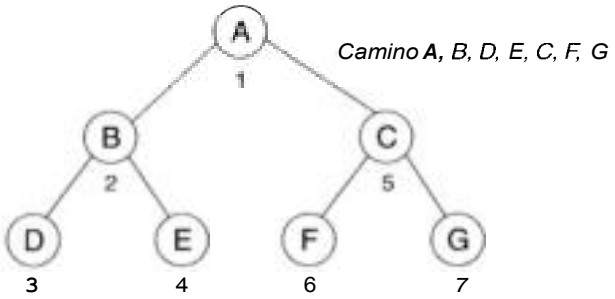



**Figura 16.22.** Recorrido preorden de un árbol binario.

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

A continuación se visita el subárbol derecho de A, que es un árbol que contiene los nodos C, F y G. De nuevo siguiendo el orden NID, se visita primero el nodo C, a continuación F (izquierdo) y, por Último, G (derecho). En consecuencia el orden del recorrido preorden para el árbol de la Figura 16.22 es A-B-D-E-C-F-G.

Un refinamiento del algoritmo es:

```
algoritmo preOrden  (val raiz <puntero nodos>)
Recorrer un arbol binario en secuencia nodo-izdo-dcho
Pre raiz es el nodo de entrada del árbol o subárbol
Post cada nodo se procesa en orden
1 si (raiz no es nulo)
  1  procesar (raiz)
  2  preOrden (raiz -> subarbolIzdo)
  3  preOrden (raiz -> subarbolDcho
2  return
```

La función preorden muestra el código fuente en C del algoritmo ya citado anteriormente. El tipo de los datos es entero.

```
typedef int TipoElemento;
struct nodo {
  TipoElemento datos;
  struct nodo *hijo_izdo, *hijo_dcho;
};
typedef struct nodo Nodo;

void preorden (Nodo *p)
{
  if (p)
  {
    printf("%d ",p -> datos);
    PreOrden(p -> hijo-izdo);
    PreOrden(p -> hijo-dcho);
  }
}
```

### *Gráficas de las llamadas recursivas de preorden*

El recorrido recursivo de un árbol se puede mostrar gráficamente por dos métodos distintos: 1) *paseo* ***preorden*** del árbol; 2) recorrido algorítmico.

Un medio gráfico para visualizar el recorrido de un árbol es imaginar que se está dando un «*paseo*» alrededor del árbol comenzando por la raíz y siguiendo el sentido contrario a las agujas del reloj, un nodo a continuación de otro sin pasar dos veces por el mismo nodo. El camino señalado por una línea continua que comienza en el nodo 1 (Fig. 16.21) muestra el recorrido preorden completo. En el caso de la Figura 16.22 el recorrido es A B D E C F G.

El otro medio gráfico de mostrar el recorrido algorítmico recursivo es similar a las diferentes etapas del algoritmo. Así la primera llamada procesa la raíz del árbol A. A continuación se llama recursivamente a procesar subárbol izquierdo, procesa el nodo B. La tercera llamada procesa el nodo D, que es un subárbol formado por un único nodo. En ese punto, se llama en preorden, con un puntero nulo, que produce un retorno inmediato al subárbol D para procesar a su subárbol derecho. Debido a que el subárbol derecho de D es también nulo, se vuelve al nodo B de modo que va a procesar (visitar) su subárbol derecho, E. Después de procesar el nodo E, se hacen dos llamadas más, una con el puntero izquierdo *null* de E y otra con su puntero derecho *null.* Como el subárbol B ha sido totalmente procesado, se vuelve a la raíz del árbol y se procesa su subárbol derecho, C. Después de procesar C, llama para procesar su subárbol izquierdo F. Se hacen dos llamadas con *null,* vuelve al nivel donde está el nodo C para procesar su rama derecha G. Aún se realizan dos llamadas más, una al subárbol izquierdo *null* y otra al subárbol derecho. Entonces se retorna en el árbol, se concluye el recorrido del árbol.

### 16.7.2. Recorrido *enorden*

El recorrido **en orden** ***(inorder)*** procesa primero el subárbol izquierdo, después el raíz y a continuación el subárbol derecho. El signiticado de ***in*** es que la raíz se procesa entre los subárboles. Si el árbol no está vacío, el método implica los siguientes pasos:

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

El algoritmo correspondiente es:

```
Enorden(A)

si el arbol no esta vacio entonces
  inicio
    Recorrer el subarbol izquierdo
    Visitar el nodo raiz
    Recorrer el subarbol derecho
  fin
```

Un refinamiento del algoritmo es:

```
algoritmo enOrden (val raiz <puntero a nodos>)
Recorrer un árbol binario en la secuencia izquierdo-nodo-derecho
  pre raíz en el nodo de entrada de un árbol o subárbol
  post cada nodo se ha de procesar en orden
1   si (raíz no es nulo)
    1 enorden (raiz -> subarbol Izquierdo)
    2 procesar (raiz)
    3 enOrden (raiz->subarbolDerecho)
2   retorno
fin enorden
```

En el árbol de la Figura 16.23, los nodos se han numerado en el orden en que son visitados durante el recorrido ***enorden.*** El primer subárbol recorrido es el subárbol izquierdo del nodo raíz (árbol cuyo nodo contiene la letra B. Este subárbol consta de los nodos B, D y E y es a su vez otro árbol con el nodo B como raíz, por lo que siguiendo el orden IND, se visita primero D, a continuación B (nodo raíz) y, por último, E (derecha). Después de la visita a este subárbol izquierdo se visita el nodo raíz A y, por Último, se visita el subárbol derecho que consta de los nodos C, F y G. **A** continuación, siguiendo el orden IND para el subárbol derecho, se visita primero F, después C (nodo raíz) **y,** por Último, G. Por consiguiente, el orden del recorrido inorden de la Figura 16.23 es D-B-E-A-F-C-G.

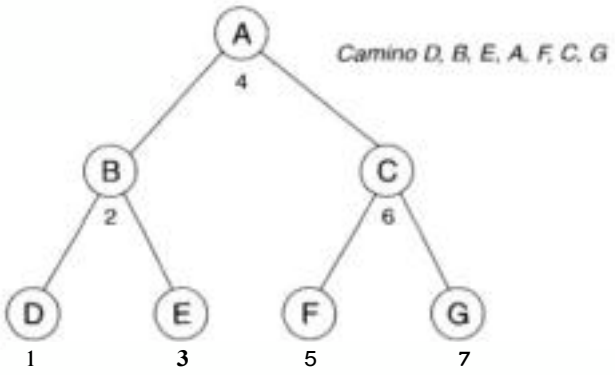



**Figura 16.23.** Recorrido *enorden* de un árbol binario.

La siguiente función visita y escribe el contenido de los nodos de un árbol binario de acuerdo al recorrido *EnOrden*. La función tiene como parámetro un puntero al nodo raíz del árbol.

```
void enorden (Nodo *p)
{
  if (p)
  {
    enorden(p -> hijo-izqdo);        /* recorrer subárbol izquierdo */
    printf("%d ",p -> datos);        /* visitar la raíz */
    enorden (p -> hijo-dcho);        /* recorrer subárbol derecho */
  }
}
```

## 16.7.3. Recorrido *postorden*

El recorrido postorden **(IDN)** procesa el nodo raíz (*post*) después de que los subárboles izquierdo y derecho se han procesado. Se comienza situándose en la hoja más a la izquierda y se procesa. A continuación se procesa su subárbol derecho. Por último se procesa el nodo raíz. Las etapas del algoritmo son:

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

El algoritmo recursivo para un árbol A es:

```
si A no esta vacio entonces
  inicio
    Postorden (subarbol izquierdo del raíz de A)
    Postorden (subarbol derecho del raíz de A)
    Visitar la raíz de A
  fin
```

El refinamiento del algoritmo es:

```
algoritmo postorden (val raiz <puntero a nodo>)
```

***Recorrer un árbol binario en secuencia izquierda-derecha-nodo***
pre *raíz es el nodo de entrada de un árbol a un subárbol*
post *cada nodo ha sido procesado en orden*

```
1 Si (raíz no es nulo)
    1postOrden (raíz -> SubarbolIzdo)
    2postOrden (raíz -> SubarbolDcho)
    3procesar (raiz)
2 retorno
fin postorden
```

Si se utiliza el recorrido postorden del árbol de la Figura 16.24, se visita primero el subárbol izquierdo A. Este subárbol consta de los nodos B, D y E y siguiendo el orden IDN, se visitará primero D (izquierdo), luego E (derecho) y, por Último, B (nodo). A continuación, se visita el subárbol derecho A que consta de los nodos C, F y G. Siguiendo el orden IDN para este árbol, se visita primero F (izquierdo), después G (derecho) y, por Último, C (nodo). Finalmente se visita el raíz A (nodo). Así el orden del recorrido postorden del árbol de la Figura 16.24 es D-E-B-F-G-C-A.

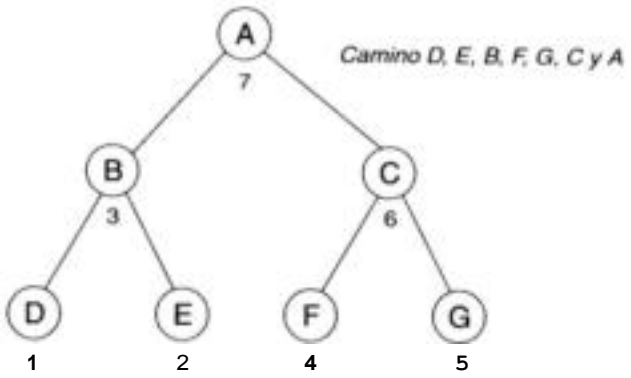



**Figura 16.24.** Recorrido ***postorden*** de un árbol binario.

**La** función `postorden` que implementa en C el código fuente del algoritmo correspondiente

```
void postorden  (Nodo *p)
{
  if (p)
  {
    postorden (p -> hijo-izqdo);
    postorden (p -> hijo-dcho);
    printf("%d ",p -> datos);
  }
}
```

### *Nota de programación modular*

**La visita al nodo raíz del árbol que se representa mediante una sentencia `printf()` podría representarse también con una función `visitar`**

```
void visitar (Nodo *p)
{
   printf("%d ",p -> datos);
}
```

**La función `postorden` quedaría así:**

```
void postorden (Nodo *p)
{
  if (p)
  {
    postorden (p -> hijo_izqdo);
    postorden (p -> hijo_dcho);
    visitar (p);
  }
}
```

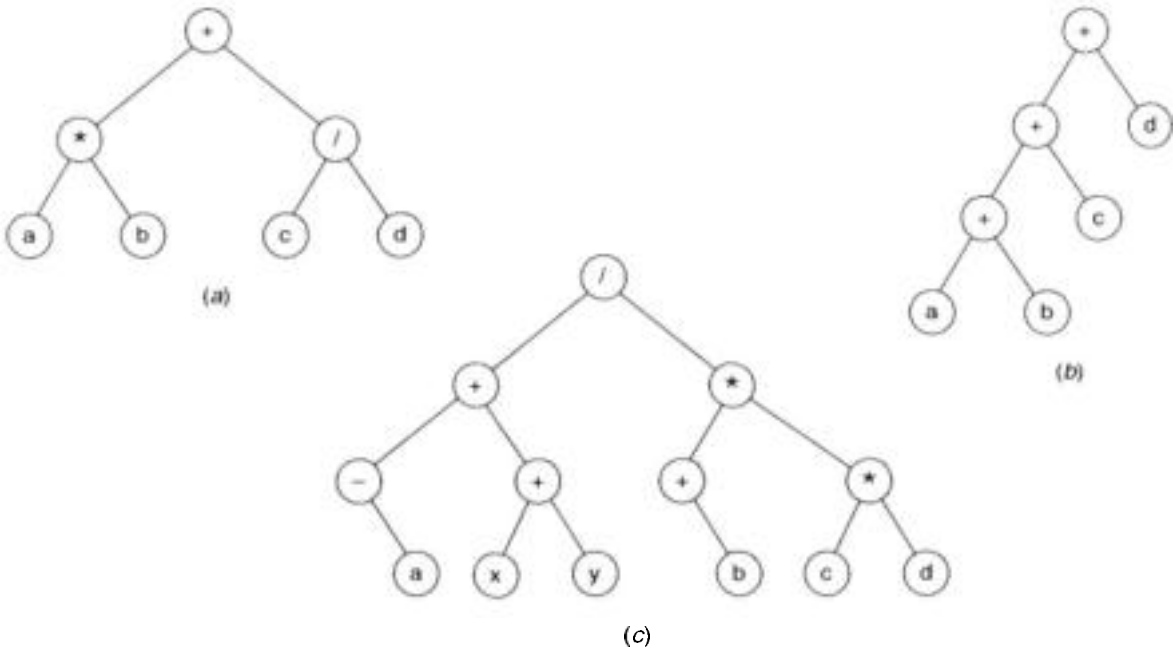



Figura 16.25. Árboles de expresión.

**Ejercicio 16.2**

***Si la función*** `visitar()` ***se reemplaza por la .sentencia.***

```
printf("%d ",t -> dato);
```

***deducir los elementos de los árboles binarios siguientes en cada uno de los tres recorridos fundamentales.***

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

| | *Árbola* | *Árbol b* | *Árbol c* |
|---|---|---|---|
| *PreOrden* | +*ab/cd | +++abcd | /+- a + xy * +b * cd |
| *EnOrden* | a*c+c/d | a+b+c+d | -a + x + y / + b * c * d |
| *PostOrden* | ab*cd/+ | ab+c+d+ | a - xy ++ b + cd ** / |

## 16.7.4. Profundidad de un árbol binario

La profundidad de un árbol binario es una característica que se necesita conocer con frecuencia durante el desarrollo de una aplicación con árboles. La función `Profundidad` evalúa la ***profundidad*** de un ***árbol*** binario. Para ello tiene un parámetro que es un puntero a la raíz del árbol.

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

El método de cálculo de la profundidad de los subárboles utiliza llamadas recursivas a la función `Profundidad` con punteros a los respectivos subárboles como parámetros de la misma. La fun-

ción `Profundidad`devuelve como resultado la profundidad del subárbol mas profundo más I (la misma del raíz).

```
int Profundidad (Nodo *p)
{
  if (!p)
    return 0 ;
  else
  {
    int profundidad1 = Profundidad (p -> hijo-izqdo);
    int profundidadD = Profundidad (p -> hijo-dcho);
    if (profundidad1 > profundidadD)
     return profundidad1 + 1;
    else
     return profundidadD + 1;
  }
}
```

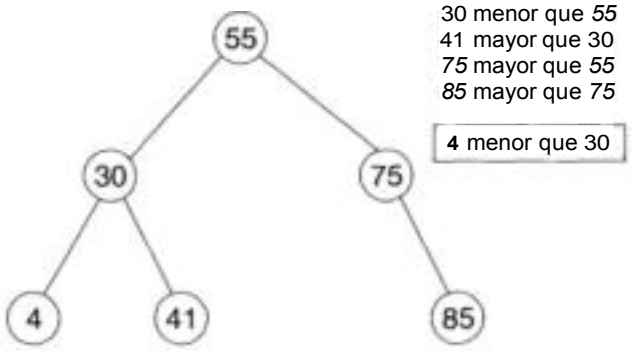
## 16.8. ÁRBOL BINARIO DE BÚSQUEDA

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

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

**Ejemplo 16.8**

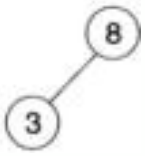
***Árbol binario de búsqueda.***



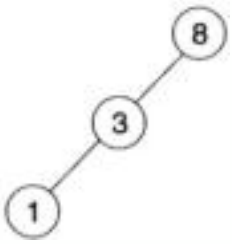

### 16.8.1. Creación de un árbol binario de búsqueda

Supongamos que se desea almacenar los números **8** 3 1 20 10 5 4 en un árbol binario de búsqueda. Siguiendo la regla, dado un nodo en el árbol todos los datos a su izquierda deben ser menores que todos los datos del nodo actual, mientras que todos los datos a la derecha deben ser mayores que los datos. Inicialmente el árbol está vacío y se desea insertar el 8. La única elección es almacenar el 8 en el raíz:

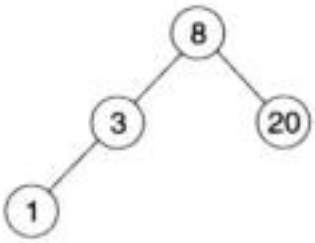
A continuación viene el 3. Ya que 3 es menor que 8, el 3 debe ir en el subárbol izquierdo.



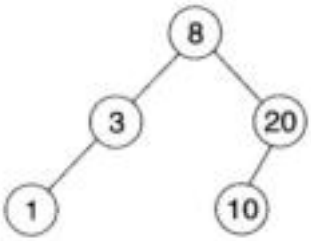

A continuación se ha de insertar 1 que es menor que 8 y que 3, por consiguiente irá a la izquierda y debajo de 3.

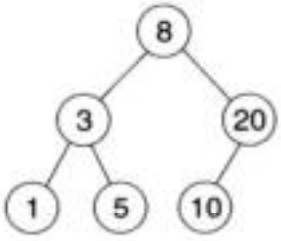


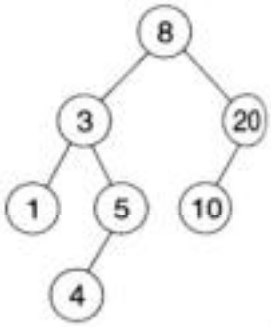

El siguiente número es 20, mayor que 8, lo que implica debe ir a la derecha de 8.




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



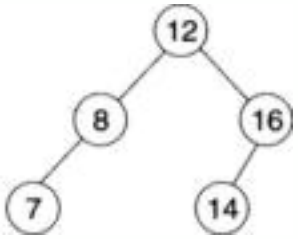




Una propiedad de los árboles binarios de búsqueda es que no son únicos para los mismos datos.

### Ejemplo 16.9

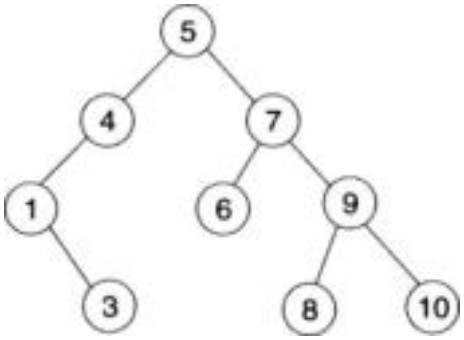
***Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 14.***

***Solución***




### Ejemplo 16.10

***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.***

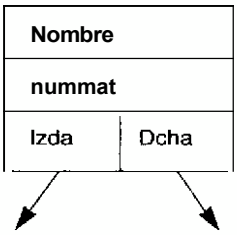



## 16.8.2. Implementación de un nodo de un árbol binario de búsqueda

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

Declaración de tipos

| | |
|---|---|
| Nombre | Tipo de dato cadena (`string`) |
| Matrícula | Tipo entero |




```
struct nodo {
   int nummat;
   char nombre[30];
   struct nodo *izda, *dcha;
};
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, char* n)

  Nodo* t;
  t = (Nodo*)malloc(sizeof (Nodo));
  t -> nummat = id;
  strcpy(t->nombre,n);
  t ->izda = t -> dcha = NULL;
  return t;
I
```

## 16.9. 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 tenemos la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo.
- *Inserción* de un nodo.
- *Kecorriúo* de un árbol.
- *Borrado* de un nodo.

### 16.9.1. 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.

***Buscar una información específica***

Si se desea encontrar un nodo en el árbol que contenga la información sobre una persona específica. La función `buscar` tiene dos parámetros, un puntero al árbol y un número de matrícula para la persona requerida. Como resultado, la función devuelve un puntero al nodo en el que se almacena la información sobre esa persona; en el caso de que la información sobre la persona no se encuentra 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 la persona, 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 el número de matrícula requerido es más pequeño o mayor que el número de matrícula 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 paráinetro.

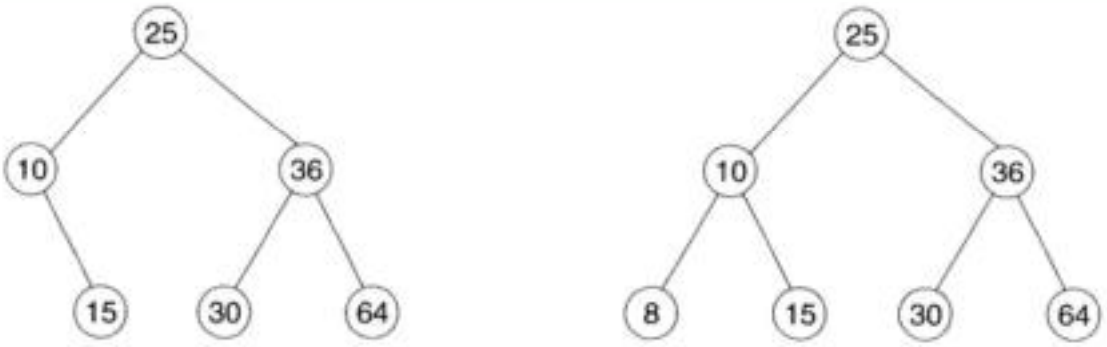
El código C de la función `buscar`. es:

```
Nodo* buscar (Nodo*p, int buscado)
{
```

```
  if (!p)
    return 0;
  else if (buscado == p -> nummat)
    return p;
  else if (buscado < p -> nummat)
    return buscar (p -> izda, buscado);
  else
    return buscar (p -> dcha, buscado);
}
```

## 16.9.2. 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).



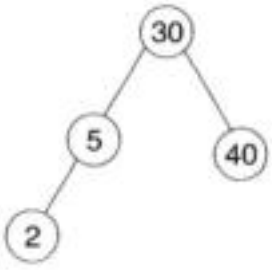

*Antes de insertar 8* *Después de insertar 8*

**Figura 16.26.** Inserción en un árbol binario de búsqueda.

Por ejemplo, considérese el caso de añadir el nodo 8 al árbol de la Figura 16.26. 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.

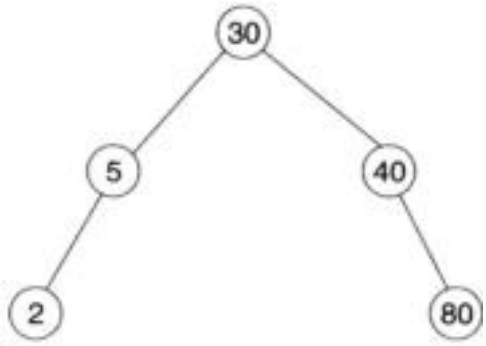
### Ejemplo 16.11

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

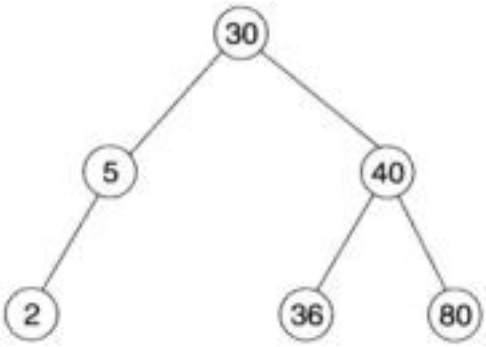


A continuación insertar un elemento con clave **36** en el árbol binario de búsqueda resultante.

**Solución**




*(a)* Inserción de 80




*(a)* Inserción de 36

### 16.9.3. Función `insertar()`

La función `insertar`que pone nuevos nodos es sencilla. Se deben declarar tres argumentos: un puntero al raíz del árbol, el nuevo nombre y número de matrícula de la persona. La función creará un nuevo nodo para la nueva persona 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, int nuevomat, char  *nuevo-nombre)
{
  if (!(*raiz))
    *raiz = CrearNodo(nuevo_mat, nuevo-nombre);
  else if (nuevomat < (*raiz) -> nummat)
    insertar (&((*raiz) -> izda), nuevomat, nuevo-nombre);
  else
    insertar (&((*raiz) -> dcha), nuevomat, nuevo-nombre);
}
```

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 número de matrícula de la nueva persona sea más pequeño o mayor que el número de matrícula en la raíz del árbol.

### 16.9.4. Eliminación

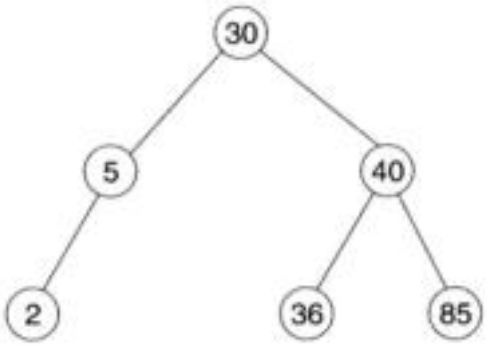
La operación de ***eliminación*** de un nodo es también una extensión de la operación de búsqueda, si bien **más** compleja que la inserción debido a que el nodo a suprimir puede ser cualquiera **y** la operación de supresión debe mantener la estructura de árbol binario de búsqueda después de la eliminación de datos. 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 clave (inmediatamente superior o inmediatamente inferior) con objeto de mantener la estructura de árbol binario.

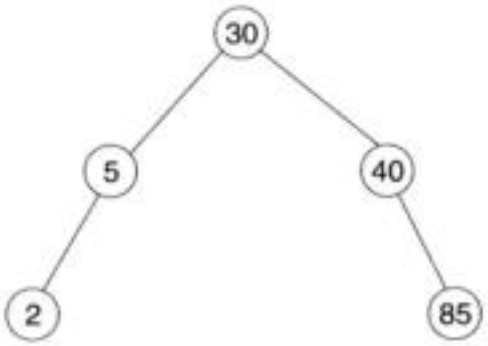
---

**Ejemplo 16.12**

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

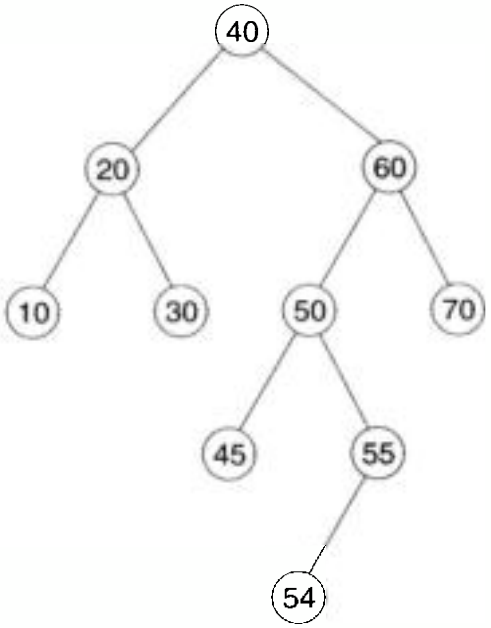



El árbol resultante es:

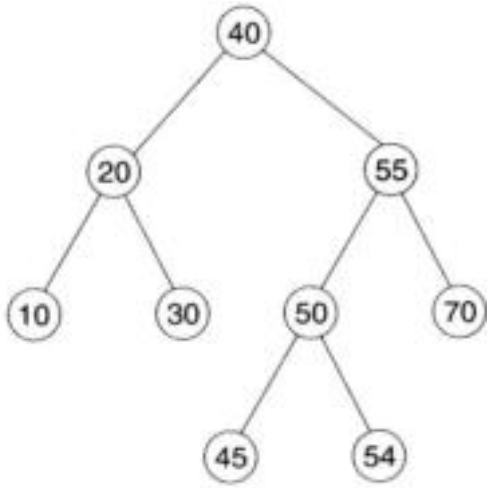


### Ejemplo 16.13

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




Se reemplaza 60 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 *5.5* al raíz del subárbol y se reajusta el árbol.




### Ejercicio 16.3

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

***Análisis***

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

Las operaciones que se van a implementar son las de `insertar`, `eliminar`, `buscar` y `visualizar` el árbol. Los algoritmos de las tres primeras operaciones ya están descritos anteriormente. La operación de `visualizar` va a consistir en un recorrido en `inorden`,cada vez que se visite el nodo raíz se escribe los datos del estudiante.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct nodo {
  int nummat;
  char nombre[30];
  struct nodo *izda, *dcha;
};
typedef struct nodo Nodo;
Nodo* CrearNodo(int id, char* n);
Nodo* buscar (Nodo* p, int buscado);
void insertar (Nodo** raiz, int nuevo-mat, char  *nuevo-nombre);
void eliminar (Nodo** r, int mat);
void visualizar (Nodo* r);

int main()
{
  int nm;
  char nom[30];
  Nodo* R = 0;

      /* Crea el árbol */
  do{
   printf ("Numero de matricula (0 -> Fin): "); scanf("%d%*c",&nm);
   if (nm)
   {
     printf ("Nombre: "); gets (nom);
     insertar(&R,nm,nom);
   I
  }while (nm);
   /* Opciones de escribir el  árbol o borrar una 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(R);
   I
   else if (nm == 2){
    int cl;
    printf ("Clave:"); scanf("%d",&cl);
    eliminar(&R,cl);
   }
  }while (nm != 3);

  return 1;
I

Nodo* CrearNodo(int id, char* n)
{
```

```
  Nodo* t;
  t = (Nodo") malloc(sizeof(Nodo));
  t -> nummat = id;
  strcpy(t->nombre,n);
  t -> izda = t-> dcha = NULL;
  return t;
}

Nodo* buscar (Nodo* p, int buscado)

  if (!p)
    return 0;
  else if (buscado == p -> nummat)
    return p;
  else if (buscado < p -> nummat)
    return buscar (p -> izda, buscado);
  else
    return buscar (p -> dcha, buscado);
}

void insertar (Nodo** raiz, int nuevomat, char  *nuevo—nombre)
{
  if (!(*raiz))
   *raiz = CrearNodo(nuevo_mat, nuevo—nombre);
  else if (nuevomat < (*raiz) -> nummat)
    insertar (&((*raiz) -> izda), nuevomat, nuevo—nombre);
  else
    insertar (&((*raiz) -> dcha), nuevo—mat, nuevo—nombre);
}

void visualizar (Nodo" r)
{
  if (r)
  {
   visualizar(r -> izda);
   printf("Matricula %d \t %s \n",r->nummat,r->nombre);
   visualizar(r -> dcha);
  }
}

void eliminar (Nodo** r, int mat)

  if (!(*r))
    printf("!! Registro con clave %d no se encuentra !!. \n",mat);
  else if (mat < (*r)->nummat)
    eliminar(&(*r)->izda, mat);
  else if (mat > (*r)->nummat)
    eliminar(&(*r)->dcha,mat);
  else /* Matricula encontrada */
  {
    Nodo* q; /* puntero al nodo a suprimir */
    q = (*r);
    if (q->izda == NULL)
     (*r) = q->dcha;
    else if (q->dcha == NULL)
     (*r) = q->izda;
    else
```

```
    { /* tiene rama izda y dcha. Se reemplaza por el mayor
                 de los menores */
     Nodo* a, *p;
     P = q;
     a = q->izda;
     while (a->dcha){
      p = a;
      a = a->dcha;
     }
     q->nummat = a->nummat;
     strcpy(q->nombre,a->nombre);
     if (p == q)
      p->izda = a->izda;
     else
      p->dcha = a->izda;
     q = a;
    }
    free(q);
  }
}
```

### 16.9.5. Recorridos de un árbol

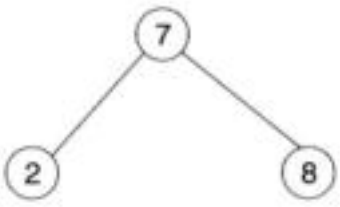
Existen dos tipos de recorrido de los nodos de un árbol: el recorrido en anchura y el recorrido en profundidad. En *el recorrido en anchura* se visitan los nodos por niveles. Para ello se utiliza una estructura auxiliar tipo cola en la que después de mostrar el contenido de un nodo, empezando por el nodo raíz, se almacenan los punteros correspondientes a sus hijos izquierdo y derecho. De esta forma si recorremos los nodos de un nivel, mientras mostramos su contenido, almacenamos en la cola los punteros a todos los nodos del nivel siguiente.

El *recorrido en profundidad* se realiza por uno de tres métodos recursivos: *preorden, inorden y postorden*. El primer método consiste en visitar el nodo raíz, su árbol izquierdo y su árbol derecho, por este orden. El recorrido *inorden* visita el árbol izquierdo, a continuación el nodo raíz y finalmente el árbol derecho. El recorrido *postorden* consiste en visitar primero el árbol izquierdo, a continuación el derecho y finalmente el raíz.

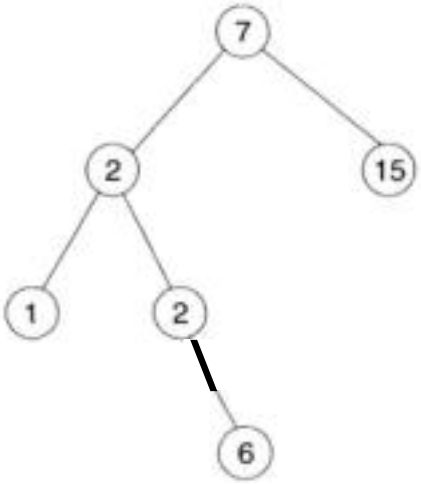
| | | | |
|---|---|---|---|
| *preorden* | Raíz | Izdo | Dcho |
| *en orden* | Izdo | Raíz | Dcho |
| *postorden* | Izdo | Dcho | Raíz |

### 16.9.6. Determinación de la altura de un árbol

La *altura* de un árbol dependerá del criterio que se siga para definir dicho concepto. **Así, si** en el caso de un árbol que tiene nodo raíz, se considera que su altura es 1, la altura del árbol es 2, y la altura del árbol



es **4.** Por último, si la altura de un árbol con un nodo es 1, la altura de un árbol vacío (el puntero es NULL) es 0.




### *Nota*

**La altura de un** árbol **es 1 más que la mayor de las alturas de sus subárboles izquierdo y derecho.**

**A** continuación, en el Ejemplo 16.14 se escribe una función entera que devuelve la altura de un árbol.

---

**Ejemplo 16.14**

*Función que determina la altura de un árbol binario de manera recursiva. Se considera que la altura de un árbol vacío es 0; si no está vacío. la altura es 1 + máximo entre las alturas de rama izquierda y derecha.*

```
int altura(Nodo* r)
{
  if (r == NULL)
    return 0;
  else
    return  (1 + max(altura(r->izda),altura(r->dcha)));
}
```

## 16.10. APLICACIONES DE ÁRBOLES EN ALGORITMOS DE EXPLORACIÓN

Los algoritmos recursivos de recorridos de árboles son el fundamento de muchas aplicaciones de árboles. Proporcionan un acceso ordenado y metódico a los nodos y a sus datos. Vamos a considerar en esta sección una serie de algoritmos de recorrido usuales en numerosos problemas de programación, tales como: contar el número de nodos hoja, calcular la profundidad de un árbol, imprimir un árbol o copiar y eliminar árboles.

### 16.10.1. Visita a los nodos de un árbol

En muchas aplicaciones se desea explorar (recorrer) los nodos de un árbol pero sin tener en cuenta un orden de recorrido preestablecido. En esos casos, el cliente o usuario es libre para utilizar el algoritmo oportuno.

La función `ContarHojas` recorre el árbol y cuenta el número de nodos hoja. Para realizar esta operación se ha de visitar cada nodo comprobando si es un nodo hoja. El recorrido utilizado será el ***postorden.***

```
/* Función ContarHojas
   la función utiliza recorrido postorden
   en cada visita se comprueba si el nodo es un nodo hoja
  (no tiene descendientes)
*/
void contarhojas(Nodo* r, int* nh)
{
  if (r != NULL)
  {
    contarhojas(r -> izda, nh);
    contarhojas(r -> dcha, nh);
       /* procesar raíz: determinar si es hoja */
    if (r->izda==NULL && r->dcha==NULL) (*nh)++;
  }
}
```

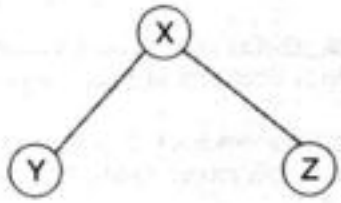
La función `eliminarbol` utiliza un recorrido postorden para liberar todos los nodos del árbol binario. Este recorrido asegura la liberación de la memoria ocupada por un nodo después de haber liberado su rama izquierda y derecha.

```
/*                    Función eliminarbol
  Recorre en postorden el árbol. Procesar la raíz, en esta
  función es liberar el nodo con free().
*/
void eliminarbol(Nodo* r)
{
  if (r != NULL)
  {
    eliminarbol(r -> izda);
    eliminarbol(r -> dcha);
    printf("\tNodo borrado: %d ",r->nummat);
    free(r);
  }
}
```

## 16.11. RESUMEN

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

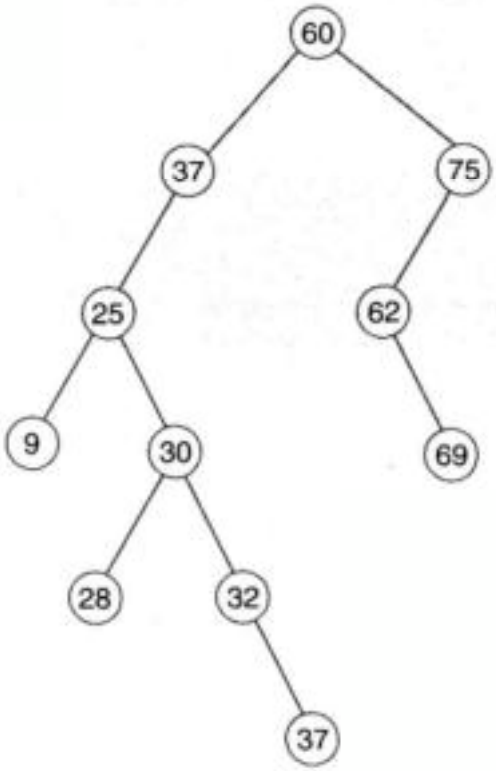
**La estructura árbol más utilizada normalmente es el *árbol binario.* Un árbol binario es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho.**

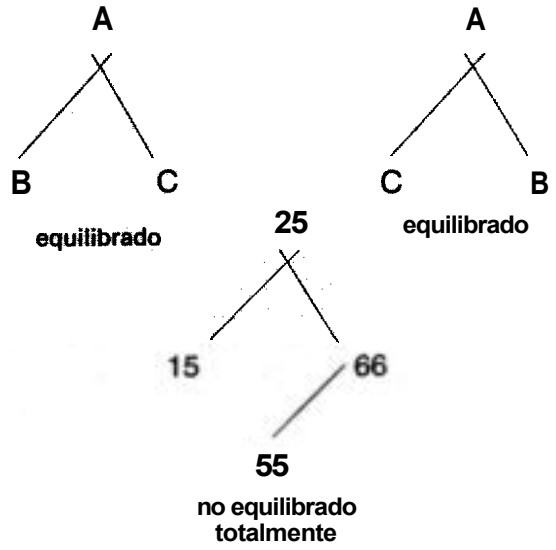



En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un padre. X es un *antecesor o ascendente* del elemento Y.

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.

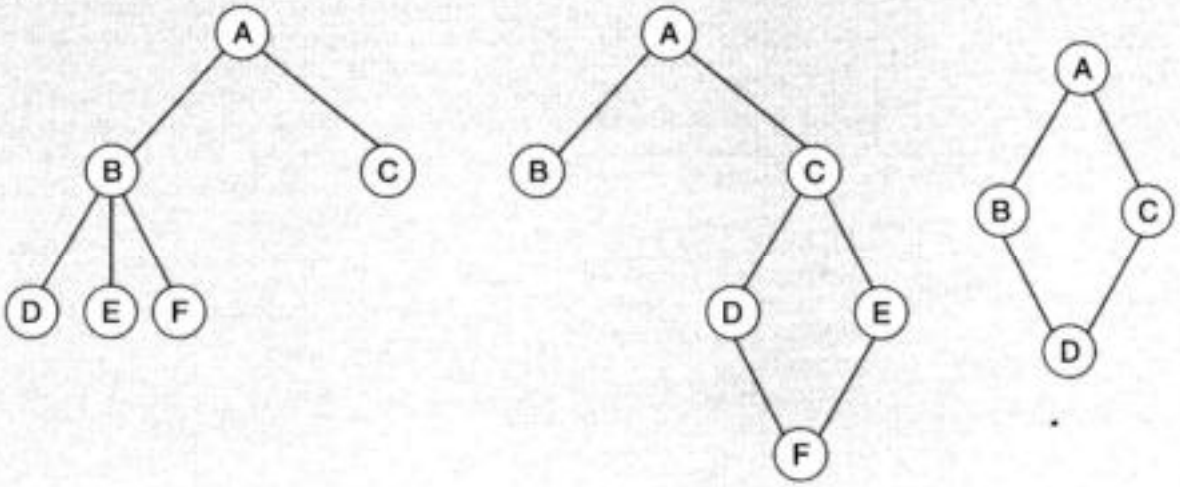


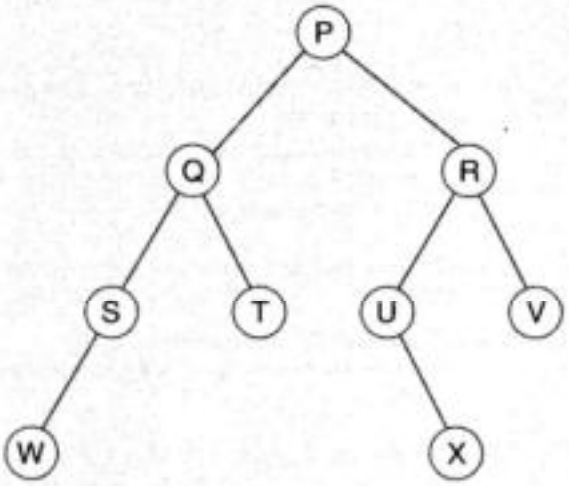

Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan fundamentalmente para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

## 16.12. EJERCICIOS

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




**16.2.** Considérese el árbol siguiente:




*a*) ¿Cuál es su altura?
*b*) ¿Está el árbol equilibrado? ¿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`.

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

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

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

**16.4.** Para el árbol del ejercicio 2 hacer recorridos utilizando los órdenes siguientes: `NDI, DNI, DIN`.

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

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

**16.6.** El recorrido preorden de un cierto árbol binario produce.

ADFGHKLPQRWZ

y en recorrido ***enorden*** produce

GFHKDLAWRQPZ

Dibujar el árbol binario.

**16.7.** Escribir una función no recursiva que cuente las hojas de un árbol binario.

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

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

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

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

**16.11.** Diseñar una función recursiva de búsqueda, que devuelva un puntero a un elemento en un árbol binario de búsqueda; si no está el elemento, devuelva `NULL`.

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

## 16.13. PROBLEMAS

**16.1.** 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 de 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 ORDE-NADOS POR NUMERO 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 |

**163.** Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición.

Hacer uso de la estructura árbol binario de búsqueda, cada nodo del árbol que tenga una palabra y su frecuencia.

**163.** 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.

**16.4.** Escribir una función booleana IDENTICOS que permita decir si dos árboles binarios son iguales.

**16.5.** Diseñar un programa interactivo que permita dar altas, bajas, listar, etc., en un árbol binario de búsqueda

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

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

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

**16.8.** Diseñar procedimientos no recursivos que listen los nodos de un árbol en inorden, preorden y postorden.

**16.9.** Dados dos árboles binarios de búsqueda indicar mediante un programa si los árboles tienen o no elementos comunes.

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

**16.11.** Un árbol binario de búsqueda puede implementarse con un array. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición **I** del array, su hijo izquierdo se encuentra en la posición **2*1** y su hijo derecho en la posición **21 + 1.** Diseñar a partir de esta representación los correspondientes procedimientos y funciones para gestionar interactivamente un árbol de números enteros. (Comente el inconveniente de esta representación de cara al máximo y mínimo número de nodos que pueden almacenarse.)

**16.12.** Una matriz de *N* elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar ordene ascendentemente la cadena de caracteres.

**16.13.** Dado un árbol binario de búsqueda diseñe un procedimiento que liste los nodos del árbol ordenados descendentemente.