CAPÍTULO 13

Estructura de datos no lineales (árboles y grafos)

INTRODUCCIÓN

Las *estructuras dinámicas lineales de datos —listas enlazadas, pilas y colas—* tienen grandes ventajas de flexibilidad sobre las representaciones contiguas; sin embargo, tienen un punto débil: son listas secuenciales, es decir, están dispuestas de modo que es necesario moverse a través de ellas una posición cada vez (cada elemento tiene un siguiente elemento). Esta linealidad es típica de cadenas, de elementos que pertenecen a una sola dimensión: campos en un registro, entradas en una pila, entradas en una cola y de nodos en una lista enlazada simple. En este capítulo se tratarán las *estructuras de datos no lineales* que resuelven los problemas que plantean las listas lineales y en las que cada elemento puede tener diferentes "siguientes" elementos, que introducen el concepto de estructuras de bifurcación. Estos tipos de datos se llaman *árboles*.

Asimismo, este capítulo introduce a una estructura matemática importante que tiene aplicaciones en ciencias tan diversas como la sociología, química, física, geografía y electrónica. Estas estructuras se denominan *grafos*.

13.1. INTRODUCCIÓN

Las estructuras de datos que han sido examinadas hasta ahora en este libro son lineales. A cada elemento le correspondía siempre un "siguiente" elemento. La linealidad es típica de cadenas, de elementos de arrays o listas, de campos en registros, entradas en pilas o colas y nodos en listas enlazadas.

En este capítulo se examinarán las estructuras de datos *no lineales*. En estas estructuras cada elemento puede tener diferentes "siguientes" elementos, que introduce el concepto de estructuras de bifurcación.

Las estructuras de datos no lineales son *árboles* y *grafos*. A estas estructuras se les denomina también *estructuras multienlazadas.*

13.2. ÁRBOLES

El árbol es una estructura de datos fundamental en informática, muy utilizada en todos sus campos, porque se adapta a la representación natural de informaciones homogéneas organizadas y de una gran comodidad y rapidez de manipulación. Esta estructura se encuentra en todos los dominios (campos) de la informática, desde la pura *algorítmica* (métodos de clasificación y búsqueda...) a la *compilación* (árboles sintácticos para representar las expresiones o producciones posibles de un lenguaje) o incluso los dominios de la inteligencia artificial (árboles de juegos, árboles de decisiones, de resolución, etc.).

Las estructuras tipo árbol se usan principalmente para representar datos con una relación jerárquica entre sus elementos, como son árboles genealógicos, tablas, etc.

Un árbol A es un conjunto finito de uno o más nodos, tales que:

1. Existe un nodo especial denominado RAIZ(v_1) del árbol.
2. Los nodos restantes (v_2, v_3, ..., v_n) se dividen en $m >= 0$ conjuntos disjuntos denominado A_1, A_2, ..., A_m, cada uno de los cuales es, a su vez, un árbol. Estos árboles se llaman *subárboles* del RAIZ.

La definición de árbol implica una estructura recursiva. Esto es, la definición del árbol se refiere a otros árboles. Un árbol con ningún nodo es un *árbol nulo*; no tiene raíz.

La Figura 13.1 muestra un árbol en el que se ha rotulado cada nodo con una letra dentro de un círculo. Esta es una notación típica para dibujar árboles.

Los tres subárboles del raíz A son B, C y D, respectivamente. B es la raíz de un árbol con un subárbol E. Este subárbol no tiene subárbol conectado. El árbol C tiene dos subárboles, F y G.

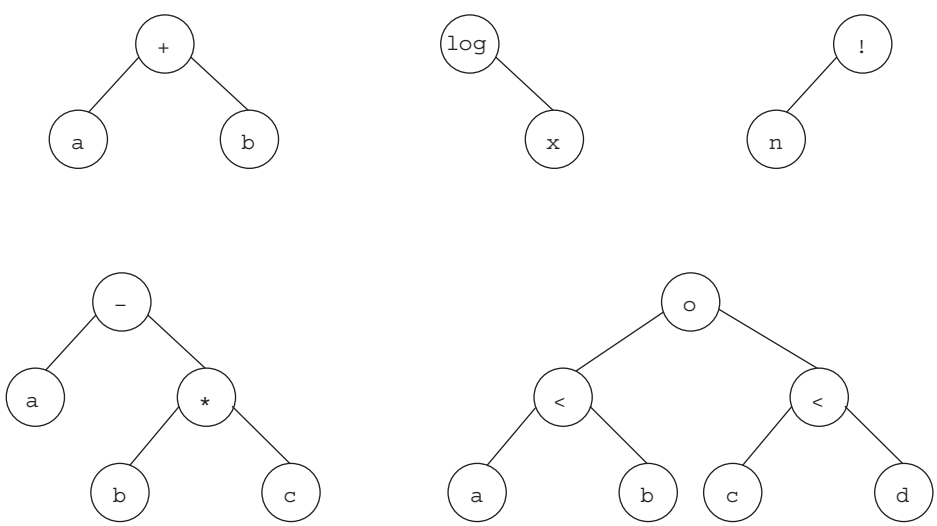


Figura 13.1. Diferentes árboles.

13.2.1. Terminología y representación de un árbol general

La representación y terminología de los árboles se realiza con las típicas notaciones de las relaciones familiares en los árboles genealógicos: padre, hijo, hermano, ascendente, descendiente, etc. Sea el árbol general de la Figura 13.2.

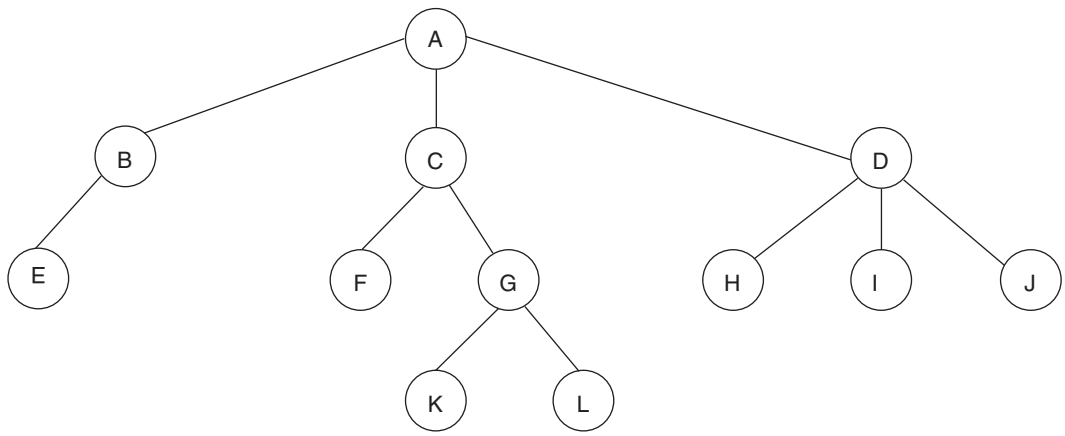


Figura 13.2. Árbol general.

Las definiciones a tener en cuenta son:

- *Raíz* del árbol. Todos los árboles que no están vacíos tienen un único nodo raíz. Todos los demás elementos o nodos se derivan o descienden de él. El nodo raíz no tiene *padre*, es decir, no es el hijo de ningún elemento.
- *Nodo*, son los vértices o elementos del árbol.
- *Nodo terminal* u *hoja (leaf node)* es aquel nodo que no contiene ningún subárbol (los nodos terminales u hojas del árbol de la Figura 13.2 son `E, F, K, L, H` y `J`).
- A cada nodo que no es hoja se asocia uno o varios subárboles llamados *descendientes (offspring)* o *hijos*. De igual forma, cada nodo tiene asociado un antecesor o *ascendiente* llamado *padre*.
- Los nodos de un mismo padre se llaman *hermanos*.
- Los nodos con uno o dos subárboles —no son hojas ni raíz— se llaman *nodos interiores* o *internos*.
- Una colección de dos o más árboles se llama *bosque (forest)*.
- Todos los nodos tienen un solo padre —excepto el *raíz*— que no tiene padre.
- Se denomina *camino* el enlace entre dos nodos consecutivos y *rama* es un camino que termina en una hoja.
- Cada nodo tiene asociado un número de *nivel* que se determina por la longitud del camino desde el raíz al nodo específico. Por ejemplo, en el árbol de la Figura 13.2.

Nivel 0 `A`
Nivel 1 `B, C, D`
Nivel 2 `E, F, G, H, I, J`
Nivel 3 `K, L`

- La *altura* o *profundidad* de un árbol es el número máximo de nodos de una rama. Equivale al nivel más alto de los nodos más uno. El *peso* de un árbol es el número de nodos terminales. La *altura* y el *peso* del árbol de la Figura 13.2 son 4 y 7, respectivamente.

Las representaciones gráficas de los árboles —además de las ya expuestas— pueden ser las mostradas en la Figura 13.3.

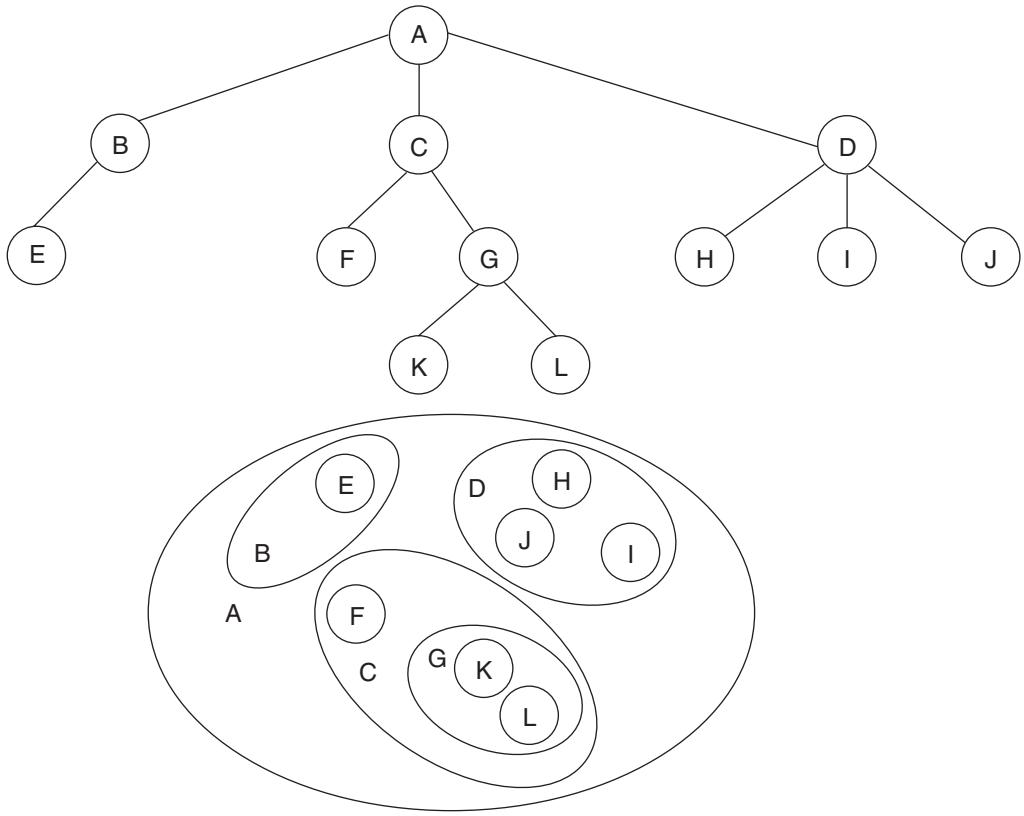


Figura 13.3 Representaciones de árboles.

13.3. ÁRBOL BINARIO

Existe un tipo de árbol denominado *árbol binario* que puede ser implementado fácilmente en una computadora.

Un *árbol binario* es un conjunto finito de cero o más nodos, tales que:

- Existe un nodo denominado raíz del árbol.
- Cada nodo puede tener 0, 1 o 2 subárboles, conocidos como *subárbol izquierdo* y *subárbol derecho*.

La Figura 13.4 representa diferentes tipos de árboles binarios:

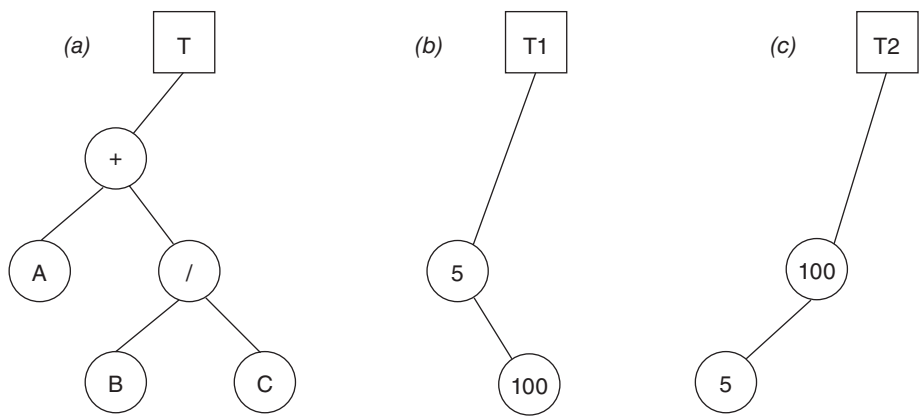


Figura 13.4. Ejemplos de árboles binarios: *(a)* expresión árbol a + b/c; *(b)* y *(c)* dos árboles diferentes con valores enteros.

13.3.1. Terminología de los árboles binarios

Dos árboles binarios se dice que son *similares* si tienen la misma estructura, y son *equivalentes* si son similares y contienen la misma información (Figura 13.5).

Un árbol binario está *equilibrado* si las alturas de los dos subárboles de cada nodo del árbol se diferencian en una unidad como máximo.

altura (subárbol izquierdo) – altura (subárbol derecho) ≤ 1

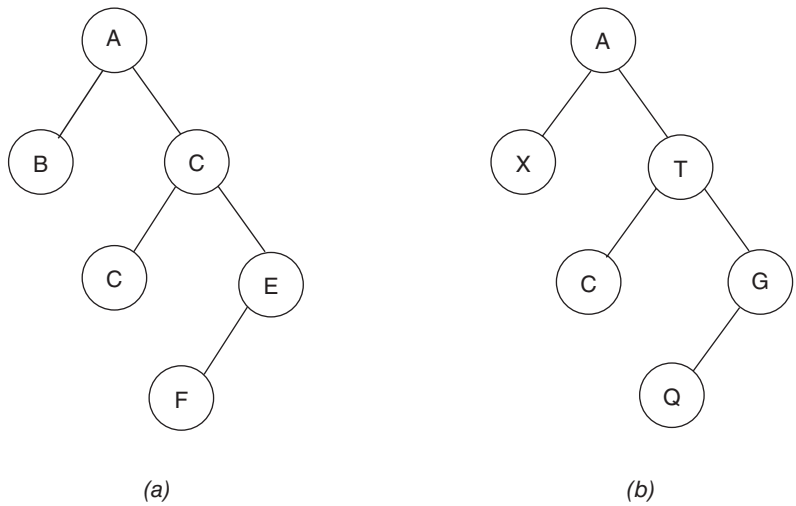


Figura. 13.5. Árboles binarios: *(a)* similares, *(b)* equivalentes.

El procesamiento de árboles binarios equilibrados es más sencillo que los árboles no equilibrados. En la Figura 13.6 se muestran dos árboles binarios de diferentes alturas y en la Figura 13.7, árboles equilibrados y sin equilibrar.

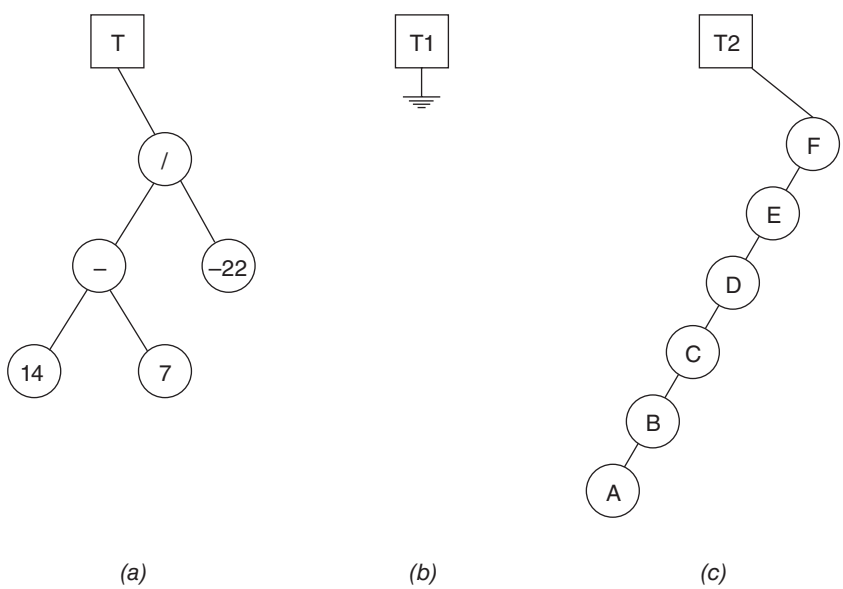


Figura 13.6. Árboles binarios de diferentes alturas: *(a)* altura 3, *(b)* árbol vacío, altura 0, *(c)* altura 6.

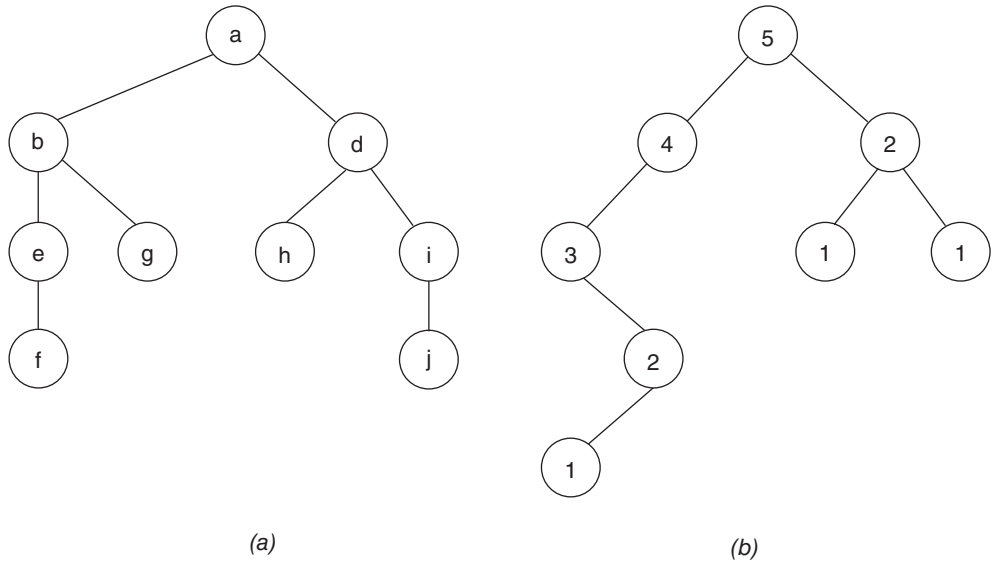


Figura 13.7. Arboles binarios: (*a*) equilibrados, (*b*) no equilibrados.

13.3.2. Árboles binarios completos

Un árbol binario se llama *completo* si todos sus nodos tienen exactamente dos subárboles, excepto los nodos de los niveles más bajos que tienen cero. Un árbol binario completo, tal que todos los niveles están llenos, se llama *árbol binario lleno*.

En la Figura 13.8 se ilustran ambos tipos de árboles.

Un árbol binario T de nivel h puede tener como máximo $2^h - 1$ nodos.

La altura de un árbol binario lleno de n nodos es $\log_2(n + 1)$. A la inversa, el *número máximo de nodos* de un árbol binario de altura h será $2^h - 1$. En la Figura 13.9 se muestra la relación matemática que liga los nodos de un árbol.

Por último, se denomina *árbol degenerado* un árbol en el que todos sus nodos tienen solamente un subárbol, excepto el último.

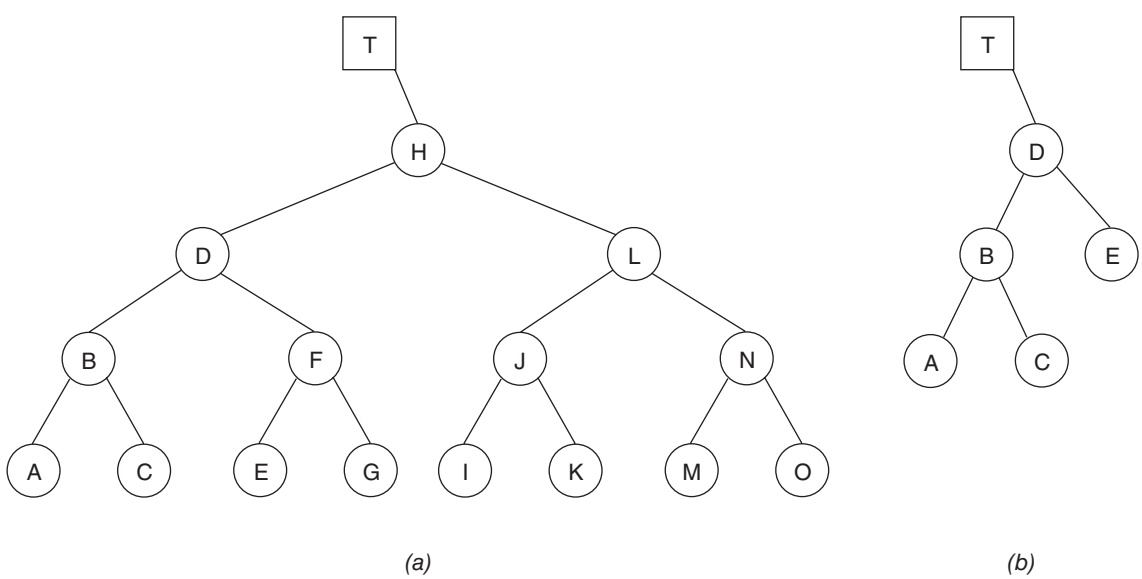


Figura 13.8. *(a)* árbol binario lleno de altura 4, *(b)* árbol binario completo de altura 3.

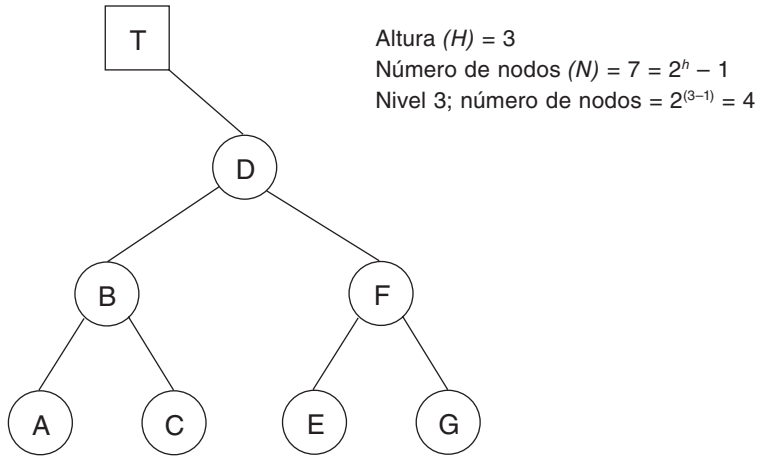


Figura 13.9. Relaciones matemáticas de un árbol binario.

13.3.3. Conversión de un árbol general en árbol binario

Dado que los árboles binarios es la estructura fundamental en la teoría de árboles, será preciso disponer de algún mecanismo que permita la conversión de un árbol general en un árbol binario.

Los árboles binarios son más fáciles de programar que los árboles generales. En éstos es imprescindible deducir cuántas ramas o caminos se desprenden de un nodo en un momento dado. Por ello, y dado que de los árboles binarios siempre se cuelgan como máximo dos subárboles, su programación será más sencilla.

Afortunadamente existe una técnica para convertir un árbol general a formato de árbol binario. Supongamos que se tiene el árbol A y se quiere convertir en un árbol binario B. El algoritmo de conversión tiene tres pasos fáciles:

1. La raíz de B es la raíz de A.
2. *a*) Enlazar al nodo raíz con el camino que conecta el nodo más a la izquierda (su hijo).
   b) Enlazar este nodo con los restantes descendientes del nodo raíz en un camino, con lo que se forma el nivel 1.
   c) A continuación, repetir los pasos *a*) y *b*) con los nodos del nivel 2, enlazando siempre en un mismo camino todos los hermanos —descendientes del mismo nodo—. Repetir estos pasos hasta llegar al nivel más alto.
3. Girar el diagrama resultante 45° para diferenciar entre los subárboles izquierdo y derecho.

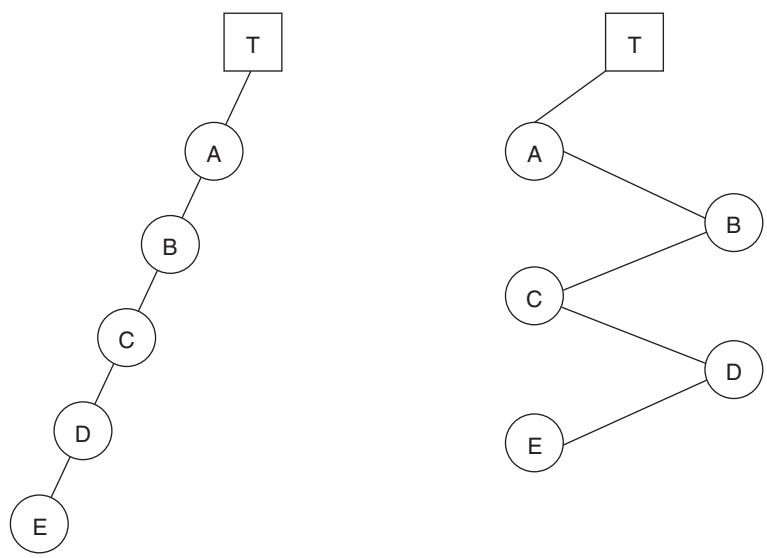
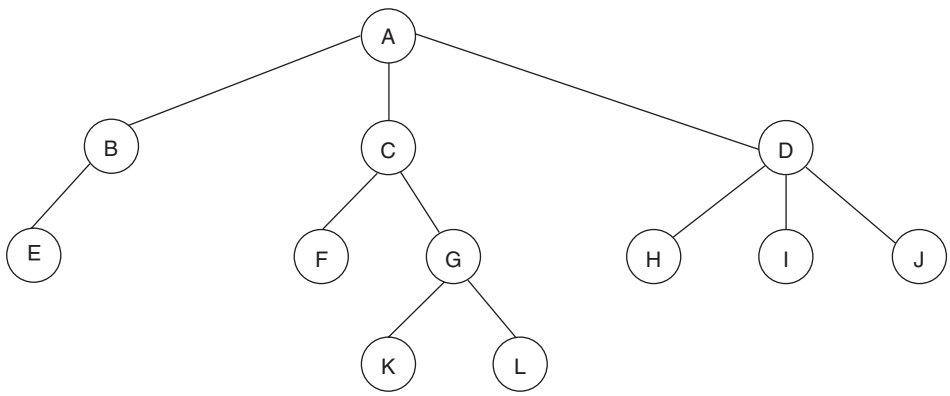


Figura 13.10. Árboles degenerados.

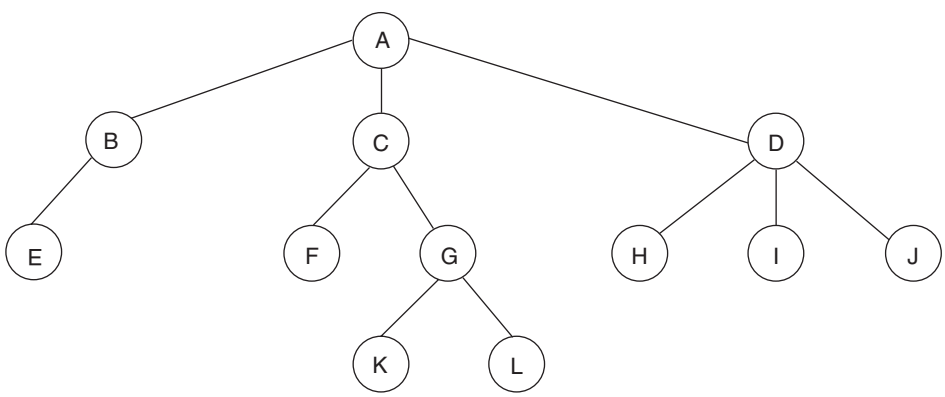
EJEMPLO 13.1

Convertir el árbol general T en un árbol binario.

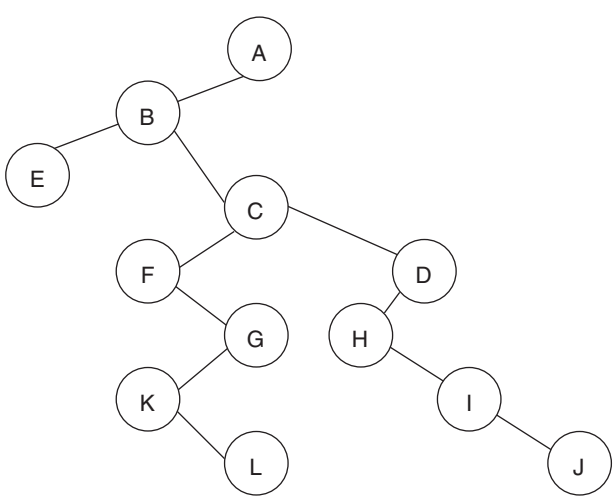


Siguiendo los pasos del algoritmo.

Paso 1:



Paso 2:



Paso 3:

Obsérvese que no existe camino entre E y F, debido a que no son descendientes del árbol original, ya que ellos tienen diferentes padres B y C.

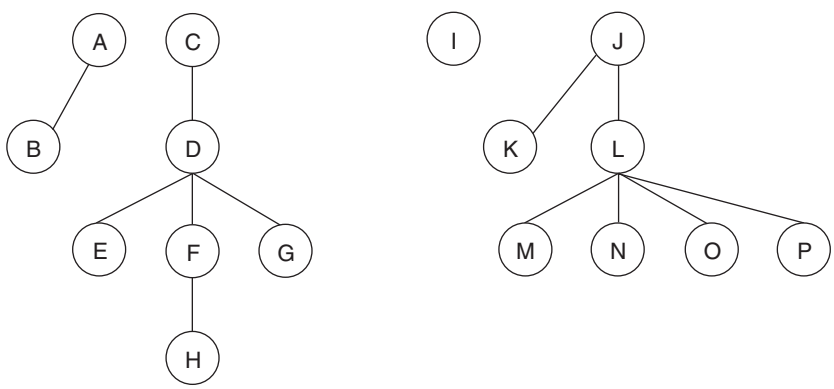
En el árbol binario resultante los punteros izquierdos son siempre de un nodo padre a su primer hijo (más a la izquierda) en el árbol general original. Los punteros derechos son siempre desde un nodo de sus descendientes en el árbol original.

EJEMPLO 13.2

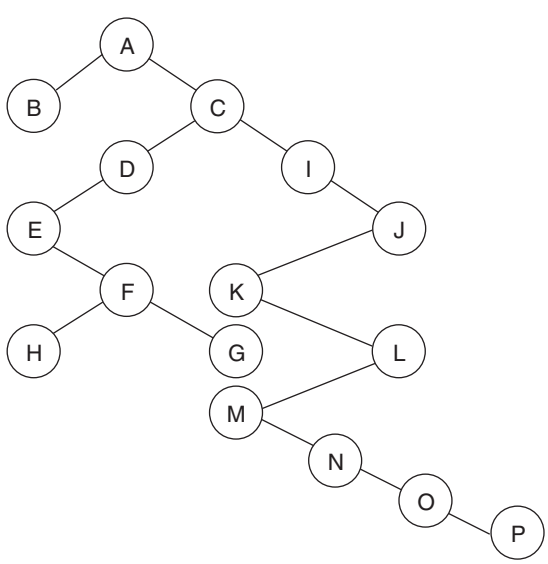
El algoritmo de conversión puede ser utilizado para convertir un bosque de árboles generales a un solo árbol binario.

El bosque siguiente puede ser representado por un árbol binario.

Bosque de árboles:

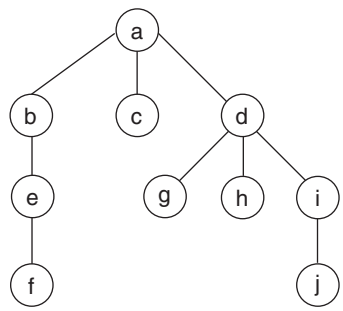


Árbol binario equivalente



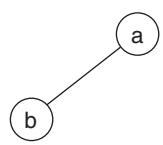
EJEMPLO 13.3

Convertir el árbol general en árbol binario.

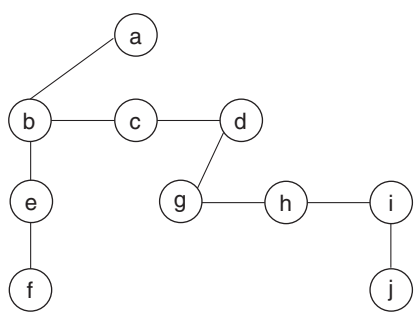


Solución

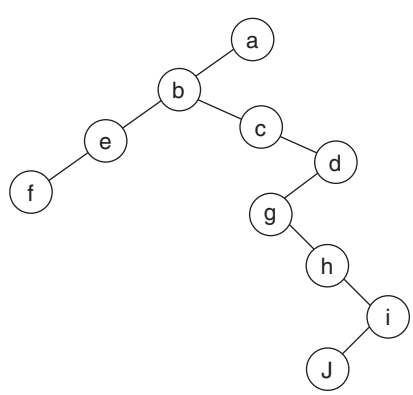
Paso 1:



Paso 2:



Paso 3:



13.3.4. Representación de los árboles binarios

Los árboles binarios pueden ser representados de dos modos diferentes:

- *Mediante punteros* (lenguajes C y C++).
- *Mediante arrays* o *listas enlazadas*.
- *Vinculando nodos, objetos* con mienbros que referencian otros objetos del mismo tipo.

13.3.4.1. Representación por punteros

Cada nodo de un árbol será un registro que contiene al menos tres campos:

- Un campo de datos con un tipo de datos.
- Un puntero al nodo del subárbol izquierdo (que puede ser **nulo**-`null`).
- Un puntero al nodo del subárbol derecho (que puede ser **nulo**-`null`).

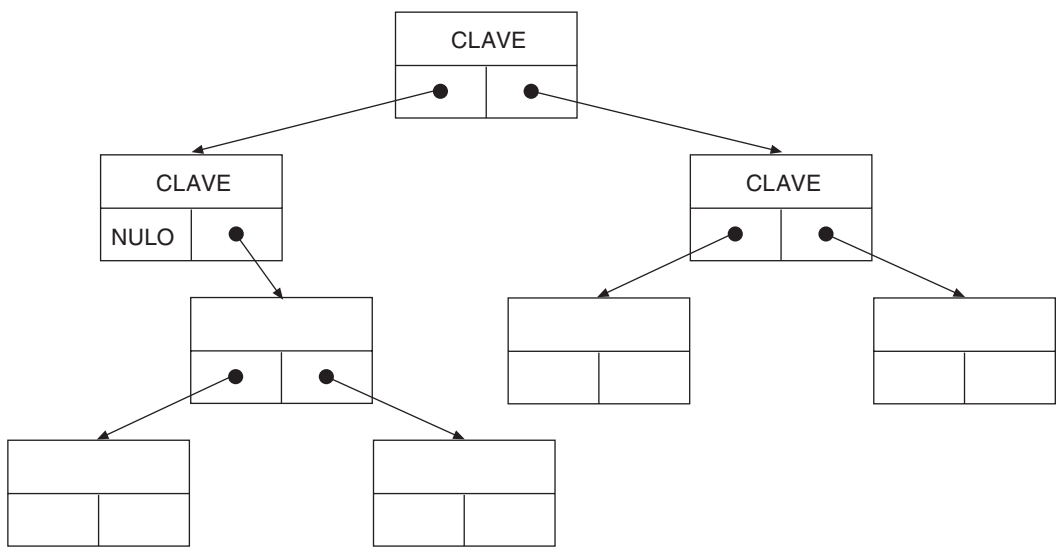
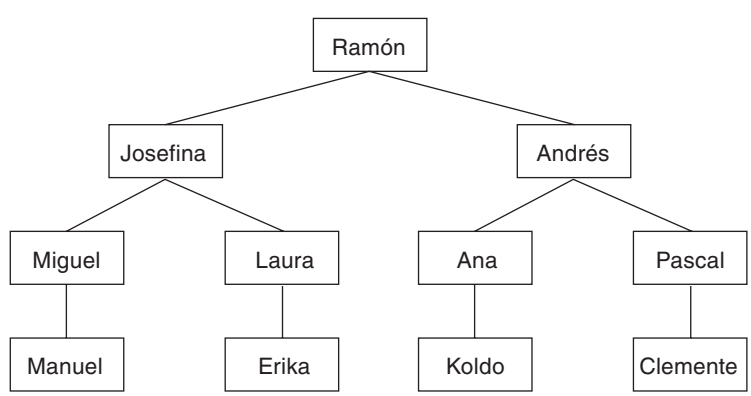


Figura 13.11. Representación de un árbol con punteros.



En lenguaje algorítmico se tendrá:

```
tipo nodo_arbol
  puntero_a nodo_arbol: punt
  registro : nodo_arbol
```

```
    <tipo_elemento> : elemento
    punt: subiz, subder
fin_registro
```

13.3.4.2. Representación por listas enlazadas

Mediante una lista enlazada se puede siempre representar el árbol binario de la Figura 13.12.

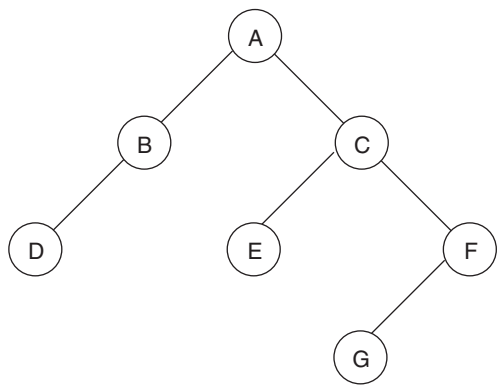


Figura 13.12. Árbol binario.

Nodo del árbol: campo 1 `INFO` (nodo)
campo 2 `IZQ` (nodo)
campo 3 `DER` (nodo)

El árbol binario representado como una lista enlazada se representa en la Figura 13.13.

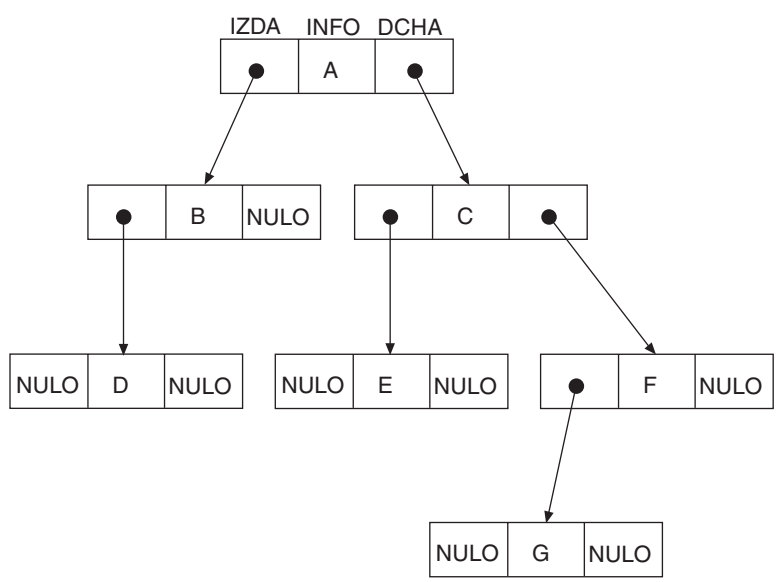


Figura 13.13. Árbol binario como lista enlazada.

13.3.4.3. Representación por arrays

Existen diferentes métodos; uno de los más fáciles es mediante tres arrays lineales paralelos que contemplan el campo de información y los dos punteros de ambos subárboles. Así, por ejemplo, el nodo raíz `RAMÓN` tendrá dos punteros

`IZQ:(9)` —JOSEFINA— y `DER:(16)` —ANDRÉS—, mientras que el nodo `CLEMENTE`, al no tener descendientes, sus punteros se consideran cero (`IZQ:0`, `DER:0`).

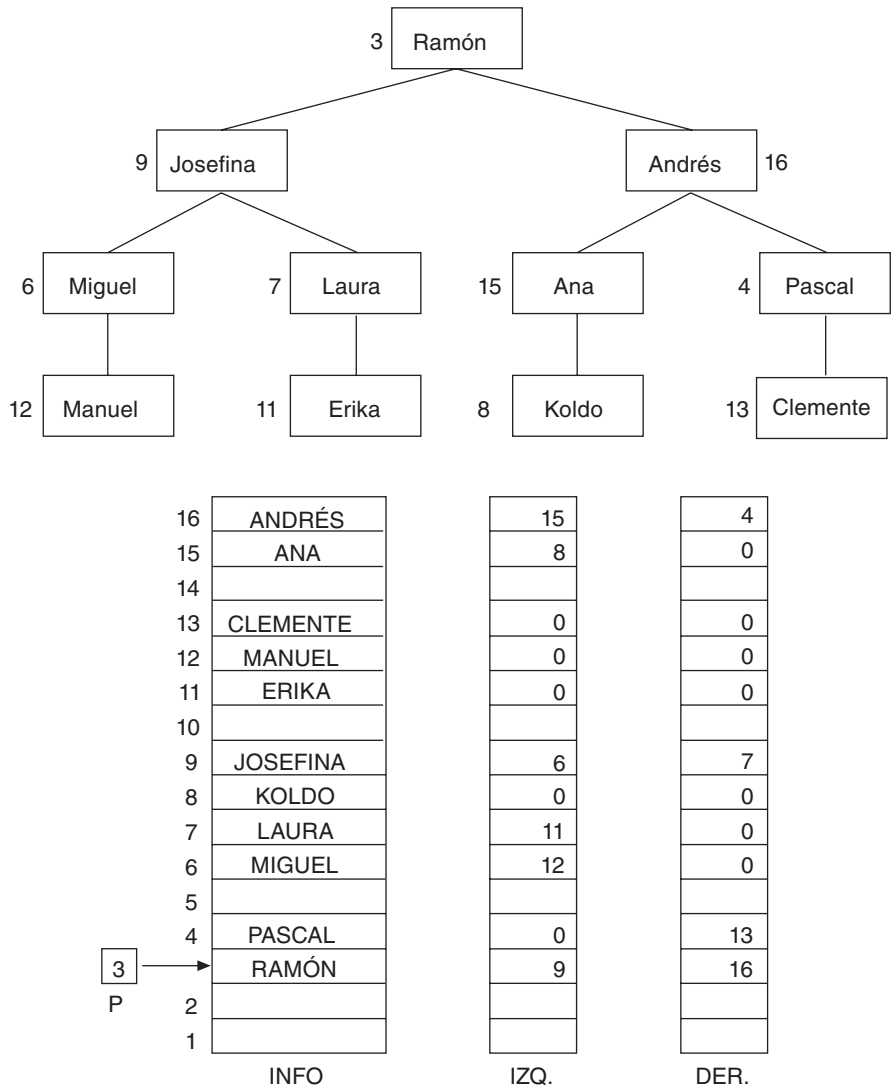
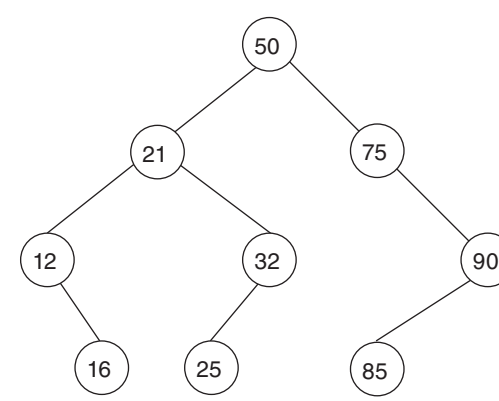


Figura 13.14. Árbol binario como arrays.

Otro método resulta más sencillo —un array lineal—. Para ello se selecciona un array lineal `ARBOL`.

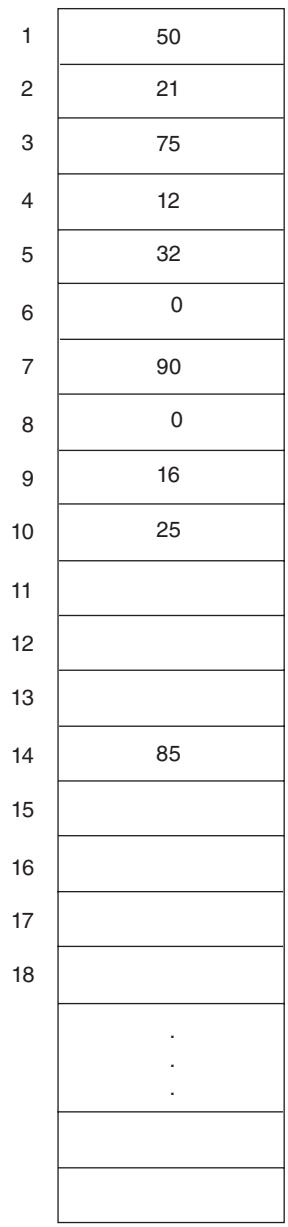


El algoritmo de transformación es:

```
1. La raíz del árbol se guarda en ARBOL [1].
2. si un nodo n está en ARBOL[i] entonces
   su hijo izquierdo se pone en ARBOL[2*i]
   y su hijo derecho en ARBOL[2*i + 1]
   si un subárbol está vacío, se le da el valor NULO.
```

Este sistema requiere más posiciones de memoria que nodos tiene el árbol. Así, la transformación necesitará un array con 2^{h+2} elementos si el árbol tiene una profundidad h. En nuestro caso, como la profundidad es 3, requerirá 32 posiciones (2^5), aunque si no se incluyen las entradas nulas de los nodos terminales, veremos cómo sólo necesita catorce posiciones.

Árbol



Un tercer método, muy similar al primero, sería la representación mediante un array de registros.

P 3

	INFO	IZQ	DER
1			
2			
3	RAMÓN	9	16
4	PASCAL	0	13
5			
6	MIGUEL	12	0
7	LAURA	11	0
8	KOLDO	0	0
9	JOSEFINA	6	7
10			
11	ERIKA	0	0
12	MANUEL	0	0
13	CLEMENTE	0	0
14			
15	ANA	8	0
16	ANDRÉS	15	4

13.3.5. Recorrido de un árbol binario

Se denomina *recorrido de un árbol* el proceso que permite acceder una sola vez a cada uno de los nodos del árbol. Cuando un árbol se recorre, el conjunto completo de nodos se examina.

Existen muchos modos para recorrer un árbol binario. Por ejemplo, existen seis diferentes recorridos generales en profundidad de un árbol binario, simétricos dos a dos.

Los algoritmos de recorrido de un árbol binario presentan tres tipos de actividades comunes:

- *Visitar* el nodo raíz.
- *Recorrer* el subárbol izquierdo.
- *Recorrer* el subárbol derecho.

Estas tres acciones repartidas en diferentes órdenes proporcionan los diferentes recorridos del árbol en profundidad. Los más frecuentes tienen siempre en común recorrer primero el subárbol izquierdo y luego el subárbol derecho. Los algoritmos que lo realizan llaman *pre-orden, post-orden, in-orden* y su nombre refleja el momento en que se visita el nodo raíz. En el in-orden el raíz está en el medio del recorrido, en el pre-orden el raíz está el primero y en el post-orden el raíz está el último:

Recorrido pre-orden

1. Visitar el raíz.
2. Recorrer el subárbol izquierdo en pre-orden.
3. Recorrer el subárbol derecho en pre-orden.

Recorrido in-orden

1. Recorrer el subárbol izquierdo en in-orden.
2. Visitar el raíz.
3. Recorrer el subárbol derecho en in-orden.

Recorrido post-orden

1. Recorrer el subárbol izquierdo en post-orden.
2. Recorrer el subárbol derecho en post-orden.
3. Visitar el raíz.

Obsérvese que todas estas definiciones tienen naturaleza recursiva.
En la Figura 13.15 se muestran los recorridos de diferentes árboles binarios.

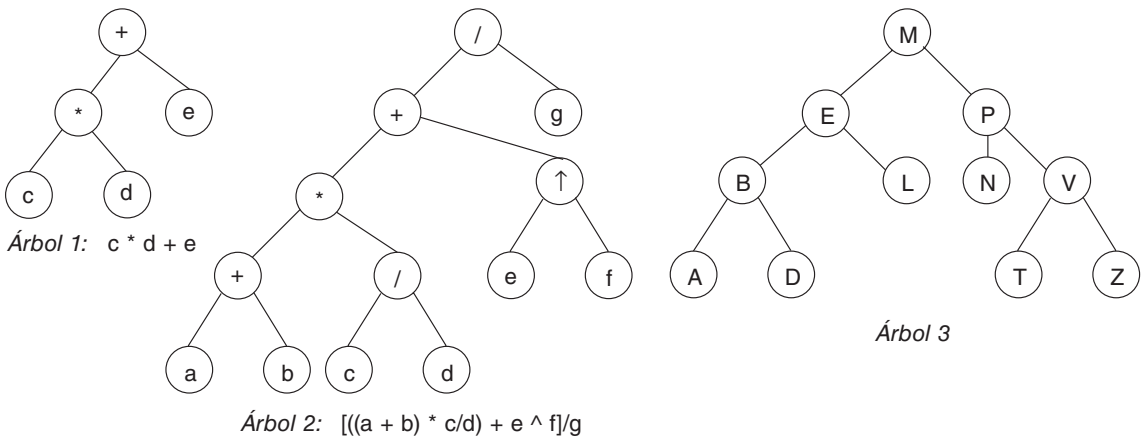
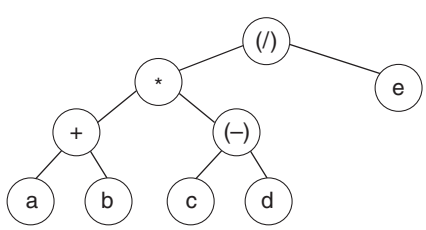


Figura 13.15. Recorrido de árboles binarios.

Árbol 1	*Pre-orden*	`+ * c d e`
	In-orden	`c * d + e`
	Post-orden	`c d * e +`
Árbol 2	*Pre-orden*	`/ + * + a b / c d ^ e f g`
	In-orden	`a + b * c / d + e ^ f / g`
	Post-orden	`a b + c d / * e f ^ + g /`
Árbol 3	*Pre-orden*	`MEBADLPNVTZ`
	In-orden	`ABDELMNPTVZ`
	Post-orden	`ADBLENTZVPM`

EJEMPLO 13.4

Calcular los recorridos del árbol binario.

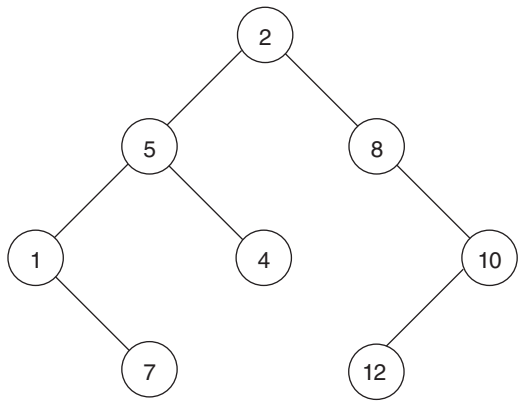


Solución

recorrido pre-orden	`/ * + ab - cde`
recorrido in-orden	`a + b * c - d/e`
recorrido post-orden	`ab + cd - * e/`

EJEMPLO 13.5

Realizar los recorridos del árbol binario.



Solución

recorrido pre-orden	2 5 1 7 4 8 10 12
recorrido in-orden	1 7 5 4 2 8 12 10
recorrido post-orden	7 1 4 5 12 10 8 2

13.4. ÁRBOL BINARIO DE BÚSQUEDA

Recordará del Capítulo 10, "Ordenación, búsqueda e intercalación", que para localizar un elemento en un array se podía realizar una búsqueda lineal; sin embargo, si el array era grande, una búsqueda lineal era ineficaz por su lentitud, especialmente si el elemento no estaba en el array, ya que requería la lectura completa del array. Se ganaba tiempo si se clasificaba el array y se utilizaba una búsqueda binaria. Sin embargo, en un proceso de arrays las inserciones y eliminaciones son continuas, por lo que esto se hará complejo en cualquier método.

En los casos de gran número de operaciones sobre arrays o listas, lo que se necesita es una estructura donde los elementos puedan ser eficazmente localizados, insertados o borrados. Una solución a este problema es una variante del árbol binario que se conoce como *árbol binario de búsqueda o árbol binario clasificado (binary search tree)*.

El árbol binario de búsqueda se construirá teniendo en cuenta las siguientes premisas:

- El primer elemento se utiliza para crear el nodo raíz.
- Los valores del árbol deben ser tales que pueda existir un orden (entero, real, lógico o carácter e incluso definido por el usuario si implica un orden).
- En cualquier nodo todos los valores del subárbol izquierdo del nodo son menor o igual al valor del nodo. De modo similar, todos los valores del subárbol derecho deben ser mayores que los valores del nodo.

Si estas condiciones se mantienen, es sencillo probar que el recorrido *in-orden* del árbol produce los valores clasificados por orden. Así, por ejemplo, en la Figura 13.16 se muestra un árbol binario.

Los tres recorridos del árbol son:

pre-orden	P F B H G S R Y T W Z
in-orden	B F G H P R S T Y W Z
post-orden	B G H F R T W Z Y S P

En esencia, un árbol binario contiene una clave en cada nodo que satisface las tres condiciones anteriores. Un árbol con las propiedades anteriores se denomina *árbol binario de búsqueda*.

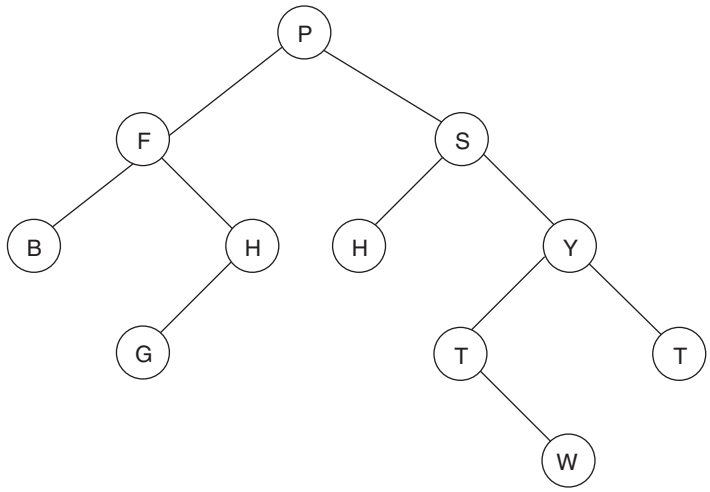


Figura 13.16. Árbol binario.

EJEMPLO 13.6

Se dispone de un array que contiene los siguientes caracteres:

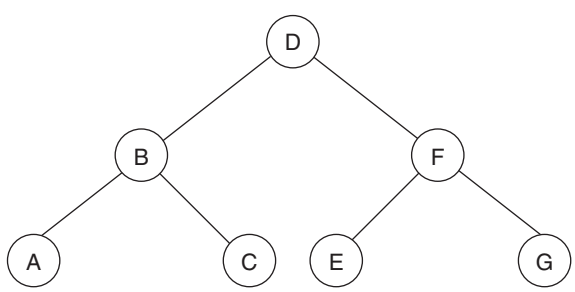
```
D  F  E  B  A  C  G
```

Construir un árbol binario de búsqueda.

Los pasos para la construcción del algoritmo son:

1. Nodo raíz del árbol: `D`.
2. El siguiente elemento se convierte en el descendente derecho, dado que `F` alfabéticamente es mayor que `D`.
3. A continuación, se compara `E` con el raíz. Dado que `E` es mayor que `D`, pasará a ser un hijo de `F` y como `E < F` será el hijo izquierdo.
4. El siguiente elemento `B` se compara con el raíz `D` y como `B < D` y es el primer elemento que cumple esta condición, `B` será el hijo izquierdo de `D`.
5. Se repiten los pasos hasta el último elemento.

El árbol binario de búsqueda resultante sería:



EJEMPLO 13.7

Construir el árbol binario de búsqueda correspondiente a la lista de números.

```
4  19  -7  49  100  0  22  12
```

El primer valor, como ya se ha comentado, es la raíz del árbol: es decir, 4. El siguiente valor, 19, se compara con 4; como es más grande se lleva al subárbol derecho de 4. El siguiente valor, –7, se compara con el raíz y es menor que su valor, 4; por tanto, se mueve al subárbol izquierdo. La Figura 13.17 muestra los sucesivos pasos.

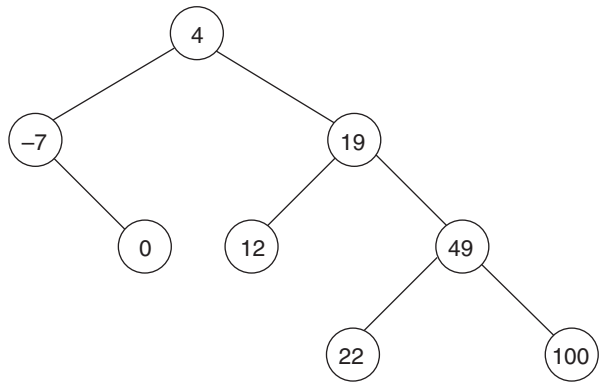
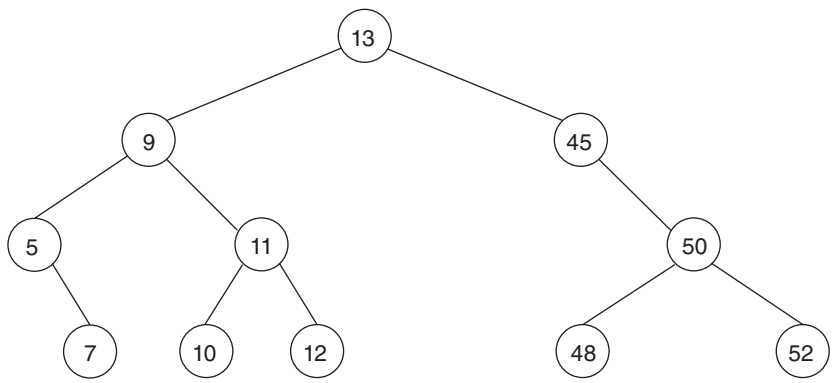


Figura 13.17. Construcción de un árbol binario.

13.4.1. Búsqueda de un elemento

La búsqueda en un árbol binario ordenado es dicotómica, ya que a cada examen de un nodo se elimina aquel de los subárboles que no contiene el valor buscado (valores todos inferiores o todos superiores).



El algoritmo de búsqueda del elemento —clave `x`— se realiza comparándolo con la clave del raíz del árbol. Si no es el mismo, se pasa al subárbol izquierdo o derecho, según el resultado de la comparación, y se repite la búsqueda en ese subárbol. La terminación del procedimiento se producirá cuando:

- Se encuentra la clave.
- No se encuentra la clave; se continúa hasta encontrar un subárbol vacío.

```
procedimiento buscar (E punt: RAIZ;
                      E <tipo_elemento>: elemento;
                      S punt: actual, anterior)
  var
    logico: encontrado
  inicio
    encontrado ← falso
    anterior ← nulo
    actual ← raiz
    mientras no encontrado Y (actual<>nulo) hacer
      si actual→.elemento = elemento entonces
```

```
            encontrado ← verdad
        si_no
          anterior ← actual
          si actual→.elemento > elemento entonces
              actual ← actual→.izdo
          si_no
              actual ← actual→.dcho
          fin_si
        fin_si
      fin_mientras
      si no encontrado entonces
          escribir('no existe', elemento)
      si_no
          escribir( elemento, 'existe')
      fin_si
  fin_procedimiento
  //< tipo_elemento> en este algoritmo es un tipo de dato simple
```

13.4.2. Insertar un elemento

Para insertar un elemento en el árbol A se ha de comprobar, en primer lugar, que el elemento no se encuentra en el árbol, ya que su caso no precisa ser insertado. Si el elemento no existe, la inserción se realiza en un nodo en el que al menos uno de los dos punteros `izq` o `der` tenga valor `nulo`.

Para realizar la condición anterior se desciende en el árbol a partir del nodo raíz, dirigiéndose de izquierda a derecha de un nodo, según que el valor a insertar sea inferior o superior al valor del campo clave `INFO` de este nodo. Cuando se alcanza un nodo del árbol en que no se puede continuar, el nuevo elemento se engancha a la izquierda o derecha de este nodo en función de que su valor sea inferior o superior al del nodo alcanzado.

El algoritmo de inserción del elemento `x` es:

```
procedimiento insertar (E/S punt: raiz;
                        E <tipo_elemento> : elemento)
var
  punt : nuevo,
         actual,
         anterior
inicio
  buscar (raiz, elemento, actual, anterior)
  si actual<> NULO entonces
      escribir ('elemento duplicado')
  si_no
      reservar (nuevo)
      nuevo→.elemento ← elemento
      nuevo→.izdo ← nulo
      nuevo→.dcho ← nulo
      si anterior = nulo entonces
        raiz ← nuevo
      si_no
        si anterior→.elemento > elemento entonces
            anterior→.izdo ← nuevo
        si_no
            anterior→.dcho ← nuevo
        fin_si
      fin_si
    fin_si
  fin_procedimiento
  // <tipo_elemento> es un tipo simple
```

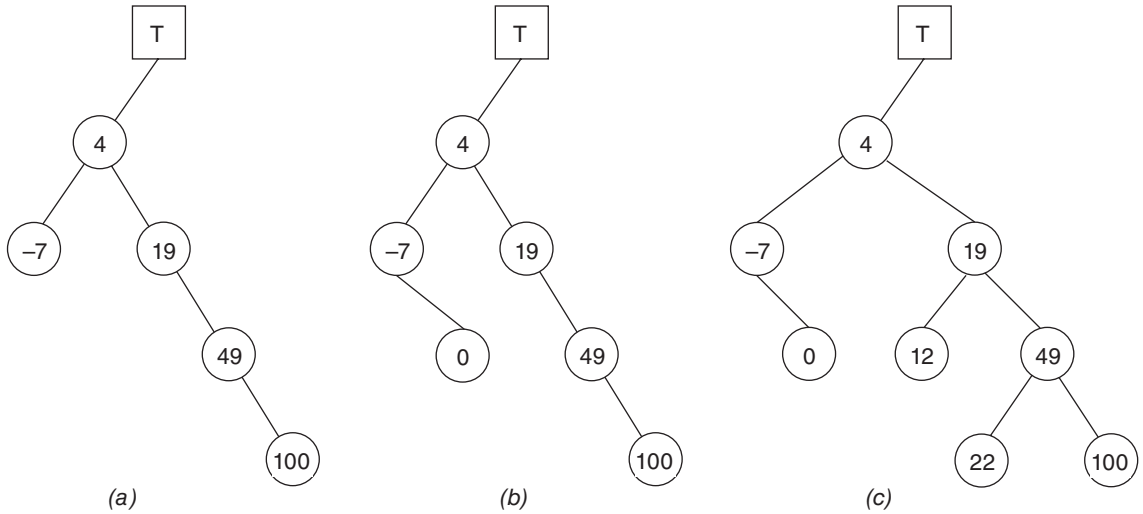


Figura 13.18. Inserciones en un árbol de búsqueda binaria: (*a*) insertar 100, (*b*) insertar (0), (*c*) insertar 22 y 12.

Para insertar el valor `x` en el árbol binario ordenado se necesitará llamar al subprograma `insertar`.

13.4.3. Eliminación de un elemento

La eliminación de un elemento debe conservar el orden de los elementos del árbol. Se consideran diferentes casos, según la posición del elemento o nodo en el árbol:

- Si el elemento es una hoja, se suprime simplemente.
- Si el elemento no tiene más que un descendiente, se sustituye entonces por ese descendiente.
- Si el elemento tiene dos descendientes, se sustituye por el elemento inmediato inferior situado lo más a la derecha posible de su subárbol izquierdo.

Para poder realizar estas acciones será preciso conocer la siguiente información del nodo a eliminar:

- Conocer su posición en el árbol.
- Conocer la dirección de su padre.
- Conocer si el nodo a eliminar tiene hijos, si son 1 o 2 hijos, y en el caso de que sólo sea uno, si es hijo derecho o izquierdo.

La Figura 13.19 muestra los tres posibles casos de eliminación de un nodo: *(a)* eliminar `C`, *(b)* eliminar `F`, *(c)* eliminar `B`.

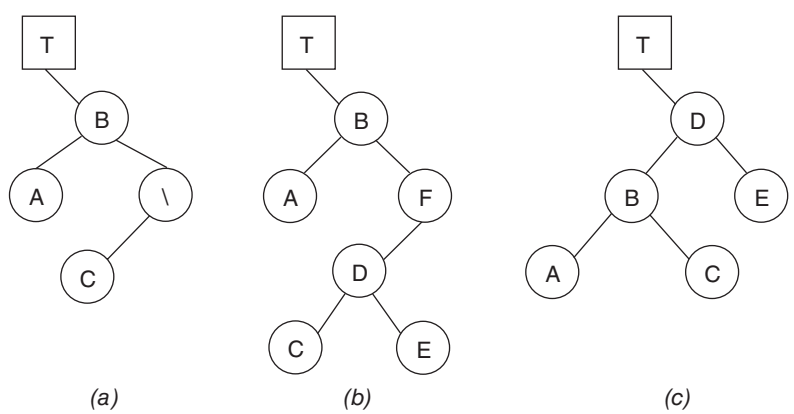


Figura 13.19. Casos posibles de eliminación de un nodo.

En la Figura 13.20 se muestra el caso de eliminación de un nodo con un subárbol en un gráfico comparativo antes y después de la eliminación.

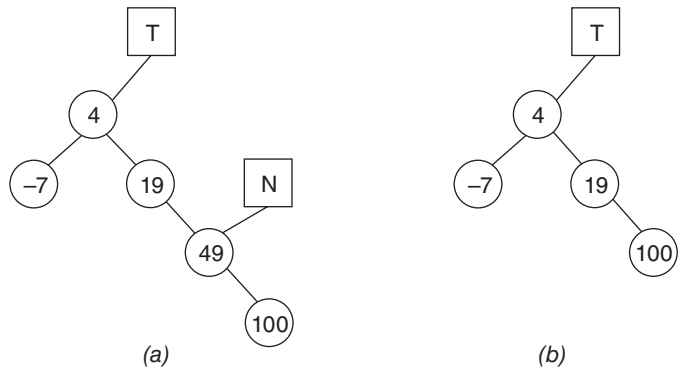


Figura 13.20. Eliminación de un nodo con un subárbol.

En la Figura 13.21 se muestra el caso de la eliminación de un nodo (27) que tiene dos subárboles no nulos. En este caso se busca el nodo sucesor cuyo campo de información le siga en orden ascendente, es decir, 42, se intercambia entonces con el elemento que se desea borrar, 27.

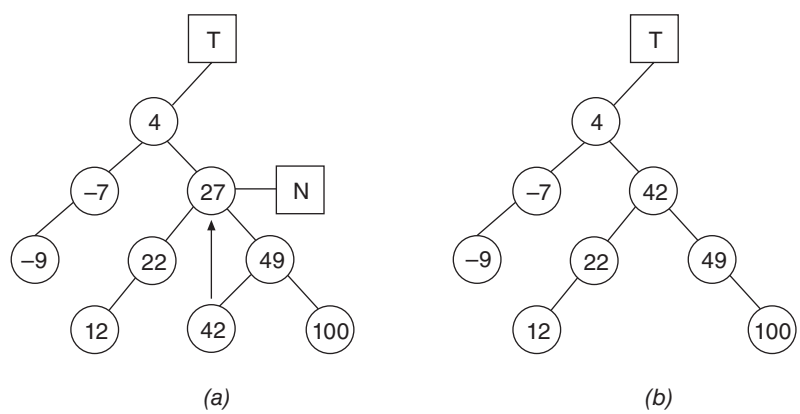
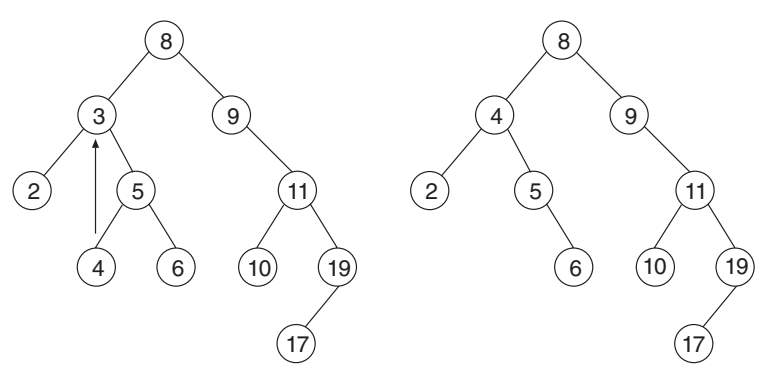


Figura 13.21. Eliminación de un nodo con dos subárboles no nulos.

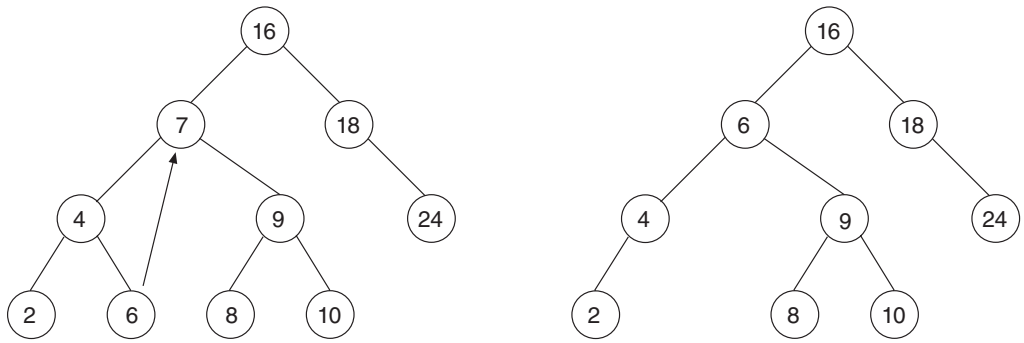
EJEMPLO 13.8

Deducir los árboles resultantes de eliminar el elemento 3 en el árbol A y el elemento 7 en el árbol B.

Árbol A

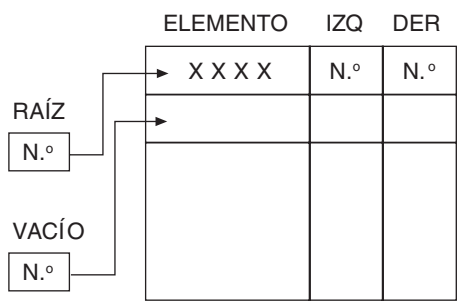


Árbol B. En este caso se busca el nodo sucesor cuyo campo de información le siga en orden decreciente, es decir, 6.



Árbol binario mediante arrays

Los árboles deben ser tratados como estructuras dinámicas. No obstante, si el lenguaje no tiene punteros podremos simularlos mediante arrays.



`Izdo` y `dcho` serán dos campos numéricos para indicar la posición en que están los hijos izquierdo y derecho. El valor 0 indicaría que no tiene hijo.

Trataremos el array como una lista `ENLAZADA` y necesitaremos una `LISTA DE VACIOS` y una variable `VACIO` que apunte al primer elemento de la lista de vacíos. Para almacenar la lista de vacíos es indiferente que utilicemos el campo `Izdo` o `Dcho`.

EJEMPLO 13.9

```
algoritmo arbol_binario_mediante_arrays

const
  Max = <expresion>
tipo
  registro: TipoElemento
     ... : ...
     ... : ...
  fin_registro
  registro: TipoNodo
     TipoElemento : Elemento
     Entero       : Izdo, Dcho
  fin_registro
  array[1..Max] de TipoNodo : Arr

var
  Arr          : a
```

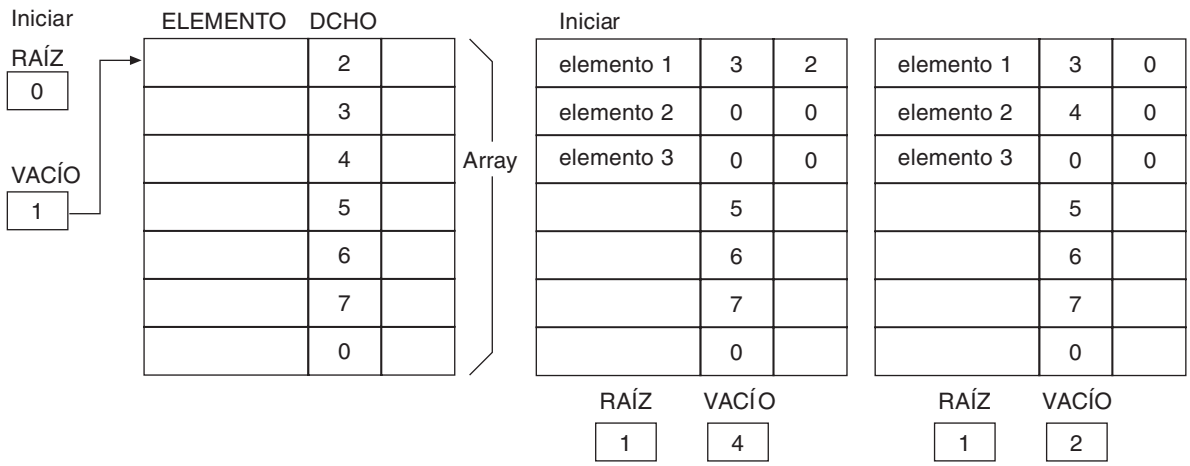
```
  Entero       : opcion
  TipoElemento : elemento
  Entero       : raiz
  Entero       : vacio

inicio
  iniciar(a,raiz,vacio)
  repetir
    menu
    escribir ('OPCIÓN: ')
    repetir
      leer (opcion)
    hasta_que (opcion >= 0) Y (opcion <= 3)
    según_sea opcion hacer
      1 :
            listado (a,raiz)
            escribir ('INTRODUZCA NUEVO ELEMENTO: ')
            proc_leer (elemento)
            altas (a, elemento, raiz, vacio)
            listado (a,raiz)
            pausa
      2 :
            listado (a,raiz)
            escribir ('INTRODUZCA ELEMENTO A DAR DE BAJA: ')
            proc_leer (elemento)
            bajas (a, elemento, raiz, vacio)
            listado (a,raiz)
            pausa
      3:
            listado (a,raiz)
            pausa
    fin_según
  hasta_que opcion = 0
fin

procedimiento pausa
var
  cadena : c
inicio
  escribir('PULSE RETURN PARA CONTINUAR')
  leer(c)
fin_procedimiento

procedimiento menu
inicio
  escribir ('1.- ALTAS')
  escribir ('2.- BAJAS')
  escribir ('3.- LISTADO')
  escribir ('0.- FIN')
fin_procedimiento

procedimiento iniciar(S Arr: a;  S Entero: raiz, vacio)
var
  Entero: i
inicio
  raiz ← 0
```

```
  vacio ← 1
  desde i ← 1 hasta Max-1 hacer
    a[i].dcho ← i+1
  fin_desde
  a[Max].dcho ← 0
fin_procedimiento

logico función ArbolVacio(E Entero: raiz)
inicio
  si raiz=0 entonces
     devolver(verdad)
  si_no
     devolver(falso)
  fin_si
fin_función

logico funcion ArbolLleno(E Entero: vacio)
inicio
  si vacio=0 entonces
      devolver(verdad)
  si_no
      devolver(falso)
  fin_si
fin_función

procedimiento inorden(E/S Arr: a;   E Entero: raiz)
inicio
  si raiz <> 0 entonces
      inorden(a,a[raiz].Izdo)
      proc_escribir(a[raiz].elemento)
      inorden(a,a[raiz].Dcho)
  fin_si
fin_procedimiento

procedimiento preorden(E/S Arr: a;   E Entero: raiz)
inicio
  si raiz <> 0 entonces
      proc_escribir(a[raiz].elemento)
      preorden(a,a[raiz].Izdo)
      preorden(a,a[raiz].Dcho)
  fin_si
fin_procedimiento

procedimiento postorden(E/S Arr: a;   E Entero: raiz)
inicio
  si raiz <> 0 entonces
      postorden(a,a[raiz].Izdo)
      postorden(a,a[raiz].Dcho)
      proc_escribir(a[raiz].elemento)
  fin_si
fin_procedimiento

procedimiento buscar(E/S Arr: a; E entero: raiz;
                     E TipoElemento: elemento;
                     S entero: act, ant)

var
  logico: encontrado
```

```
inicio
  encontrado ← falso
  act ← raiz
  ant ← 0
  mientras no encontrado y (act<>0) hacer
    si igual(elemento, a[act].elemento) entonces
       encontrado ← verdad
    si_no
       ant ← act
       si mayor(a[act].elemento, elemento) entonces
           act ← a[act].Izdo
       si_no
           act ← a[act].Dcho
       fin_si
    fin_si
  fin_mientras
fin_procedimiento

procedimiento altas(E/S Arr: A; E TipoElemento: elemento;
                    E/S entero: raiz, vacio)
var
  entero: act,ant,auxi
inicio
  si vacio <> 0 entonces
      buscar(a, raiz, elemento, act, ant)
      si act <> 0 entonces
         escribir('ESE ELEMENTO YA EXISTE')
      si_no
         auxi ← vacio
         vacio ← a[auxi].Dcho
         a[auxi].elemento ← elemento
         a[auxi].Izdo ← 0
         a[auxi].Dcho ← 0
         si ant = 0 entonces
            raiz ← auxi
         si_no
            si mayor(a[ant].elemento, elemento) entonces
               a[ant].Izdo ← auxi
            si_no
               a[ant].Dcho ← auxi
            fin_si
         fin_si
      fin_si
  fin_si
fin_procedimiento

procedimiento bajas(E/S Arr: A; E TipoElemento: elemento;
                    E/S entero: raiz, vacio)
var
  entero: act, ant, auxi
inicio
  buscar(a, raiz, elemento, act, ant)
  si act = 0 entonces
     escribir('ESE ELEMENTO NO EXISTE')
  si_no
     si (a[act].Izdo = 0) Y (a[act].Dcho = 0) entonces
```

```
          si ant = 0 entonces
            raiz ← 0
          si_no
            si a[ant].Izdo = act entonces
               a[ant].Izdo ← 0
            si_no
               a[ant].Dcho ← 0
            fin_si
          fin_si
      si_no
         si (a[act].Izdo <> 0) Y (a[act].Dcho <> 0) entonces
             ant ← act
             auxi ← a[act].Izdo
             mientras a[auxi].Dcho <> 0 hacer
                 ant ← auxi
                 auxi ← a[auxi].Dcho
             fin_mientras
             a[act].Elemento ← a[auxi].Elemento
             si ant = act entonces
                a[ant].Izdo ← a[auxi].Izdo
             si_no
                a[ant].Dcho ← a[auxi].Izdo
             fin_si
             act ← auxi
         si_no
             si a[act].Dcho <> 0 entonces
                si ant = 0 entonces
                   raíz ← a[act].Dcho
                si_no
                   si a[ant].Izdo = act entonces
                      a[ant].Izdo ← a[act].Dcho
                   si_no
                      a[ant].Dcho ← a[act].Dcho
                   fin_si
                fin_si
             si_no
                si ant = 0 entonces
                   raíz ← a[act].Izdo
                si_no
                   si a[ant].Dcho = act entonces
                      a[ant].Dcho ← a[act].Izdo
                   si_no
                      a[ant].Izdo ← a[act].Izdo
                   fin_si
                fin_si
             fin_si
         fin_si
      fin_si
      a[act].Dcho ← vacio
      vacio ← act
   fin_si
fin_procedimiento

procedimiento listado (E Arr: a;   E Entero: raiz)
inicio
   escribir ('INORDEN: ')
```

```
  inorden (a, raiz)
  escribir ('PREORDEN: ')
  preorden (a, raiz)
  escribir ('POSTORDEN: ')
  postorden (a, raiz)
fin_procedimiento
```



13.5. GRAFOS

Los grafos son otra estructura de datos no lineal y que tiene gran número de aplicaciones. El estudio del análisis de grafos ha interesado a los matemáticos durante siglos y representa una parte importante de la teoría combinatoria en matemáticas. Aunque la teoría de grafos es compleja y amplia, en esta sección se realizará una introducción a la teoría de grafos y a los algoritmos que permiten su solución por computadora.

Los árboles binarios representan estructuras jerárquicas con limitaciones de dos subárboles por cada nodo. Si se eliminan las restricciones de que cada nodo puede apuntar a dos nodos —como máximo— y que cada nodo puede estar apuntado por otro nodo —como máximo— nos encontramos con un grafo.

Ejemplos de grafos en la vida real los tenemos en la red de carreteras de un estado o región, la red de enlaces ferroviarios o aéreos nacionales, etc.

En una red de carreteras los nudos de la red representan los *vértices* del grafo y las carreteras de unión de dos ciudades los *arcos*, de modo que a cada arco se asocia una información tal como la distancia, el consumo en gasolina por automóvil, etc.

Los grafos nos pueden ayudar a resolver problemas como éste. Supóngase que ciertas carreteras del norte del Estado han sido bloqueadas por una reciente tormenta de nieve. ¿Cómo se puede saber si todas las ciudades de ese Estado se pueden alcanzar por carretera desde la capital o si existen ciudades aisladas? Evidentemente existe la solución del estudio de un mapa de carreteras; sin embargo, si existen muchas ciudades, la obtención de la solución puede ser ardua y costosa en tiempo. Una computadora y un algoritmo adecuado de grafos solucionarán fácilmente el problema.

13.5.1. Terminología de grafos

Formalmente un *grafo* es un conjunto de puntos —una estructura de datos— y un conjunto de líneas, cada una de las cuales une un punto a otro. Los puntos se llaman *nodos* o *vértices* del grafo y las líneas se llaman *aristas* o *arcos (edges)*.

Se representan el conjunto de vértices de un grafo dado `G` por V_G y el conjunto de arcos por A_G. Por ejemplo, en el grafo `G` de la Figura 13.23:

$V_G = \{a, b, c, d\}$
$A_G = \{1, 2, 3, 4, 5, 6, 7, 8\}$