Unidad 2: Estructuras de Datos

Tema IV. Estructuras de Datos compuestos enlazados

Árboles binarios. Árbol binario de búsqueda.
Grafos. Clasificación. Recorrido de un grafo.

PRIMERA PARTE

Prof. ADJ. (ord) Vallejos, Oscar Adolfo - Fa.Ce.Na.

Árboles

Los árboles constituyen estructuras de datos jerarquizados, y tienen multitud de aplicaciones.

- Análisis de circuitos, Representación de estructuras de fórmulas matemáticas
- Organización de datos en bases de datos
- Representación de la estructura sintáctica en compiladores.
- En muchas otras áreas de las ciencias del computador.

Un árbol está constituido por una colección de elementos denominados nodos, uno de los cuales se distingue con el nombre de raíz, junto con una relación de 'parentesco' que establece una estructura jerárquica sobre los nodos.

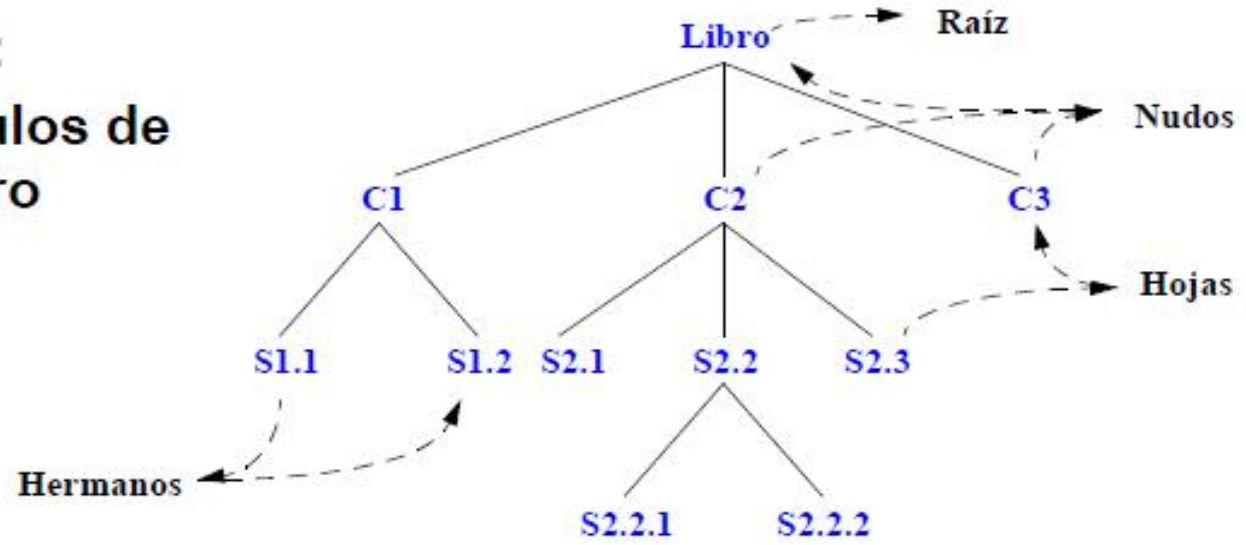
Cada nodo tiene un padre (excepto el raíz) y puede tener cero o más hijos.

Árboles

- **Un árbol es una estructura de datos jerarquizada aplicada sobre una colección de elementos u objetos (nodos). Que se puede definir en forma recursiva.**
  - Una estructura vacía o
  - un elemento o clave de información (nodo) más un número finito de estructuras tipo árbol, disjuntos, llamados subárboles. Si dicho número de estructuras es inferior o igual a 2, se tiene un árbol binario.
- **Es, por tanto, una estructura no secuencial**.
- **Cada dato reside en un nodo, y existen relaciones de parentesco entre nodos.**

Ejemplo:

Capítulos de un libro



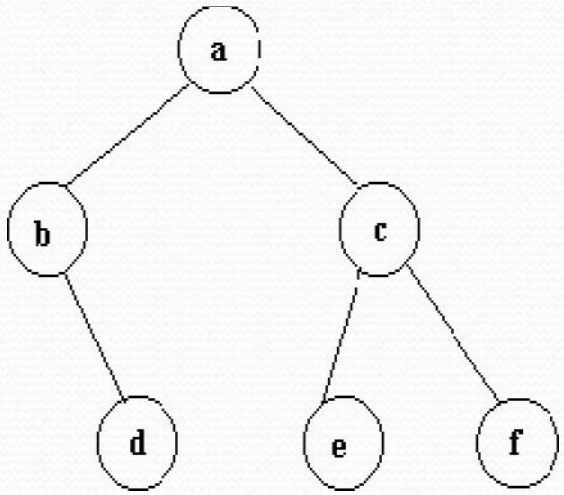
Otra definición nos da el árbol como un tipo de grafo

Un árbol es un grafo acíclico, conexo y no dirigido. Es decir, es un grafo no dirigido en el que existe exactamente un camino entre todo par de nodos.

Esta definición permite implementar un árbol y sus operaciones empleando las representaciones que se utilizan para los grafos.

Formas de representación

Mediante un grafo



Mediante un diagrama encolumnado

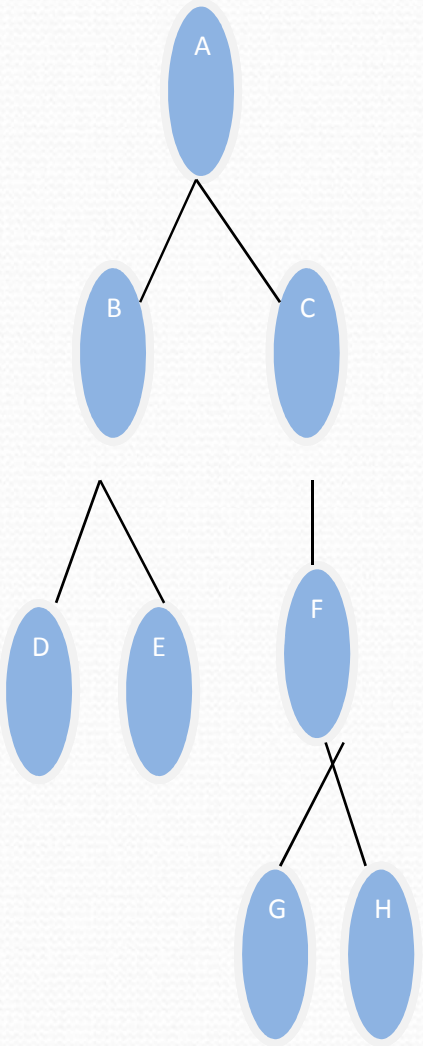
a
b
d
c
e
f

Nomenclatura sobre árboles

Árboles

- Raíz: es aquel elemento que no tiene antecesor; ejemplo: *A*
- Rama: arista entre dos nodos. Es decir, es el camino que termina en un hoja (C, F, H).
- Antecesor: un nodo X es es antecesor de un nodo Y si por alguna de las ramas de X se puede llegar a Y.
- Sucesor: un nodo X es sucesor de un nodo Y si por alguna de las ramas de Y se puede llegar a X.
- Grado de un nodo: el número de descendientes directos que tiene. Ejemplo:
    - *C* tiene grado 1,
    - *D* tiene grado 0,
    - *A* tiene grado 2.

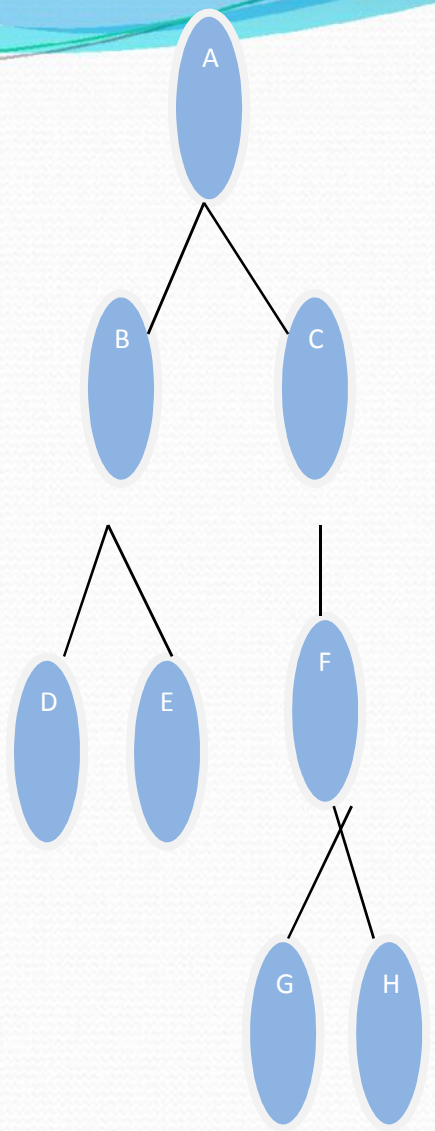
  El grado del árbol será entonces el máximo grado de todos los nodos del árbol. (2).
- Hoja: nodo que no tiene descendientes: grado 0. Ejemplo: *D*
- Nodo interno: aquel que tiene al menos un descendiente. Es decir, no es raiz ni hoja (F).
- Nivel: número de ramas que hay que recorrer para llegar de la raíz a un nodo. Ejemplo: el nivel del nodo *A* es 1 (es un convenio), el nivel del nodo *E* es 3.
- Altura: el nivel más alto del árbol. En el ejemplo de la figura 1 la altura es 4.
- Anchura: es el mayor valor del número de nodos que hay en un nivel. En la figura, la anchura es 3.
- Un nodo “B” es descendiente directo de un nodo “A”, si el nodo “B” es apuntado por el nodo “A”. ***“B” es hijo de “A”***.
- Los nodos son hermanos cuando son descendientes directos de un mismo nodo. ***Hijos de un mismo padre.*** (B y C).
- Una colección de dos o mas árboles se llama bosque.



Árboles

Nomenclatura sobre árboles (continua)

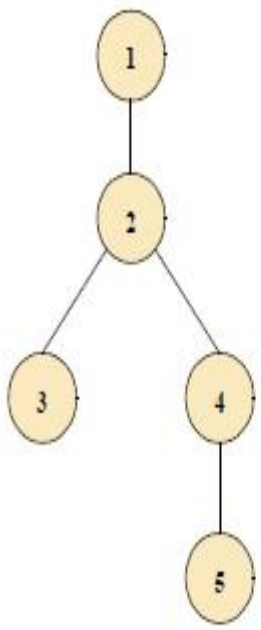
- Todos los nodos tienen un solo padre a excepción del nodo raíz (no tiene padre).
- Todo nodo que no tiene descendientes directos (hijos), se conoce con el nombre de ***terminal u hoja*** (D, E, G, H)
- Camino: Es el recorrido que enlaza nodos consecutivos. (secuencia de nodos tales que cada uno es hijo del anterior). (A , C, F).. Longitud del camino: nº de nodos que tiene
- Descendiente: un nodo es descendiente de otro si hay un camino del segundo al primero



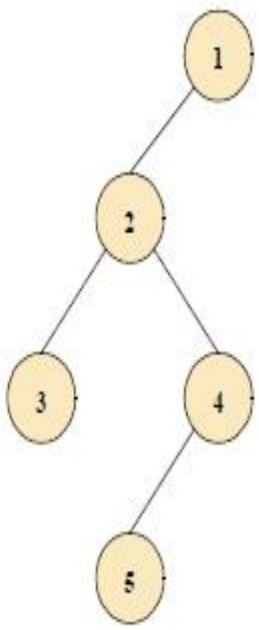
Aclaraciones: se ha denominado *A* a la raíz, pero se puede observar según la figura que cualquier nodo podría ser considerado raíz, basta con *girar* el árbol. Podría determinarse por ejemplo que *C* fuera la raíz, y *A* y *F* los sucesores inmediatos de la raíz *C*. Sin embargo, en las implementaciones sobre un computador que se realizan a continuación es necesaria una jerarquía, es decir, que haya una única raíz.

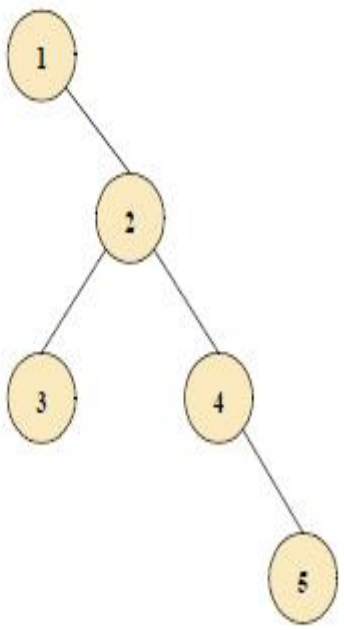
Árboles binarios

Un árbol binario es un árbol orientado y ordenado, en el que cada nodo puede tener un hijo izquierdo y un hijo derecho



Un árbol ordinario





Dos árboles binarios

Búsquedas en árboles binarios

- Los árboles binarios se adaptan muy bien para buscar elementos de forma eficiente.

- Para ello, todos los elementos se almacenan en el árbol ordenados:
  - Todos los descendientes izquierdos de un nodo son menores que él
  - Todos los descendientes derechos de un nodo son mayores que él

- En este caso, la búsqueda es O(log n) en el caso promedio, si el árbol está equilibrado (s las hojas están aproximadamente a la misma profundidad).

Árboles binarios

- Es posible representar estructuras de datos más complejas que las listas haciendo uso de los punteros. Las listas se han definido como registros que contienen datos y un puntero al siguiente nodo de la lista; una generalización del concepto de lista es el árbol, donde se permite que cada registro del tipo de dato dinámico tenga más de un enlace.

- La naturaleza lineal de las listas hace posible, y fácil, definir alguna operaciones de modo iterativo.

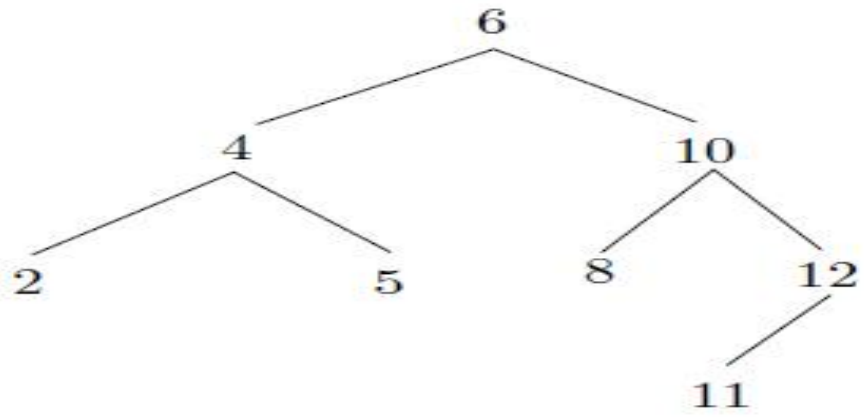
- Los árboles son estructuras de datos recursivas más generales que una lista y son apropiados para aplicaciones que involucran algún tipo de jerarquía (tales como los miembros de una familia o los trabajadores de una organización), o de ramificación (como los árboles de juegos), o de clasificación y/o búsqueda.

Árboles de búsqueda

Como un caso particular de árbol binario se encuentran los árboles binarios de búsqueda (o árboles de búsqueda binaria).

que son aquellos árboles en los que el valor de cualquier nodo es mayor que el valor de su hijo izquierdo y menor que el de su hijo derecho. Según la definición dada, no puede haber dos nodos con el mismo valor en este tipo de árbol.

La utilidad de los árboles binarios de búsqueda reside en que si buscamos cierta componente, podemos decir en qué mitad del árbol se encuentra comparando solamente con el nodo raíz.



Datos: 6, 4, 2, 10, 5, 12, 8, 11

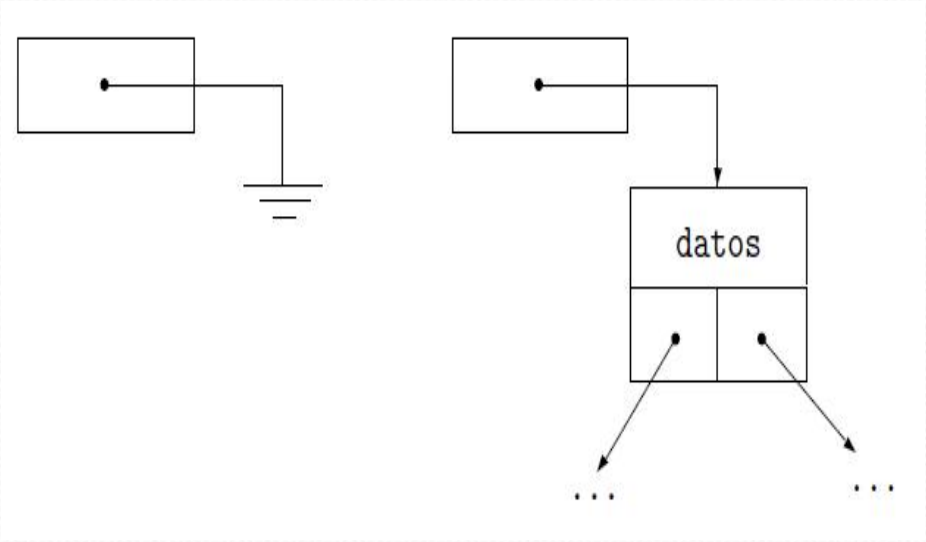
Árboles de búsqueda

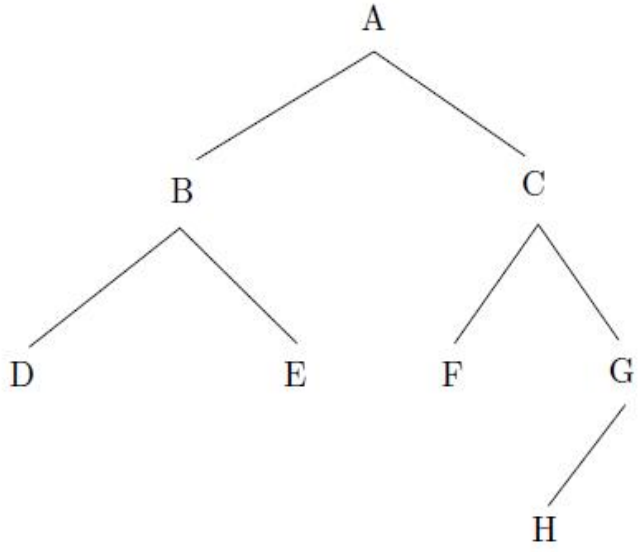
Los nodos de un árbol binario de búsqueda se pueden enumerar en orden creciente siguiendo un recorrido en inorden.

Una mejora de los árboles de búsqueda consiste en añadir un campo clave en cada nodo y realizar las búsquedas comparando los valores de dichas claves en lugar de los valores del campo contenido. De esta forma, pueden existir en el árbol dos nodos con el mismo valor en el campo contenido pero con clave distinta. En este texto se implementan los árboles de búsqueda sin campo clave para simplificar la presentación; la modificación de la implementación para incluir un campo clave es un ejercicio trivial.

Árboles binarios

La definición recursiva de árbol es muy sencilla: Un árbol o es vacío o consiste en un nodo que contiene datos y punteros hacia otros árboles. En este apartado sólo trataremos con árboles binarios, que son árboles en los que cada nodo tiene a lo sumo dos descendientes.





Árboles binarios

La definición en "C" del tipo árbol binario es sencilla: cada nodo del árbol va a tener dos punteros en lugar de uno.

```
typedef struct _nodo {

   int dato;
   struct _nodo *derecho;
   struct _nodo *izquierdo;

} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Arbol;
```

Árboles de búsqueda

Operaciones básicas

Consulta

Inserción

Eliminación de nodos

Las dos primeras son de fácil implementación, haciendo uso de la natural recursividad de los árboles. La operación de eliminación de nodos es, sin embargo, algo más compleja.

Buscar un elemento en un árbol ABB

- Si el árbol está vacío, terminamos la búsqueda: el elemento no está en el árbol.
- Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda con éxito.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

```
int Buscar(Arbol a, int dat) {
 pNodo actual = a;
 while(!Vacio(actual)) {
    if(dat == actual->dato)
      return TRUE;/* dato encontrado (2) */
      else
      if(dat < actual->dato)
      actual = actual->izquierdo;
      else
         if(dat > actual->dato)
         actual = actual->derecho;
   } return FALSE;
}
```

Árboles de búsqueda

Inserción de un nuevo nodo

Inserta un nuevo nodo en un árbol binario de búsqueda árbol; la inserción del nodo es tarea fácil, todo consiste en encontrar el lugar adecuado donde insertar el nuevo nodo, esto se hace en función de su valor de manera similar a lo visto en el ejemplo anterior.

```
procedure Insertar(datoNuevo: tElem; var arbol: tArbol);
  {Efecto: se añade ordenadamente a arbol un nodo de contenido
  datoNuevo}
begin
  if arbol = nil then begin
    New(arbol);
    with arbol^ do begin
      hIzdo:= nil;
      hDcho:= nil;
      contenido:= datoNuevo
    end  {with}
  end {then}
  else with arbol^ do
    if datoNuevo < contenido then
      Insertar(datoNuevo,hIzdo)
    else if datoNuevo > contenido then
      Insertar(datoNuevo,hDcho)
    else {No se hace nada: entrada duplicada}
end; {Insertar}
```

Insertar un elemento en un árbol ABB

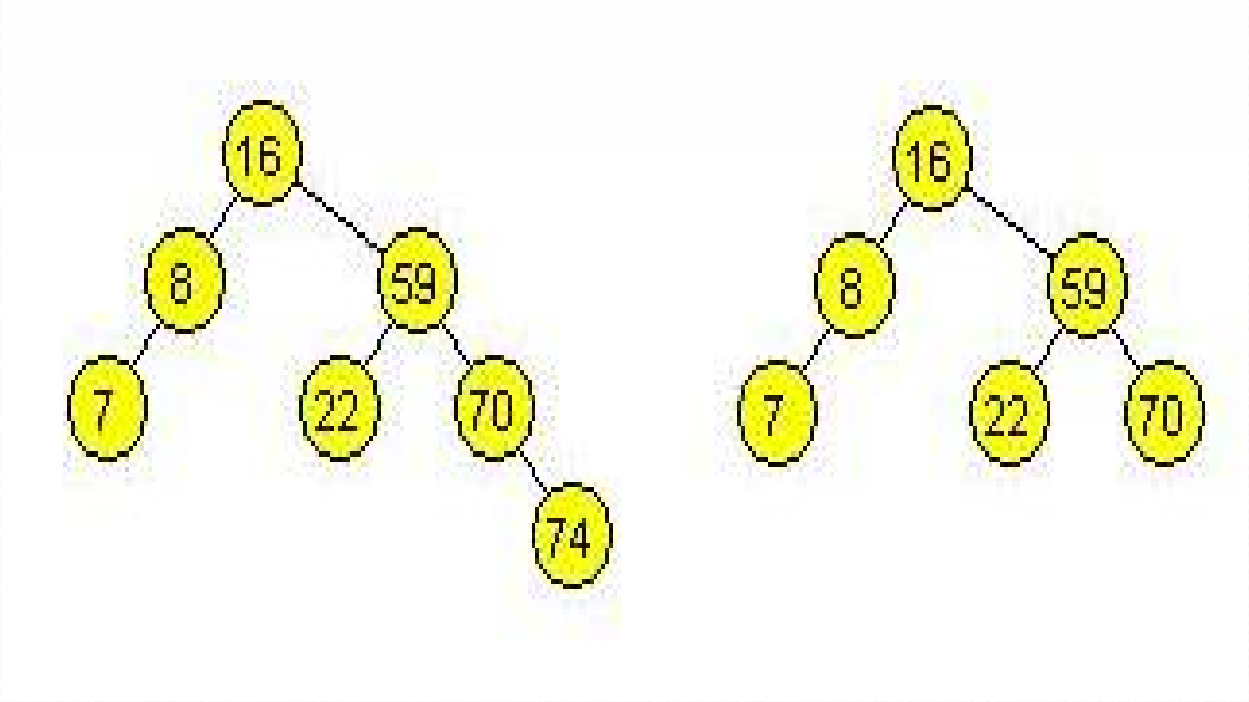
- Padre = NULL
- nodo = Raiz
- Bucle: mientras actual no sea un árbol vacío o hasta que se encuentre el elemento.
  - Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo: Padre=nodo, nodo=nodo->izquierdo.
  - Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho: Padre=nodo, nodo=nodo->derecho.
- Si nodo no es NULL, el elemento está en el árbol, por lo tanto salimos.
- Si Padre es NULL, el árbol estaba vacío, por lo tanto, el nuevo árbol sólo contendrá el nuevo elemento, que será la raíz del árbol.
- Si el elemento es menor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol izquierdo de Padre.
- Si el elemento es mayor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol derecho de Padre.

```
void Insertar(Arbol *a, int dat) {
   pNodo padre = NULL;
   pNodo actual = *a;
   while(!Vacio(actual) && dat != actual->dato) {
      padre = actual;
      if(dat < actual->dato) actual = actual->izquierdo;
       else if(dat > actual->dato)
      actual = actual->derecho;
   }
      if(!Vacio(actual))
      return;
      if(Vacio(padre))
      *a = (Arbol)malloc(sizeof(tipoNodo));
      (*a)->dato = dat;
      (*a)->izquierdo = (*a)->derecho = NULL;
      }
         else
         if(dat < padre->dato) {
      actual = (Arbol)malloc(sizeof(tipoNodo));
      padre->izquierdo = actual;
      actual->dato = dat;
      actual->izquierdo = actual->derecho = NULL;
       } else
      if(dat > padre->dato) {
         actual = (Arbol)malloc(sizeof(tipoNodo));
         padre->derecho = actual;
         actual->dato = dat;
         actual->izquierdo = actual->derecho = NULL;
    }
 }
```

Árboles de búsqueda

Supresión de un nodo

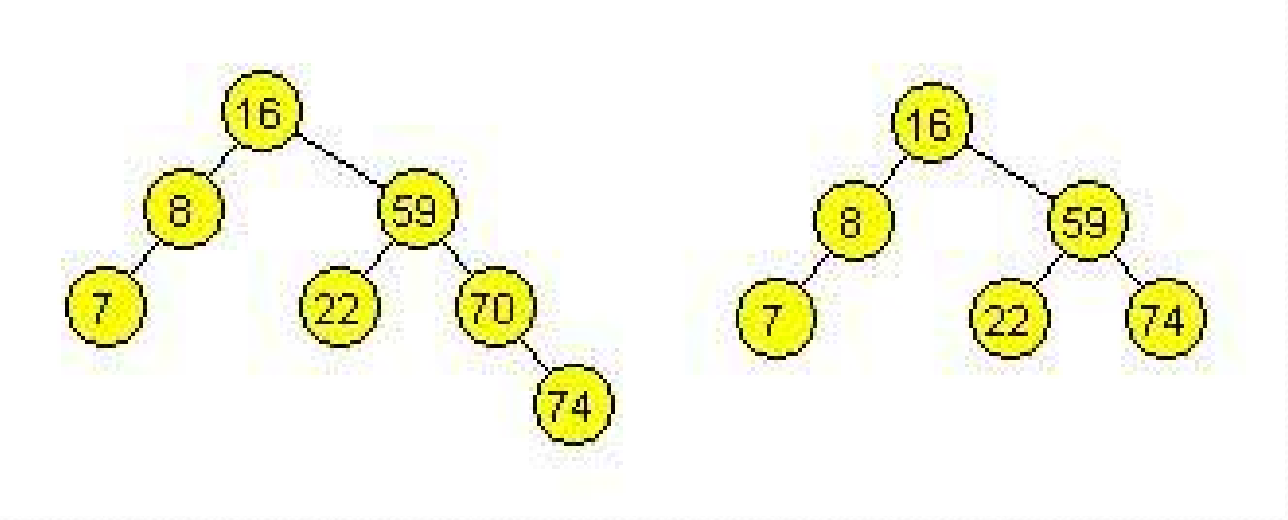
Una hoja del árbol: Si el nodo por eliminar es una hoja, entonces basta con destruir su variable asociada (Delete) y, posteriormente, asignar nil a ese puntero.



Árboles de búsqueda

Supresión de un nodo

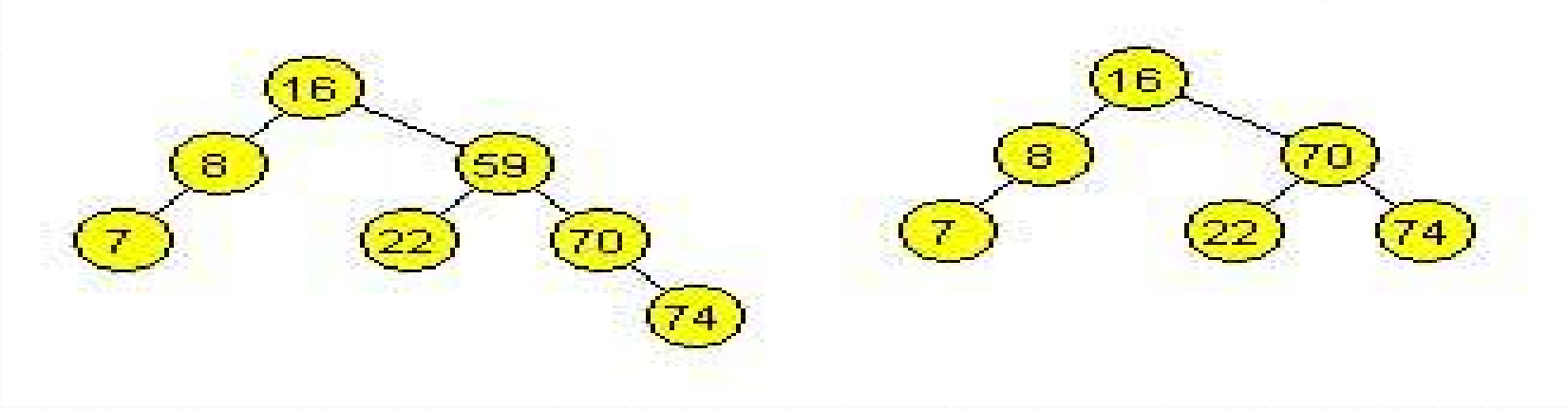
Un nodo con un sólo hijo: Si el nodo por eliminar sólo tiene un subárbol, se usa la misma idea que al eliminar un nodo interior de una lista: hay que "*saltarlo*" conectando directamente el nodo anterior con el nodo posterior y desechando el nodo por eliminar.



Árboles de búsqueda

Supresión de un nodo

C) Un nodo con dos hijos: consiste en eliminar el nodo deseado y recomponer las conexiones de modo que se siga teniendo un árbol de búsqueda. En primer lugar, hay que considerar que el nodo que se coloque en el lugar del nodo eliminado tiene que ser mayor que todos los elementos de su subárbol izquierdo, luego la primera tarea consistirá en buscar tal nodo; de éste se dice que es el predecesor del nodo por eliminar (>en qué posición se encuentra el nodo predecesor?). Una vez hallado el predecesor el resto es bien fácil, sólo hay que copiar su valor en el nodo por eliminar y desechar el nodo predecesor.



Árboles de búsqueda

Se presenta un esbozo en seudocódigo del algoritmo en cuestión; la implementación en C se deja como ejercicio

```
Determinar el número de hijos del nodo N a eliminar
si N no tiene hijos entonces
   eliminarlo
en otro caso si N sólo tiene un hijo H entonces
   Conectar H con el padre de N
en otro caso si N tiene dos hijos entonces
   Buscar el predecesor de N
   Copiar su valor en el nodo a eliminar
   Desechar el nodo predecesor
```

Eliminar un elemento de un árbol ABB

- Padre = NULL
- Si el árbol está vacío: el elemento no está en el árbol, por lo tanto salimos sin eliminar ningún elemento.
- Si el valor del nodo raíz es igual que el del elemento que buscamos, estamos ante uno de los siguientes casos:
  - El nodo raíz es un nodo hoja:
    - Si 'Padre' es NULL, el nodo raíz es el único del árbol, por lo tanto el puntero al árbol debe ser NULL.
    - Si raíz es la rama derecha de 'Padre', hacemos que esa rama apunte a NULL.
    - Si raíz es la rama izquierda de 'Padre', hacemos que esa rama apunte a NULL.
    - Eliminamos el nodo, y salimos.
  - El nodo no es un nodo hoja:
    - Buscamos el 'nodo' más a la izquierda del árbol derecho de raíz o el más a la derecha del árbol izquierdo. Hay que tener en cuenta que puede que sólo exista uno de esos árboles. Al mismo tiempo, actualizamos 'Padre' para que apunte al padre de 'nodo'.
    - Intercambiamos los elementos de los nodos raíz y 'nodo'.
    - Borramos el nodo 'nodo'. Esto significa volver a (3), ya que puede suceder que 'nodo' no sea un nodo hoja.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

```
void Borrar(Arbol *a, int dat) {
 pNodo padre = NULL; /* (1) */
 pNodo actual;
 pNodo nodo;
 int aux;
 actual = *a;
 while(!Vacio(actual)) {
/* Búsqueda (2) else implícito */
     if(dat == actual->dato) {
     if(EsHoja(actual)) {
          if(padre)
               if(padre->derecho == actual)
      padre->derecho = NULL;
     else if(padre->izquierdo == actual)
     padre->izquierdo = NULL;
     free(actual);
     actual = NULL;
     return;
}
     else {
     padre = actual;
if(actual->derecho) {
 nodo = actual->derecho;
 while(nodo->izquierdo) {
 padre = nodo;
 nodo = nodo->izquierdo;
 }
 }
else {
 nodo = actual->izquierdo;
 while(nodo->derecho) {
 padre = nodo;
 nodo = nodo->derecho;
 }
 }
aux = actual->dato;
actual->dato = nodo->dato;
 nodo->dato = aux;
 actual = nodo;
     }
 }
else {
padre = actual;
if(dat > actual->dato)
actual = actual->derecho;
else
if(dat < actual->dato)
actual = actual->izquierdo;
          }
     }
}
```

Comprobar si el árbol está vacío

```
int Vacio(Arbol r) {
 return r==NULL;
 }
```

Contar número de nodos

- Para contar los nodos podemos recurrir a cualquiera de los tres modos de recorrer el árbol: inorden, preorden o postorden y como acción incrementamos el contador de nodos. Para implementar este algoritmo recurrimos a dos funciones:

```
int NumeroNodos(Arbol a, int *contador)
{
 *contador = 0;
 auxContador(a, contador);
 return *contador;
}

void auxContador(Arbol nodo, int *c) {
(*c)++; /* Acción: incrementar número de nodos. (Preorden) */
if(nodo->izquierdo)
auxContador(nodo->izquierdo, c);
 /* Rama izquierda */
if(nodo->derecho)
auxContador(nodo->derecho, c); /* Rama derecha */
}
```

Arboles binarios

Recorrido de un árbol binario: EN PROFUNDIDAD y EN AMPLITUD

Definir un algoritmo de recorrido de un árbol binario no es una tarea directa ya que, al no ser una estructura lineal, existen distintas formas de recorrerlo.

En particular, al llegar a un nodo podemos realizar una de las tres operaciones siguientes:

Leer el valor del nodo.

Seguir por el hijo izquierdo.

Seguir por el hijo derecho.

El orden en el que se efectúen las tres operaciones anteriores determinaría el orden en el que los valores de los nodos del árbol son leídos. Si se postula que siempre se leerá antes el hijo izquierdo que el derecho, entonces existen tres formas distintas de recorrer un árbol:

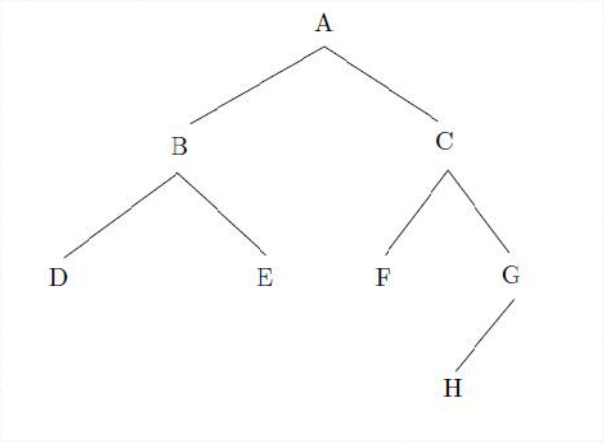
Árboles binarios

Preorden: Primero se lee el valor del nodo y después se recorren los subárboles.

Esta forma de recorrer el árbol también recibe el nombre de recorrido primero en profundidad. Se leerá así: ABDECFGH.

Inorden: En este tipo de recorrido, primero se recorre el subárbol izquierdo, luego se lee el valor del nodo y, finalmente, se recorre el subárbol derecho. Se leerá así: DBEAFCHG.

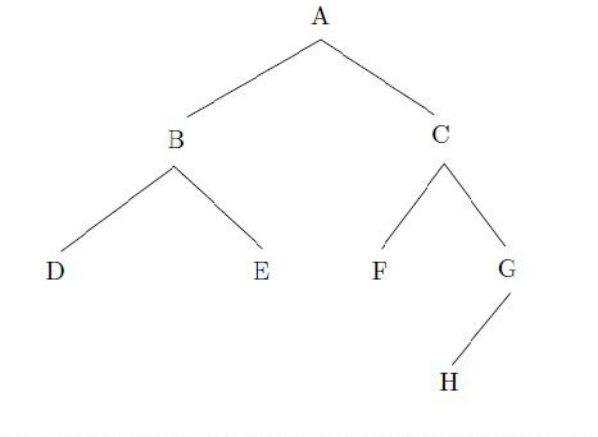
Postorden: En este caso, se visitan primero los subárboles izquierdo y derecho y después se lee el valor del nodo. Se leerá así: DEBFHGCA.



Árboles binarios

Un procedimiento recursivo para recorrer el árbol en profundidad (en Pre - Orden) sería:

```
void PreOrden(Arbol a, void
(*func)(int*)) { func(&(a->dato));
/* Aplicar la función al dato del nodo actual */
if(a->izquierdo) PreOrden(a-
>izquierdo, func);
/* Subárbol izquierdo */
if(a->derecho) PreOrden(a->derecho,
func);
/* Subárbol derecho */
}
```

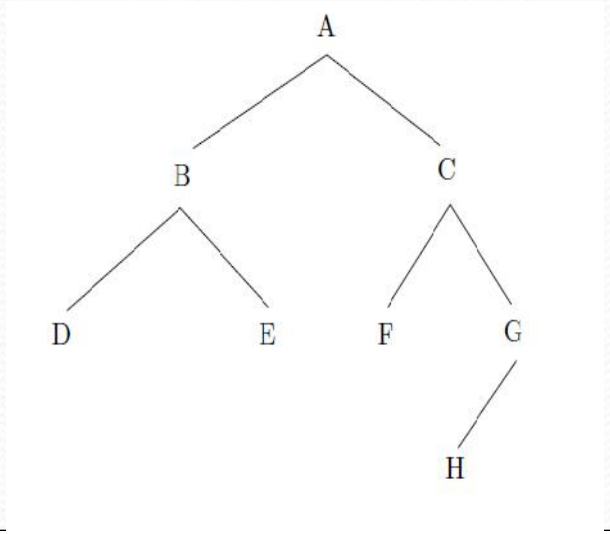


Se leerá así: ABDECFGH

Árboles binarios

Un procedimiento recursivo para recorrer el árbol en profundidad (en In Orden) sería:

```
void InOrden(Arbol a, void (*func)(int*)) {
if(a->izquierdo) InOrden(a->izquierdo, func);
/* Subárbol izquierdo */
func(&(a->dato));
/* Aplicar la función al dato del nodo actual */
if(a->derecho) InOrden(a->derecho, func);
/* Subárbol derecho */
}
```

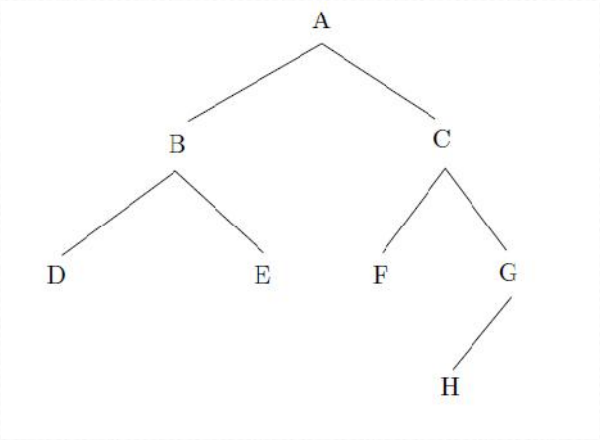


Se leerá así: DBEAFCHG.

Árboles binarios

Recorrido en profundidad (postorden): se visitan primero el subárbol izquierdo, después el subárbol derecho, y por último el nodo actual.

```
void PostOrden(Arbol a, void (*func)(int*)) {
if(a->izquierdo) PostOrden(a->izquierdo, func);
/* Subárbol izquierdo */
if(a->derecho) PostOrden(a->derecho, func);
/* Subárbol derecho */
func(&(a->dato));
/* Aplicar la función al dato del nodo actual */
}
```



Se leerá así: DEBFHGCA

La ventaja del recorrido en postorden es que permite borrar el árbol de forma consistente. Es decir, si visitar se traduce por borrar el nodo actual, al ejecutar este recorrido se borrará el árbol o subárbol que se pasa como parámetro. La razón para hacer esto es que no se debe borrar un nodo y después sus subárboles, porque al borrarlo se pueden perder los enlaces, y aunque no se perdieran se rompe con la regla de manipular una estructura de datos inexistente. Una alternativa es utilizar una variable auxiliar, pero es innecesario aplicando este recorrido.

Recorrido en amplitud (Arbol – Grafo)

Consiste en ir visitando el árbol por niveles. Primero se visitan los nodos de nivel 1 (como mucho hay uno, la raíz), después los nodos de nivel 2, así hasta que ya no queden más.

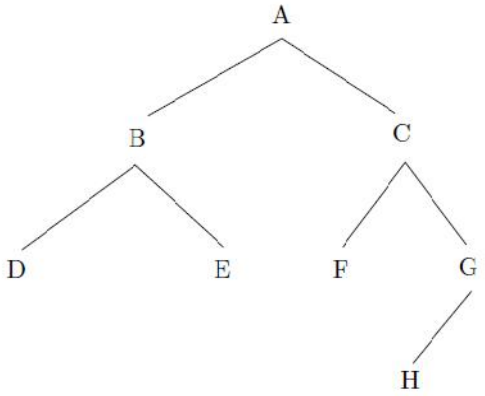
En este caso el recorrido no se realizará de forma recursiva sino iterativa, utilizando una cola como estructura de datos auxiliar.

El procedimiento consiste en encolar (si no están vacíos) los subárboles izquierdo y derecho del nodo extraido de la cola, y seguir desencolando y encolando hasta que la cola esté vacía.

En la codificación que viene a continuación no se implementan las operaciones sobre colas.

```
procedure amplitud(a : tArbol);
var
  cola : tCola;  { las claves de la cola serán de tipo árbol binario }
  aux : tArbol;

begin
  if a <> NIL then begin
    CrearCola(cola);
    encolar(cola, a);
    While Not colavacia(cola) Do begin
      desencolar(cola, aux);
      visitar(aux);
      if aux^.izq <> NIL encolar(cola, aux^.izq);
      if aux^.der <> NIL encolar(cola, aux^.der)
    end
  end
end;
```



Se leería: ABCDEFGH

Resumen …

1.- DEFINICIÓN DE LA ESTRUCTURA DE DATOS ÁRBOL: Es una estructura no lineal de datos homogéneos tal que establece una jerarquía entre sus elementos.

2.- ÁRBOLES BINARIOS: Es un árbol que o bien esta vacío, o bien esta formado por un nodo o elemento Raíz y dos arboles binarios, llamados subarbol izquierdo y subarbol derecho. Es un árbol que tiene o 0,1, 2 hijos su nodo. Como máximo dos hijos por elem.

TERMINOLOGÍA:

NODO: cada uno de los elementos de un árbol.

SUCESORES DE UN NODO: son los elementos de su subarbol izquierdo y de su subarbol derecho.

NODO HIJO: son los sucesores directos de un Nodo.

NODO TERMINAL O NODO HOJA: es aquel que no tiene hijos.

NIVEL DE UN NODO: es un número entero que se define como o para la raíz y uno más que el nivel de su padre para cualquier otro nodo. n=o Para la Raíz. para cualquier otro nodo 1+ el nivel de su padre.

RAMA: es cualquier camino que se establece entre la raíz y un nodo terminal.

ALTURA O PROFUNDIDAD DE UN ARBOL: es el máximo nivel de los nodos de un árbol que coincide con el número de nodos de la rama más larga menos 1(-1)

Grafos

Un grafo es un objeto matemático que se utiliza para representar circuitos, redes, etc. Los grafos son muy utilizados en computación, ya que permiten resolver problemas muy complejos.

Los grafos, constituyen estructuras de datos en las que se pueden expresar relaciones de conexión entre diversos elementos denominados vértices.

Cada conexión se representa por un dato llamado arista

El estudio de grafos es una rama de la algoritmia muy importante.

Los grafos tienen gran cantidad de aplicaciones:

- Representación de circuitos electrónicos analógicos y digitales
- Representación de caminos o rutas de transporte entre localidades
- Representación de redes de computadores.

Uno de los problemas más importantes en los grafos es el de encontrar el camino de coste mínimo.

Definición

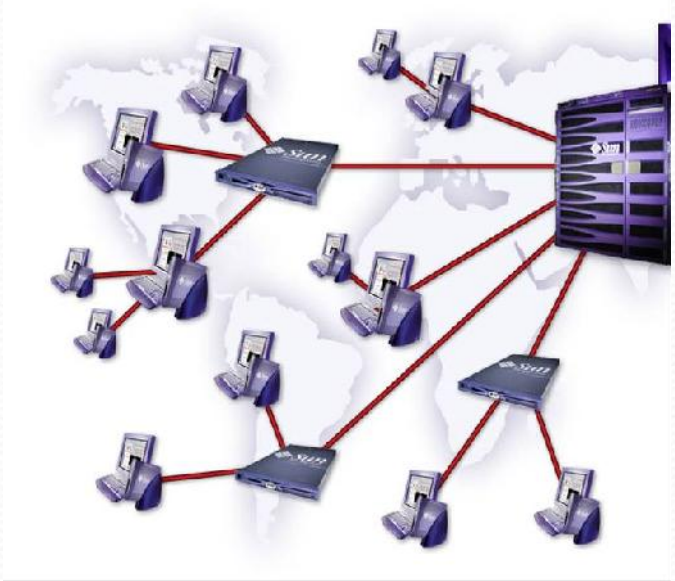
Son estructuras de datos NO lineales.

Se los puede definir como un conjunto de puntos o nodos, y un conjunto de lìneas o aristas, tal que cada una de ellas une un punto con otro punto.

Conjunto de puntos X = x1, x2, x3, … xn)
Conjunto de lineas o aristas se denota con la siguiente expresion

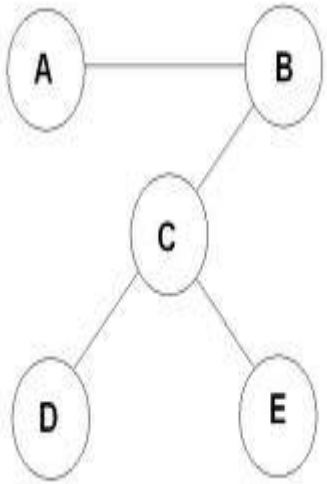
L = { Lij / Xi y Xj estan conectados}

Por lo tanto el grafo se define como el conjunto de nodos X, y las relaciones entre los mismos, L.



Glosario

- Un grafo consta de *vértices* (o nodos) y *aristas*.
- Los vértices son objetos que contienen información y las aristas son conexiones entre vértices.
- Para representarlos, se suelen utilizar puntos para los vértices y líneas para las conexiones, aunque hay que recordar siempre que la definición de un grafo no depende de su representación.
- Un *camino* entre dos vértices es una lista de vértices en la que dos elementos sucesivos están conectados por una arista del grafo. Así, el camino ABCD es un camino que comienza en el vértice A y pasa por los vértices B y C (en ese orden) y al final va del C al D.
- El grafo será *conexo* si existe un camino desde cualquier nodo del grafo hasta cualquier otro. Si no es conexo constará de varias *componentes conexas*.



Glosario

Un *camino simple* es un camino desde un nodo a otro en el que ningún nodo se repite (no se pasa dos veces).

Si el camino simple tiene como primer y último elemento al mismo nodo se denomina *ciclo*.

Cuando el grafo no tiene ciclos tenemos un *árbol*.

Varios árboles independientes forman un *bosque*.

Un *árbol de expansión* de un grafo es una reducción del grafo en el que solo entran a formar parte el número mínimo de aristas que forman un árbol y conectan a todos los nodos.

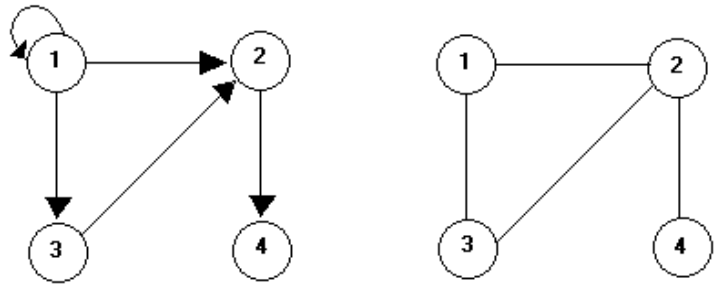
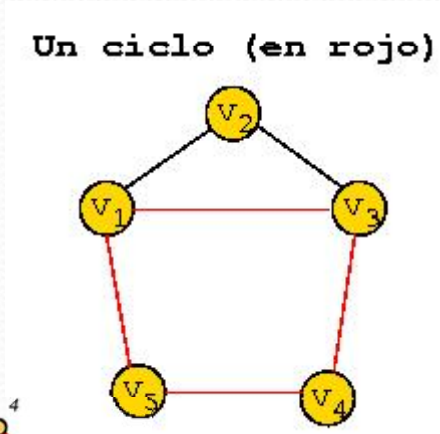
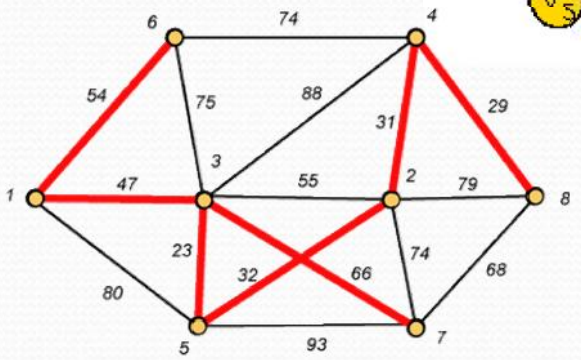


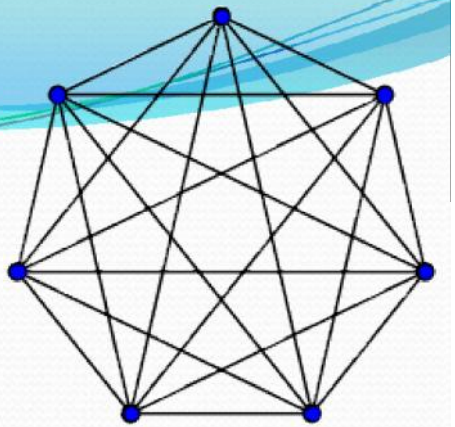
Figura 2:Ejemplos simples de un grafo dirigido y uno no dirigido.

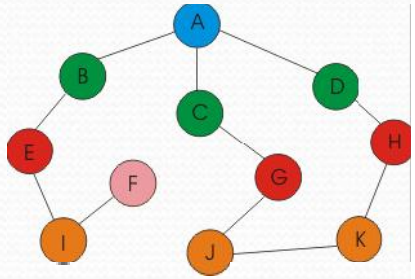


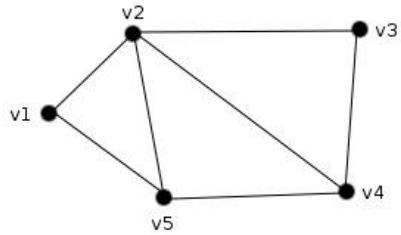


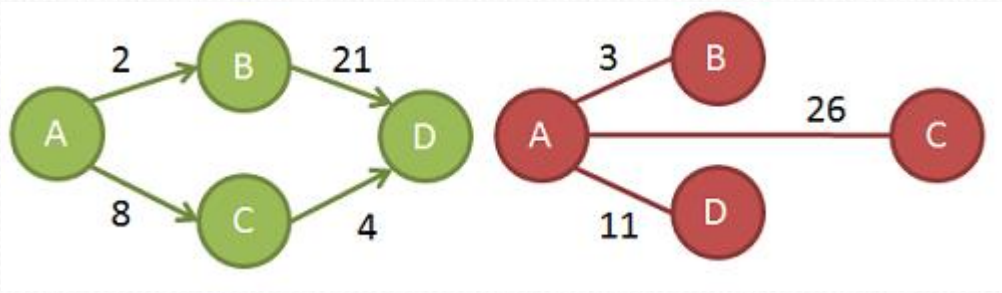
Glosario

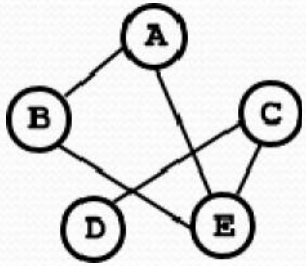
- Según el número de aristas que contiene:
  - Un grafo es *completo* si cuenta con todas las aristas posibles (es decir, todos los nodos están conectados con todos)
  - *Un grafo es disperso* si tiene relativamente pocas aristas.
  - Un grafo es *denso* si le faltan pocas para ser completo.
- Grafos NO dirigidos: Las aristas son la mayor parte de las veces bidireccionales, es decir, si una arista conecta dos nodos A y B se puede recorrer tanto en sentido hacia B como en sentido hacia A.
- Grafo dirigido: Sin embargo, en ocasiones tenemos que las uniones son unidireccionales. Estas uniones se suelen dibujar con una flecha.
- Grafo dirigido y ponderado: Cuando las aristas llevan un coste asociado (un entero al que se denomina *peso*) el grafo es *ponderado*.
- Una *red* es un grafo dirigido y ponderado.

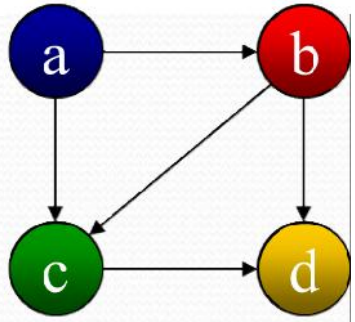












Representaciones

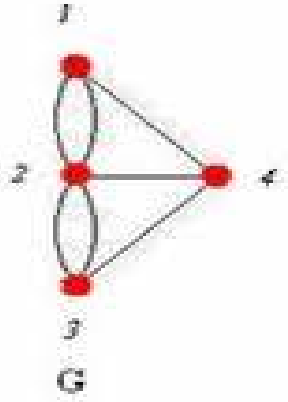
- Una característica especial en los grafos es que podemos representarlos utilizando dos estructuras de datos distintas.
- En los algoritmos que se aplican sobre ellos veremos que adoptarán tiempos distintos dependiendo de la forma de representación elegida.
- En particular, los tiempos de ejecución variarán en función del número de vértices y el de aristas, por lo que la utilización de una representación u otra dependerá en gran medida de si el grafo es denso o disperso.

Representación por matriz de adyacencia

Representación por lista de adyacencia

Representación por matriz de adyacencia

- Es la forma más común de representación y la más directa.
- Consiste en una tabla de tamaño V x V, en que la que a[i, j] tendrá como valor 1 si existe una arista del nodo i al nodo j. En caso contrario, el valor será 0.
- Cuando se trata de grafos ponderados en lugar de 1 el valor que tomará será el peso de la arista.
- Si el grafo es no dirigido hay que asegurarse de que se marca con un 1 (o con el peso) tanto la entrada a[i, j] como la entrada a[j, i], puesto que se puede recorrer en ambos sentidos.



$$A(G) = \begin{pmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Representación por matriz de adyacencia

```
int V,A;
int a[maxV][maxV];
void inicializar() {
 int i,x,y,p;
 char v1,v2;
 // Leer V y A
memset(a,0,sizeof(a));
 for (i=1; i<=A; i++) {
    scanf("%c %c%d\n",&v1,&v2,&p);
    x= v1-'A';
    y=v2-'A';
    a[x][y]=p;
    a[y][x]=p;
    }
 }
```

Representación por lista de adyacencia

Otra forma de representar un grafo es por medio de listas que definen las aristas que conectan los nodos.

Lo que se hace es definir una lista enlazada para cada nodo, que contendrá los nodos a los cuales es posible acceder.

Es decir, un nodo A tendrá una lista enlazada asociada en la que aparecerá un elemento con una referencia al nodo B si A y B tienen una arista que los une.

Obviamente, si el grafo es no dirigido, en la lista enlazada de B aparecerá la correspondiente referencia al nodo A.

Las listas de adyacencia serán estructuras que contendrán un valor entero (el número que identifica al nodo destino), así como otro entero que indica el coste en el caso de que el grafo sea ponderado.

El conjunto de vertices se representa como un vector, el cual tiene un número máximo de posiciones que es igual al número máximo de vertices del grafo . Tambien se llena con valores booleanos .

1	2	3	4	5	6
T	T	T	T	T	T

Representación por lista de adyacencia

```
struct nodo { int v; int p; nodo *sig; };
int V,A; // vértices y aristas del grafo struct nodo
*a[maxV], *z;
void inicializar() {
int i,x,y,peso;
char v1,v2;
struct nodo *t;
z=(struct nodo*)malloc(sizeof(struct nodo));
z->sig=z;
for (i=0; i<V; i++)
   a[i]=z; for (i=0; i<A; i++) {
   scanf("%c %c %d\n",&v1,&v2,&peso);
   x=v1-'A'; y=v2-'A';
   t=(struct nodo *)malloc(sizeof(struct nodo));
   t->v=y;
   t->p=peso;
   t->sig=a[x];
   a[x]=t;
   t=(struct nodo *)malloc(sizeof(struct nodo));
   t->v=x; t->p=peso;
   t->sig=a[y];
   a[y]=t;
   }
}
```

Operaciones: PROCEDIMIENTO GRAFOVACIO O INICIAR GRAFO .

Estática .

```
PROCEDURE GRAFO_VACIO ( VAR GRAFO : TIPOGRAFO ) ;
VAR
X , Y : INTEGER ;
BEGIN
     FOR X := 1 TO N DO
     BEGIN
     GRAFO.VERTICES [ X ] := FALSE ;
                    FOR I:= 1 TO N DO
                    BEGIN
                    GRAFO.ARCOS [ X , Y ]:= FALSE ;
                    END ;
     END ;
END ;
```

Dinamica

```
PROCEDURE GRAFO_VACIO ( VAR GRAFO : TIPOGRAFO ) ;
BEGIN
          GRAFO ^ . SIG:= NIL ;
          GRAFO ^ . ADYA := NIL ;
END ;
```

AÑADIR VERTICE y arco

Estática

```
PROCEDURE AÑADE_VER ( VAR GRAFO : TIPOGRAFO ; VERT :  TIPOVERTICE ) ;
BEGIN
    GRAFO . VERTICE [ VERT ] := TRUE ;
END ;
```

```
PROCEDURE AÑADE_ARC ( VAR GRAFO : TIPOGRAFO ; ARC : TIPOARCO ) ;
BEGIN
      IF ( GRAFO . VERTICES [ ARC . ORIGEN ] = TRUE )   AND
         ( GRAFO . VERTICES [ ARC . DESTINO ] = TRUE ) THEN
           GRAFO . ARCOS [ ARC . ORIGEN , ARC . DESTINO ] := TRUE ;
END ;
```

3.4.- BORRAR VERTICE y arco

Estática .

```
PROCEDURE BORRA_VER ( VAR GARFO : TIPOGRAFO ; VER :
   TIPOVERTICE ) ;
BEGIN
   GRAFO . VERTICE [ VER ] := FALSE ;
END ;
```

```
PROCEDURE BORRA_ARC ( VAR GRAFO : TIPOGRAFO ; ARC :
   TIPOARCO ) ;
BEGIN
   GARFO . ARCOS [ ARC . ORIGEN , ARC . DESTINO ] := FALSE ;
END ;
```

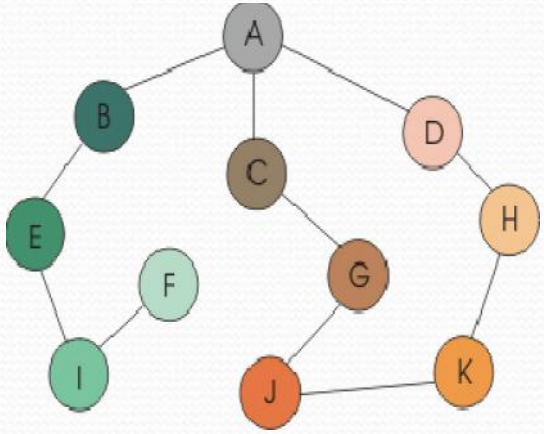
```
PROCEDURE BORRA_ARC (VAR GRAFO : TIPOGRAFO ; ARC :
TIPOARCO ) ;
VAR
POS1 , POS2 : TIPOGRAFO ;
AUX , ANT : PUNTEROARCO ;
ENC : BOOLEAN ;
BEGIN
BUSCAR ( GRAFO , ARC . ORGIGEN , POS1 ) ; --- Buscamos el origen
IF ( POS1 <> NIL ) THEN
BEGIN
BUSCAR ( GRAFO , ARC . DESTINO , POS2 ); --- Buscamos destino
IF ( POS2 <>NIL ) THEN
BEGIN
IF ( POS1^. ADYA^. INFO = ARC. DESTINO ) THEN
BEGIN
AUX := POS1 ^. ADYA ;
Elimimamos sí es el 1º ---------- POS1 ^. ADYA := AUX ^. ADYA ;
DISPOSE ( AUX ) ;
END ;
ELSE
BEGIN
ANT := POS1 ^. ADYA ;
AUX := ANT ^. ADYA ;
ENC := FALSE ;
WHILE ( AUX <> NIL ) AND ( NOT ENC ) DO
BEGIN
IF ( AUX ^. INFO = ARC . DESTINO ) THEN BEGIN
ENC := TRUE ;
ANT ^. ADYA := AUX ^. ADYA ;
DISPOSE ( AUX ) ;
END;
ELSE
BEGIN
ANT := AUX ;
AUX := AUX ^. ADYA ;
END;
END;
END;
END;
END;
```

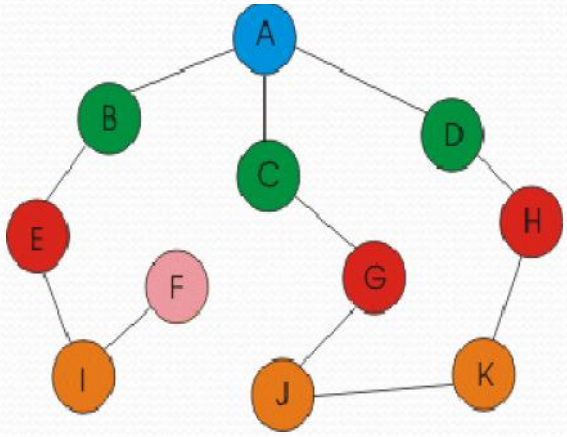
BORRAR VERTICE y arco Dinamica

Exploración de grafos

Recorrer un grafo significa pasar por todos sus vertices y procesar la información que de esto se desprende, dependiendo del problema que se nos planteó . Este recorrido se puede hacer de dos maneras distintas.

- En profundidad. Una forma sencilla de recorrer los vértices es mediante una función recursiva, cuya base es la estructura de datos pila.
- En anchura. La sustitución de la recursión (pila) por una cola nos proporciona el segundo método de búsqueda o recorrido, la búsqueda en amplitud o anchura
- Si estan almacenados en orden alfabético, tenemos que el orden que seguiría el recorrido en profundidad sería el siguiente: A-B-E-I-F-C-G-J-K-H-D.
- En un recorrido en anchura el orden sería :A-B-C-D-E-G-H-I-J-K-F





Búsqueda en profundidad

- Se implementa de forma recursiva, aunque también puede realizarse con una pila.

- Se utiliza un array val para almacenar el orden en que fueron explorados los vértices. Para ello se incrementa una variable global id (inicializada a o) cada vez que se visita un nuevo vértice y se almacena id en la entrada del array val correspondiente al vértice que se está explorando.

- La siguiente función realiza un máximo de V (el número total de vértices) llamadas a la función visitar, que implementamos aquí en sus dos variantes: representación por matriz de adyacencia y por listas de adyacencia.

Búsqueda en amplitud o anchura

- La diferencia fundamental respecto a la búsqueda en profundidad es el cambio de estructura de datos: una cola en lugar de una pila.

- En esta implementación, la función del array val y la variable id es la misma que en el método anterior.

Bibliografía

Libros

ALGORITMOS, DATOS Y PROGRAMAS con aplicaciones en Pascal, Delphi y Visual Da Vinci. De Guisti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2. Capitulo 9.

PROGRAMACIÓN; Castor F. Herrmann,María E. Valesani.; 2001; Editorial: MOGLIA S.R.L..ISBN: 9874338326. Capitulo 4. Estructuras de Datos No Lineales.

Internet

Algoritmos y Programación en Pascal. Cristóbal Pareja Flores y Otros. Cap. 16, y 17.