

Introducción. Árboles generales. Terminología. Representación de un árbol. Árboles binarios. Equilibrio. Árboles binarios completos. Estructura de un árbol binario. Recorrido de un árbol. Profundidad de un árbol binario. Árbol binario de búsqueda. Creación de un Árbol binario de búsqueda. Implementación de un nodo en arbol binario de búsqueda. Operaciones en árbol binario de búsqueda: búsqueda, inserción, recorrido y eliminación. Ejemplos. Grafos. Partes de un grafo. Terminología. Tipos de grafos. Representación. Matriz de adyacencia. Ejemplo de aplicación.



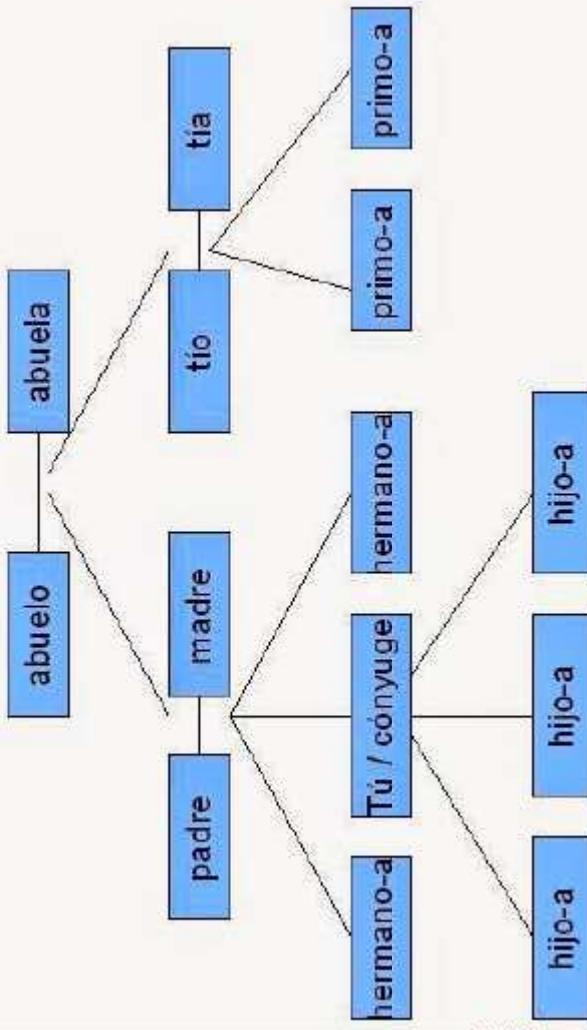


El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales* al contrario de los arrays y las listas enlazadas que constituyen *estructuras lineales*.

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 este apartado se estudiará el concepto de árbol general y los tipos de árboles más usuales, binario y binario de búsqueda.

## Árbol genealógico



Desde el punto de vista conceptual, un árbol es un objeto que comienza con una raíz, y se extiende en varias ramificaciones o líneas, cada una de las cuales puede extenderse en ramificaciones hasta terminar finalmente en una hoja.

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 si a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general.

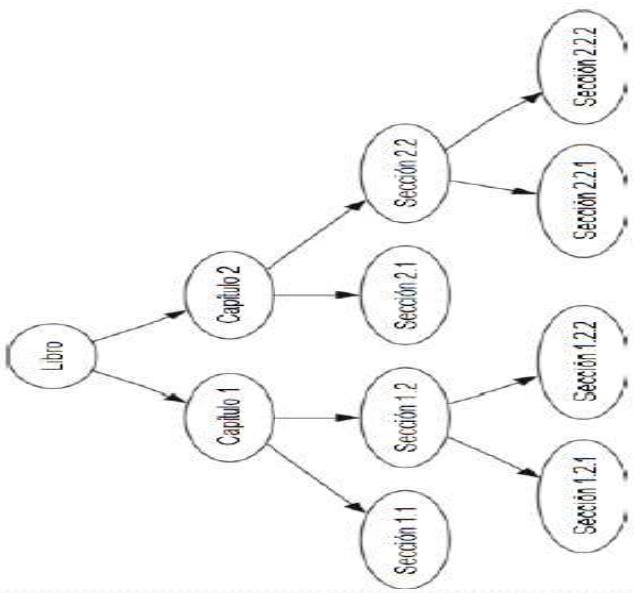
Un **árbol** consta de un conjunto finito de elemento, denominados **nodos** y un conjunto finito de líneas dirigidas, denominadas **ramas**, que conectan nodos. El número de ramas asociado con un nodo es el **grado** del nodo.

**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  $n > 0$  conjuntos disjuntos tales que  $T_1 \dots T_n$ , en donde cada uno de estos conjuntos es un **árbol**. A  $T_1 T_2 \dots T_n$  se le denomina **subárboles** de la raíz.

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

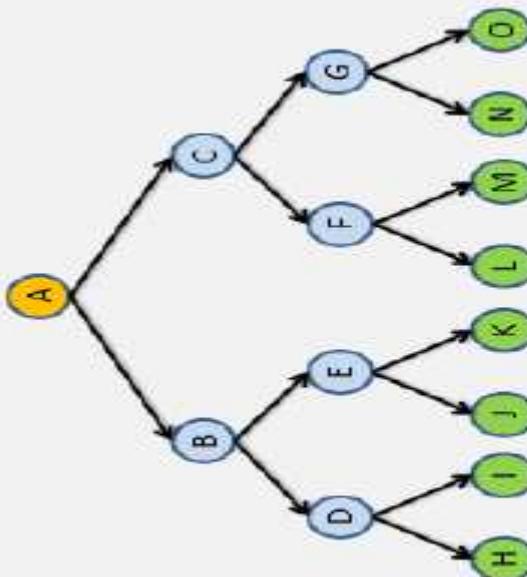


Un árbol es una estructura de datos jerarquizada aplicada sobre una colección de elementos u objetos (nodos). Que se puede definir en forma recursiva. Es, por tanto, una estructura no secuencial. Cada dato reside en un nodo, y existen relaciones de parentesco entre nodos.

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.

V.S.

Estructura jerárquica



Estructura lineal



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

## TERMINOLOGIA

**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 el 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, E, G, H

**Nodo interno:** aquel que tiene al menos un descendiente. Es decir, no es raíz ni hoja (F).

**Nivel:** de un nodo es su distancia al raíz. La raíz tiene una distancia 0 de sí misma. Los hijos del raíz están a un nivel 1, sus hijos están en el nivel 2 y así sucesivamente.

**Altura:** La altura o profundidad de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. En el ejemplo 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) (D E) (G H)

**Bosque:** Una colección de dos o más árboles.

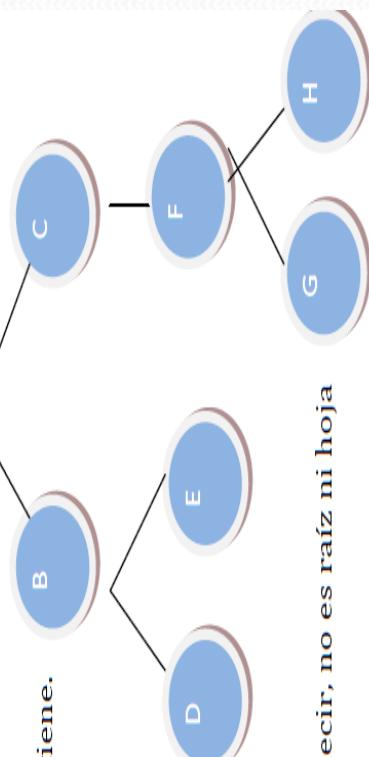
**Padre:** Todos los nodos tienen un solo padre a excepción del nodo raíz (no tiene padre. Ej.: A).

Todo nodo que no tiene descendientes directos (hijos), se conoce con el nombre de **terminal u hoja** (D, E, G, H)

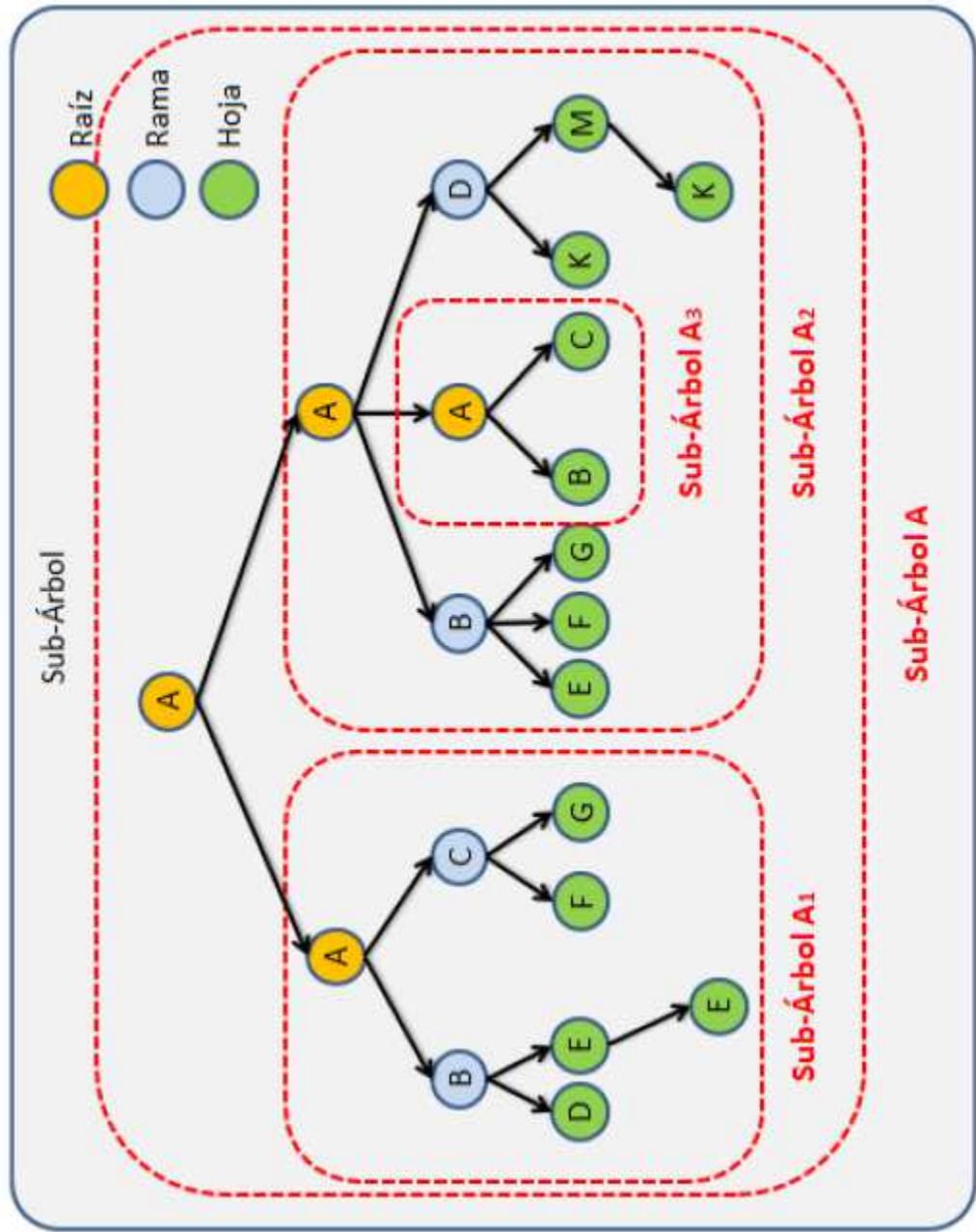
**Camino:** es una secuencia de nodos en los que cada nodo es adyacente al siguiente.  
Ejemplo (A, C, F).

**Longitud del camino:** n° de nodos que tiene

**Descendiente:** un nodo es descendiente de otro si hay un camino del segundo al primero

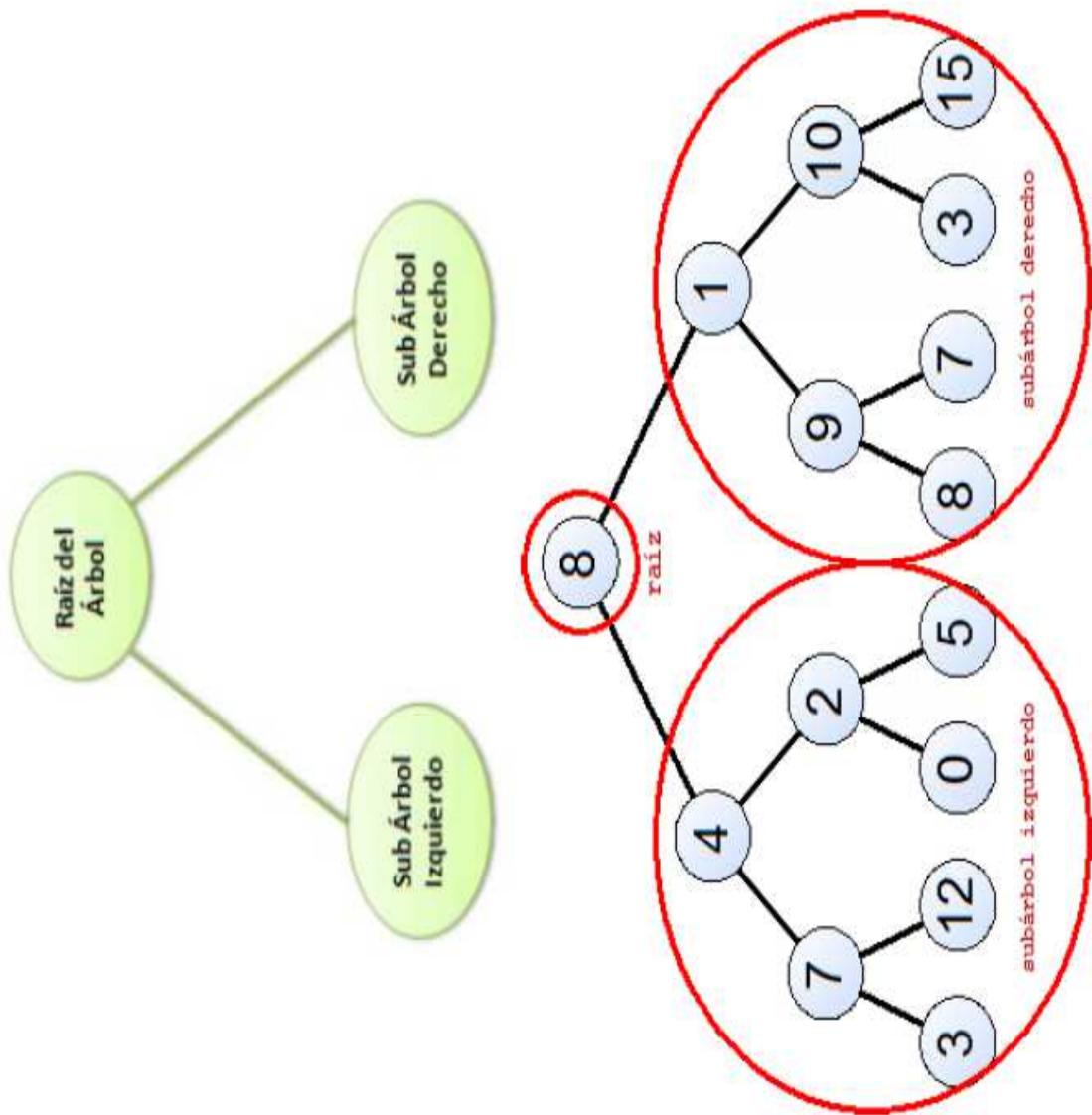


Un árbol se divide en **subárboles**. Un subárbol es cualquier estructura conectada por debajo del raíz. Cada nodo de un subárbol que se define por el nodo y todos los descendientes del nodo. El primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En el ejemplo (BDE) es un subárbol.



## ARBOLES 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 más hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha *hijo derecho*.



## ARBOLES BINARIOS

Un arbol binario es una estructura recursiva. Cada nodo es el raiz de su propio subarbol y tiene hijos, que son raices de arboles llamados los subarboles derecho e izquierdo del nodo, respectivamente.

Un arbol binario se divide en tres subconjuntos disjuntos:

- {R}            *Nodo raiz*
- {I1,I2,...In}    *Subarbol izquierdo de R*
- {D1,D2,...Dn}   *Subarbol derecho de R*

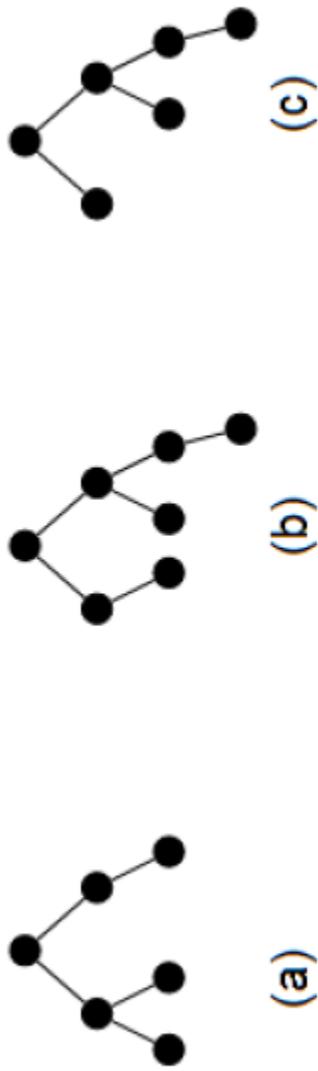
La **definición en "C"** del tipo árbol binario es sencilla: cada nodo del árbol va a tener dos punteros en lugar de uno.

```
typedef struct _nodo {  
    int dato;  
    struct _nodo *derecho;  
    struct _nodo *izquierdo;  
} tipoNodo;  
typedef tipoNodo *pNodo;  
typedef tipoNodo *Arbol;
```

## 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 solo un camino de bifurcación o de ramas, el que conduce al nodo deseado. De modo similar, los nodos de nivel 2 de un árbol solo pueden ser accedidos siguiendo solo dos ramos del árbol.

La característica anterior nos conduce a que 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 de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si definimos la altura del subárbol izquierdo como  $H_i$  y la altura del subárbol derecho como  $H_d$  entonces el factor de equilibrio del árbol B se determina por la siguiente fórmula:  $B = H_d - H_i$ .



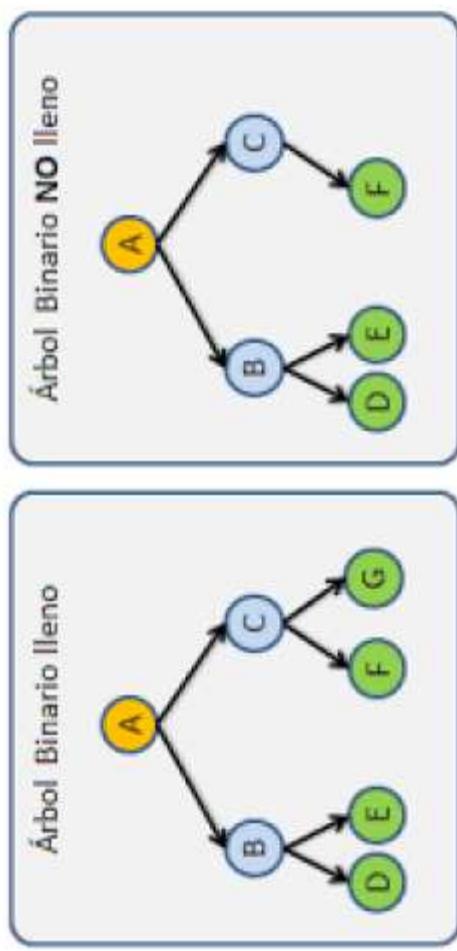
Utilizando esta fórmula el equilibrio del nodo raíz de los árboles de la figura anterior son:

- (a) 0, (b) 1, (c) 2

## ARBOLÉS BINARIOS COMPLETOS

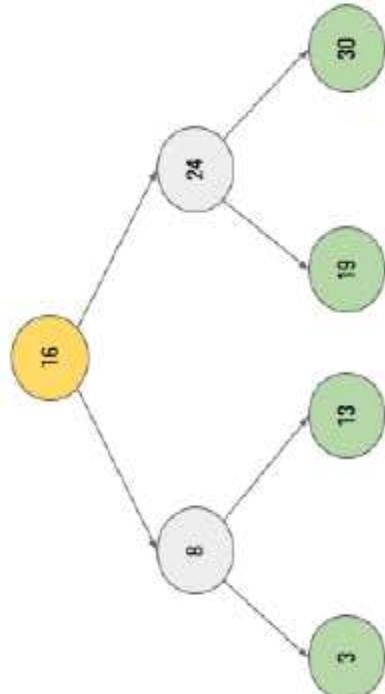
Un arbol binario **completo** de profundidad  $n$  es un arbol en el que para cada nivel, del 0 al nivel  $n-1$  tiene un conjunto lleno de nodos y todos los nodos hoja a nivel  $n$  ocupan las posiciones mas a la izquierda del arbol.

Un arbol binario completo que contiene  $2^n$  nodos a nivel  $n$  es un **arbol lleno**. Un arbol lleno es un arbol binario que tiene el maximo numero de entradas para su altura. Esto sucede cuando el ultimo nivel esta lleno.

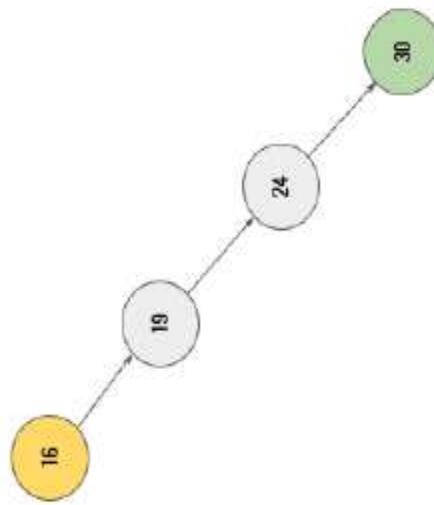


Llamamos **árbol degenerado** en el que hay un solo nodo hoja y cada nodo no hoja solo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

Árbol binario balanceado

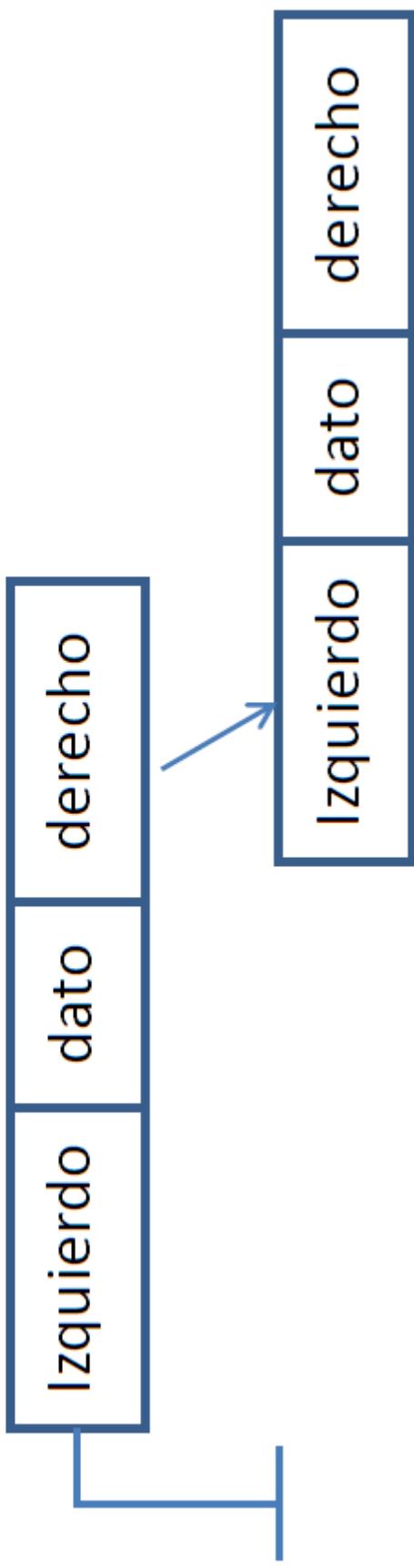


Árbol binario degenerado



## 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 al 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 árbol vacío.

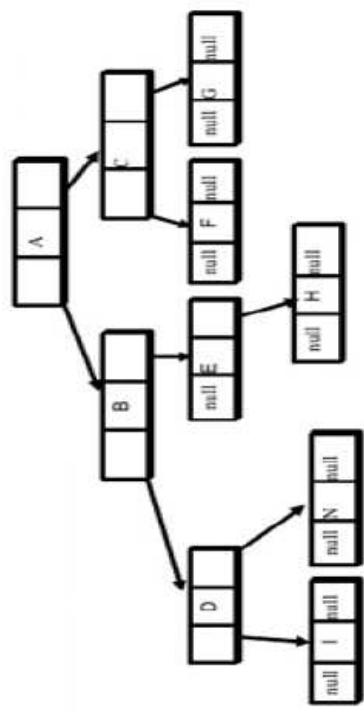


El algoritmo correspondiente a la estructura de un árbol es el siguiente:

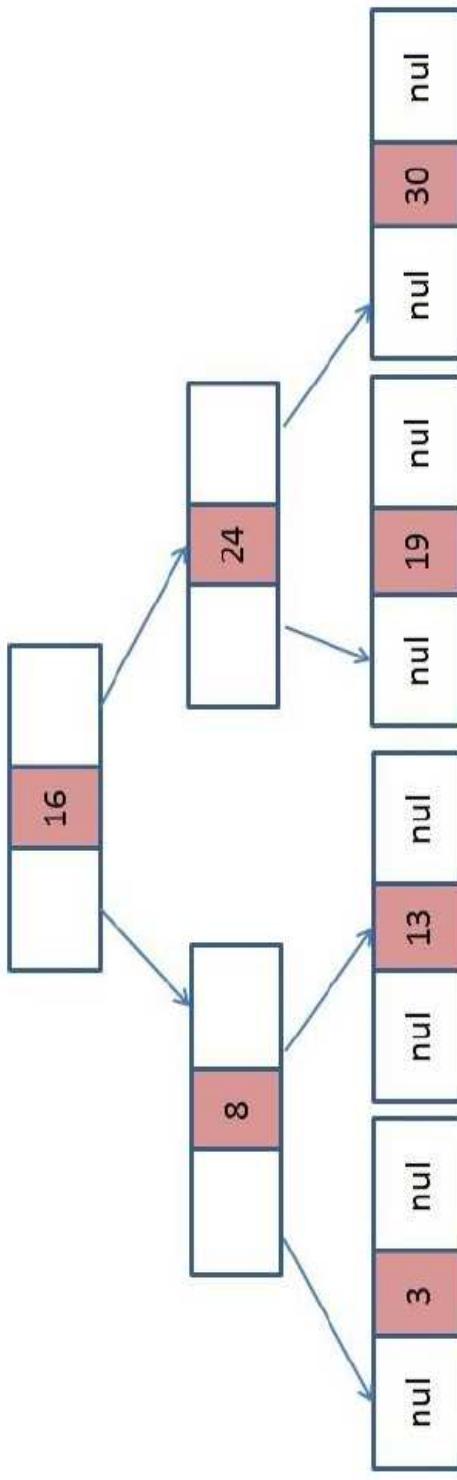
```
Nodo
    subarbolIzquierdo      <- Puntero a nodo
    datos                  <- Tipodato
    subarbolDerecho        <- Puntero a nodo
fin nodo
```

Ejemplo de árbol binario y su estructura de nodos

```
TypeDef int TipoElemento;
TypeDef struc nodo
{
    TipoElemento dato;
    Struct nodo *izq;
    Struct nodo *der;
} Nodo;
TypeDef Nodo* ArbolBinario;
```



Representación de la figura del árbol binario balanceado anterior



## RECORRIDO DE UN ARBOL

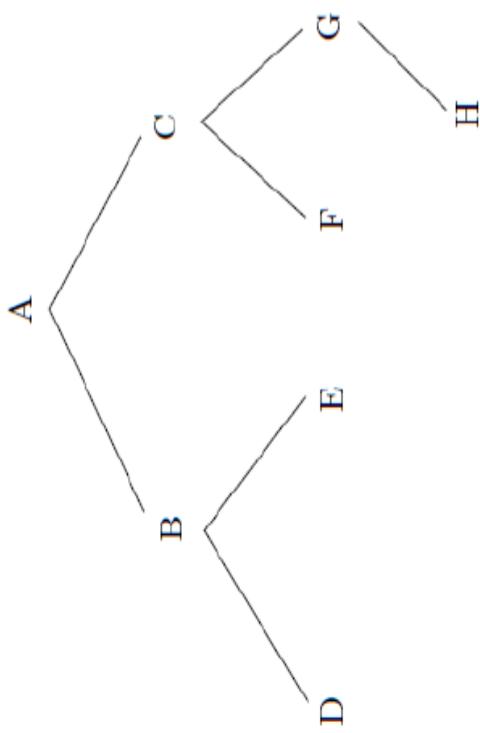
En ciencias de la computación, el **recorrido** de árboles se refiere al proceso de visitar de una manera sistemática, exactamente una vez, cada nodo en una estructura de datos de **árbol** (examinando y/o actualizando los datos en los nodos). Tales **recorridos** están clasificados por el orden en el cual son visitados los nodos.

En ese orden de ideas, el recorrido de un ‘árbol binario’ se lleva a cabo en tres sentidos: **Preorden, Inorden y Postorden**. A continuación se detalla cada caso.

## 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.



El algoritmo correspondiente para un arbol T sería:

**si** T no es vacío **entonces**  
**Inicio**

    Visitar el raíz de T

    Preorden (subárbol izquierdo del raíz de T)

    Preorden (subárbol derecho del raíz de T)

**Fin**

```
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 */  
}
```

## 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.

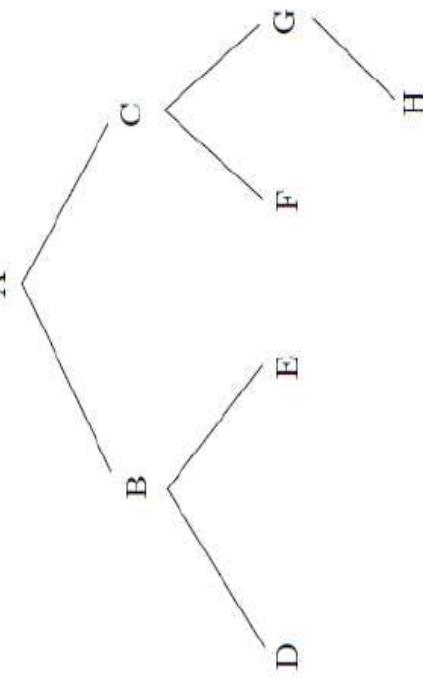
El algoritmo correspondiente enorden sería:

**Si** el arbol no está vacío **entonces**  
**Inicio**

    Recorer el subárbol izquierdo  
    Visitar el nodo raíz  
    Recorer el subárbol derecho

**Fin**

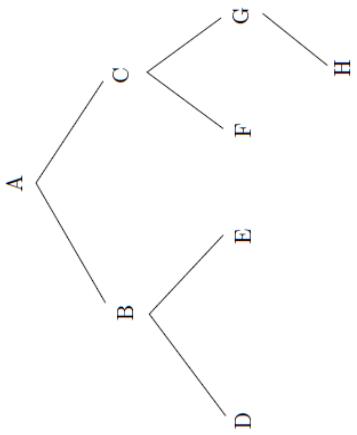
```
void InOrden(Arbol a, void (*func) (int*)) { if (a->izquierdo)
InOrden(a->izquierdo, func); /* Subárbol izquierdo */
func (&(a->dato));
/* Aplicar la función al dato del nodo actual */
if (a->derecho)
    InOrden(a->derecho,
func);
/* Subárbol derecho */
}
```



## POSTORDEN

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

El algoritmo correspondiente postorden sería:



**Si T no es vacío entonces**

**Inicio**

Postorden (subárbol izquierdo del raíz de T)

Postorden (subárbol derecho del raíz de T)

Visitar el raíz de T

**Fin**

```
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
    */
}
```

# 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.

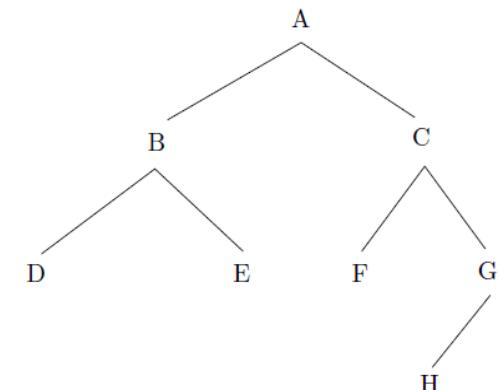
En este caso el recorrido no se realizará de forma recursiva sino iterativa, utilizando una cola como estructura de datos auxiliar.

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

## PROFUNDIDAD DE UN ARBOL 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 tienen un parámetro que es un puntero a la raíz del árbol.

El caso más sencillo de cálculo de la profundidad es cuando un 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 uno por separado. Las variables profundidadI, profundidadD almacenarán las profundidades de los subárboles izquierdo y derecho respectivamente.

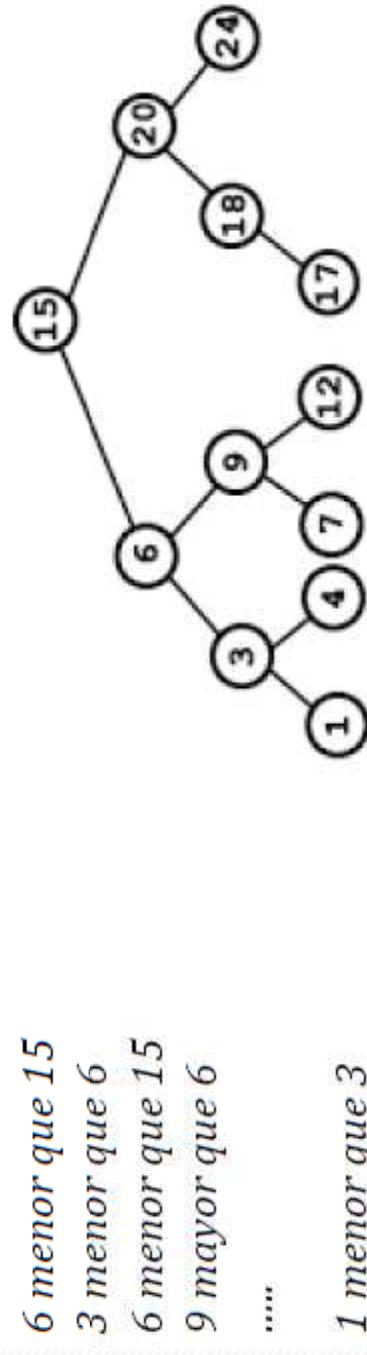
El método de cálculo de la profundidad de los subárboles utiliza llamadas recursivas a la función profundidad con punteros a los respectivos subárboles como parámetros de la misma. La función profundidad devuelve como resultado la profundidad del subárbol más profundo más 1 (la misma del raíz).

```
int Profundidad (Nodo *p)
{
    if (!p)
        Return 0;
    else
    {
        int profundidadI = Profundidad (p->hijo_izqdo);
        int profundidadD = Profundidad (p->hijo_dcho);
        if (profundidadI > profundidadD)
            return profundidadI +1;
        else
            return profundidadD +1;
    }
}
```

## ARBOL BINARIO DE BUSQUEDA

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 termino utilizando algoritmo de búsqueda binaria.

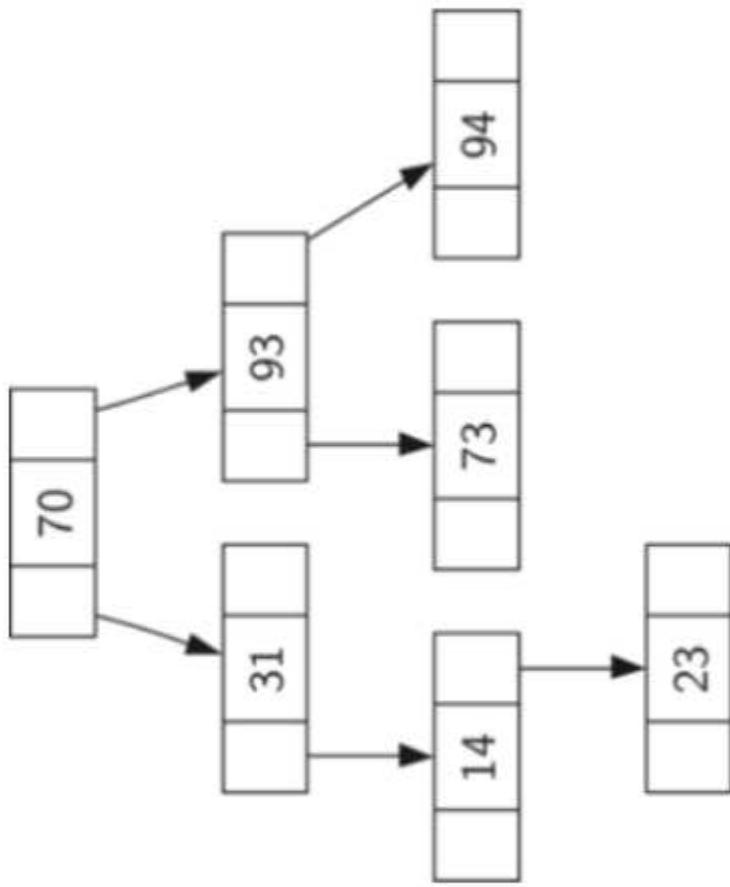
Un **árbol binario de búsqueda (ABB)** es aquel que dado que nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos.



Un **árbol binario de búsqueda(ABB)** es un **árbol binario** con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo  $x$  son menores que el elemento almacenado en  $x$  y todos los elementos almacenados en el subárbol derecho de  $x$  son mayores que el elemento almacenado en  $x$ .

## CREACION DE UN ARBOL BINARIO DE BUSQUEDA

Ahora que usted ya sabe lo que es un árbol binario de búsqueda, veremos cómo se construye. El árbol de búsqueda de la figura representa los nodos que existen después de haber insertado las siguientes claves en el orden mostrado: 70,31,93,94,14,23,73.



## IMPLEMENTACION DE UN NODO DE UN ARBOL BINARIO DE BUSQUEDA

Un árbol binario de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. Vemos a continuación 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 libreta en su universidad (dato entero).

### Declaración de tipos

<i>Nombre</i>	Tipo de dato cadena (string)
<i>Número de libreta</i>	Tipo entero

Nombre	
numLibreta	
Izda	dcha

```
struct nodo {  
    int numLibreta;  
    char nombre [30];  
    struct nodo *izda, *dcha;  
};  
TypeDef struct nodo Nodo;
```

## CREACION DE UN NODO

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

```
Nodo* CrearNodo (int id, char* n)
{
    Nodo* t;
    t = (Nodo*) malloc (sizeof (Nodo));
    t -> numLibreta = id;
    strcpy (t->nombre, n);
    t -> izda = t -> dcha = NULL;
    return t;
}
```

## OPERACIONES EN ARBOL BINARIO DE BUSQUEDA

Como vimos hasta ahora, 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. Las operaciones básicas son:

- *Búsqueda de un nodo*
- *Inserción de un nodo*
- *Recorrido del arbol*
- *Eliminación de un nodo*

## BUSQUEDA

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 derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

### *Si buscamos una información específica*

Si se desea encontrar un nodo en el árbol que contenga la información de un determinado alumno. La función buscar tiene dos parámetros, el puntero al árbol y un numero de libreta del alumno a buscar. 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 esta vacío. En caso afirmativo 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 libreta requerida es más pequeña o mayo que el número de legajo del nodo raíz.
3. La función de búsqueda se consigue llamando recursivamente a la función buscar con un puntero al subárbol izquierdo o derecho como parámetro.

El código C de la función buscar es:

```
Nodo* buscar (Nodo*, p, int buscado)
{
    if (!p)
        return 0;
    else if (buscado == p->numLibreta)
        return p;
    else if (buscado < p->numLibreta)
        return buscar (p ->izda, buscado);
    else
        return buscar (p ->dcha, buscado);
}
```

## INSERCIÓN DE UN NODO

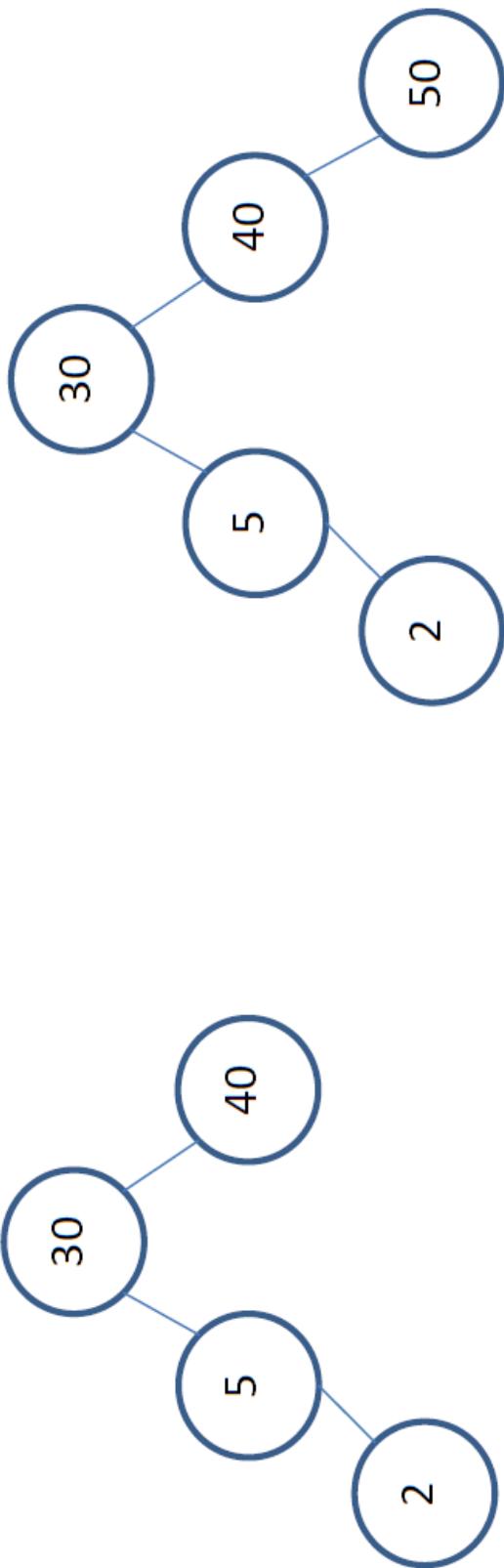
Una característica importante que debe poseer el algoritmo de inserción es que el árbol resultante de una inserción en un árbol de búsqueda ha de ser también de búsqueda. En esencia, el algoritmo de inserción se apoya en la localización de un elemento, de modo que si se encuentra el elemento (clave) 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).

*Ejemplo. Insertar un elemento con clave 50 en el siguiente árbol binario de búsqueda:*

30 ( 5 (2) 40 )

Resultado

30 ( 5 (2) 40 (50) )



## Funció n insertar()

Esta función es sencilla. Se deben declarar tres argumentos: un puntero al raíz del árbol, el nuevo nombre y número de libreta del alumno. 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 colocara como nodo hoja.
3. Enlazar el nuevo nodo al árbol

El código C de la función insertar es:

```
Void insertar (Nodo** raiz, int nueva_libreta, char *nuevo_nombre)
{
    if (!(*raiz))
        *raiz = CrearNodo (nueva_libreta, nuevo_nombre);
    else if (nueva_libreta < (*raiz) -> nueva_libreta)
        insertar (&(*raiz) -> izda), nueva_libreta, nuevo_nombre);
    else
        insertar (&(*raiz) -> dcha, nueva_libreta,
nuevo_nombre);
}
```

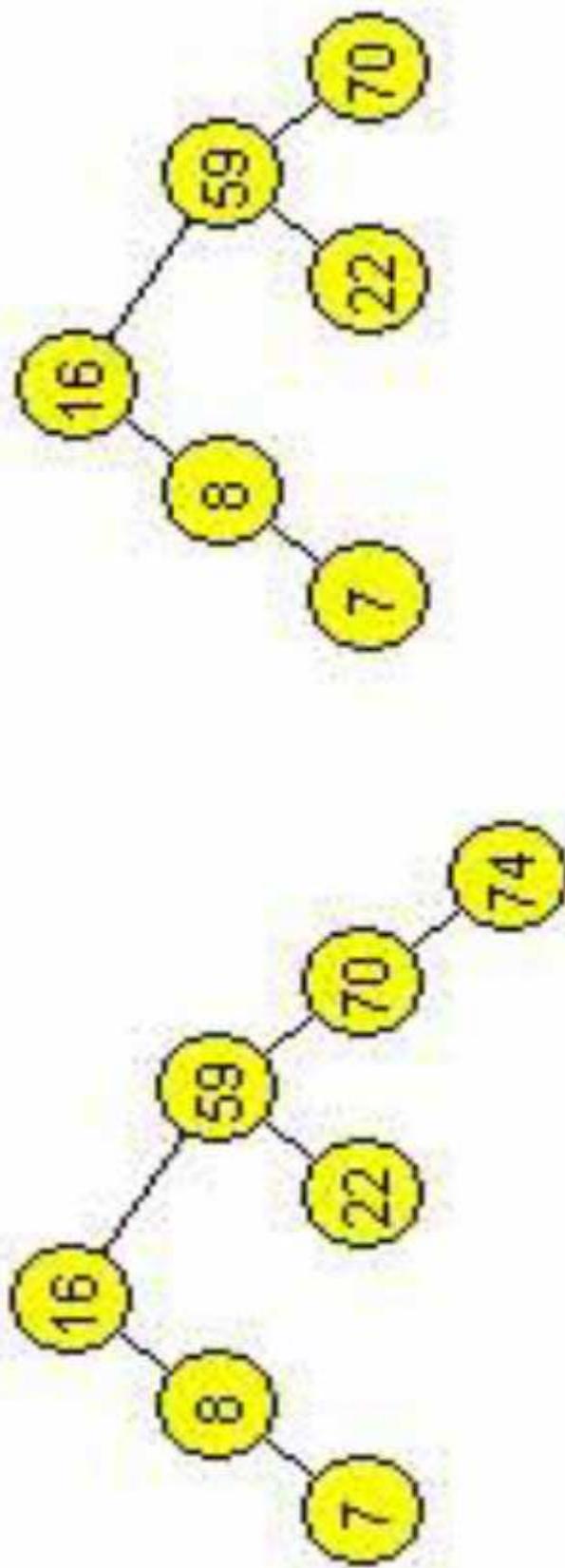
## ELIMINACIÓN DE UN NODO

La operación de *eliminación* de un nodo es también una extensión de la operación de búsqueda, si bien un poco más compleja que la operación de inserción dado que se puede suprimir cualquier nodo y se debe seguir manteniendo la estructura de árbol binario de búsqueda.

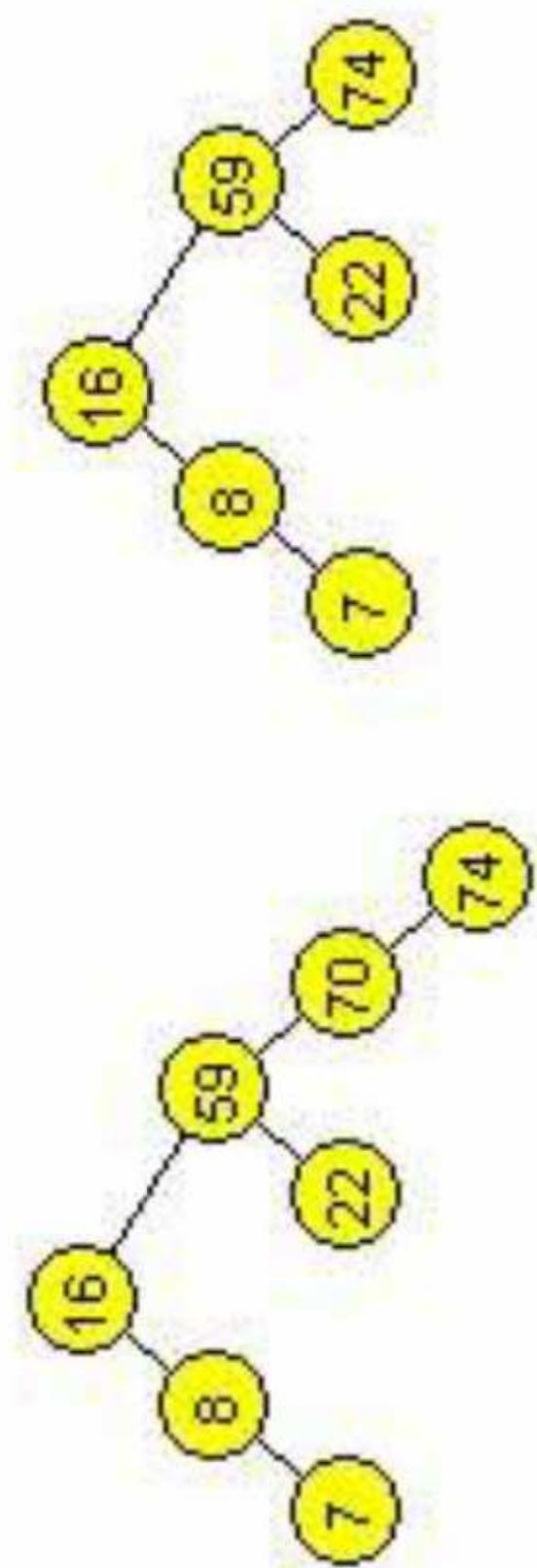
Los pasos a seguir son:

1. Buscar 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 este ocupa el nodo mas próximo en clave (inmediatamente superior o inmediatamente inferior) con el objeto de mantener la estructura de árbol binario.

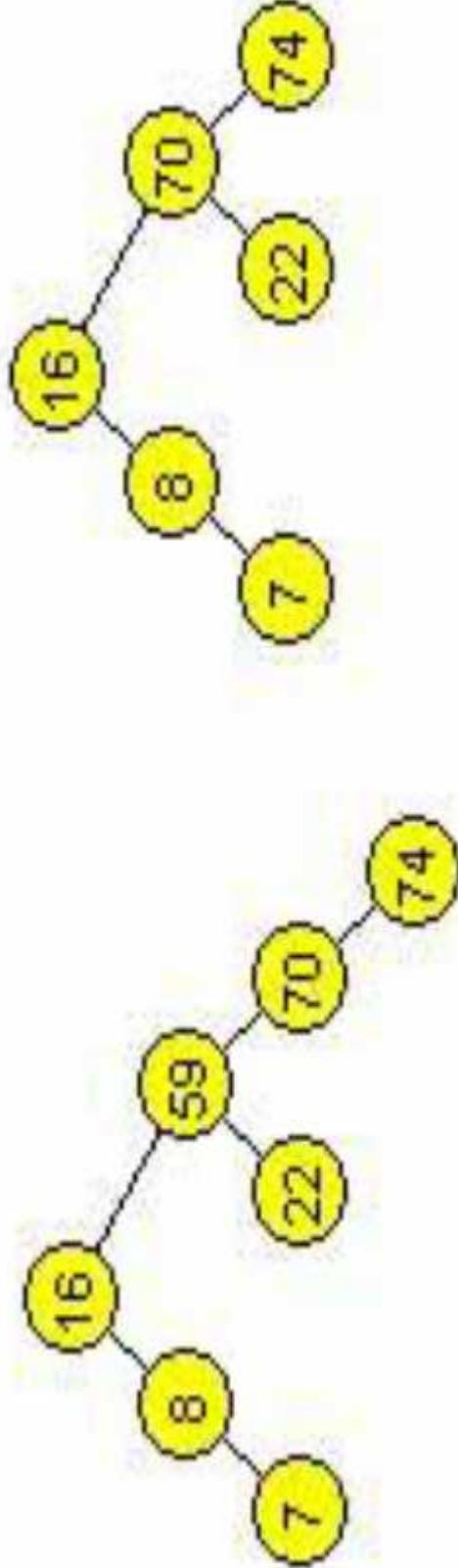
**Una hoja del árbol:** Si el nodo por eliminar es una hoja, entonces basta con destruir su variable asociada (Delete) y, posteriormente, asignar nil a ese puntero.



**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 desecharando el nodo por eliminar.



**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 predecessor del nodo por eliminar (>en qué posición se encuentra el nodo predecessor?). Una vez hallado el predecessor el resto es bien fácil, sólo hay que copiar su valor en el nodo por eliminar y desechar el nodo predecessor.





A continuacion algunos ejemplos ....

## Cargar un Arbol binario de búsqueda y mostrar los elementos en INORDEN – PREORDEN - POSORDEN

```
struct nodo{
    int nro;
    struct nodo *izq, *der;
};

typedef struct nodo *ABB;

/* es un puntero de tipo nodo que hemos llamado ABB, que ultizaremos
para mayor facilidad de creacion de variables */
```

```
ABB crearNodo(int x)
{
    ABB nuevoNodo = new(struct nodo);
    nuevoNodo->nro = x;
    nuevoNodo->izq = NULL;
    nuevoNodo->der = NULL;

    return nuevoNodo;
```

```
void insertar(ABB &arbol, int x)
{
    if(arbol==NULL)
    {
        arbol = crearNodo(x);
    }
    else if(x < arbol->nro)
        insertar(arbol->izq, x);
    else if(x > arbol->nro)
        insertar(arbol->der, x);
}
```

```
void preOrden(ABB arbol)
{
    if(arbol!=NULL)
    {
        cout << arbol->nro << " ";
        preOrden(arbol->izq);
        preOrden(arbol->der);
    }
}
```

```
void enOrden(ABB arbol)
{
    if(arbol!=NULL)
    {
        enOrden(arbol->izq);
        cout << arbol->nro << " ";
        enOrden(arbol->der);
    }
}
```

```
void postOrden(ABB arbol)
{
    if(arbol!=NULL)
    {
        postOrden(arbol->izq);
        postOrden(arbol->der);
        cout << arbol->nro << " ";
    }
}
```

```
void verArbol(ABB arbol, int n)
{
    if(arbol==NULL)
        return;
    verArbol(arbol->der, n+1);
    for(int i=0; i<n; i++)
        cout<<"  ";
    cout<< arbol->nro <<endl;
    verArbol(arbol->izq, n+1);
}
```

```
int main()
{
    ABB arbol = NULL; // creado Arbol
    int n; // numero de nodos del arbol
    int x; // elemento a insertar en cada nodo
    cout << "\n\t\t ..[ ARBOL BINARIO DE BUSQUEDA ].. \n\n";
    cout << " Numero de nodos del arbol: ";
    cin >> n;
    cout << endl;
    for(int i=0; i<n; i++)
    {
        cout << " Numero del nodo " << i+1 << ": ";
        cin >> x;
        insertar( arbol, x);
    }
    cout << "\n Mostrando ABB \n\n";
    verArbol( arbol, 0);
    cout << "\n Recorridos del ABB";
    cout << "\n\n En orden : "; enOrden(arbol);
    cout << "\n\n Pre Orden : "; preOrden(arbol);
    cout << "\n\n Post Orden : "; postOrden(arbol);
    cout << endl << endl;
    system("pause");
    return 0;
}
```

```
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;
        }
    }
}
```

## GRAFOS

Los grafos no son más que la versión general de un árbol, es decir, cualquier nodo de un grafo puede apuntar a cualquier otro nodo de éste (incluso a él mismo).

Este tipo de estructuras de datos tienen una característica que lo diferencia de las estructuras que hemos visto hasta ahora: los grafos se usan para almacenar datos que están relacionados de alguna manera (relaciones de parentesco, puestos de trabajo, ...); por esta razón se puede decir que los grafos representan la estructura real de un problema.

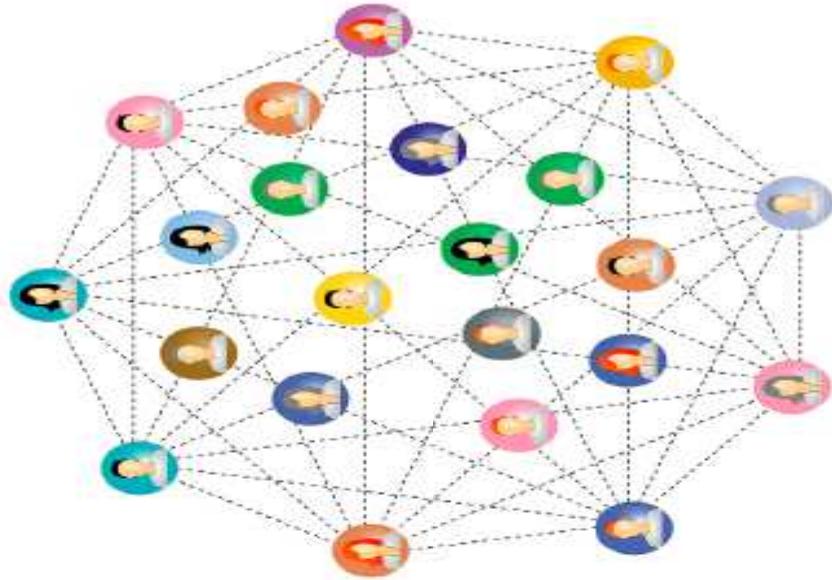
En lo que a ingeniería de telecomunicaciones se refiere, los grafos son una importante herramienta de trabajo, pues se utilizan tanto para diseño de circuitos como para calcular la mejor ruta de comunicación en Internet.

Un grafo es la representación simbólica de los elementos constituidos de un sistema o conjunto, mediante esquemas gráficos. Se puede decir también, que un grafo consiste en un conjunto de nodos (también llamados vértices) y un conjunto de arcos (aristas) que establecen relaciones entre nodos.

Es importante resaltar, que informalmente un grafo se define como  $G = (V, E)$ , siendo los elementos de  $V$  los vértices o nodos, y los elementos de  $E$ , las aristas. Formalmente, un grafo  $G$ , se define como un par ordenado,  $G = (V, E)$ , donde  $V$  es un conjunto finito y  $E$  es un conjunto que consta de dos elementos de  $V$ .

Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan los terminales y las aristas representan las conexiones inalámbricas). En fin, prácticamente cualquier problema puede representarse mediante un grafo.

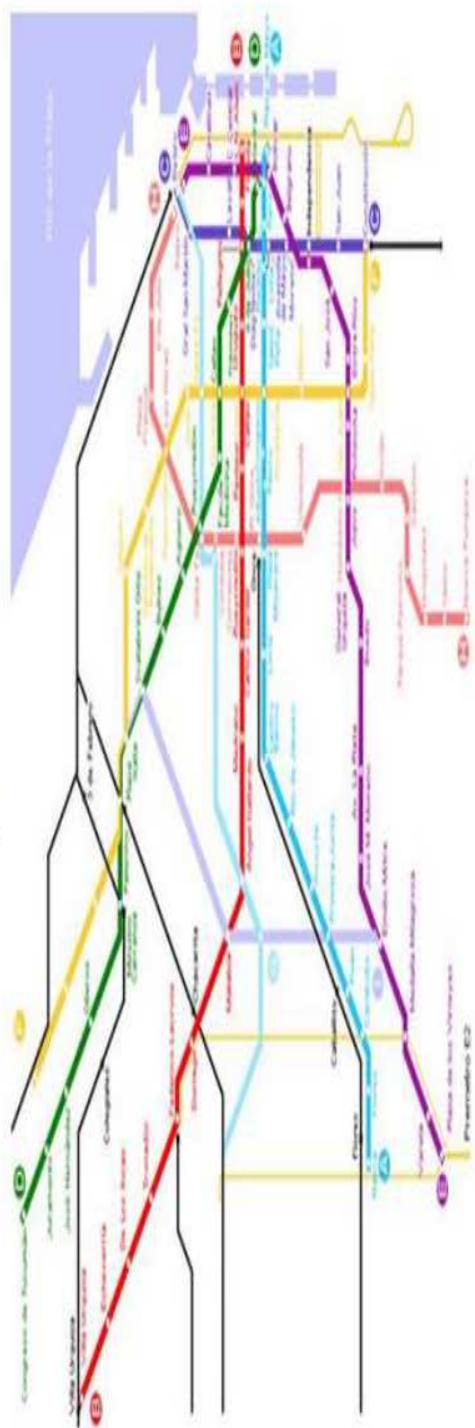
El estudio de grafos es una rama de la algoritmia muy importante.



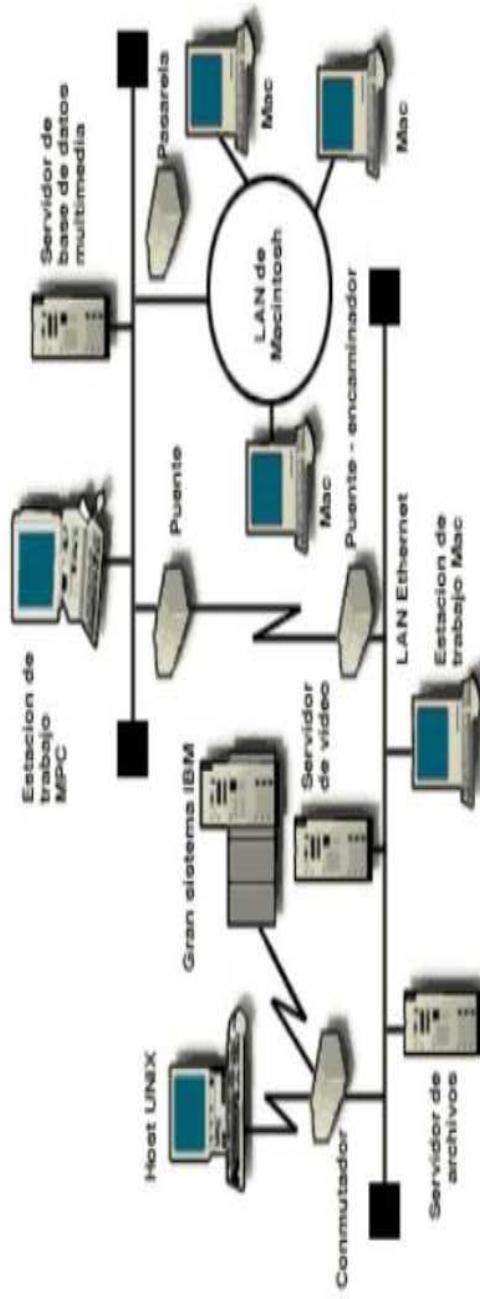
GRAFOS

**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.



Redes de transporte

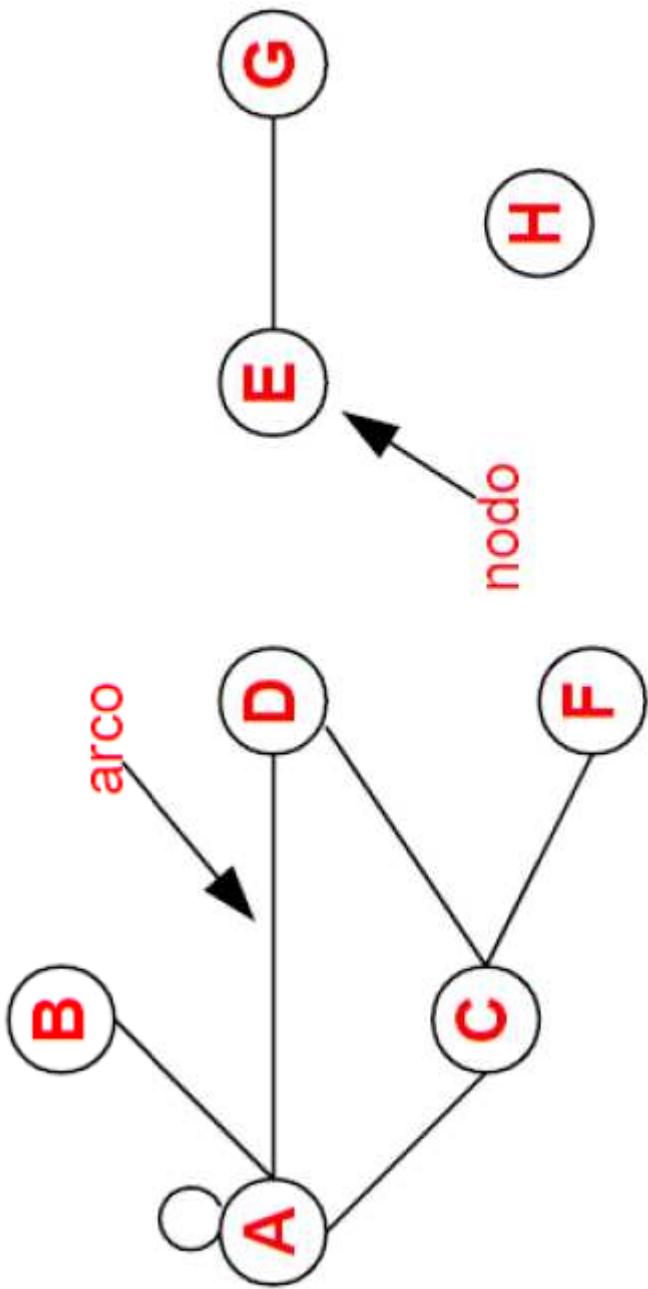


Redes de computadoras

## DECLARACION DE GRAFO

Un grafo está formado por un conjunto de nodos(o vértices) y un conjunto de arcos. Cada arco en un grafo se especifica por un par de nodos.

El conjunto de nodos es {A, B, C, D, F, G, H} y el conjunto de arcos {(A, B), (A, D), (A, C), (C, D), (C, F), (E, G), (A, A)} para el siguiente grafo

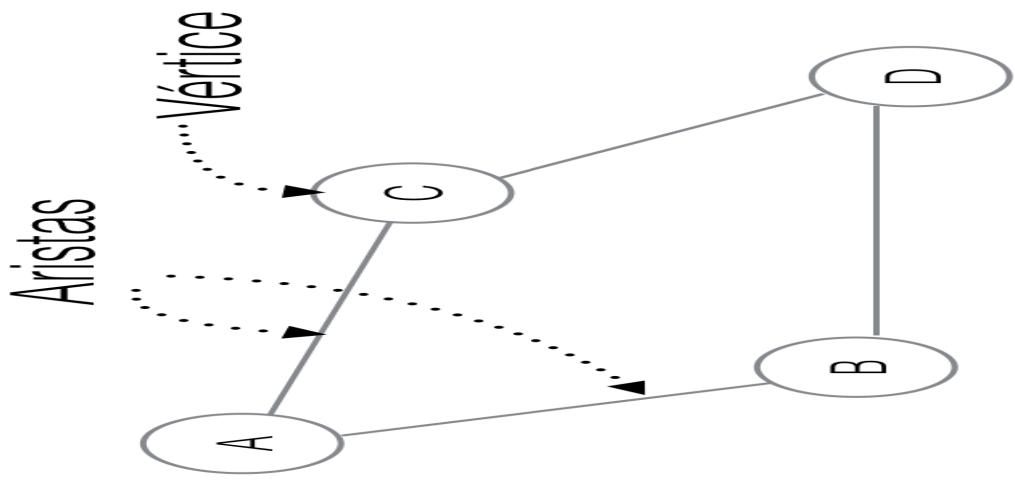


Un grafo consta de vértices (o nodos) y aristas.

**Aristas:** Son las líneas con las que se unen las aristas de un grafo y con la que se construyen también caminos. Si la arista carece de dirección se denota indistintamente  $\{a, b\}$  o  $\{b, a\}$ , siendo  $a$  y  $b$  los vértices que une.

Si  $\{a, b\}$  es una arista, a los vértices  $a$  y  $b$  se les llama sus extremos.

**Vértices:** Son los puntos o nodos con los que está conformado un grafo. Llamaremos grado de un vértice al número de aristas de las que es extremo. Se dice que un vértice es 'par' o 'ímpar' según lo sea su grado.



## TERMINOLOGÍA

Formalmente un grafo es un conjunto de puntos y un conjunto de líneas, cada una de las cuales une un punto a otro.

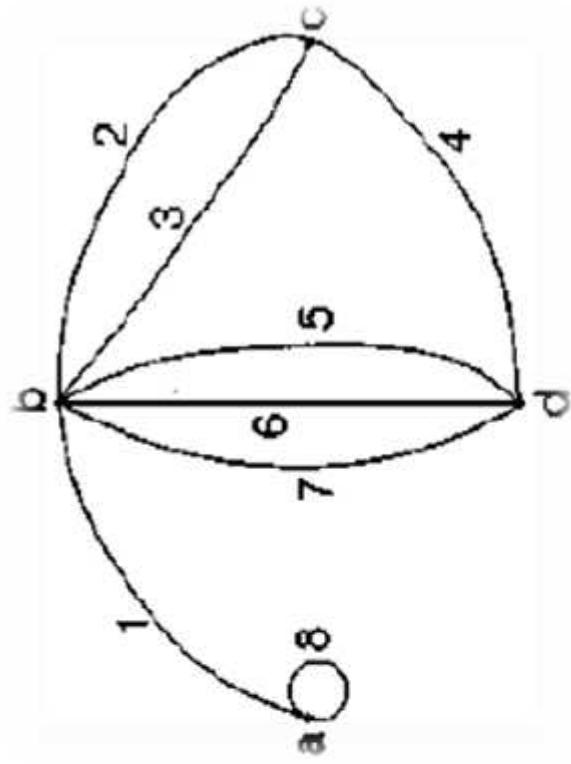
**Vértice:** Nodo.

Se representan el conjunto de vértices de un grafo  $G$  por  $V_G$  y el conjunto de arcos por  $A_G$

Por ejemplo:

$$V_G = \{a, b, c, d\}$$

$$A_G = \{1, 2, 3, 4, 5, 6, 7, 8\}$$



## TERMINOLOGÍA

El número de elementos de  $V_g$  se llama **orden del grafo**.

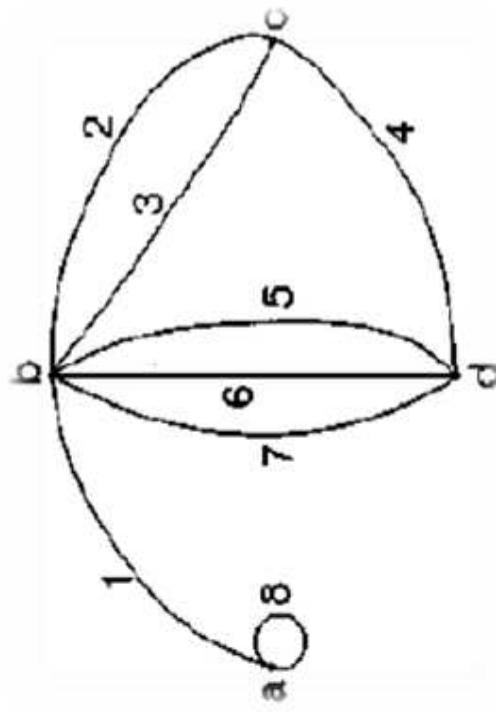
Un **grafo nulo** es un grafo de orden cero.

**Aristas:** líneas o arcos, se representa por los vértices que conecta.

La arista 3 conecta los vértices b y c, y se representa por  $V(b, c)$ .

Algunos vértices pueden conectarse con sí mismos, por ejemplo: el arco 8 tiene la forma  $V(a, a)$ .

Estas aristas se denominan **bucles o lazos**.



Un **camino** es una secuencia de uno o más arcos que conectan dos nodos. Un *camino simple* es un camino desde un nodo a otro en el que ningún nodo se repite (no se pasa dos veces).

La **longitud** de un camino es el número de arcos que comprende.

**Enlace:** Conexión entre dos vértices (nodos).

**Adyacencia:** Se dice que dos vértices son adyacentes si entre ellos hay un enlace directo.

**Vecindad:** Conjunto de vértices adyacentes a otro.

## ¿Sabías que...

Los Vértices, son los objetos representados por un punto dentro del grafo.



Las Aristas, son las líneas que unen dos vértices.



Las Arista Adyacentes, se dice que dos aristas son adyacentes si convergen sobre el mismo vértice.



Las Aristas Múltiples o Paralelas, dos aristas son múltiples o paralelas si tienen los mismos vértices en común o incidente sobre los mismos vértices.



Lazo, es una arista cuyos extremos inciden sobre el mismo vértice.



## TIPOS DE GRAFOS

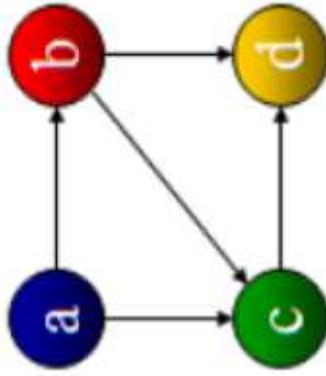
### Dirigido

Un grafo dirigido o dígrafo es un tipo de grafo en el cual las aristas tienen una dirección definida, a diferencia del grafo generalizado, en el cual la dirección puede estar especificada o no.

Al igual que en el grafo generalizado, el grafo dirigido está definido por un par de conjuntos  $G=(V,E)$ , donde:

- $V \neq \emptyset$ , un conjunto no vacío de objetos simples llamados vértices o nodos.
- $E \subseteq \{(a,b) \in V \times V; a \neq b\}$  es un conjunto de pares ordenados de elementos de  $V$  denominados aristas o arcos, donde por definición un arco va del primer nodo (a) al segundo nodo (b) dentro del par.

Por definición, los grafos dirigidos no contienen bucles (lazos).



$$V = \{a, b, c, d\}$$

$$E = \{(a, c), (a, b), (b, c), (b, d), (c, d)\}$$

## TIPOS DE GRAFOS

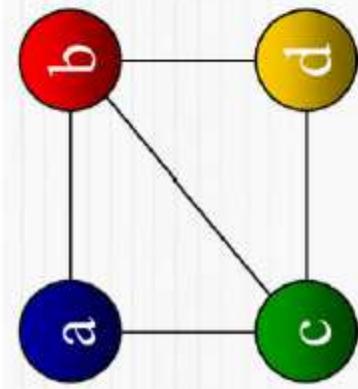
### No dirigido

Un grafo no dirigido o grafo propiamente dicho es un grafo  $G=(V,E)$  donde:

- $V \neq \emptyset$
- $E \subseteq \{x \in P(V) : |x| = 2\}$  es un conjunto de pares no ordenados de elementos de  $V$ .

Un par no ordenado es un conjunto de la forma  $\{a,b\}$ , de manera que  $\{a,b\} = \{b,a\}$ .

Para los grafos, estos conjuntos pertenecen al conjunto de potencia de  $V$ , denotado  $P(V)$ , y son de cardinalidad 2.



$$V=\{a, b, c, d\}$$

$$E=\{(a,c),(c,a),(a,b),(b,a),(b,c),(c,b),(b,d),(d,b),(c,d),(d,c)\}$$

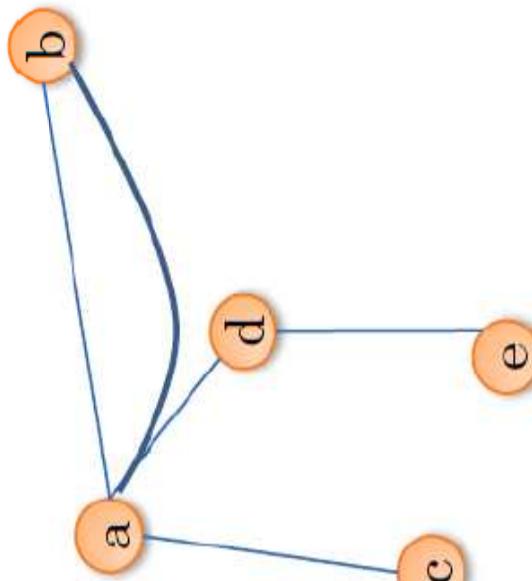
## TIPOS DE GRAFOS

### Grafo conectado

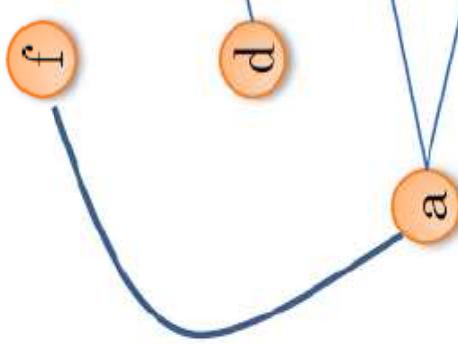
También llamado *Grafo conexo*, en matemáticas y ciencias de la computación es aquel grafo que entre cualquier par de sus vértices existe un camino (Grafo) que los une. Existe siempre un camino que une dos vértices cualesquiera.

### Grafo desconectado

Existen vértices que no están unidos por un camino.



Grafo conexo



Grafo no conexo

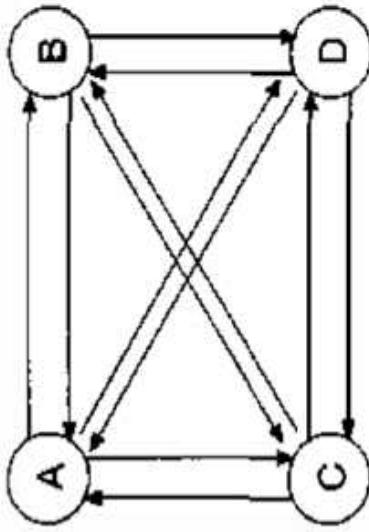
## TIPOS DE GRAFOS

### Grafo Simple

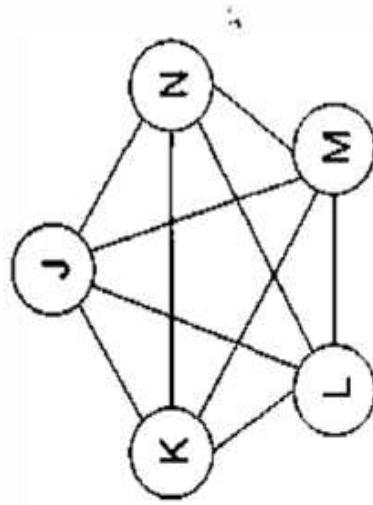
Es aquel grafo que no posee bucles o lazos. Se puede decir también, que un grafo es simple si a lo más existe una arista uniendo dos vértices cualesquiera. Esto es equivalente a decir que una arista cualquiera es la única que une dos vértices específicos. Un grafo que no es simple se denomina multigrafo.

### Grafo Completo

Un grafo completo es un grafo simple en el que cada par de vértices están unidos por una arista, es decir, contiene todas las posibles aristas. Se puede hacer referencia que un grafo completo de  $n$  vértices tiene  $n(n-1)/2$  aristas, y se nota  $K_n$ . Es un grafo regular con todos sus vértices de grado  $n-1$ . La única forma de hacer que un grafo completo se torne desconexo a través de la eliminación de vértices, sería eliminándolos todos.



(a) grafo completo dirigido



(b) grafo completo no dirigido

## TIPOS DE GRAFOS

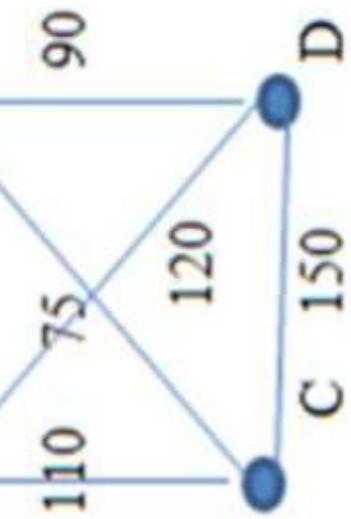
### Grafo ponderado

Un grafo ponderado o con peso es aquel en el que cada arista tiene un valor.

Los grafos con peso pueden representar situaciones de gran interés, por ejemplo los vértices pueden ser ciudades y las aristas distancias o precios del pasaje de avión entre ambas ciudades.

El grafo de la figura es un grafo ponderado, sus ponderaciones se encuentran sobre cada arista.

En este caso, por ejemplo se lee que la arista que une los vértices A y B tiene una ponderación de 100.



Ahora, imaginemos que los vértices del grafo anterior son ciudades y que sus aristas son posibles caminos entre cada ciudad, la ponderación indicaría la distancia entre ciudades.

Las ponderaciones no solo representan distancias, pueden ser valores monetarios, tiempos, entre otros.

## REPRESENTACION

**Matricial:** Usamos una matriz cuadrada de *boolean* en la que las filas representan los nodos origen, y las columnas, los nodos destinos. De esta forma, cada intersección entre fila y columna contiene un valor booleano que indica si hay o no conexión entre los nodos a los que se refiere. Si se trata de un grafo con pesos, en lugar de usar valores booleanos, usaremos los propios pesos de cada enlace y en caso de que no exista conexión entre dos nodos, rellenaremos esa casilla con un valor que represente un coste  $\infty$ . A esta matriz se le llama **Matriz de Adyacencia**.

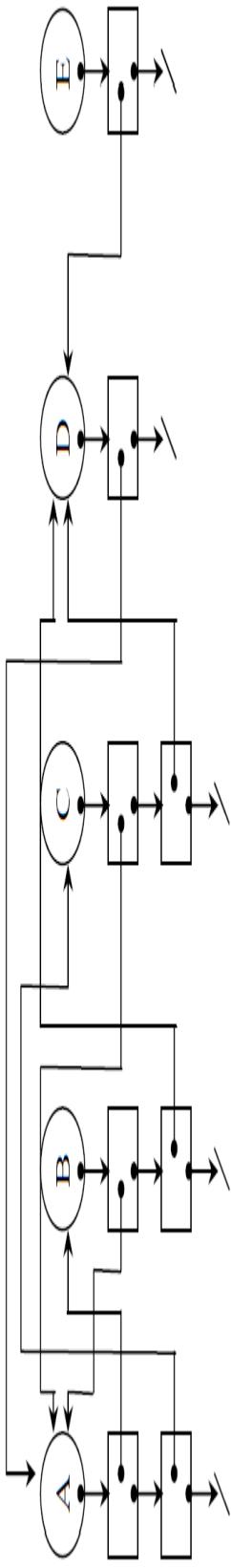
- Si no tuviera pesos:

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	1	0
C	1	0	0	1	0
D	1	0	0	0	0
E	0	0	0	1	0

- Teniendo en cuenta los pesos:

	A	B	C	D	E
A	0	16	3	$\infty$	$\infty$
B	50	0	$\infty$	8	$\infty$
C	25	$\infty$	0	12	$\infty$
D	1	$\infty$	$\infty$	0	$\infty$
E	$\infty$	$\infty$	$\infty$	2	0

**Dinámica:** Usamos listas dinámicas. De esta manera, cada nodo tiene asociado una lista de punteros hacia los nodos a los que está conectado:



## MATRIZ DE ADYACENCIA

La matriz de adyacencia **M** es una matriz de 2 dimensiones que representa las conexiones entre pares de vértices.

$$M(I, J) = \begin{cases} 1 & \text{si existe una arista } (V_i, V_j) \text{ en } A_G, V_i \text{ es adyacente a } V_j \\ 0, & \text{en caso contrario} \end{cases}$$

Las columnas y las filas de la matriz representan los vértices del grafo.

Si existe una arista desde  $i$  a  $j$  (esto es, el vértice  $i$  es adyacente a  $j$ ), se introduce el costo o peso de la arista  $i$  a  $j$ , si no existe la arista, se introduce **0**.

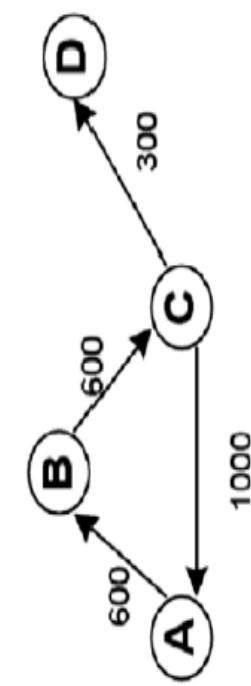
Los elementos de la diagonal principal son todos cero, ya que el costo de la arista  $i$  a  $i$  es 0.

Si  $G$  es un grafo no dirigido, la matriz es simétrica  $M(i, j) = M(j, i)$

	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	1	0	0	1
D	0	0	0	0



	A	B	C	D
A	0	600	100	0
B	600	0	600	0
C	100	600	0	300
D	0	0	300	0



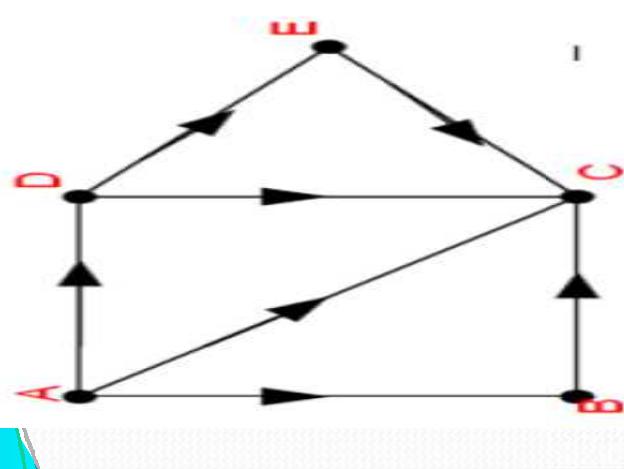
## *Código matriz de adyacencia*

```
int V,A;
int a[maxV][maxV];

void inicializar() {
    int i,x,y,p;
    char v1,v2;
    // Leer V y A

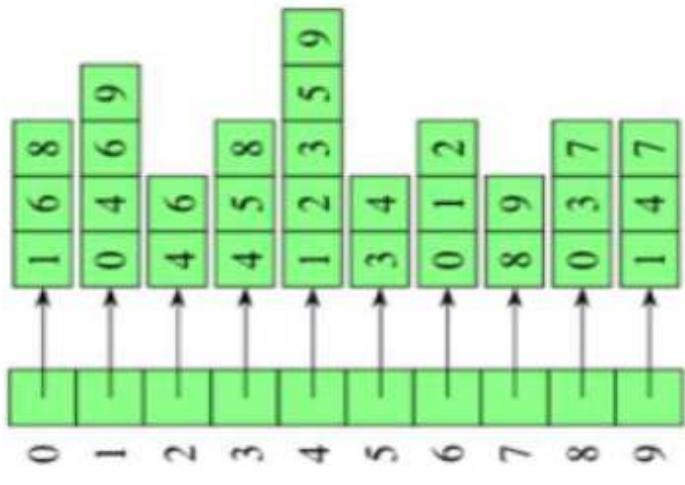
    memset(a,0,sizeof(a));
    for (i=1; i<=A; i++) {
        scanf("%c %c%d\n",&v1,&v2,&p);
        x= v1- 'A';
        y=v2- 'A';
        a[x][y]=p;
        a[y][x]=p;
    }
}
```

## LISTAS DE ADYACENCIA



Nodo	Lista de adyacencia
A	B, C, D
B	C
C	D, E
D	C
E	C

(B) Listas de adyacencia



Representar un grafo con listas de adyacencia combina las matrices de adyacencia con las listas de aristas. Para cada vértice  $i$ , almacena un arreglo de los vértices adyacentes a él. Tipicamente tenemos un arreglo de  $|V|$  listas de adyacencia, una lista de adyacencia por vértice. Aquí está una representación de una lista de adyacencia del grafo de la red social:

```
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.ARCHOS [ 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 ;
                            Eliminamos si 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;
                    END;
                END;
            END;
        END;
    END;

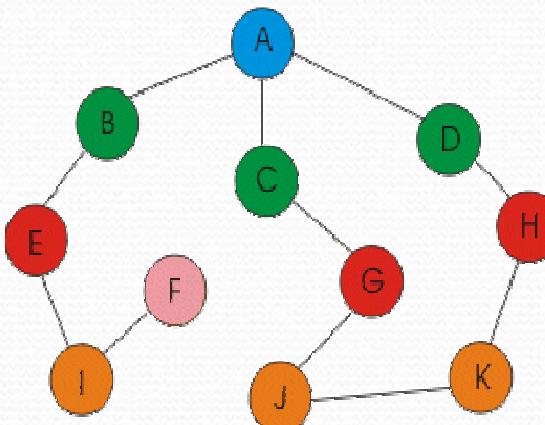
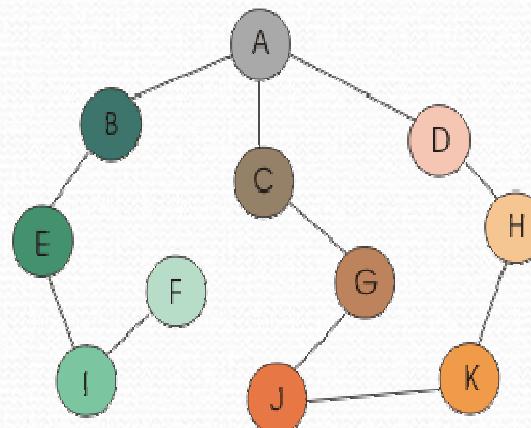
```

### BORRAR VERTICE y arco Dinamica

# Exploración de grafos

Recorrer un grafo significa pasar por todos sus vértices 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 (DFS)

- 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 0) 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.

[http://163.10.22.82/OAS/recorrido\\_grafos/dfs\\_\\_recorrido\\_en\\_profundidad2.html](http://163.10.22.82/OAS/recorrido_grafos/dfs__recorrido_en_profundidad2.html)

## *Búsqueda en amplitud o anchura (BFS)*

- 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 Giusti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2.  
Capítulo 9.

## Internet

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