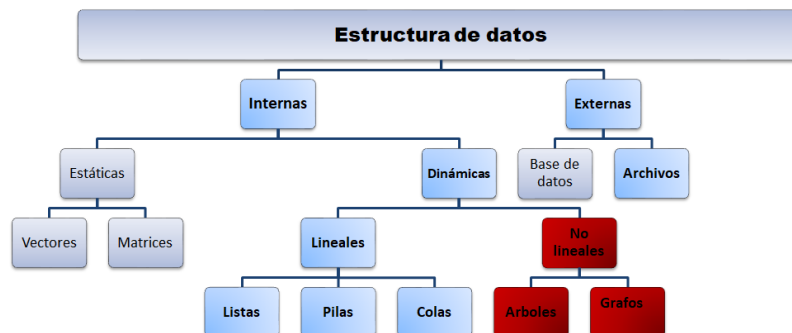


## Tema II: Estructuras Dinámicas

Estructuras no lineales. Árboles. Árboles binarios. Árbol binario de búsqueda. Recorridos en profundidad y amplitud. Grafos. Definición. Glosario. Representación por matriz de adyacencia y lista de adyacencia. Operaciones sobre grafos. Recorrido y búsqueda en profundidad. Recorrido y búsqueda en amplitud.

### 2.1 Estructuras no lineales

Las estructuras de datos no lineales, también llamadas multienlazadas, **son aquellas en las que cada elemento puede estar enlazado a cualquier otro componente**. Es decir, cada elemento puede tener varios sucesores o varios predecesores.



La estructura de datos no lineales no organiza los datos de forma consecutiva, sino que se organiza en orden ordenado. En este caso, los elementos de datos se pueden adjuntar a más de un elemento que muestra la relación jerárquica que implica la relación entre el hijo, el padre y el abuelo. En la estructura de datos no lineales, el recorrido de los elementos de datos y la inserción o eliminación no se realizan de forma secuencial.

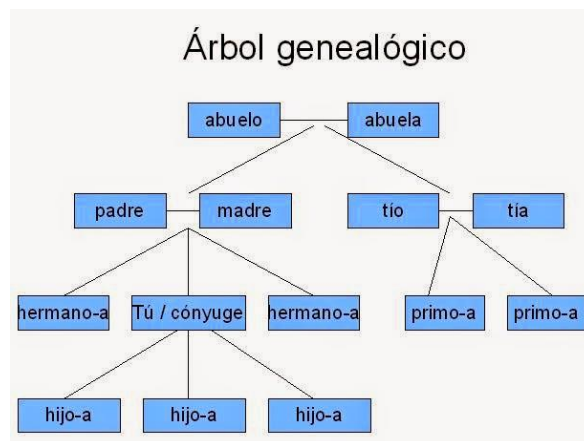
La estructura de datos no lineales utiliza la memoria de manera eficiente y no requiere de antemano la declaración de memoria. Hay dos ejemplos comunes de la estructura de datos no lineales: **árbol** y **grafo**. Una estructura de datos de árbol organiza y almacena los elementos de datos en una relación jerárquica.

Tabla comparativa estructuras lineales y no lineales		
Bases para la comparación	Estructura de datos lineales	Estructura de datos no lineales.
BASIC	Los elementos de datos se organizan de manera ordenada donde los elementos se adjuntan adyacentemente.	Organiza los datos en un orden ordenado y existe una relación entre los elementos de datos.
Recorrido de los datos	Se puede acceder a los elementos de datos de una sola vez (ejecución única).	El desplazamiento de elementos de datos en una sola vez no es posible.
Facilidad de implementación	Más simple	Complejo
Niveles involucrados	Nivel único	Nivel múltiple
Ejemplos	Arreglos, cola, pila, lista enlazada, etc.	Árbol y grafos
Utilización de la memoria	Ineficaz	Eficaz

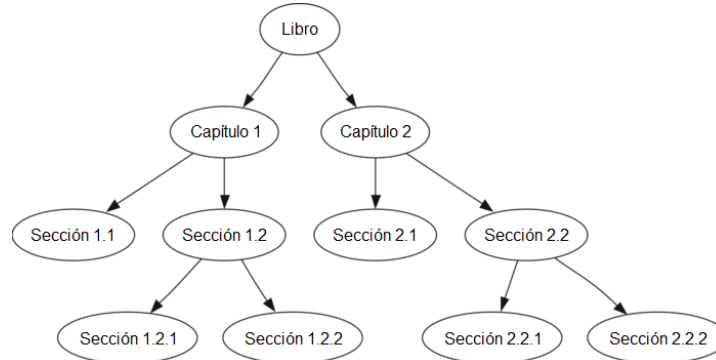
## 2.2 Árboles

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 sí 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 elementos, 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.

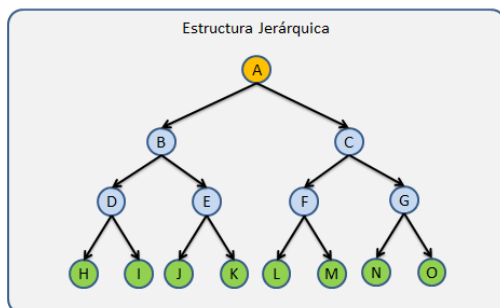


**Definición 1:** Un **árbol** consta de un conjunto finito de elementos, llamados nodos y un conjunto finito de líneas dirigidas, llamadas ramas, que conectan los nodos.

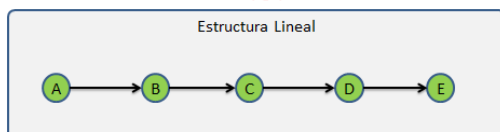
**Definición 2:** Un árbol es un conjunto de uno o más nodos tales que:

1. Hay un nodo diseñado especialmente llamado **raíz**
2. Los nodos restantes se dividen en  $n \geq 0$  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* del 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.



**V.S.**



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.

## Terminología

**Raíz:** es aquel elemento que no tiene antecesor; ejemplo: A

**Rama:** arista entre dos nodos. Es decir, es el camino que termina en una hoja (C, F, H).

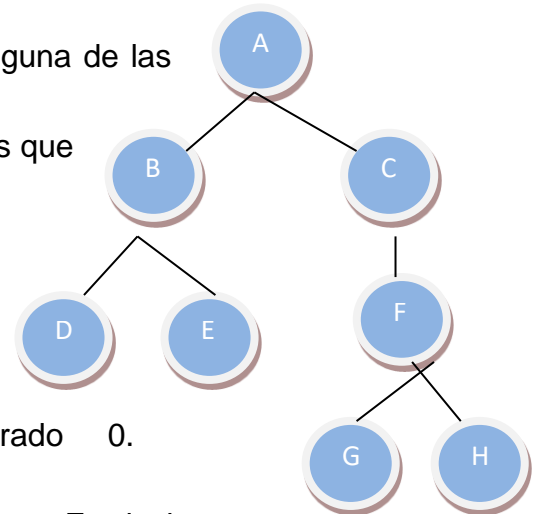
**Antecesor:** un nodo X es el antecesor de un nodo Y si por alguna de las ramas de X se puede llegar a Y.

**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 de la 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).

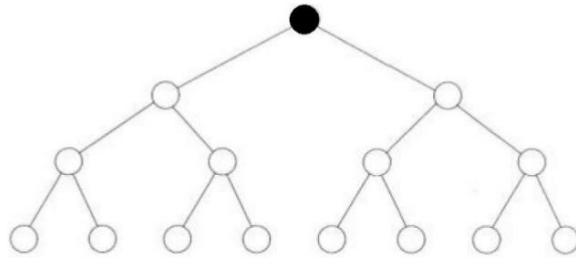
Todo nodo que no tiene descendientes directos (hijos), se conoce con el nombre de **terminal u hoja** (D, E, G, H)

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

**Longitud del camino:** nº de nodos que tiene.

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

Árbol perfectamente equilibrado.

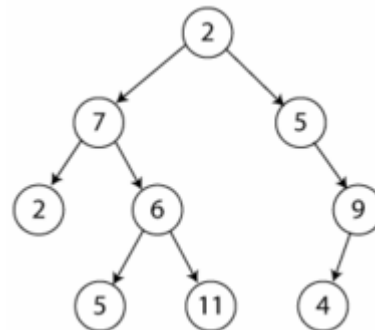


### Representación de un árbol

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

#### *Representación en nivel de profundidad*

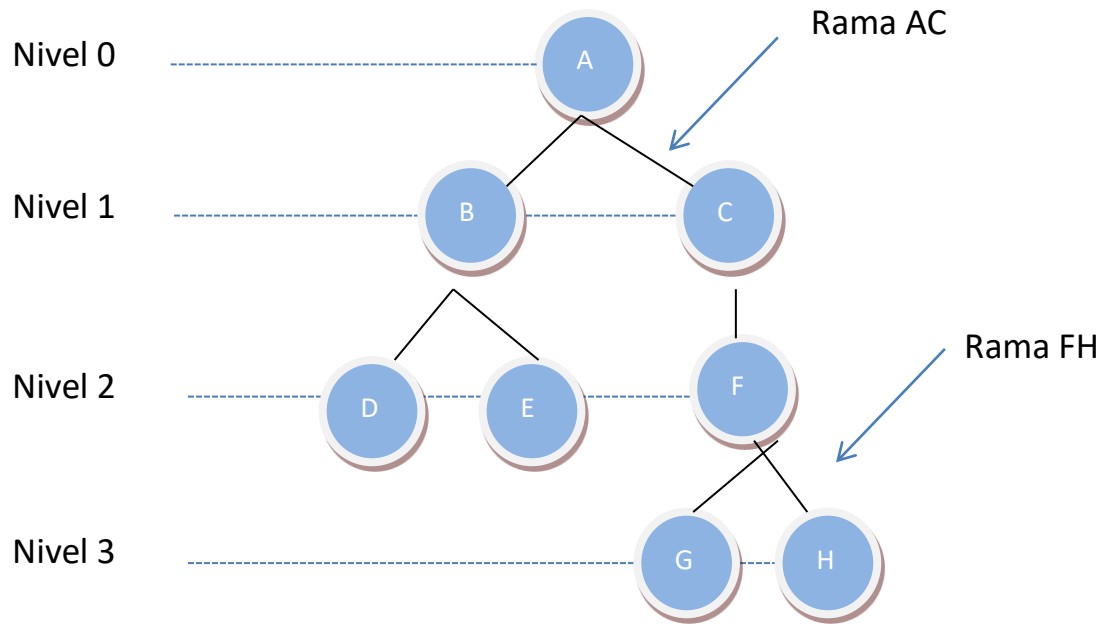
Este tipo de representación es el utilizado para representar sistemas jerárquicos en modo texto o números en situaciones tales como facturación, gestión de stock en almacenes, autopartes, etc.-



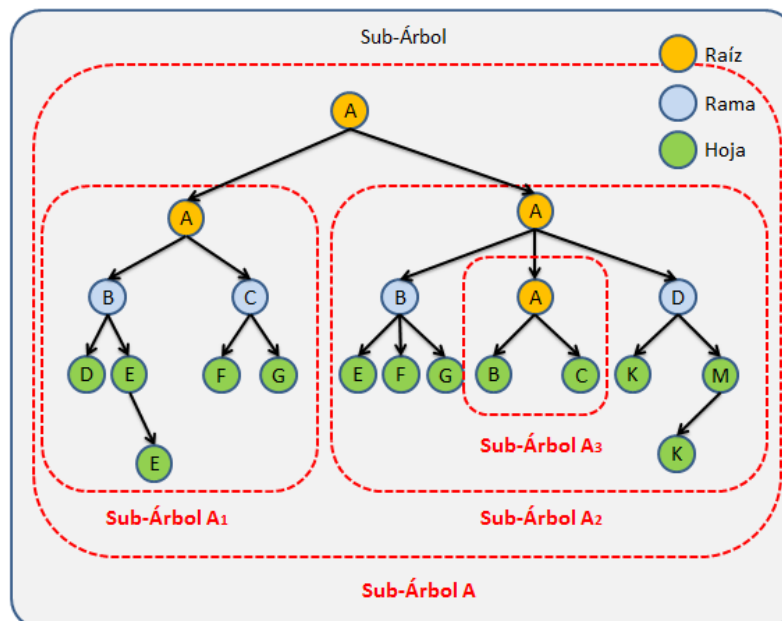
#### *Representación de listas*

Otro formato utilizado para presentar un árbol es la lista entre paréntesis. Esta es la notación utilizada con expresiones algebraicas. En cada representación, cada paréntesis abierto indica el comienzo de un nuevo nivel; cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol.

*Árbol en listas:* 2 (7(2(6(5 11)))5(9(4)))



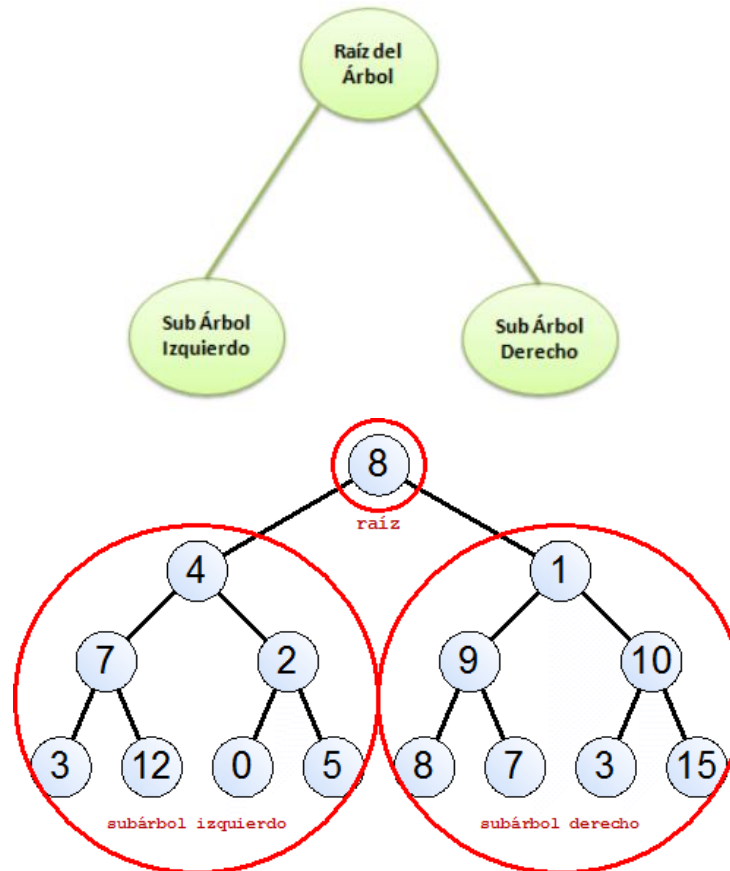
Un árbol se divide en **subárboles**. Un subárbol es cualquier estructura conectada por debajo de la raíz. Cada nodo de un árbol raíz de un subárbol que se define por el nodo y todos los descendientes del nodo. El primer nodo de un subárbol se conoce como la raíz del subárbol y se utiliza para nombrar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



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

### 2.2.1 Árboles binarios

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



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

Un árbol binario se divide en tres subconjuntos disjuntos:

$\{R\}$       *Nodo raíz*

$\{I_1, I_2, \dots, I_n\}$       *Subárbol izquierdo de R*

$\{D_1, D_2, \dots, D_n\}$       *Subárbol 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;
```

```

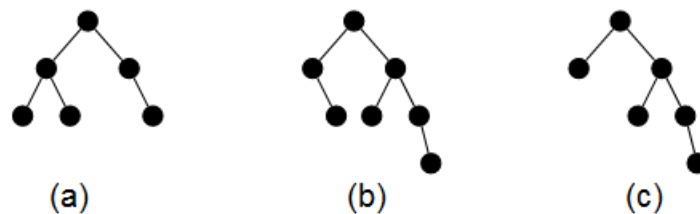
    struct _nodo *izquierdo;
} tipoNodo;
typedef tipoNodo *pNodo;
typedef tipoNodo *Arbol;

```

### 2.2.1.1 Equilibrio

La distancia de un nodo al raíz determina la eficiencia con la que puede ser localizado. Por ejemplo dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo 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. El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si definimos la altura del subárbol izquierdo como  $H_i$  y la altura del subárbol derecho como  $H_d$  entonces el factor de equilibrio del árbol B se determina por la siguiente fórmula:  $B = H_d - H_i$ .

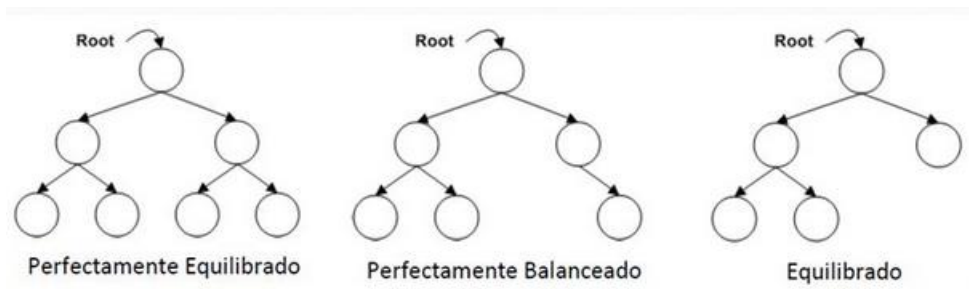


Utilizando esta fórmula el equilibrio del nodo raíz de los árboles de la figura anterior son:

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

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también perfectamente equilibrados.

Un árbol binario está equilibrado si la altura de sus árboles difiere en no más de uno (su factor de equilibrio es -1,0,+1) y sus subárboles son también equilibrados.



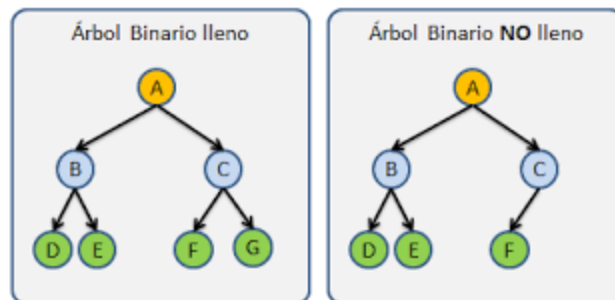


Un árbol binario está *equilibrado* si bien es vacío o bien cumple que la diferencia de alturas de sus dos hijos es como mucho 1 y además ambos están equilibrados.

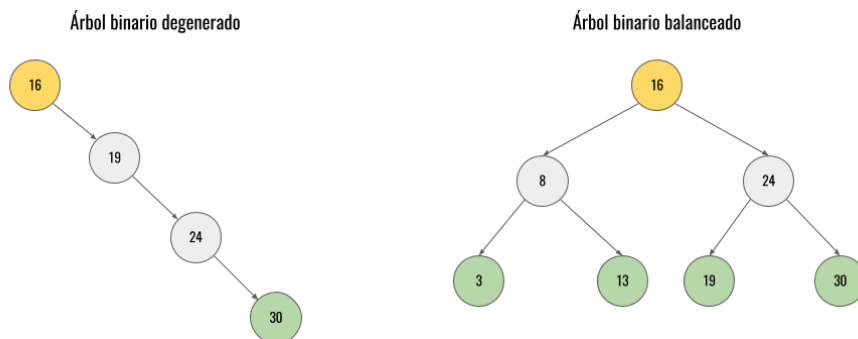
### 2.2.1.2 Árboles binarios completos

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

Un árbol binario completo que contiene  $2^n$  nodos a nivel  $n$  es un **árbol lleno**. Un árbol lleno es un árbol binario que tiene el máximo número de entradas para su altura. Esto sucede cuando el último nivel está lleno.



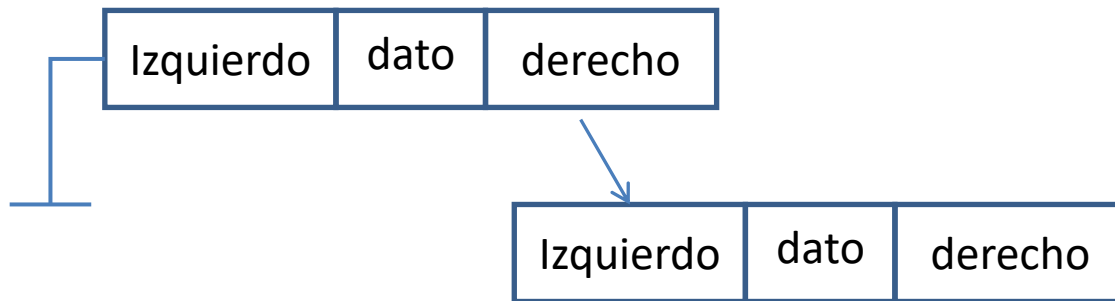
Llamamos **árbol degenerado** en el que hay un solo nodo hoja y cada nodo no hoja solo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.



### 2.2.1.3 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:

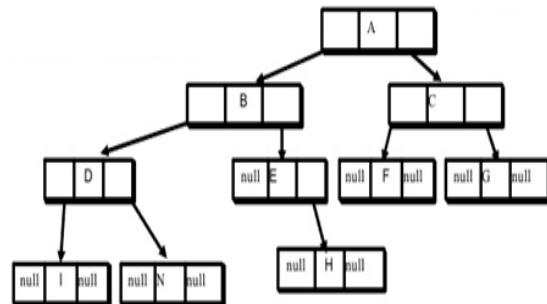
```

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

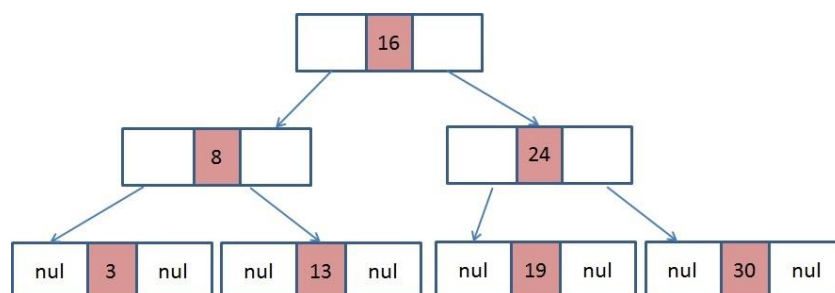
Ejemplo de árbol binario y su estructura de nodos

```

Typedef int TipoElemento;
Typedef struct nodo
{
    TipoElemento dato;
    Struct nodo *izq;
    Struct nodo *der;
} Nodo;
Typedef Nodo* ArbolBinario;
    
```



Representación de la figura del árbol binario balanceado anterior



### 2.2.2 Árbol binario de búsqueda

Los arboles vistos hasta ahora no tienen un orden definido; sin embargo, los arboles binarios ordenados tienen sentido. Estos árboles se denominan **árboles binarios de búsqueda**, debido a que se pueden buscar en ellos termino utilizando al 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.

6 menor que 15

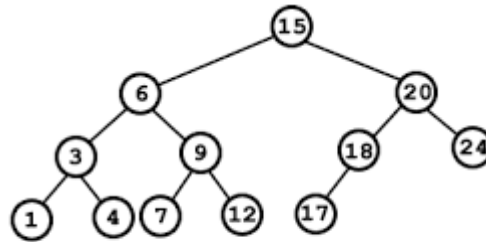
3 menor que 6

6 menor que 15

9 mayor que 6

.....

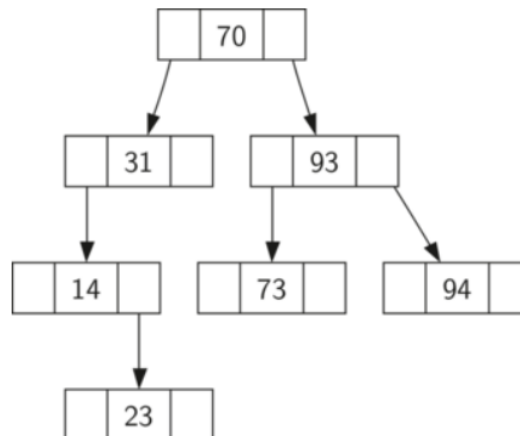
1 menor que 3



*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  $x$ .*

#### 2.2.2.1 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.



Dado que 70 fue la primera clave insertada en el árbol, es la raíz. A continuación, 31 es menor que 70, por lo que se convierte en el hijo izquierdo de 70. Luego, 93

es mayor que 70, por lo que se convierte en el hijo derecho de 70. Ahora tenemos dos niveles del árbol llenos, así que la siguiente clave va a ser el hijo izquierdo o derecho de 31 o 93. Dado que 94 es mayor que 70 y 93, se convierte en el hijo derecho de 93. Similarmente 14 es menor que 70 y 31, por lo que se convierte en el hijo izquierdo de 31. 23 es también menor que 31, por lo que debe estar en el subárbol izquierdo de 31. Sin embargo, es mayor que 14, por lo que se convierte en el hijo derecho de 14.

### 2.2.2.2 Implementación de un nodo de un árbol binario de búsqueda

Un árbol binario de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. 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*

*Nombre*                      Tipo de dato cadena (string)  
*Numero de libreta*      Tipo entero

Nombre	
numLibreta	
izda	dcha

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

### Creación 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;  
}
```

### 2.2.2.3 Operaciones en el arbol binario de busqueda

Como vimos hasta ahora, los arboles 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:

- A. *Búsqueda de un nodo*
- B. *Inserción de un nodo*
- C. *Recorrido del árbol*
- D. *Eliminación de un nodo*

#### A. Búsqueda

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

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

```

## B. Inserción de un nodo

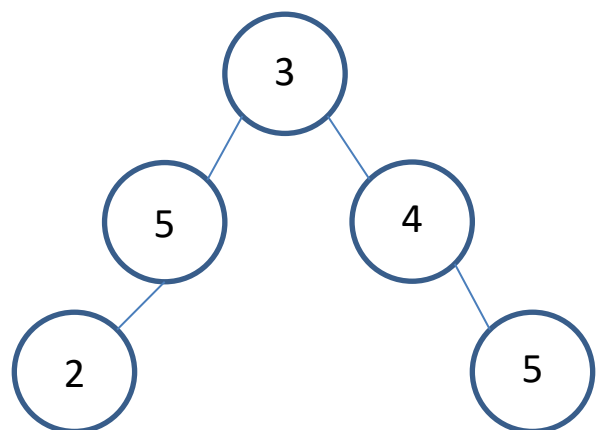
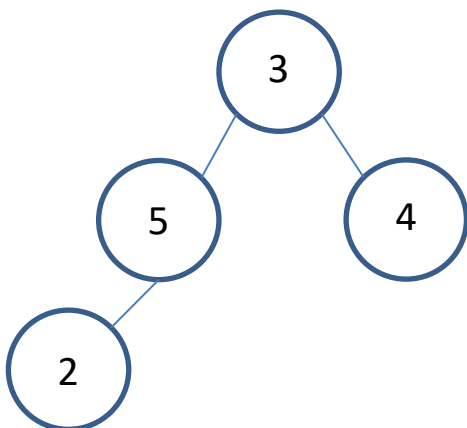
Una característica importante que debe poseer el algoritmo de inserción es que el árbol resultante de una inserción en un árbol de búsqueda ha de ser también de búsqueda. En esencia, el algoritmo de inserción se apoya en la localización de un elemento, de modo que si se encuentra el elemento (clave)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



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

## C. Recorrido del árbol

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

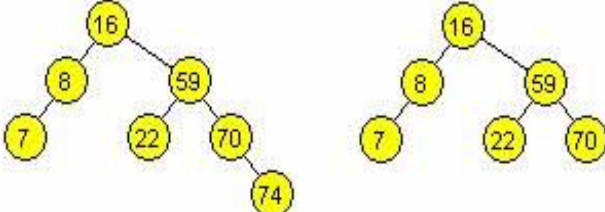
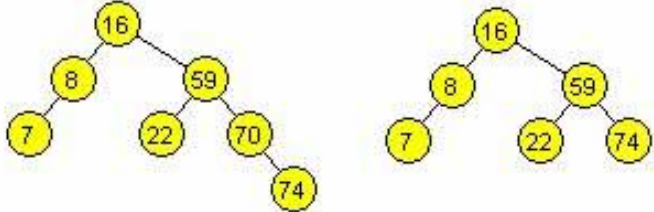
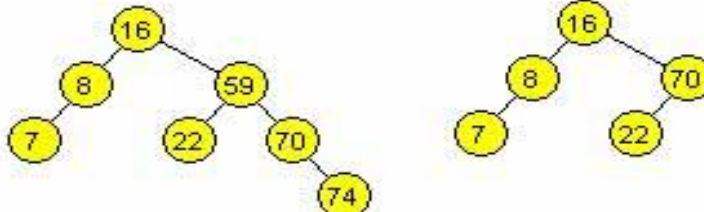
El *recorrido en profundidad* se realiza por uno de tres métodos recursivos: *preorden*, *inorden* y *postorden*. (ver apartado 3.3.4)

#### D. 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 más próximo en clave (inmediatamente superior o inmediatamente inferior) con el objeto de mantener la estructura de árbol binario.

Casos	
<b>Una hoja del árbol:</b> Si el nodo por eliminar es una hoja, entonces basta con destruir su variable asociada ( <code>delete</code> ) y, posteriormente, asignar <code>nil</code> a ese puntero.	
<b>Un nodo con un sólo hijo:</b> 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.	
<b>Un nodo con dos hijos:</b> 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	



<p>izquierdo, luego la primera tarea consistirá en buscar tal nodo; de éste se dice que es el predecesor del nodo por eliminar (&gt;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.</p>	
--	--

## Ejemplos

### 1) Insertar elemento

```
int main()
{
    ABB arbol = NULL;    // creado Arbol

    int n; // número 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;
}
```

## 2) Crear e insertar primer nodo

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

typedef struct nodo *ABB;
/* es un puntero de tipo nodo que hemos llamado ABB, que
ulitizaremos 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);
}
```

## 3) Recorrer un árbol

```
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);
    }

```

### 2.2.3 Recorridos en profundidad y amplitud

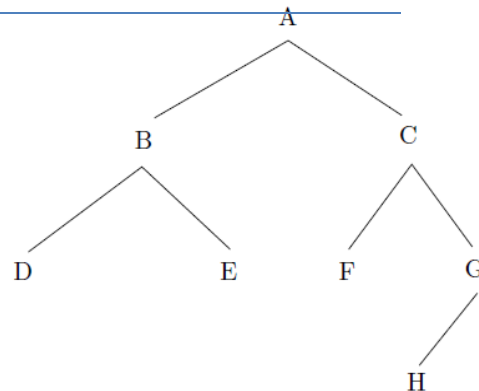
En ciencias de la computación, el **recorrido** de árboles se refiere al proceso de visitar de una manera sistemática, exactamente una vez, cada nodo en una estructura de datos de **árbol** (examinando y/o actualizando los datos en los nodos). Tales **recorridos** están clasificados por el orden en el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

**Si** el árbol no está vacío **entonces**

**Inicio**

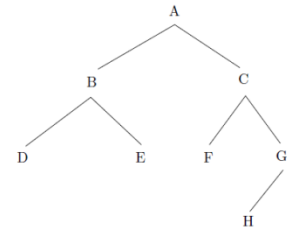
Recorrer el subárbol izquierdo

Visitar el nodo raíz

Recorrer 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 */
}
```

## Profundidad de un árbol binario

---

La profundidad de un árbol binario es una característica que se necesita conocer con frecuencia durante el desarrollo de una aplicación con árboles. La función profundidad evalúa la *profundidad* de un árbol binario. Para ello 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 una por separado. Las variables `profundidadI`, `profundidadD` almacenarán las profundidades de los subárboles izquierdo y derecho respectivamente.

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

## 2.3 Grafos

### 2.3.1 Definición

Son estructuras de datos NO lineales.

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

Conjunto de puntos  $X = x_1, x_2, x_3, \dots x_n$

Conjunto de líneas o aristas se denota con la siguiente expresión

$$L = \{ L_{ij} / X_i \text{ y } X_j \text{ est\u00e1n conectados} \}$$

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

Los grafos no son m\u00e1s que la versi\u00f3n general de un \u00e1rbol, es decir, cualquier nodo de un grafo puede apuntar a cualquier otro nodo de \u00e9ste (incluso a \u00e9l mismo).

Este tipo de estructuras de datos tienen una caracter\u00edstica que lo diferencia de las estructuras que hemos visto hasta ahora: los grafos se usan para almacenar datos que est\u00e1n relacionados de alguna manera (relaciones de parentesco, puestos de trabajo, ...); por esta raz\u00f3n se puede decir que los grafos representan la estructura real de un problema.

En lo que a ingenier\u00eda de telecomunicaciones se refiere, los grafos son una importante herramienta de trabajo, pues se utilizan tanto para dise\u00f1o de circuitos como para calcular la mejor ruta de comunicaci\u00f3n en Internet.

Un grafo es la representaci\u00f3n simb\u00f3lica de los elementos constituidos de un sistema o conjunto, mediante esquemas gr\u00e1ficos. Se puede decir tambi\u00e9n, que un grafo consiste en un conjunto de nodos (tambi\u00e9n llamados v\u00e9rtices) 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 v\u00e9rtices 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 pr\u00e1ctico, los grafos permiten estudiar las interrelaciones entre unidades que interact\u00faan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los v\u00e9rtices representan los terminales y las aristas representan las conexiones inal\u00e1mblicas). En fin, pr\u00e1cticamente cualquier problema puede representarse mediante un grafo.



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

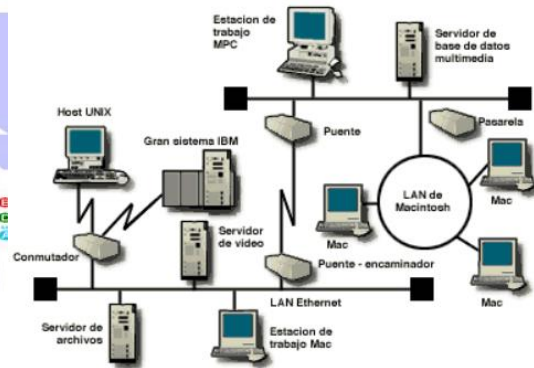
Los grafos tienen gran cantidad de aplicaciones:

- Representaci\u00f3n de circuitos electr\u00f3nicos anal\u00f3gicos y digitales
- Representaci\u00f3n de caminos o rutas de transporte entre localidades
- Representaci\u00f3n de redes de computadores.

## Redes de transporte



## Redes de computadoras



### 2.3.2 Glosario

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

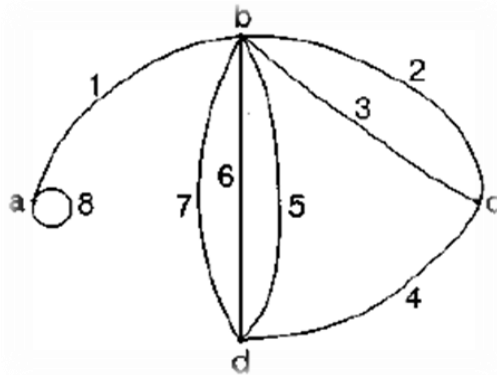
**Vértice:** Nodo.

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

Por ejemplo:

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

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

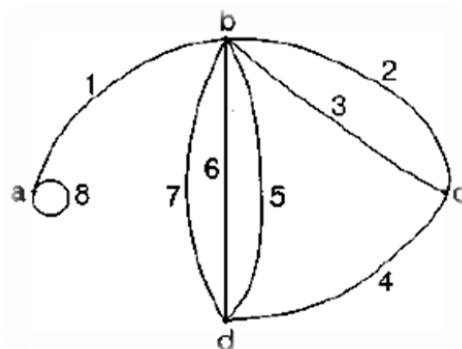


El número de elementos de  $V_G$  se llama **orden del grafo**.

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.



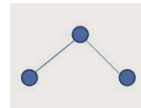
Los Vértices, son los objetos representados por un punto dentro del grafo.



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

Los grafos se puede clasificar en:

### Dirigido

---

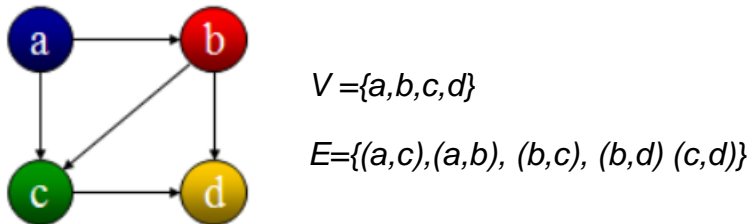
Un grafo dirigido o dígrafo es un tipo de grafo en el cual las aristas tienen una dirección definida, a diferencia del grafo generalizado, en el cual la dirección puede estar especificada o no.

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).



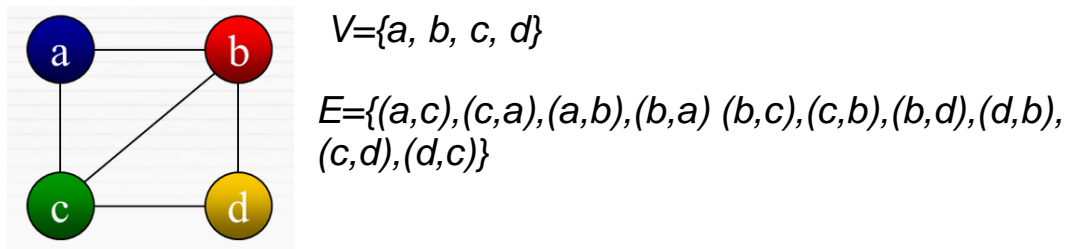
*No dirigido*

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.



### *Grafo conectado*

---

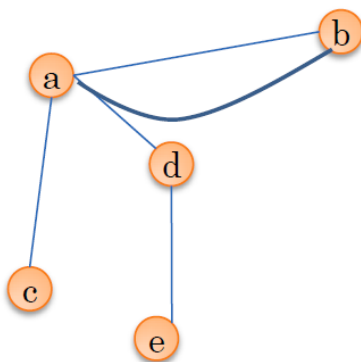
También llamado *Grafo conexo*, en matemáticas y ciencias de la computación es aquel grafo que entre cualquier par de sus vértices existe un camino (Grafo) que los une.

Existe siempre un camino que une dos vértices cualesquiera.

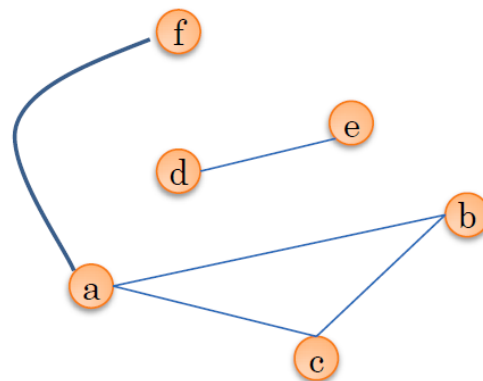
### *Grafo desconectado*

---

Existen vértices que no están unidos por un camino.



Grafo conexo



Grafo no conexo

### *Grafo Simple*

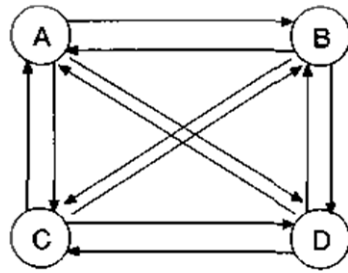
---

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 uniéndolos 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.

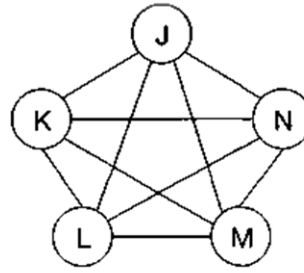
### *Grafo Completo*

---

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



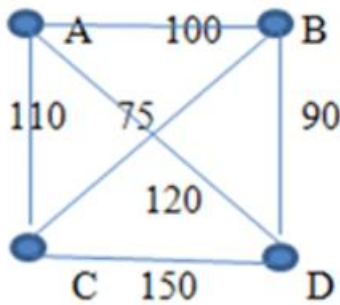
(b) grafo completo no dirigido

### 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.

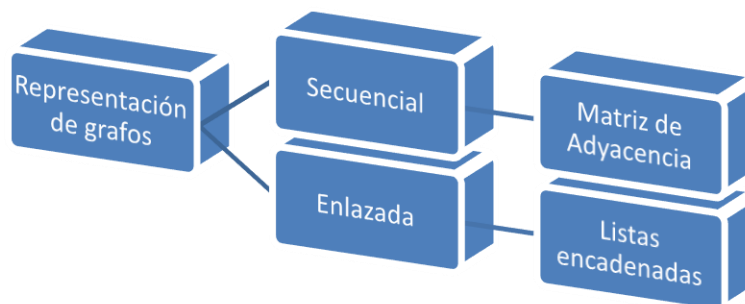


En este caso, por ejemplo se lee que la arista que une los vértices A y B tiene una ponderación de 100.

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.

### 2.3.3 Representación por matriz de adyacencia y lista de adyacencia



### 2.3.3.1 Matriz de adyacencia

Existen dos formas estándar de mantener un grafo  $G$  en la memoria de una computadora:

**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**.

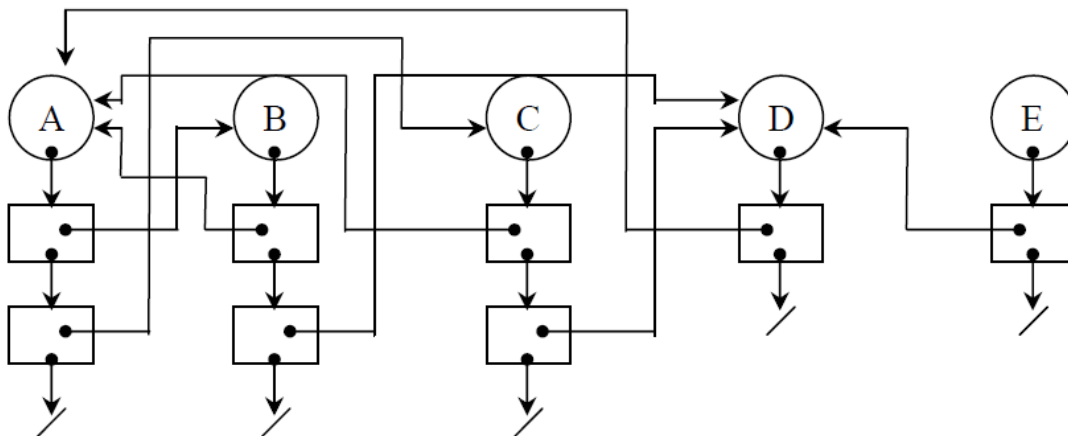
- Si no tuviera pesos:

	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:



### 2.3.3.1 Matriz de adyacencia

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

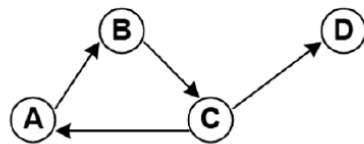
$$M(i, j) = \begin{cases} 1 & \text{si existe una arista } (V_i, V_j) \text{ en } A_G, V_i \text{ es adyacente a } V_j \\ 0, & \text{en caso contrario} \end{cases}$$

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

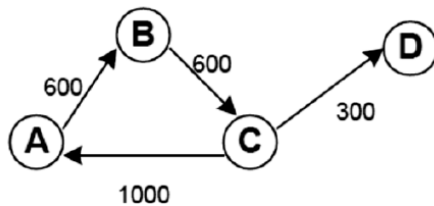
Si existe una arista desde *i* a *j* (esto es, el vértice *i* es adyacente a *j*), se introduce el costo o peso de la arista *i* a *j*, si no existe la arista, se introduce **0**.

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  $M(i, j) = M(j, i)$

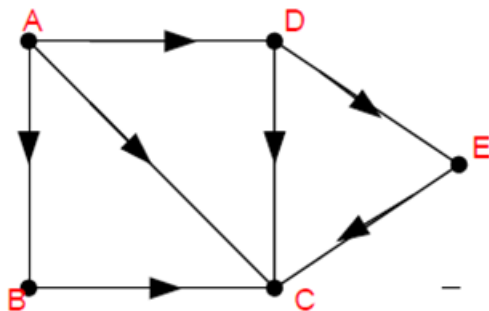


	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	1	0	0	1
D	0	0	0	0



	A	B	C	D
A	0	600	1000	0
B	600	0	600	0
C	1000	600	0	300
D	0	0	300	0

### 2.3.3.2 Listas de adyacencia

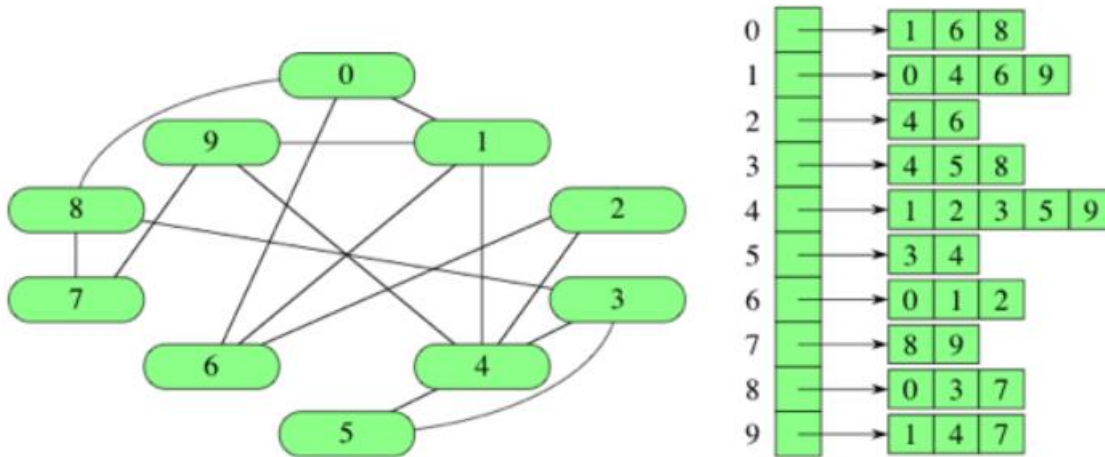


(A) Grafo G

Nodo	Lista de adyacencia
A	B, C, D
B	C
C	C, E
D	C, E
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. Típicamente 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:



### 2.3.4 Operaciones sobre grafos

En los grafos, como en todas las estructuras de datos, las dos operaciones básicas son insertar y borrar. En este caso, cada una de ellas se desdobra en dos, para insertar/eliminar vértices e insertar/eliminar aristas.

#### *Insertar vértice*

La operación de inserción de un nuevo vértice es una operación muy sencilla, únicamente consiste en añadir una nueva entrada en la tabla de vértices (estructura de datos que almacena los vértices) para el nuevo nodo. A partir de ese momento el grafo tendrá un vértice más, inicialmente aislado, ya que ninguna arista llegará a él.

#### *Insertar arista*

Esta operación es también muy sencilla. Cuando se inserte una nueva arista en el grafo, habrá que añadir un nuevo nodo a la lista de adyacencia (lista que almacena los nodos a los que un vértice puede acceder mediante una arista) del nodo origen, así si se añade la arista (A,C), se deberá incluir en la lista de adyacencia de A el vértice C como nuevo destino.

### 2.3.5 Recorrido y búsqueda en profundidad

La búsqueda en profundidad se usa cuando queremos probar si una solución entre varias posibles cumple con ciertos requisitos; como sucede en el problema del camino que debe recorrer un caballo en un tablero de ajedrez para pasar por las 64 casillas del tablero.

### 2.3.6 Recorrido y búsqueda en amplitud

Recorrido en amplitud (Breadth First Search, BFS) es otra forma sistemática de visitar los vértices. Este enfoque se denomina en amplitud porque desde cada vértice  $v$  que se visita se busca en forma tan amplia como sea posible, visitando todos los vértices adyacentes a  $v$ . Es una generalización del recorrido por niveles de un árbol.

La búsqueda por anchura se usa para aquellos algoritmos en donde resulta crítico elegir el mejor camino posible en cada momento del recorrido.

En la figura a continuación se muestra un grafo no conectado donde las flechas naranjas indican el recorrido del algoritmo BFS.

