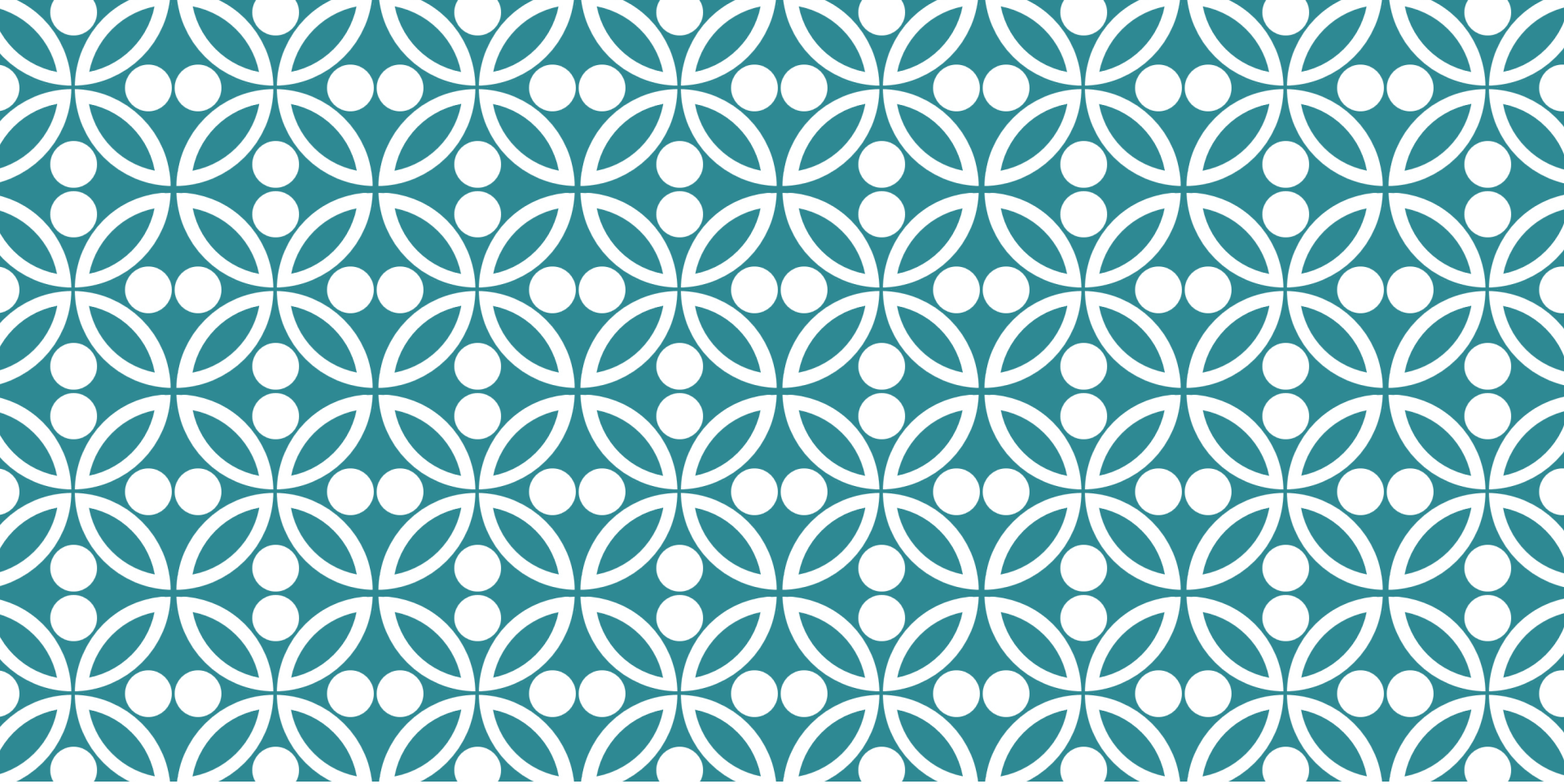


ÁRBOLES BINARIOS

Algoritmos y Estructuras de Datos II

Lic. Ana María Company

QUÉ ES UN ÁRBOL?

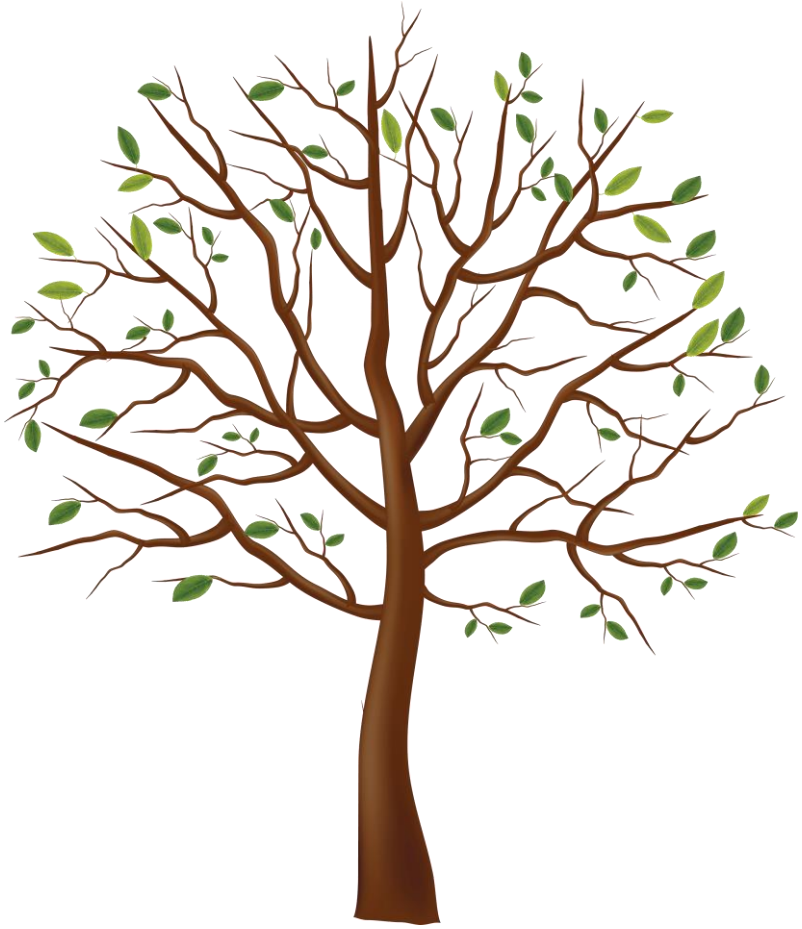
- Es una generalización del concepto de ***lista***, donde se permite que cada registro del tipo de dato dinámico tenga más de un enlace.
- Los árboles son estructuras de datos recursivas más generales que una lista y son apropiados para aplicaciones que involucran algún tipo de jerarquía (ej: miembros de una familia, trabajadores de una organización), o de ramificación (como los árboles de juegos), o de clasificación y/o búsqueda.
- La definición recursiva de árbol es muy sencilla:
- ***Un árbol o es vacío o consiste en un nodo que contiene datos y punteros hacia otros árboles.***

QUÉ ES UN ÁRBOL?

- Un árbol A es un conjunto finito de 1 o más nodos:
  - Existe un nodo especial denominado **RAIZ**(v_1) del árbol
  - Los nodos restantes (v_2, v_3, …, v_n) se dividen en m >= 0 conjuntos disjuntos denominado A_1, A_2, …, A_m , cada uno de los cuales, es a su vez un árbol, que se llaman ***subárboles del RAIZ***
- Un árbol con ningún nodo es un ***árbol nulo***: no tiene raíz.

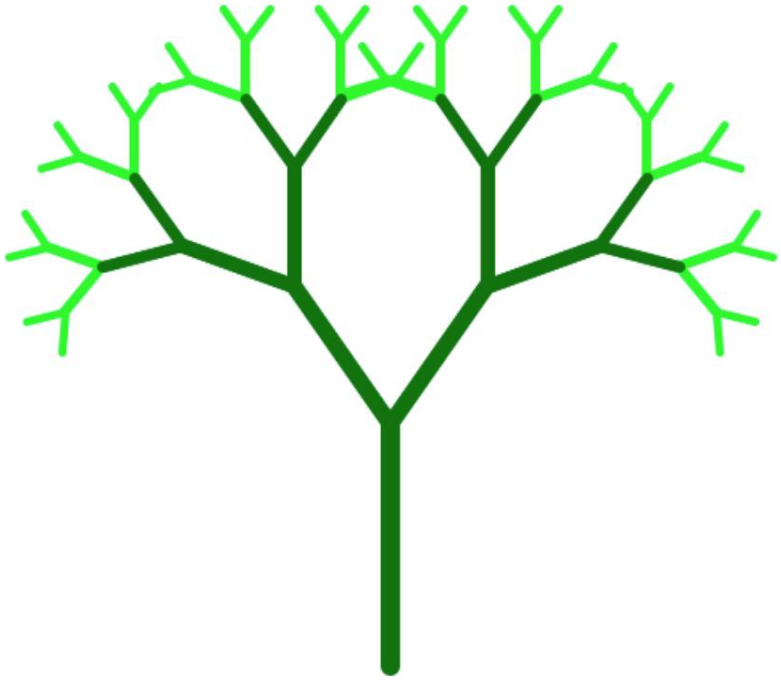
TERMINOLOGÍA

- *Raíz*
- *Hijos*
- *Hojas*
- *Ascendientes*
- *Descendientes*
- *Hermanos*
- *Padres*

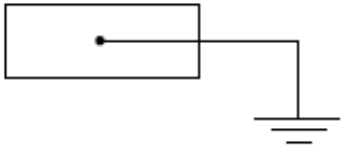


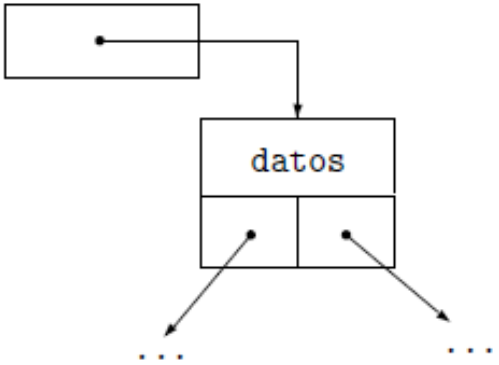
ÁRBOL BINARIO

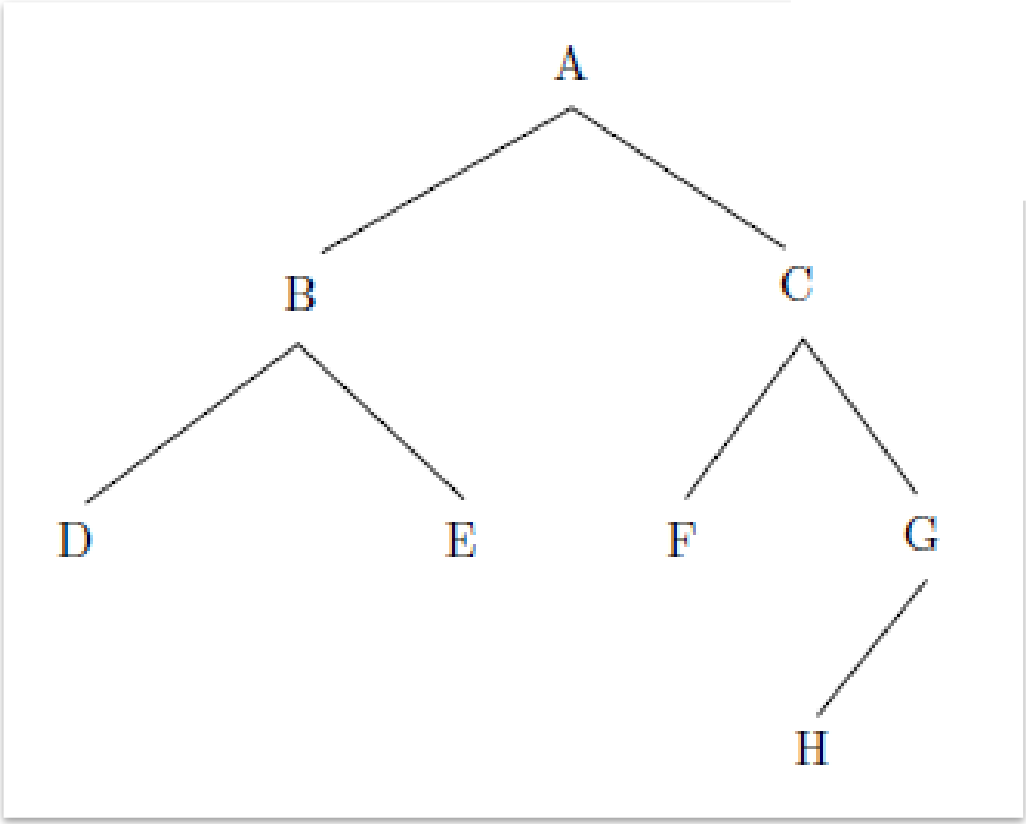
- Un árbol binario es un conjunto finito de cero o más nodos:
  - Existe un nodo denominado ***raíz*** del árbol.
  - Cada nodo puede tener 0, 1 ó 2 subárboles, conocidos como ***subárbol izquierdo*** y ***subárbol derecho***.
  - Es decir, cada nodo tiene a lo sumo dos descendientes.



ÁRBOL BINARIO - GRÁFICAMENTE







DEFINICIÓN DEL TIPO ÁRBOL BINARIO

- Cada nodo del árbol posee dos punteros en lugar de uno:

```
typedef struct nodoArbol {
   int contenido;
   struct nodoArbol * hijoIzdo;
   struct nodoArbol * hijoDcho;
}tArbol;
```

RECORRIDO DE UN ÁRBOL BINARIO

- Definir un algoritmo de recorrido de un árbol binario no es una tarea directa debido a que no es una estructura lineal, existen distintas formas de recorrerlo.
- A partir de un nodo podemos realizar alguna de las siguientes operaciones:
  - Leer el valor del nodo
  - Continuar por el hijo izquierdo
  - Continuar por el hijo derecho
- El orden en el que se efectúen las tres operaciones anteriores determinará el orden en el que los valores de los nodos del árbol son leídos.

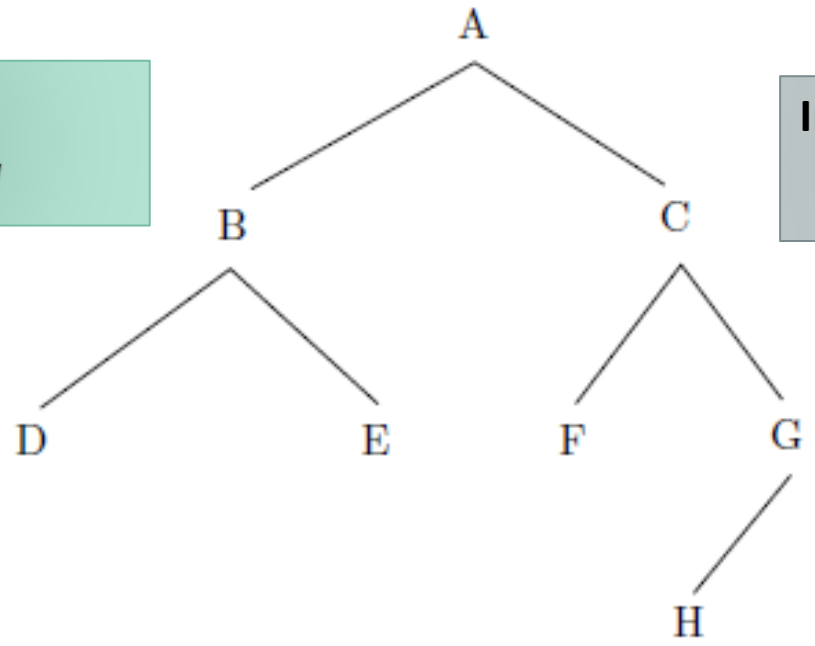
RECORRIDO DE UN ÁRBOL BINARIO

- Si se acuerda que siempre se leerá primero el hijo izquierdo y después el derecho, existen tres maneras diferentes de recorrer un árbol:
- ***Preorden***: Primero se lee el valor del nodo y después se recorren los sub-árboles.
  - Esta forma de recorrer el árbol también recibe el nombre de recorrido ***Primero en Profundidad***.
- ***Inorden***: En este tipo de recorrido, primero se recorre el sub-árbol izquierdo, luego se lee el valor del nodo y, finalmente, se recorre el sub-árbol derecho.
- ***Postorden***: En este caso, se visitan primero los sub-árboles izquierdo y derecho y después se lee el valor del nodo.

RECORRIDO DE UN ÁRBOL BINARIO - GRÁFICAMENTE

Preorden:

A B D E C F G H



Inorden:

D B E A F C H G

Postorden:

D E B F H G C A

ÁRBOLES DE BÚSQUEDA

- Son un caso particular de *árbol binario*.
- Son llamados **Árboles binarios de búsqueda** o **Árboles de búsqueda binaria**.
- Son aquellos árboles en los que el valor de cualquier nodo es mayor que el valor de su hijo izquierdo y menor que el de su hijo derecho.
- Entonces, no puede haber dos nodos con el mismo valor en este tipo de árbol.

BÚSQUEDA DE UN NODO

La versión recursiva de la función es sencilla, todo consiste en partir de la raíz y rastrear el árbol en busca del nodo en cuestión.

- si *arbol* es vacío entonces
  - Devolver error
- en otro caso si *arbol->contenido == dato* entonces
  - Devolver el puntero a la raíz de *arbol*
- en otro caso si *arbol->contenido* **>** *dato* entonces
  - Buscar en el hijo izquierdo de *arbol*
- en otro caso si *arbol->contenido* **<** *dato* entonces
  - Buscar en el hijo derecho de *árbol*

INSERCIÓN DE UN NUEVO NODO

Para la inserción del nodo, debemos encontrar el lugar conveniente donde insertar el nuevo nodo, esto se hace en función de su valor de manera similar a lo visto en el ejemplo anterior.

En pseudocódigo:

- Si *arbol es vacío* entonces
  - crear nuevo nodo
- en otro caso si *árbol->contenido* **>** *datoNuevo* entonces
  - Insertar ordenadamente en el hijo izquierdo de *arbol*
- en otro caso si *arbol->contenido* **<** *datoNuevo* entonces
  - Insertar ordenadamente en el hijo derecho de *arbol*

- La forma resultante de un árbol binario de búsqueda depende bastante del orden en el que se vayan insertando los datos:
  - Si los datos ya están ordenados el árbol degenera en una lista.

ELIMINACIÓN DE UN NODO

- La eliminación de un nodo en un árbol de búsqueda binaria no es una tarea sencilla a la hora de implementar.
- Tendremos en cuenta los siguientes casos según la condición del nodo que se desea eliminar:
  - Si el nodo es una ***hoja*** del árbol,
  - Un nodo con ***un sólo hijo***, o
  - Un nodo con ***dos hijos***.
- Los dos primeros casos no tienen mayor complejidad, sin embargo el tercer caso (un nodo con dos hijos) sí requiere un mayor nivel de detalle a la hora de su programación.

ELIMINACIÓN DE UN NODO

- Primeramente se debe ubicar el ***nodo padre del nodo por eliminar***.
- Luego analizaremos cada caso en particular:
  1. Si el nodo a eliminar es una ***hoja***, entonces solo se debe destruir su variable asociada (haciendo uso del ***free***) y, luego asignar NULL a ese puntero.
  2. 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.
  3. Si el nodo por eliminar tiene ***dos hijos*** no se puede aplicar la técnica anterior, porque entonces habrá dos nodos que conectar y no lograríamos un árbol binario. Por lo tanto, se tendrán que realizar los pasos necesarios para eliminar el nodo deseado y renovar las conexiones de modo que se siga teniendo un árbol de búsqueda.

ELIMINACIÓN DE UN NODO

- Primero, se debe tener en cuenta 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 se debe buscar ese nodo, que es el predecesor del nodo por eliminar, y se debe conocer en que posición se encuentra el nodo predecesor.
- Una vez hallado el predecesor el resto es tarea sencilla, sólo es preciso copiar su valor en el nodo por eliminar y descartar el nodo predecesor.

ELIMINACIÓN DE UN NODO - ALGORITMO

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

REFERENCIAS

- Adam Drozdek. Data Structures and Algorithms in C++. Fourth Edition.
- Aditya Y. Bhargava. Grokking Algorithms. An illustrated guide for programmers and other curious people.
- Mark Allen Weiss. Estructuras de Datos y Algoritmos. Editorial: Addison-Wesley Iberoamericana.
- Joyanes Aguilar, Luis. Programación en Pascal. 4ª Edición. Editorial: McGraw-Hill/Interamericana de España, S.A.U.
- Cristóbal Pareja Flores, Manuel Ojeda Aciego, Ángel Andeyro Quesada, Carlos Rossi Jiménez. Algoritmos y Programación en Pascal.
- Joyanes Aguilar, Luis. Fundamentos de la Programación. Algoritmos, Estructuras de Datos y Objetos. 3ª Edición. Editorial: McGraw-Hill.
- Luis Joyanes Aguilar, Ignacio Zahonero Martinez. Programación en C. Metodología, algoritmos y estructura de datos. Editorial: McGraw-Hill.

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.

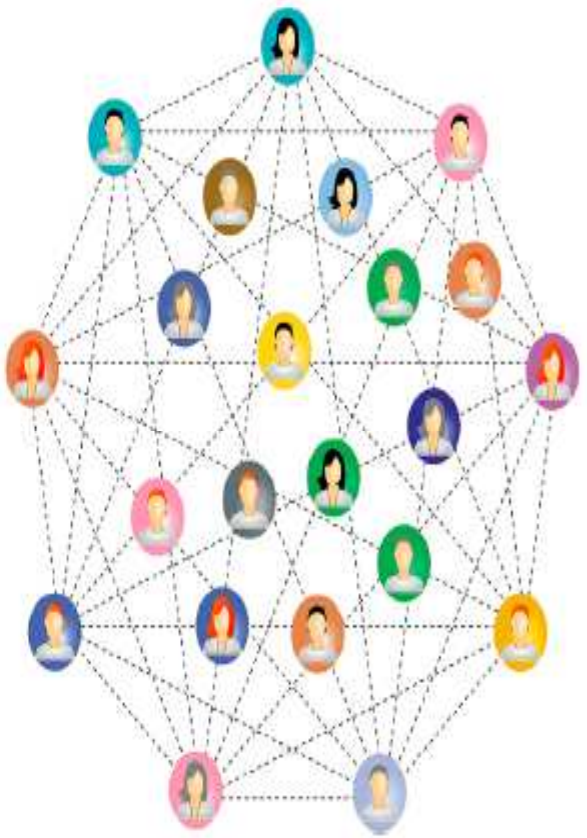
Un grafo es la representación simbólica de los elementos constituidos de un sistema o conjunto, mediante esquemas gráficos. Se puede decir también, que un grafo consiste en un conjunto de nodos (también llamados vértices) y un conjunto de arcos (aristas) que establecen relaciones entre nodos.

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.

GRAFOS

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.

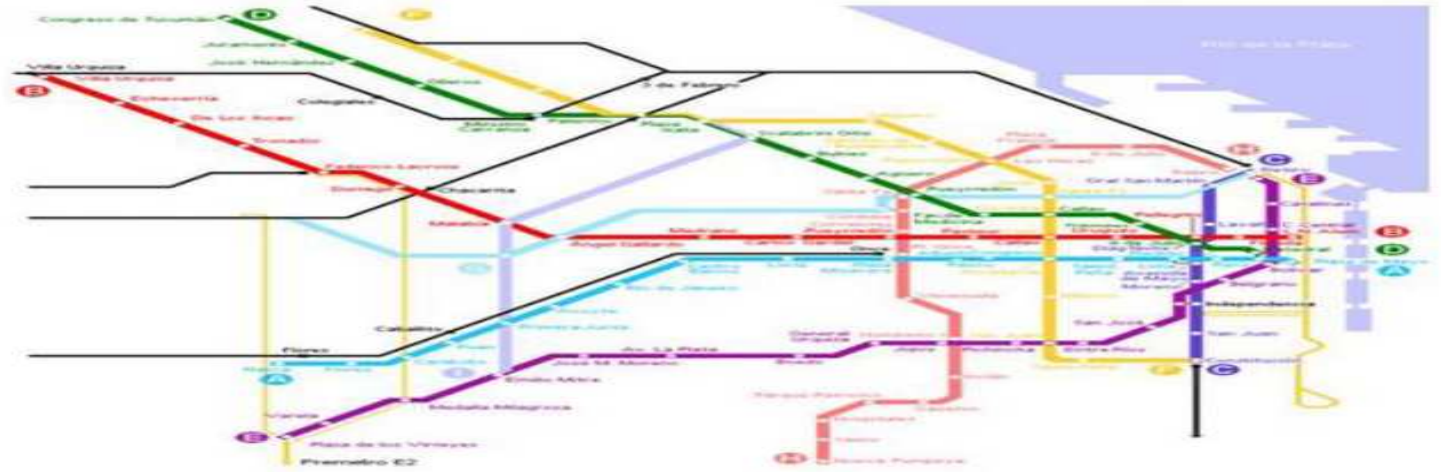
El estudio de grafos es una rama de la algoritmia muy importante.



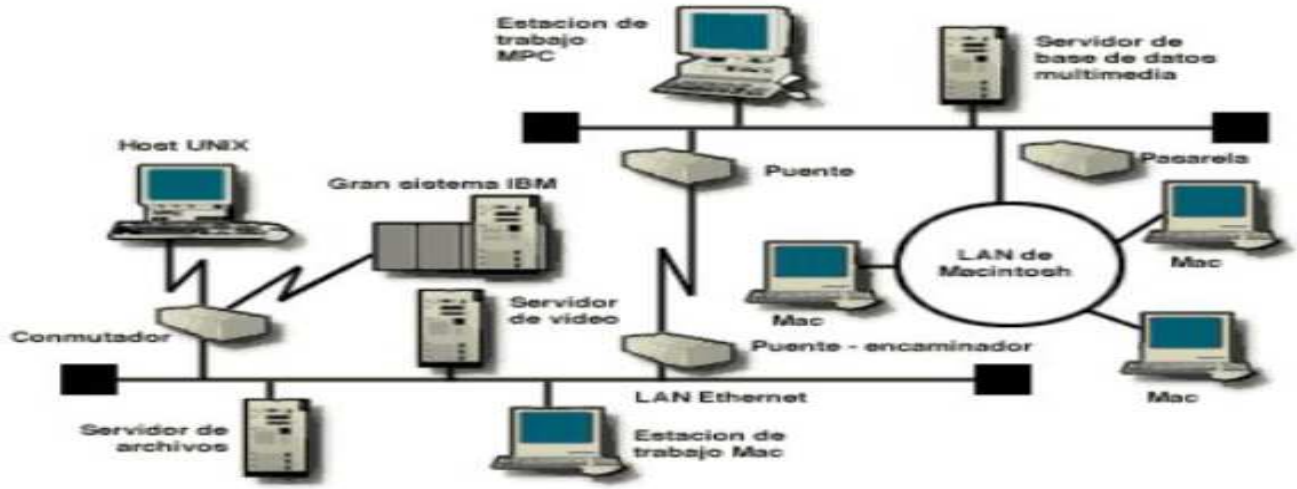
GRAFOS

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.



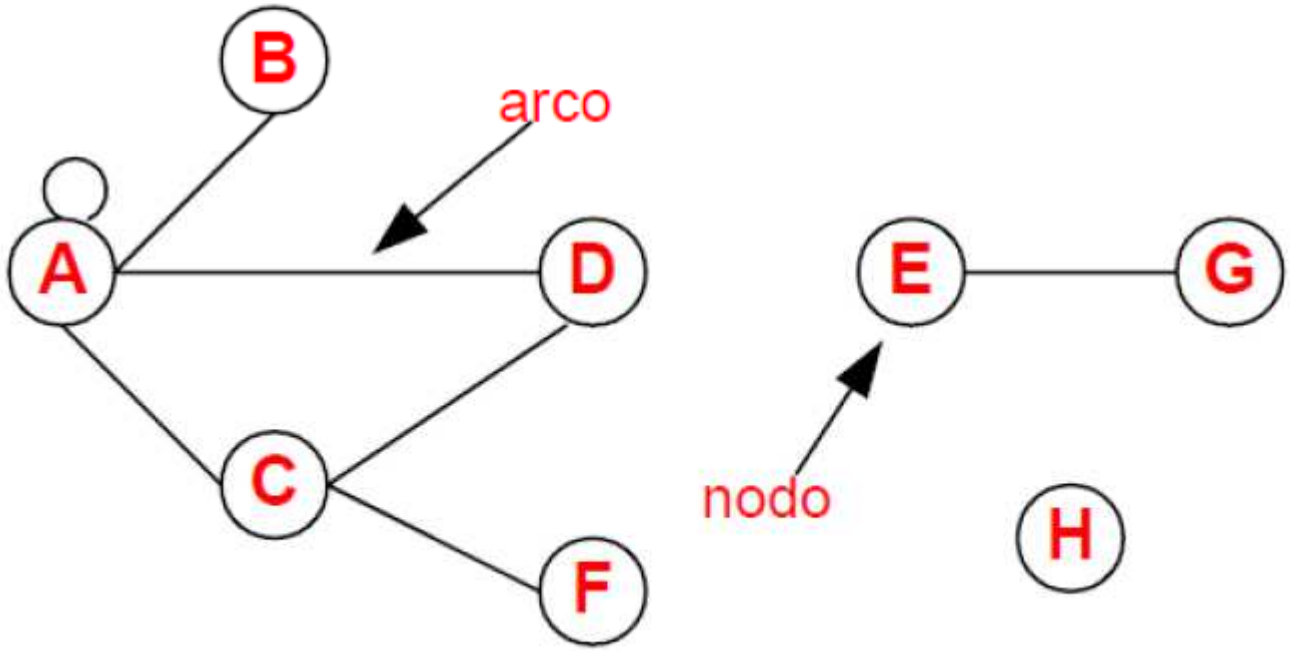
Redes de transporte



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



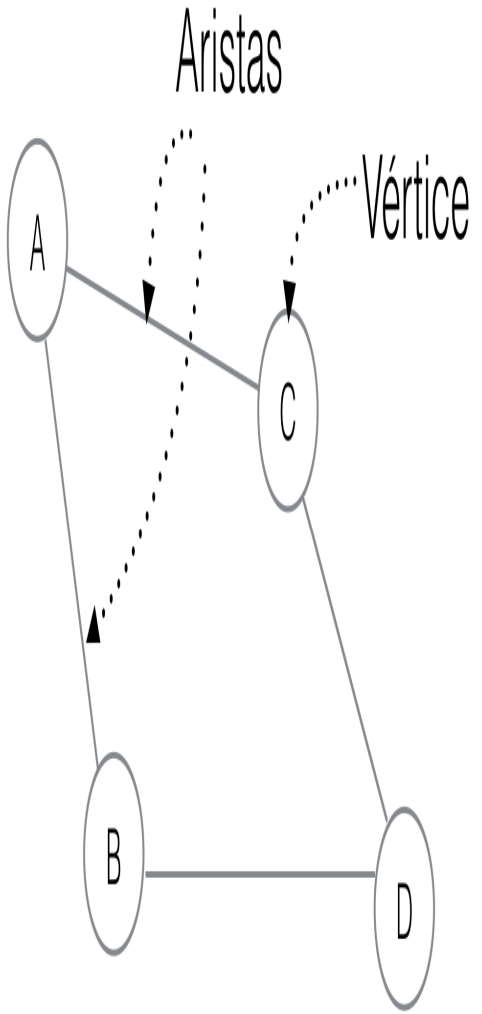
PARTES DE UN GRAFO

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 `impar' 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.

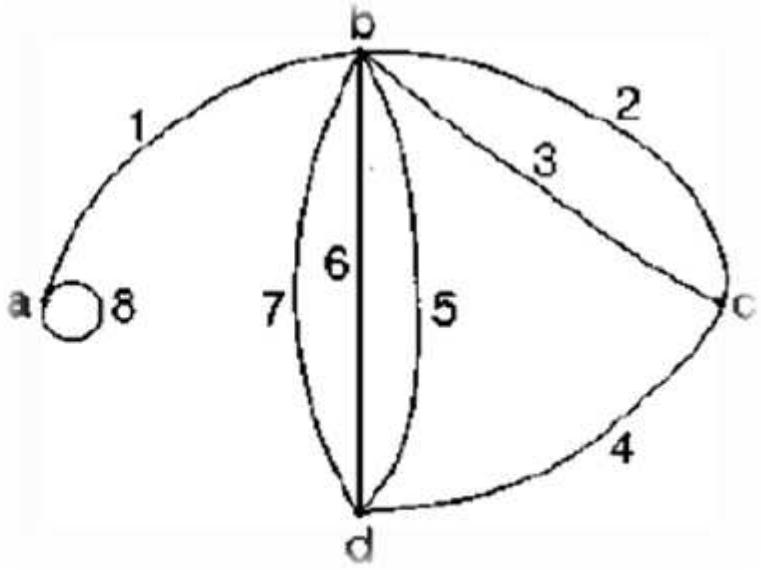
Vértice: Nodo.

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 Vg se llama **orden del grafo.**

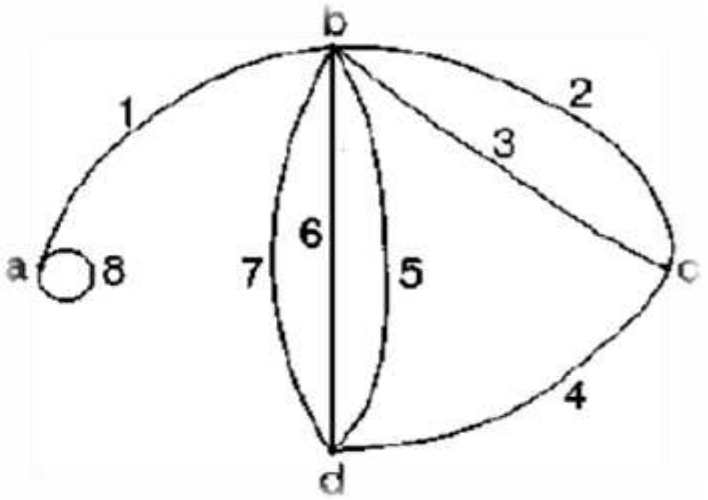
Un **grafo nulo** es un grafo de orden cero.

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.

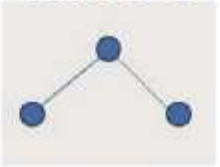
Vecindad: Conjunto de vértices adyacentes a otro.

¿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

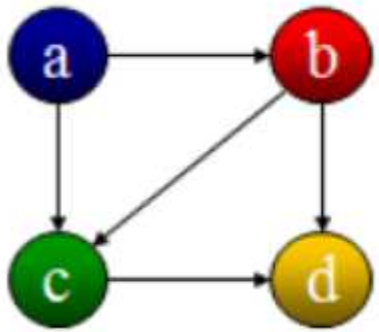
Dirigido

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:

- $V \neq 0$, 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

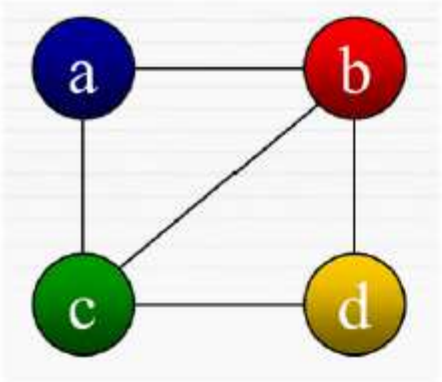
No dirigido

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

- $V \neq 0$
- $E \subseteq \{x \in P(V) : |x| = 2\}$ 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

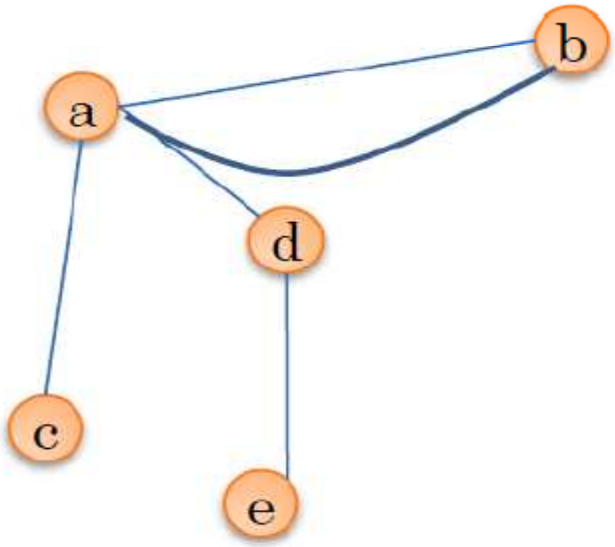
Grafo conectado

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.

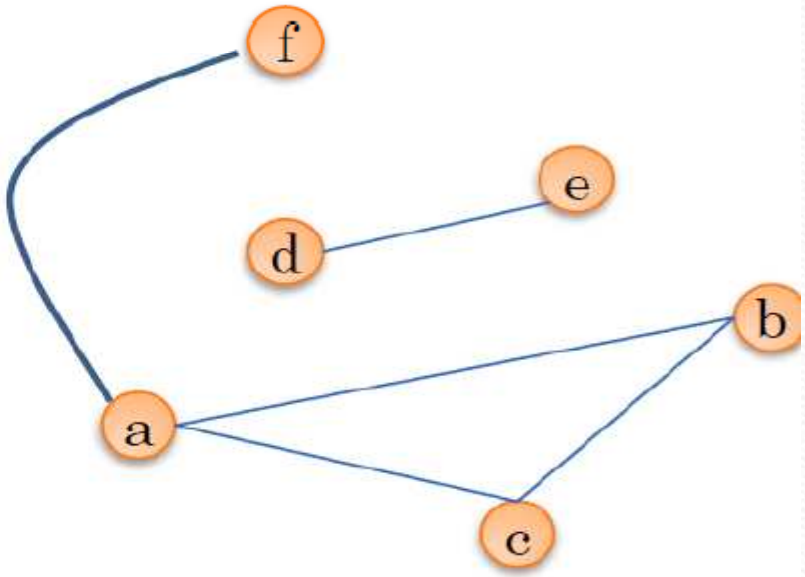
Existe siempre un camino que une dos vértices cualesquiera.

Grafo desconectado

Existen vértices que no están unidos por un camino.



Grafo conexo



Grafo no conexo

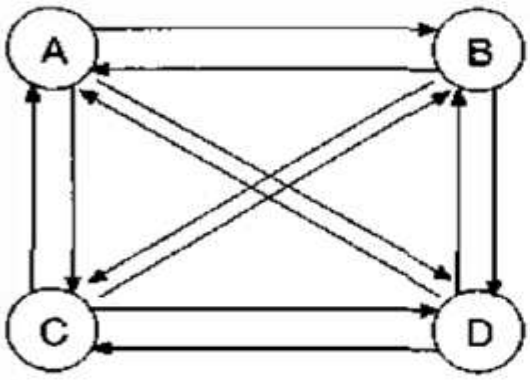
TIPOS DE GRAFOS

Grafo Simple

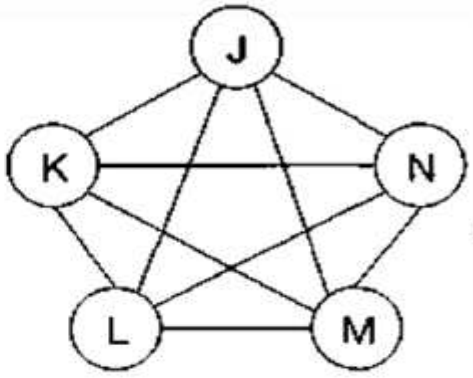
Es aquel grafo que no posee bucles o lazos. Se puede decir también, que un grafo es simple si a lo más existe una arista uniendo dos vértices cualesquiera. Esto es equivalente a decir que una arista cualquiera es la única que une dos vértices específicos. Un grafo que no es simple se denomina multigrafo.

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 disconexo a través de la eliminación de vértices, sería eliminándolos todos.



(a) grafo completo dirigido



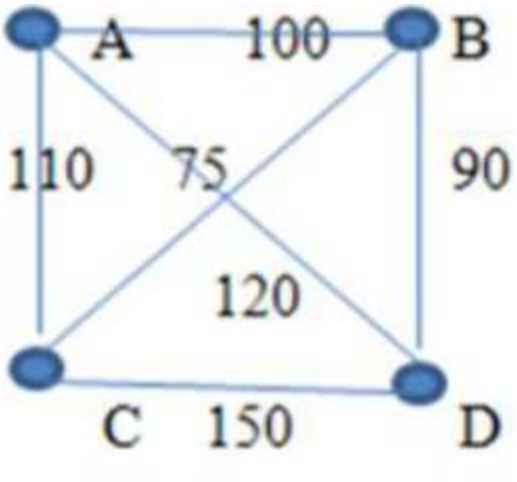
(b) grafo completo no dirigido

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.

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 ∞.
A esta matriz se le llama **Matriz de Adyacencia.**

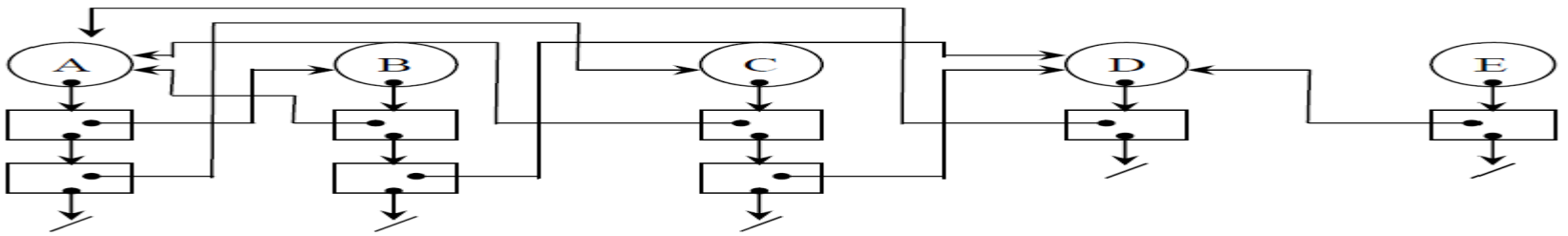
- Si no tuviera pesos:

	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	∞	∞
B	50	0	∞	8	∞
C	25	∞	0	12	∞
D	1	∞	∞	0	∞
E	∞	∞	∞	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 verticales.

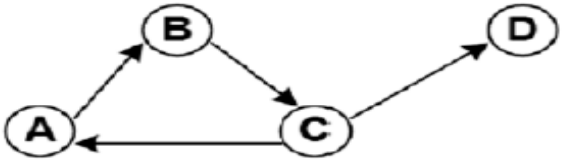
$$M(I, J) = \begin{cases} \textbf{1} \text{ si existe una arista } (V_i, V_j) \text{ en } A_G, V_i \text{ es adyacente a } V_j \\ \textbf{0}, \text{ en caso contrario} \end{cases}$$

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

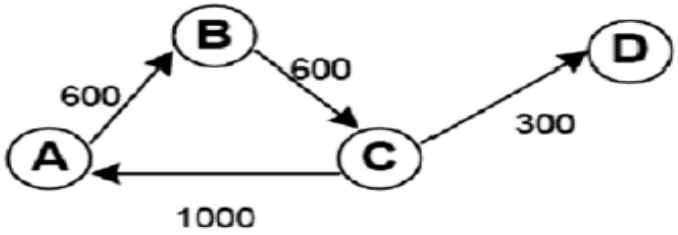
Si existe una arista desde i a j (esto es, el vértice i es adyacente a j), se introduce el costo o peso de la arista i a j, si no existe la arista, se introduce **0**.

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

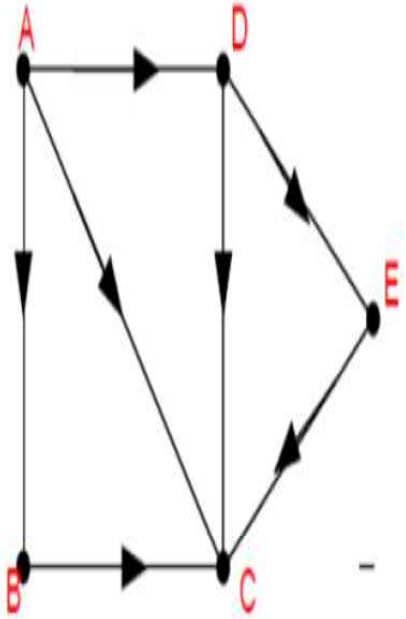


	A	B	C	D
A	0	600	1000	0
B	600	0	600	0
C	1000	600	0	300
D	0	0	300	0

Codigo 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

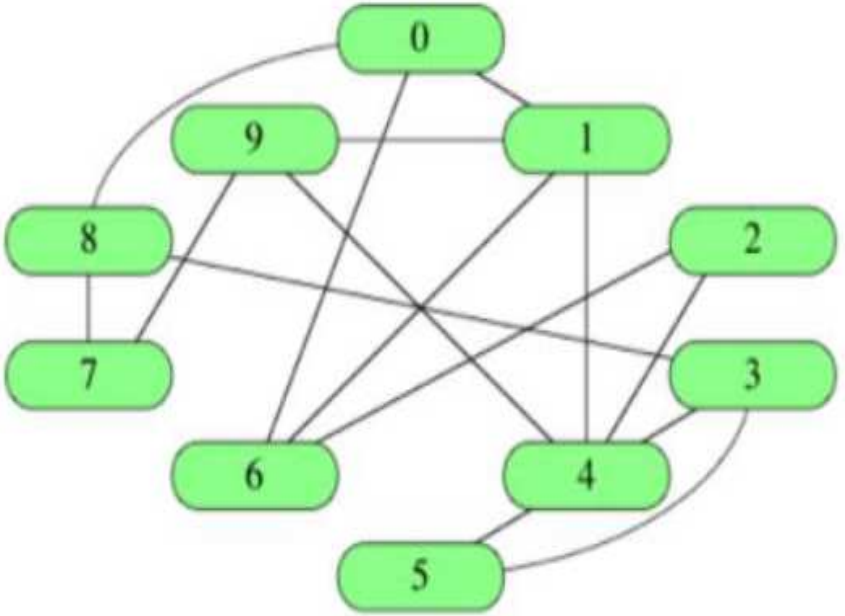


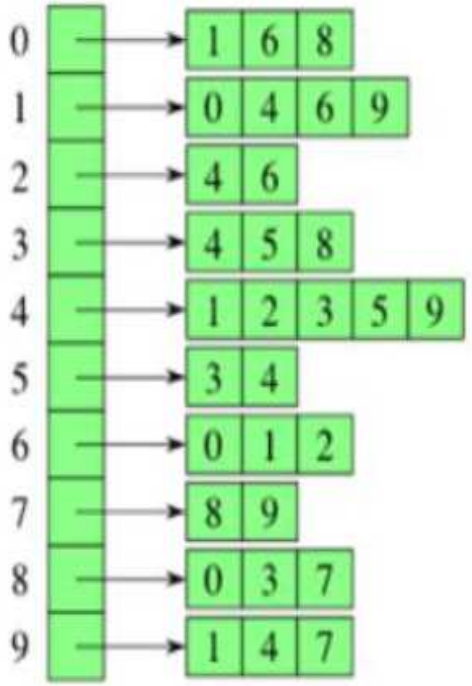
Nodo	Lista de adyacencia
A	B, C, D
B	C
C	
D	C, E
E	C

(A) Grafo G

(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:





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

Operaciones: PROCEDIMIENTO GRAFOVACIO O INICIAR GRAFO .

Estática .

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

Dinamica

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

AÑADIR VERTICE y arco

Estática

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

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

3.4.- BORRAR VERTICE y arco

Estática .

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

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

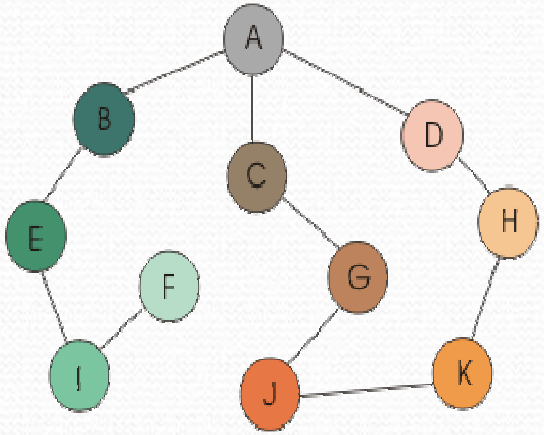
```
PROCEDURE BORRA_ARC (VAR GRAFO : TIPOGRAFO ; ARC : TIPOARCO ) ;
VAR POS1 , POS2 : TIPOGRAFO ;
AUX , ANT : PUNTEROARCO ;
ENC : BOOLEAN ;

BEGIN

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

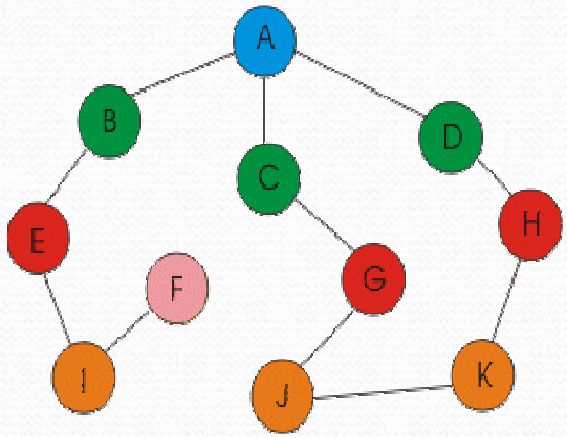
BORRAR VERTICE y arco Dinamica

Exploración de grafos

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

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





Búsqueda en profundidad (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 o) cada vez que se visita un nuevo vértice y se almacena id en la entrada del array val correspondiente al vértice que se está explorando.

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

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 Guisti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2. Capitulo 9.

Internet

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