

# TDA GRAFO

## □ GRAFOS

Sea  $V$  un conjunto de puntos:

$$V^2 = V \times V$$

- Un grafo es un par  $G = (V, E)$  con  $E \subset V^2$ .  
Los elementos de  $V$  se llaman *nodos* o *vértices*.  
Los elementos de  $E$  se llaman *lados*.
  - Un grafo se dice **dirigido** cuando el orden de los componentes de los pares de  $E$  es relevante:  $(x, y) \neq (y, x)$ .  
Sus lados se denominan *arcos*.
  - Un grafo se dice **no dirigido** cuando el orden de los componentes de los pares de  $E$  no es relevante.  
Sus lados se denominan *aristas*.
  - **Camino:** Dados  $u, v \in V$  se dice que existe un **camino** entre  $u$  y  $v$  si existe una secuencia de pares de  $E$ :  
$$(u, v_1), (v_1, v_2), \dots, (v_n, v)$$
  - Se llama *longitud* del camino al número de pares.
  - Un camino de  $u$  a  $v$  es un *ciclo* cuando  $u = v$ .
  - Un camino se dice *simple* cuando en él no hay ciclos.
  - Dados  $u, v \in V$  se dice que  $v$  es *adyacente* a  $u$  si  $(u, v) \in E$ .
- 
- Representación natural de objetos y relaciones entre éstos.
  - Grafos etiquetados y ponderados.
  - Casos particulares importantes de grafos:
    - Grafos dirigidos acíclicos (DAG).
    - Árboles.

## REPRESENTACIONES PARA GRAFOS DIRIGIDOS

### ➤ Matriz de Adyacencia.

Matriz cuadrada, A, de dimensión  $|V| \times |V|$ , tal que

- $A[i, j] = 1$  indica que el lado  $(i, j) \in E$
- $A[i, j] = 0$  indica que el lado  $(i, j) \notin E$

Problema: desperdicio de memoria para grafos dispersos.

### ➤ Listas de adyacencias.

Para cada nodo se incluye una lista con los nodos adyacentes a éste.

### ➤ Listas de cursores.

Se usan dos vectores. En el primero se incluyen por orden todas las listas de adyacencia separadas por un número que no representa vértice (p.ej.: -1). El segundo, C, con dimensión  $|V|$  tal que  $C[v]$  indica la posición del primer vector en que comienza la lista de adyacencia de v.

### ➤ Matriz de incidencia.

Matriz I de dimensión  $|V| \times |E|$  tal que

- $I[i, e] = 1$ , si  $e = (i, j)$
- $I[i, e] = -1$ , si  $e = (j, i)$
- $I[i, e] = 0$ , si  $i \notin e$ .

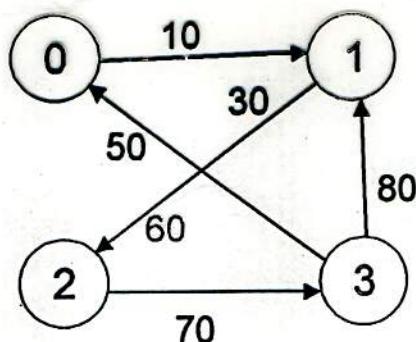
### ➤ Multilista.

Mediante una variación de la estructura básica de la Multilista podemos conseguir una implementación adecuada de la matriz de adyacencia, más eficiente para el caso de grafos dispersos.

### ➤ Estructuras duales.

Mezclando varias de las estructuras anteriores podemos obtener una representación de un grafo que resulta eficiente en la mayoría de los algoritmos a ejecutar sobre él.

## Representaciones para Grafos Dirigidos. Ejemplo



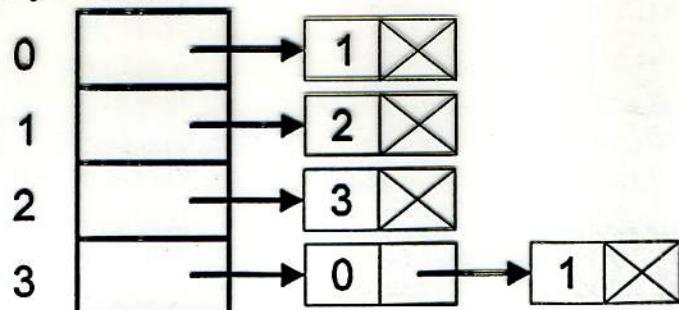
### > Matriz de Adyacencia y Costes

	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	0	0	0	1
3	1	1	0	0

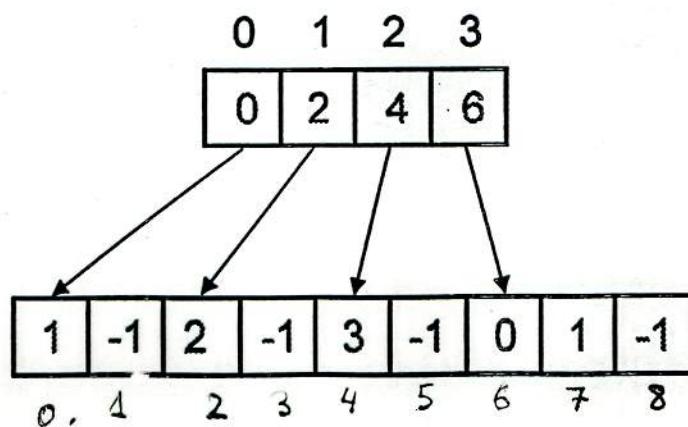
$$\begin{bmatrix} 0 & 10 & 0 & 0 \\ 0 & 0 & 30 & 0 \\ 0 & 0 & 0 & 70 \\ 50 & 80 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} - & 10 & \infty & \infty \\ \infty & - & 30 & \infty \\ \infty & \infty & - & 70 \\ 50 & 80 & \infty & - \end{bmatrix}$$

### > Listas de Adyacencia



### > Listas de cursos



## Caminos más cortos desde un nodo a los demás. (Algoritmo de Dijkstra)

Suponemos un grafo dirigido y ponderado con pesos no negativos, cuyos nodos están numerados: 0, ..., n-1. Su matriz de costes es C, de tal modo que C[i, j] indica el coste de ir desde el nodo i al nodo j. Si no existe ese lado, el coste es  $\infty$ .

- S: Conjunto de vértices {v} para los que conocemos el camino mínimo de 0 a v.
- D: Vector que almacena en la posición v, el coste de viajar desde 0 a v, pasando sólo por nodos de S.
- P: Vector de nodos, tal que P[v] es el nodo anterior a v en el camino desde 0 a v.

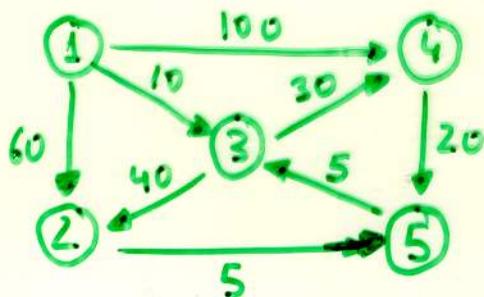
```

1:   S = {0};
2:   para (i = 1; i < n; i++)
3:     D[i] = C[0, i];
4:     P[i] = 0;
5:   para (i = 0; i < n-1; i++)
6:     Escoger un vértice w ∈ V - S tal que
       D[w] es un mínimo;
7:     insertar(S, w);
8:     para (cada vértice v ∈ V - S)
9:       si (D[w] + C[w, v] < D[v])
10:        D[v] = D[w] + C[w, v];
11:        P[v] = w;

```

- Eficiencia: Matriz de Adyacencia:  $O(n^2)$   
Lista de Adyacencia:  $O(a \log n)$
- El algoritmo es un ejemplo de Técnica "voraz" (Greedy).
- Ejemplo de aplicación: planificación de vuelos.

## EJEMPLO



ENCONTRAR CAMINOS MINIMOS DEL VERTICE 1 A LOS DEMAS

### INICIALMENTE

$$S = \{1\}$$

$$D[2] = 60; D[3] = 40; D[4] = 100; D[5] = \infty$$

$$P[i] = 1 \quad \forall i$$

### ITERACION 1

$$V - S = \{2, 3, 4, 5\} \quad w = 3 \Rightarrow S = \{1, 3\} \Rightarrow V - S = \{2, 4, 5\}$$

$$D[2] = \min(D[2], D[3] + G[3, 2]) = \min(60, 50) = 50 \Rightarrow P[2] = 3$$

$$D[4] = \min(D[4], D[3] + G[3, 4]) = \min(100, 40) = 40 \Rightarrow P[4] = 3$$

$$D[5] = \min(D[5], D[3] + G[3, 5]) = \min(\infty, \infty) = \infty$$

$$D[2] = 50; D[4] = 40; D[5] = \infty; P[2] = 3; P[4] = 3; P[5] = 1$$

### ITERACION 2

$$V - S = \{2, 4, 5\} \quad w = 4 \Rightarrow S = \{1, 3, 4\} \Rightarrow V - S = \{2, 5\}$$

$$D[2] = \min(D[2], D[4] + G[4, 2]) = \min(50, \infty) = 50; P[2] = 3$$

$$D[5] = \min(D[5], D[4] + G[4, 5]) = \min(\infty, 60) = 60; P[5] = 4$$

$$D[2] = 50; D[5] = 60; P[2] = 3; P[5] = 4;$$

### ITERACION 3

$$V - S = \{2, 5\} \quad w = 2 \Rightarrow S = \{1, 3, 4, 2\} \Rightarrow V - S = \{5\}$$

$$D[5] = \min(D[5], D[2] + G[2, 5]) = \min(60, 55) = 55; P[5] = 2;$$

$$D[5] = 55; P[5] = 2$$

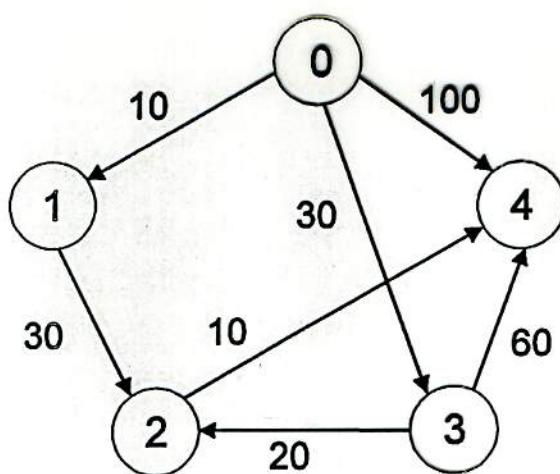
### FINALMENTE

$$w = 5 \Rightarrow S = \{1, 3, 4, 2, 5\} \rightarrow FIN$$

### CAMINO DE 1 A 5

$$P[5] = 2 \rightarrow P[2] = 3 \rightarrow P[3] = 1 \rightarrow (1, 3, 2, 5) \text{ COSTO } 55$$

### Ejemplo (Algoritmo de Dijkstra):



$$\begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 30 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

Iteración	S	w	D[1..4]
0	{0}		10, $\infty$ , 30, 100
1	{0,1}	1	10, <del>10</del> , 30, 100
2	{0,1,3}	3	10, <del>50</del> , 30, 90
3	{0,1,2,3}	2	10, <del>50</del> , 30, <del>50</del>
4	{0,1,2,3,4}	4	10, <del>50</del> , 30, <del>50</del>

Iteración	S	P[0]	P[1]	P[2]	P[3]	P[4]
0	{0}	0	0	0	0	0
1	{0,1}	0	0	1	0	0
2	{0,1,3}	0	0	<del>1</del>	0	3
3	{0,1,2,3}	0	0	<del>2</del>	0	2
4	{0,1,2,3,4}	0	0	<del>2</del>	0	2

## RECORRIDOS EN GRAFOS DIRIGIDOS

- > La resolución eficiente de muchos problemas relacionados con grafos dirigidos requiere realizar la visita sistemática de los vértices y/o arcos.
- > Existen dos recorridos principales, que son base de muchos algoritmos: búsqueda en profundidad y búsqueda en anchura.

### BÚSQUEDA EN PROFUNDIDAD

- > Generalización del recorrido en preorden de un árbol.
- marca: Vector de dimensión n. marca[v] indica si el nodo v se ha visitado o no. Suponemos un tipo enumerado: {visitado, no\_visitado} .

```

BusquedaProfundidad()
{
    para (v = 0; v < n; v++)
        marca[v] = no_visitado;
    para (v = 0; v < n; v++)
        si marca[v] == no_visitado
            bpp(v);
}

bpp(v)
{
    marca[v] = visitado;
    para cada w adyacente a v
        si marca[w] == no_visitado
            bpp(w);
}

```

- > Mejor representación: Lista de adyacencia
- > Eficiencia: O(a)

## RECORRIDO EN ANCHURA

- > Visita un nodo, después intenta visitar un vecino de éste, luego un vecino de este vecino y así sucesivamente.
- > Se usa principalmente para explorar parcialmente grafos infinitos o para encontrar el camino más corto entre dos puntos

```

BúsquedaAnchura()
{
    para cada v de V
        marca[v] = no_visitado
    para cada v de V
        si marca[v] == no_visitado
            bpa(v)
}

bpa(v)
{
    c = crear_cola();
    marca[v] = visitado;
    insertar(v, c);
    mientras (no vacia(c)) {
        u = cabeza(c);
        sacar(c);
        para cada vértice w adyacente a u {
            si marca[w] != visitado
                marca[w] = visitado;
                insertar(w, c);
        };
    };
}
;
```

## DETERMINACION DE CICLOS EN UN GRAFO

IDEA HACER SOBRE EL GRAFO UNA BPF MARCANDO EN CADA LLAMADA RECURSIVA EL VERTICE VISITADO HASTA QUE EN UNA LLAMADA ENCONTREMOS UNO PREVIAMENTE VISITADO EN CUYO CASO PARAMOS EL PROCESO PORQUE HABREMOS ENCONTRADO UN CICLO

FIN = FALSE ; CICLO = FALSE;

CICLICO (vertice v; vector\_de\_vertices visitado; int fin, ciclo;  
vertice ultimo)

{

visitado [v] = 1; i = 1;

while ( i ≤ n && !fin) do

if (M[v, i] == 1)

if visitado [i] == 0

ciclico (i, visitado, fin, ciclo, ultimo)

else {

fin = TRUE;

ciclo = TRUE;

ultimo = i;

}

else i++;

if (ciclo)

{

imprimir (v);

if (ultimo == v)

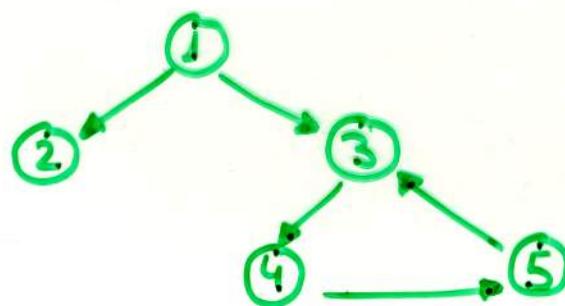
ciclo = FALSE

}

}

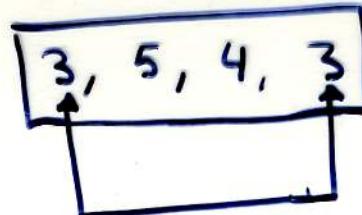
## EJEMPLO

BUSCAR UN CICLO EN EL GRAFO:



ciclico(1) { visitado[1] = 1;  
ciclico(2) { visitado[2] = 1  
ciclico(3) { visitado[3] = 1  
ciclico(4) { visitado[4] = 1  
ciclico(5) { visitado[5] = 1  
                  ciclico(3) FIN

y el ciclo sería:



### Ordenación Topológica

- > Dependencias temporales entre las tareas que componen un proyecto. (PERT).
- > Dependencias entre las asignaturas de un plan de estudios.

**Ordenación Topológica:** Ordenación lineal de los nodos de un grafo dirigido acíclico tal que si  $(i, j) \in E$  entonces i precede a j.

Se resuelve con una simple modificación del recorrido en profundidad del grafo G.

```

ClasificaciónTopológica()
{
    para (v = 0; v < n; v++)
        marca[v] = no_visitado;
    para (v = 0; v < n; v++)
        si marca[v] == no_visitado
            clasifTop(v);
}

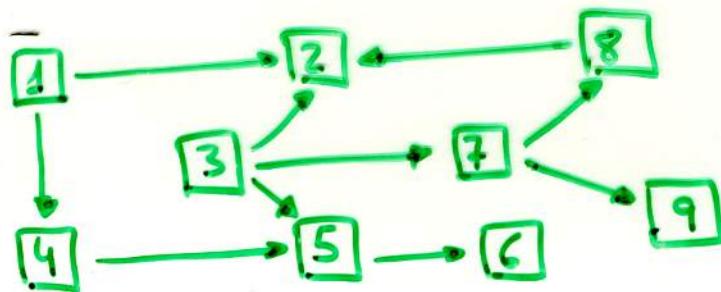
clasifTop(v)
{
    marca[v] = visitado;
    para cada w adyacente a v
        si marca[w] == no_visitado
            clasifTop(w);
    escribe(v);
}

```

En realidad, este algoritmo da una ordenación topológica de los vértices en sentido inverso. Para obtener una ordenación en sentido adecuado basta usar una pila.

## EJEMPLO

Encontrar una clasificación topológica en el grafo

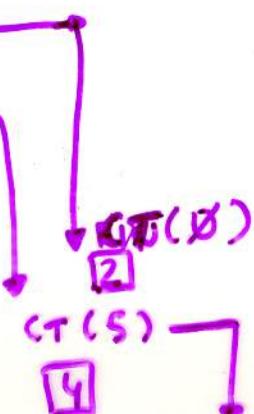


$CT(1)$

$\downarrow CT(2)$

$CT(4)$

1



→ 2, 6, 5, 4, 1

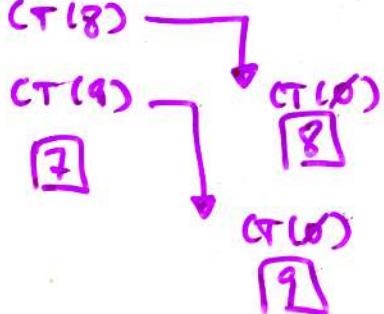
$CT(3)$

$\downarrow CT(7)$

3

$\downarrow CT(8)$

7



→ 8, 9, 7, 3

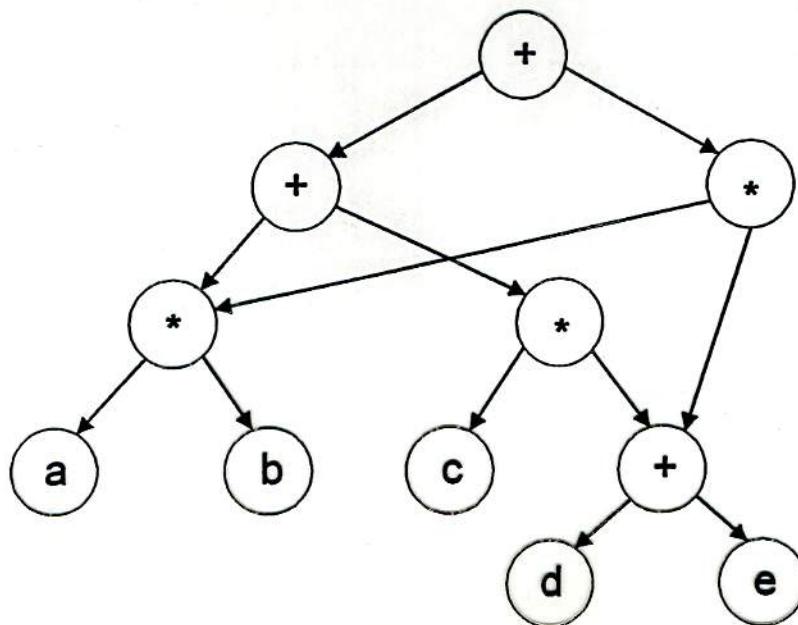
UNIMOS → 2, 6, 5, 4, 1, 8, 9, 7, 3 E INVERTIMOS:

3, 7, 9, 8, 4, 1, 5, 6, 2

(CLASIFICACION TOPOLOGICA BUSCADA)

## GRAFOS DIRIGIDOS ACÍCLICOS

➤ Expresión aritmética:  $a * b + c * (d + e) + (a * b) * (d + e)$



➤ Órdenes parciales. Un orden parcial R en un conjunto S es una relación binaria tal que:

1. (irreflexiva)  $\forall a \in S, (a, a) \notin R$

2. (transitiva)  $\forall a, b, c \in S, (a, b) \in R, (b, c) \in R \Rightarrow (a, c) \in R$ .

Ejemplos: "menor estricto que" ( $<$ ) y la "inclusión propia de conjuntos" ( $\subset$ ).

Un orden parcial se puede representar mediante un DAG.

➤ Prueba de aciclicidad

Dado un grafo  $G = (V, E)$  saber si tiene ciclos.

1. Realizar un recorrido profundidad del grafo G.

2. El grafo G es cíclico si y sólo si el recorrido produce algún arco de retroceso.

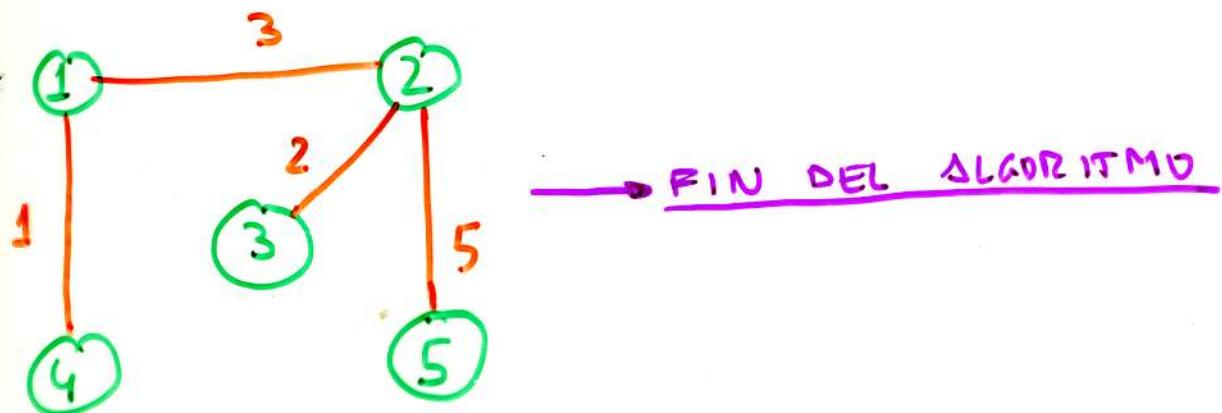
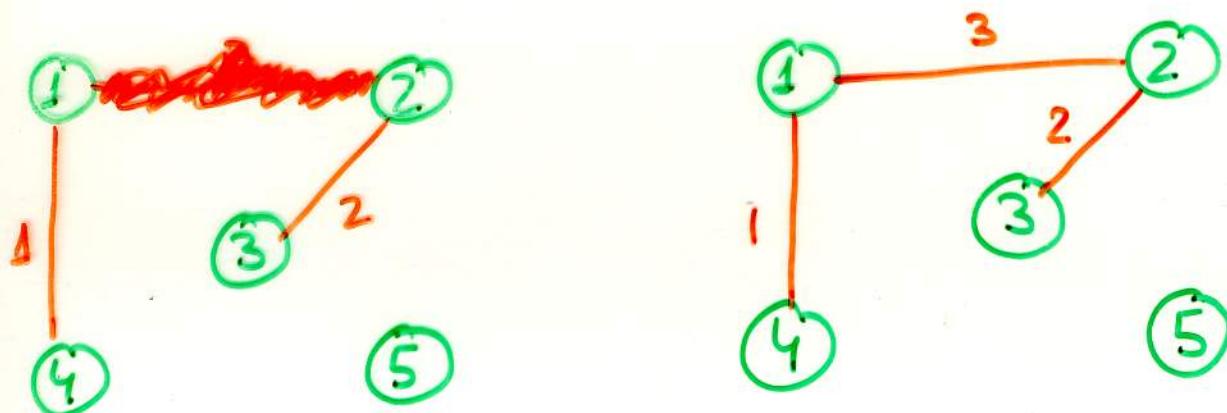
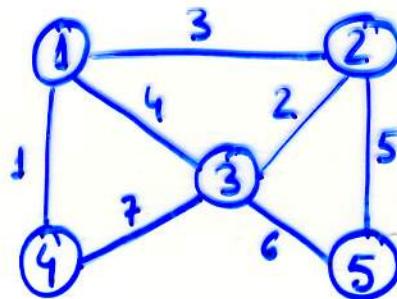
## ARBOLRES GENERADORES. ALGORITMO DE KRUSKAL

Arbol Generador } y grafo conexo aciclico uniendo con mínimo  
de mínimo peso } costo todos los vértices del grafo.

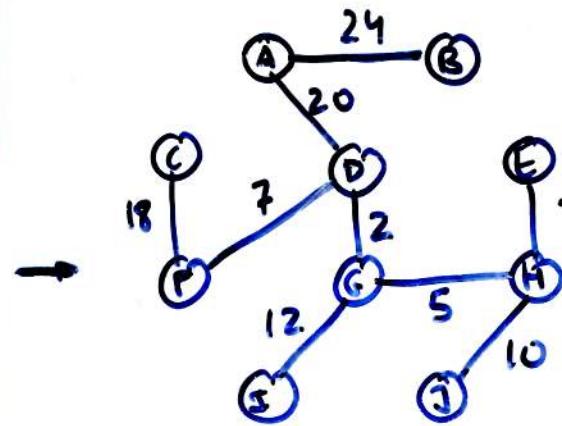
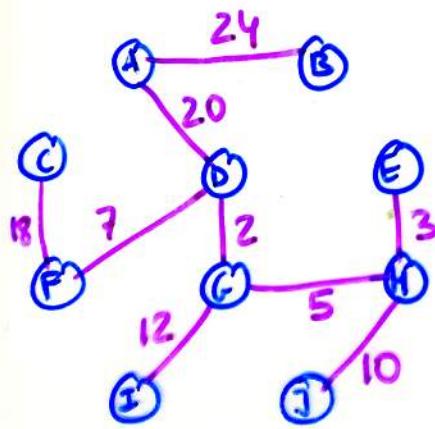
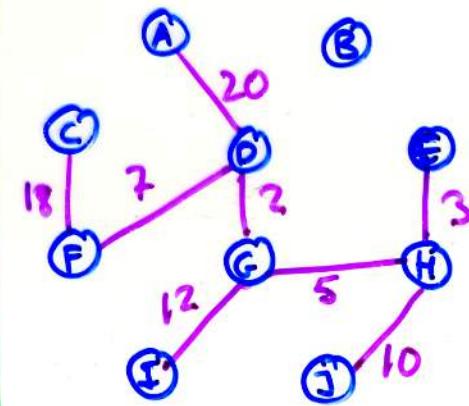
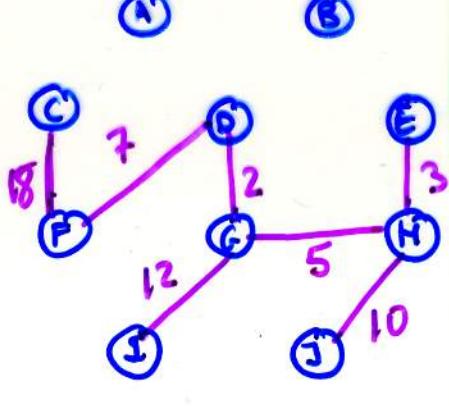
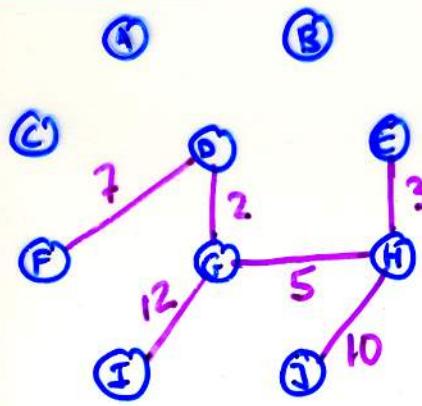
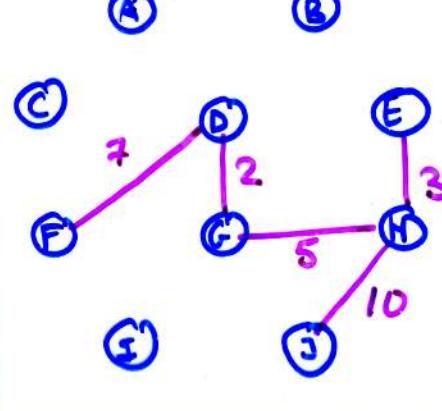
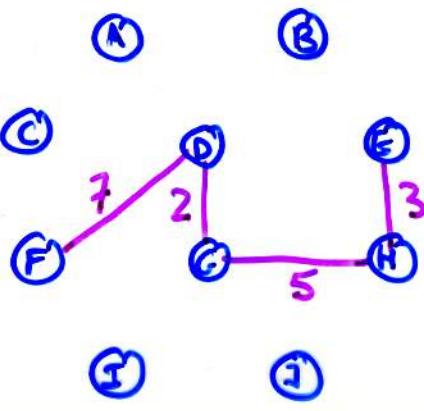
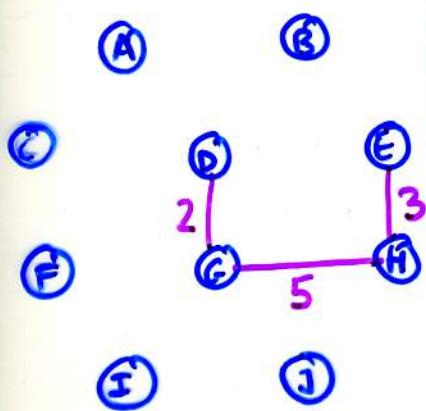
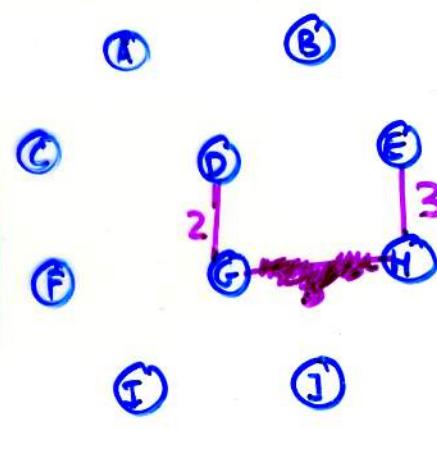
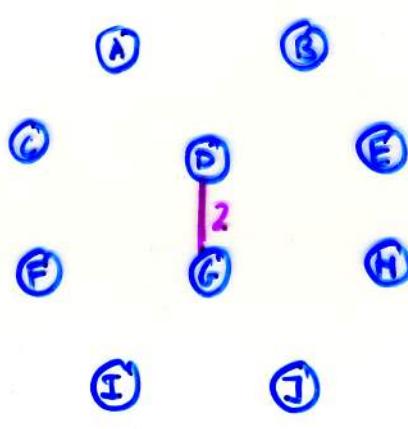
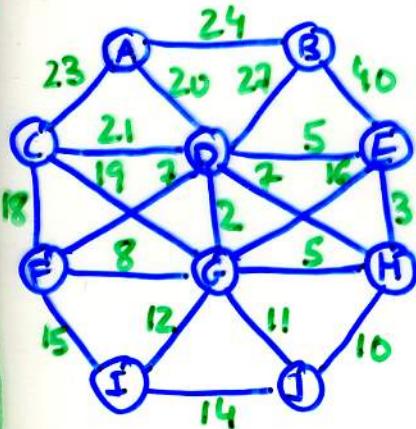
### IDEA DE KRUSKAL PARA CONSTRUIRLOS

- $G = \{V, A\}$   $V = \{1, 2, \dots, n\}$
- SE EMPIEZA POR UN GRAFO  $G' = \{V, \emptyset\}$  SIN ARCOS. ASI CADA VERTICE ES UNA COMPONENTE CONEXA.  $T = \emptyset$
- SE ORDENAN LOS ARCOS DE A EN ORDEN CRECIENTE DE COSTOS
- SI EL ARCO QUE TENEMOS QUE PROCESAR CONECTA DOS VERTICES SITUADOS EN DOS COMPONENTES CONEXAS DISTINTAS SE AGREGA EL ARCO A T Y SE FUSIONAN LAS COMPONENTES
- SE DESCARTA EL ARCO SI CONECTA DOS VERTICES CONTENIDOS EN LA MISMA COMPONENTE CONEXA (PODRIA FORMAR UN CICLO)
- EL ALGORITMO TERMINA CUANDO SOLO QUEDA UNA COMPONENTE CONEXA QUE INCLUYE TODOS LOS VERTICES DEL GRAFO.

### EJEMPLO 1



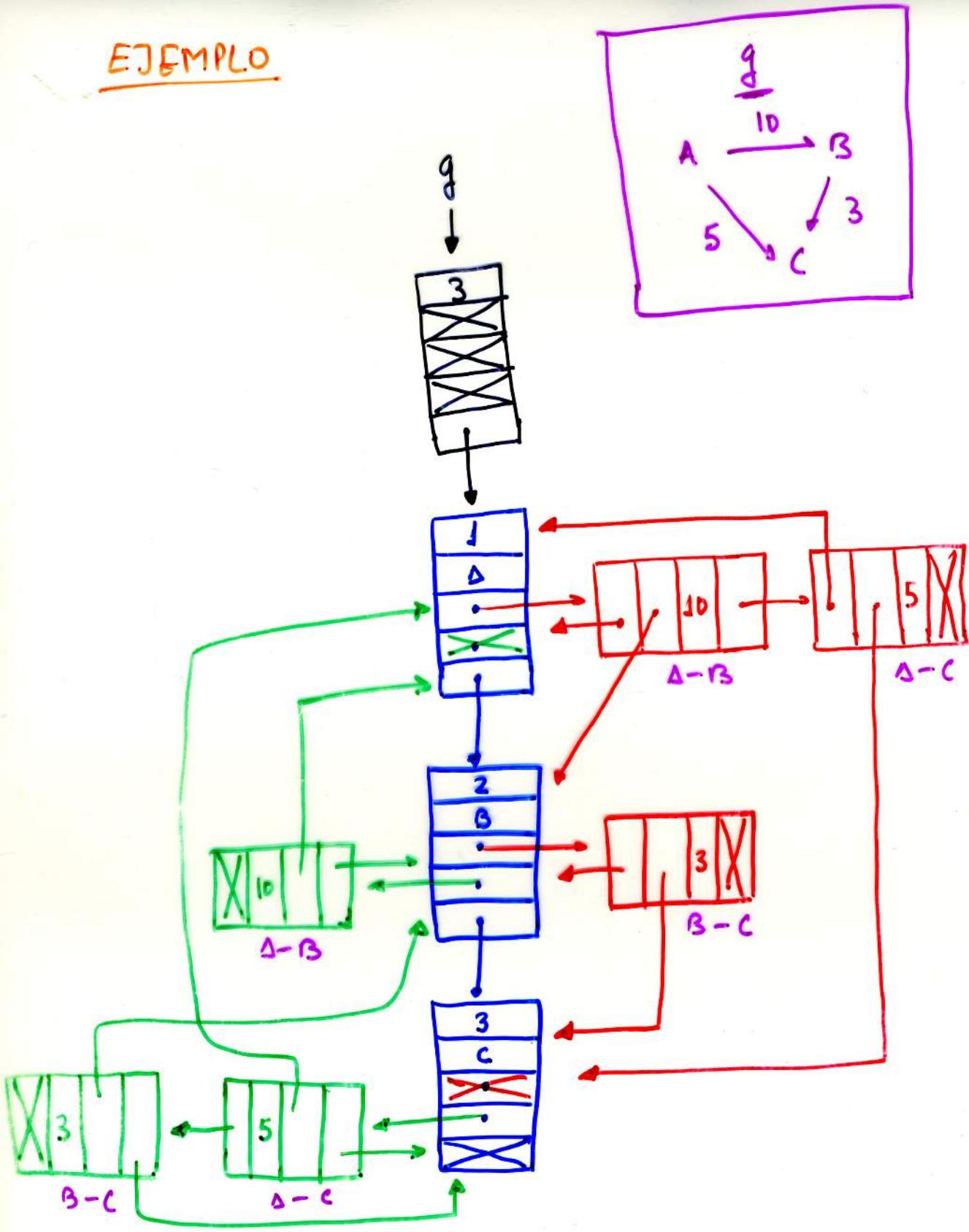
EJEMPLO 2

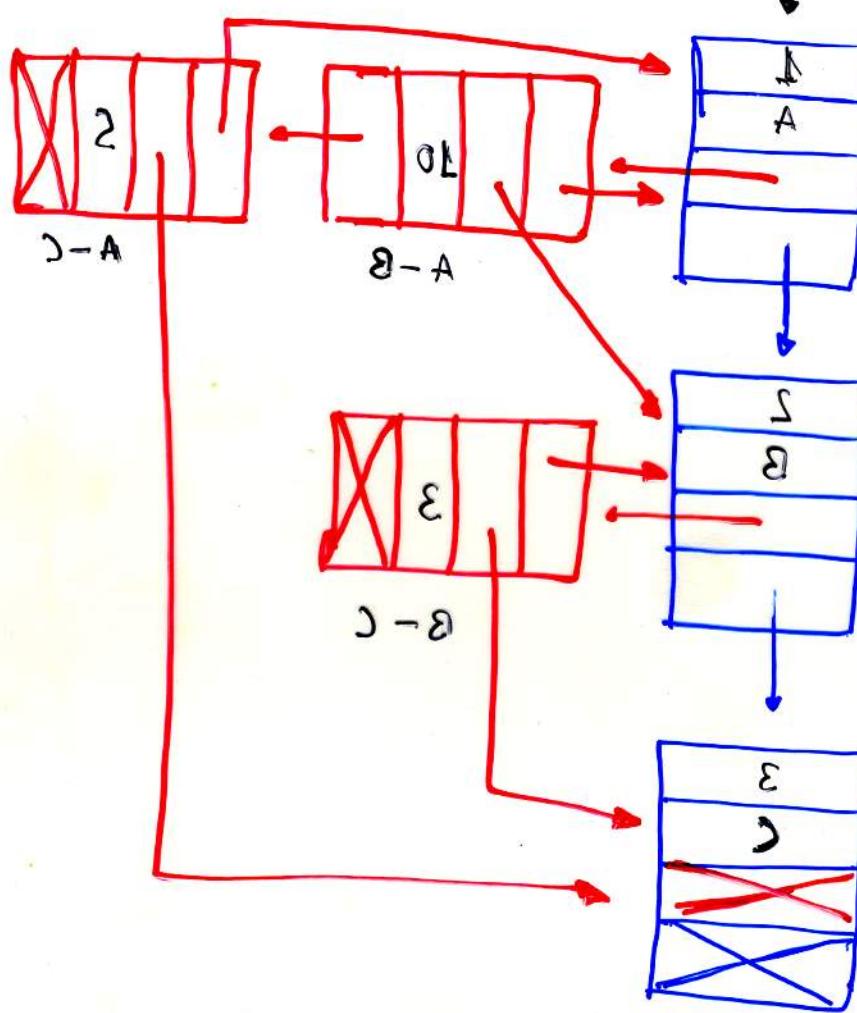
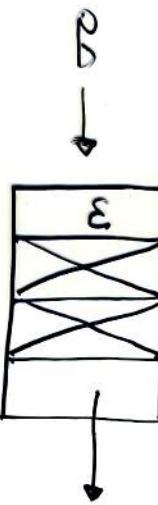
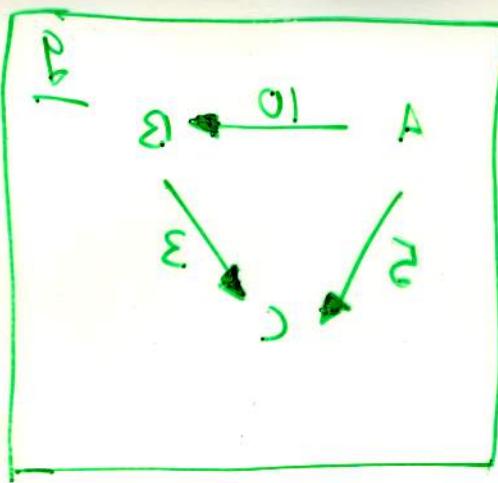


ARBOL GENERADOR DE  
MINIMO COSTO



## EJEMPLO.





template <class Tn, class Ta>

class arco {

private:

nodo<Tn, Ta> \* origen;

nodo<Tn, Ta> \* destino;

Ta valor;

public:

arco() {};

arco (const arco <Tn, Ta> & a)

{ origen = a.origen;  
destino = a.destino;  
valor = a.valor;  
};

arco (nodo <Tn, Ta> \* o, nodo <Tn, Ta> \* d, Ta v)

{ origen = o; destino = d; valor = v;  
};

Ta et~~o~~ Arco() { return valor; };

nodo <Tn, Ta> \* nodoDestino()

{ return destino;  
};

nodo <Tn, Ta> \* nodoOrigen()

{ return origen;  
};

};

template < class  $Tn$ , class  $Ta$  >

class nodo {

private:

int ind; /\* código interno de numeración \*/

$Tn$  etiq;

list < arco <  $Tn$ ,  $Ta$  > > ady;

list < arco <  $Tn$ ,  $Ta$  > > inc;

public:

nodo() {};

nodo( $Tn$  et, int i)

{ etiq = et;

ind = i;

};

nodo (const nodo <  $Tn$ ,  $Ta$  > & n)

{ ind = n.ind;

etiq = n.etiq;

ady = n.ady;

inc = n.inc;

};

$Tn$  etiqueta()

{ return etiq;

};

int indice()

{ return ind;

};

```
void nuevoAdy (nodo <Tn,Ta> * destino, Ta valor)
{
    arco <Tn,Ta> a (this, destino, valor);
    ady.push_back(a);
}

void nuevoInc (nodo <Tn,Ta> * origen, Ta valor)
{
    arco <Tn,Ta> a (origen, this, valor);
    inc.push_back(a);
}

typedef list<arco<Tn,Ta>>::iterator ady_iterator;
typedef list<arco<Tn,Ta>>::iterator inc_iterator;

ady_iterator abegin()
{
    return ady.begin();
}

ady_iterator aend()
{
    return ady.end();
}

inc_iterator ibegin() { return inc.begin(); }

inc_iterator iend() { return inc.end(); }

ady_iterator borrarAdy (ady_iterator i)
{
    return ady.erase(i);
}

inc_iterator borrarInc (inc_iterator i)
{
    return inc.erase(i);
}
```

template <class Tn, class Ta>

class grafo {

private:

list <nodo <Tn, Ta>> \*g;

int ncomp;

public:

typedef list <nodo <Tn, Ta>>::iterator nodeiterator;

typedef profundidadIterador <Tn, Ta> profIterator;

grafo();

grafo ( const grafo <Tn, Ta> & gr )

{ ncomp = gr.ncomp;

g = gr.g;

};

grafo <Tn, Ta> copiarGrafo();

void insertarNodo (Tn etg);

void insertarArco ( nodo <Tn, Ta> \* origen,  
nodo <Tn, Ta> \* destino, Tq valor);

arco <Tn, Ta> \* buscarArco ( nodo <Tn, Ta> \* o,  
nodo <Tn, Ta> \* d);

nodo <Tn, Ta> \* buscarNodo (Tn x);

nodo <Tn, Ta> \* buscarNodo (int i);

```
nodeIterator nbegin(),  
nodeIterator nend(),  
profIterator pbegin(),  
profIterator pend(),  
int numeroNodos()  
{  
    return nCount;  
}  
bool vacio()  
{  
    return nCount == 0;  
};
```

template <class Tn, class Ta>

grafo <Tn, Ta>:: grafo()

{

g = new list <nodo<Tn, Ta>>();

ncomp = 0;

}

template <class Tn, class Ta>

void grafo <Tn, Ta>:: insertarNodo (Tn etq)

{

nodo <Tn, Ta> aux (etq, ncomp);

ncomp++;

g->push\_back (aux);

}

template <class Tn, class Ta>

void grafo <Tn, Ta>:: insertarArco (nodo <Tn, Ta> \*origen,

nodo <Tn, Ta> \*destino, Ta valor)

{

origen->nuevoAdy (destino, valor);

destino->nuevoInc (origen, valor);

}

template <class Tn, class Ta>

grafo <Tn, Ta>::nodeIterator grafo <Tn, Ta>::nbegin()

{ return g->begin(); }

}

template <class Tn, class Ta>

grafo <Tn, Ta>:: nodeIterator grafo <Tn, Ta>::nend()

{ return g->end(); }

}

template <class Tn, class Ta>

nodo<Tn, Ta> \* grafo <Tn, Ta>::buscarNodo(Tn x)

{

    nodeIterator itr;

    for (itr = nbegin(); itr != nend(); )

        if (itr->etiqueta() == x)

            return &(\*itr);

        else itr++;

    return NULL;

}

```
template <class Tn, class Ta>
nodo<Tn,Ta> * grafo<Tn,Ta>:: buscarNodo (int i)
{
    int lc;
    nodeIterator itr;
    for (lc=0, itr=nbegin(); lc < i; lc++, itr++);
    return & (*itr);
}
```

```
template <class Tn, class Ta>
arco<Tn,Ta> * grafo<Tn,Ta>:: buscarArco (
    nodo<Tn,Ta> * o, nodo<Tn,Ta> * d)
{
    nodo<Tn,Ta>:: ady_iterator itr;
    bool enc=false;
    for (itr=o->abegin(); itr!=o->aend()
        &&!enc; )
        if (itr->nodoDestino() == d)
            enc=true;
        else itr++;
    if (enc) return & (*itr);
    else return NULL
}
```

```

template <class Tn, class Ta>
grafo <Tn, Ta>::nodeIterator grafo <Tn, Ta>:::
    borrarNodo (grafo <Tn, Ta>:: nodeIterator n )
{
    nodo <Tn, Ta>:: adj_iterator a, ad;
    nodo <Tn, Ta>:: iuc_iterator r, rd;
    nodo <Tn, Ta> * nady;
    nodo <Tn, Ta> * niuc, *nact;
    grafo <Tn, Ta>:: nodIterator s, old;
    for (a = n->begin(); a != n->end(); )
    {
        nact = a->nodoOrigen();
        nady = a->nodoDestino();
        for (rd = nady->begin(), rd->nodoOrigen() != nact; ++rd)
            rd = rd->nodoDestino() ->borrarNodo((rd));
        a = a->nodoOrigen() ->borrarAdy(a);
    }
    for (i = n->ibegin(); i != n->ienad(); )
    {
        nact = i->nodoDestino();
        niuc = i->nodoOrigen();
        for (ad = niuc->abegin(); ad->nodoDestino() != nact; ++ad)
            ad = ad->nodoOrigen() ->borrarAdy(ad);
        i = i->nodoDestino() ->borrarNodo((i));
    }
}

```

```
ncomp = ;  
old = s = g->erase(n);  
for (; s != nend(); ++s)  
    s->ind --;  
return old;  
}
```

```
template <class Tn, class Ta>  
void grafo<Tn, Ta>::borrarNodo (nodo<Tn, Ta> *n)  
{  
    nodeIterator itr;  
    for (itr = nbegin(); &(itr) != n; ++itr);  
    if (itr != nend())  
        borrarNodo (itr);  
}
```

template < class Tn, class Ta >

grafo < Tn, Ta > grafo < Tn, Ta >:: copiar Grafo()

{

grafo < Tn, Ta > aux;

grafo < Tn, Ta >:: nodeIterator n;

nodo < Tn, Ta >:: ady\_iterator a;

for (n = n.begin(); n != n.end(); ++n)

    aux.insertarNodo (n->etiqueta());

for (n = n.begin(); n != n.end(); ++n)

    for (a = n->abegin(); a != n->aend(); ++a)

        aux.insertarArzo (aux.buscarNodo (

            a->nodoOrigen()->etiqueta()),

            aux.buscarNodo (

            a->nodoDestino()->etiqueta()),

            a->etiquArzo());

return aux;

}