Conjuntos Disjuntos, Árboles y Gráficas

Estructuras de datos para mantener conjuntos disjuntos

- Estructura para mantener una colección $S=\{S_1,S_2,\ldots,S_k\}$ de conjuntos dinámicos disjuntos.
- Cada conjunto se identifica por un representante que es algún miembro del conjunto.
- Cada elemento del conjunto se representa con un objeto x que debe soportar las operaciones siguientes:
  - MAKE-SET(x) : nuevo conjunto con único miembro x (representante). x no puede estar en otro conjunto.
  - UNION(x,y) : une dos conjuntos dinámicos que contienen a x (S_x) y a y (S_y) como miembros en un nuevo conjunto. S_x y S_y son destruidos al terminar la operación.
  - FIND-SET(x) : regresa un apuntador al representante del único conjunto que contiene a x.

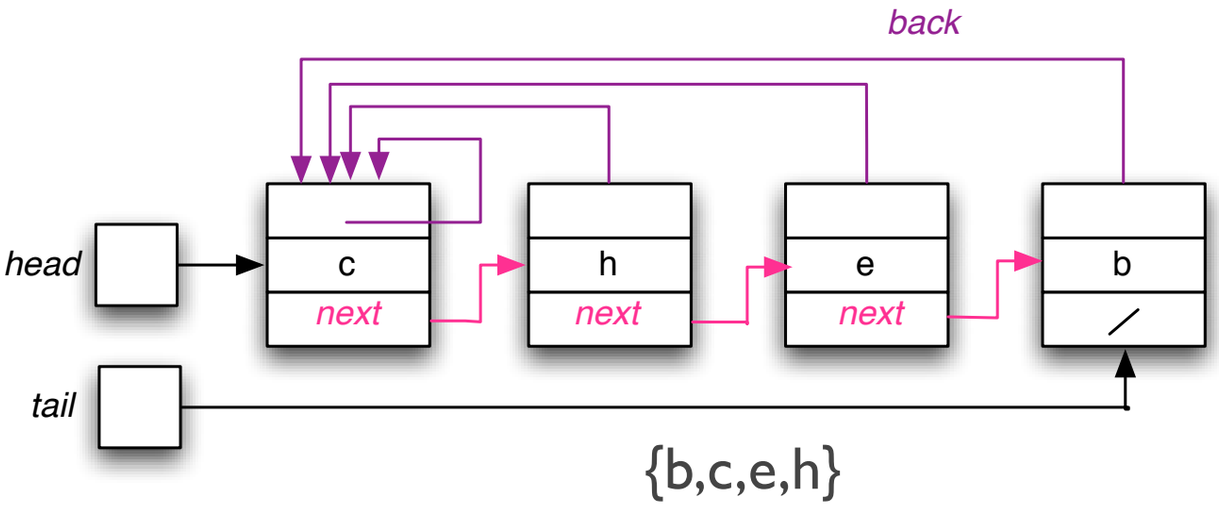
Estructuras de datos para mantener conjuntos disjuntos

- Las representaciones de conjuntos disjuntos se evalúan usando dos parámetros:
  - n : número total de operaciones MAKE-SET.
  - m: número total de operaciones MAKE-SET, UNION y FIND-SET.
- Nótese que cada operación UNION reduce el número de conjuntos disjuntos en uno.
- ¿Cuántos conjuntos disjuntos quedan después de n-1 operaciones UNION?
  - uno.
- ¿Cuál es el número máximo posible de operaciones UNION?
  - n-1.

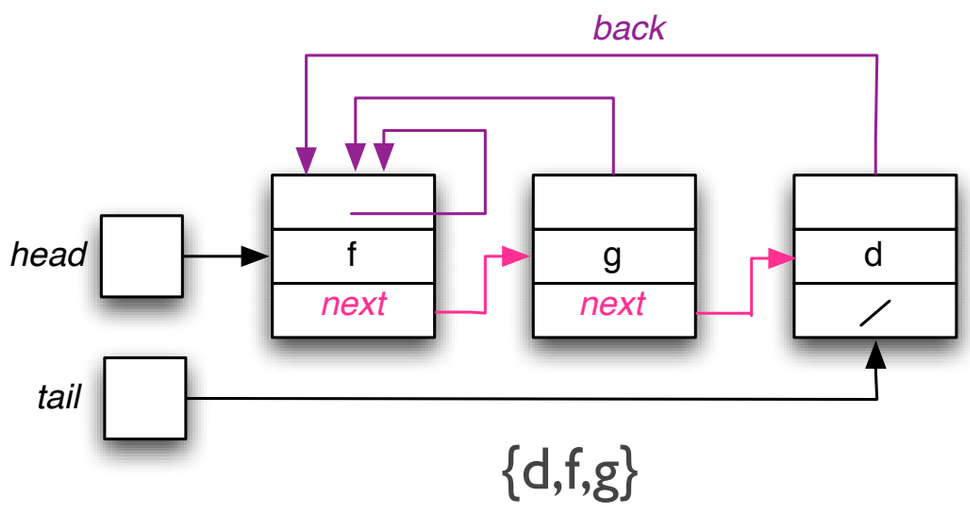
Representación de conjuntos disjuntos con listas ligadas

- El primer elemento de la lista sirve como representante del conjunto.
- Cada objeto en la lista ligada contiene:
  - un miembro del conjunto,
  - un apuntador next al objeto que contiene al siguiente miembro,
  - un apuntador back para regresar al representante.
- Cada lista ligada mantiene:
  - un apuntador head al representante,
  - un apuntador tail al final de la lista.
- Dentro de cada lista los objetos pueden aparecer en cualquier orden.

Representación de conjuntos disjuntos con listas ligadas

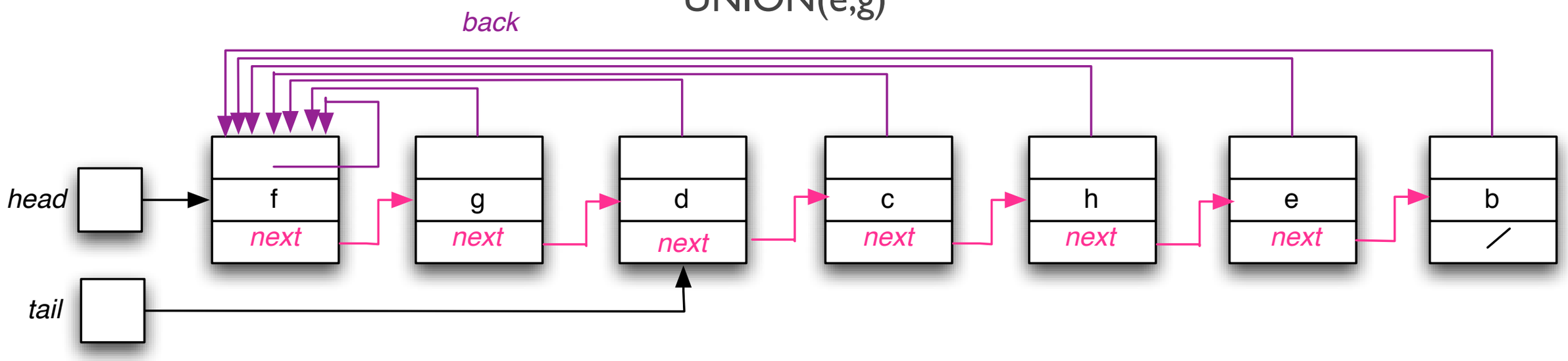


{b,c,e,h}



{d,f,g}

UNION(e,g)



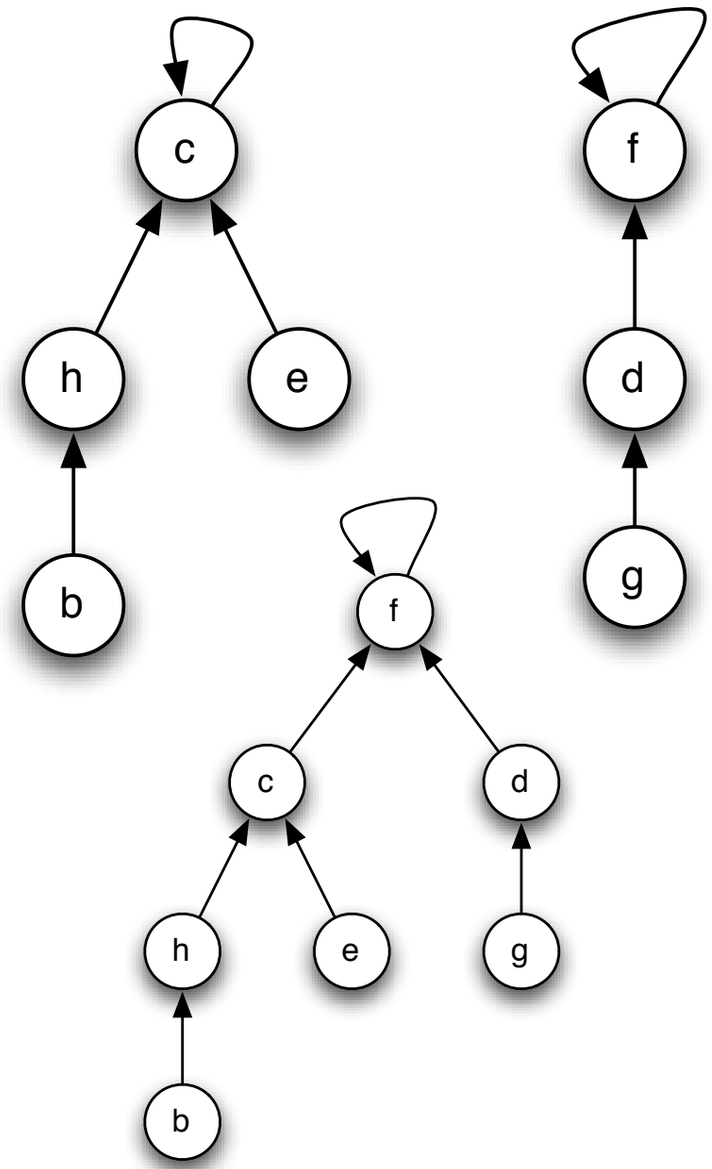
{b,c,d,e,f,g,h}

Representación de conjuntos disjuntos con listas ligadas

- ¿Cuál es el tiempo de ejecución para MAKE-SET(x)?
  - O(1): crear la lista nueva con x como único elemento.
- ¿Y de FIND-SET?
  - O(1): regresar el apuntador back de x a su representante.
- Implementación simple de UNION(x,y), ¿cuánto tiempo toma?
  - O(x): actualizar el apuntador del objeto original al representante de la lista x (que puede ser la más larga).
- Implementación con heurística de peso de UNION.
- Usando la representación de listas ligadas y una heurística de peso para UNION, una secuencia de m operaciones MAKE-SET, UNION, FIND-SET, de las cuáles n son MAKE-SET, toma O(m+n log n) tiempo de ejecución.

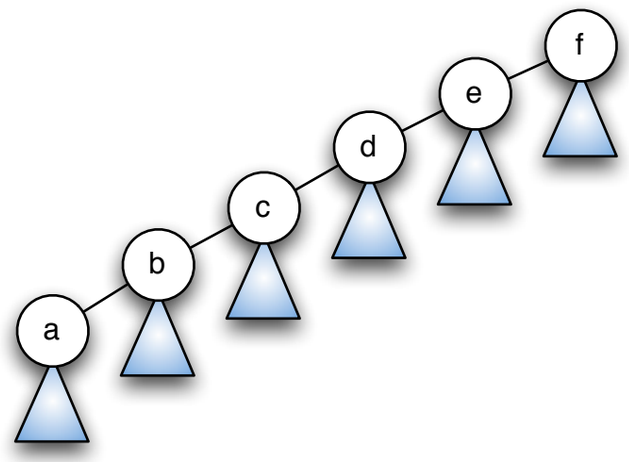
Representación de conjuntos disjuntos con árboles

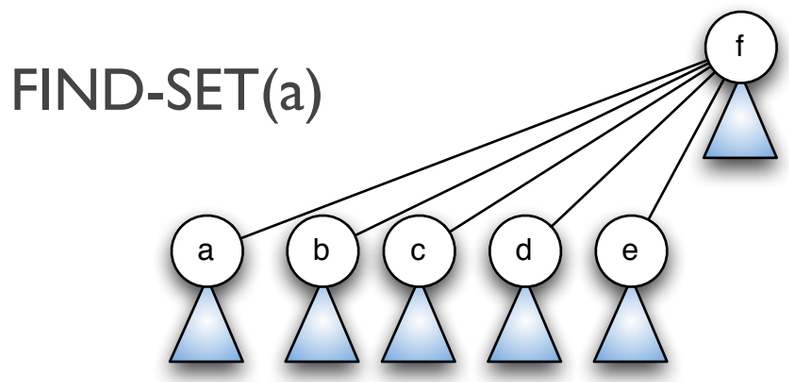
- Árboles enraizados.
- Cada nodo contiene un miembro y cada árbol representa un conjunto.
- Cada miembro apunta a su padre.
- Por si mismos no representan grandes ventajas: se usan dos heurísticas.
- Unión por rango.
- Camino por compresión.
- MAKE-SET crea un árbol nuevo con un sólo nodo.
- FIND-SET sigue los apuntadores a predecesores hasta encontrar la raíz.
- UNION hace la raíz de un árbol apuntar a la raíz de otro.



Heurísticas: unión por rango

- Unión por rango (union by rank)
  - similar a la unión por peso con listas ligadas.
  - hacer al árbol con menos nodos apuntar a la raíz del árbol con más nodos.
  - para cada nodo se mantiene su rango (número de aristas en el camino más largo entre el nodo y sus nodos dependientes).
- Compresión por camino (path compression)
  - usado durante FIND-SET para hacer que cada nodo visitado apunte directamente a la raíz.





Representación de conjuntos disjuntos con árboles

```
MAKE-SET(x)

1  p[x] ← x
2  rank[x] ← 0
```

```
FIND-SET(x)

1  if x ≠ p[x]
2       then p[x] ← FIND-SET(p[x])
3  return p[x]
```

```
UNION(x, y)

1  LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)

1  if rank[x] > rank[y]
2       then p[y] ← x
3       else p[x] ← y
4            if rank[x] = rank[y]
5                 then rank[y] ← rank[y] + 1
```

Representación de conjuntos disjuntos con árboles

- La operación UNION con una heurística de unión por rango tiene un tiempo de ejecución de $O(m \log n)$.
- La operación FIND-SET con una heurística de compresión de caminos tiene un tiempo de ejecución de $\Theta(n+f(1+\log n))$.
  - para n operaciones MAKE-SET, n-1 operaciones UNION y f operaciones FIND-SET.
- Cuando ámbas heurísticas son usadas, el tiempo de ejecución en el peor caso es $O(m\alpha(n))$, donde $\alpha(n)$ es una función que crece muy lento.
- Se ha probado que para cualquier aplicación de conjuntos disjuntos $\alpha(n) \leq 4$ (ver Cormen et. al, sección 21.4)
- Se puede ver el tiempo de ejecución con ambas heurísticas como líneal en m para todas las situaciones prácticas.

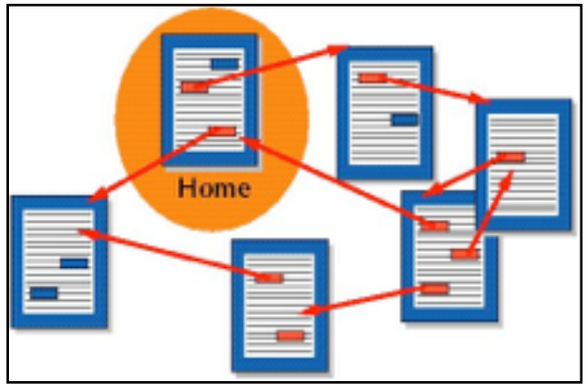
Gráficas

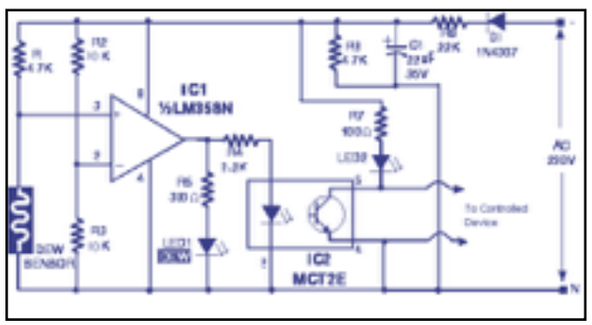
- Muchas aplicaciones computacionales involucran no solo un conjunto de elementos sino que también conexiones entre pares de elementos.

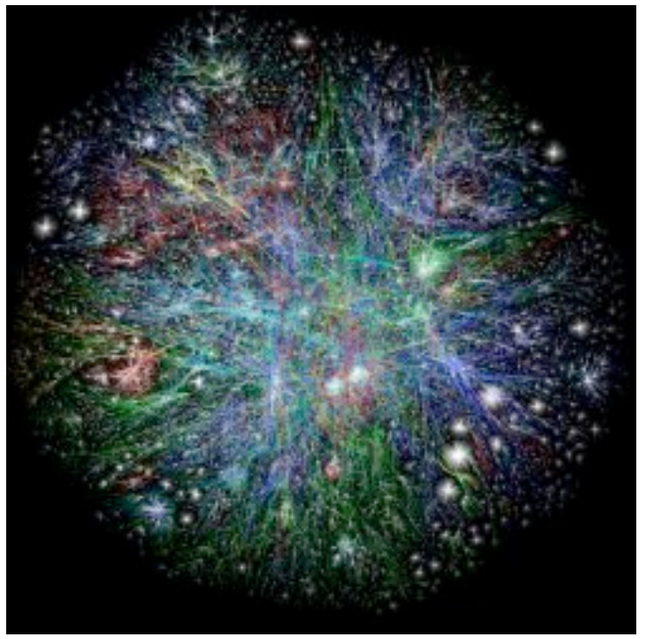
- Las relaciones que resultan de estas conexiones nos llevan a preguntas como:
  - ¿Hay forma de llegar de un elemento a otro siguiendo las conexiones?
  - ¿Cuáles y cuántos elementos se pueden alcanzar a partir de un elemento dado?
  - ¿Cuál es la mejor forma de llegar de un elemento a otro?

- Para modelar situaciones como esta se utilizan las gráficas.

Aplicaciones







- mapas
- hypertexto
- modelación mecánica
- planificación de tareas
- transacciones
- emparejamiento
- redes informáticas, internet
- estructura de programas...

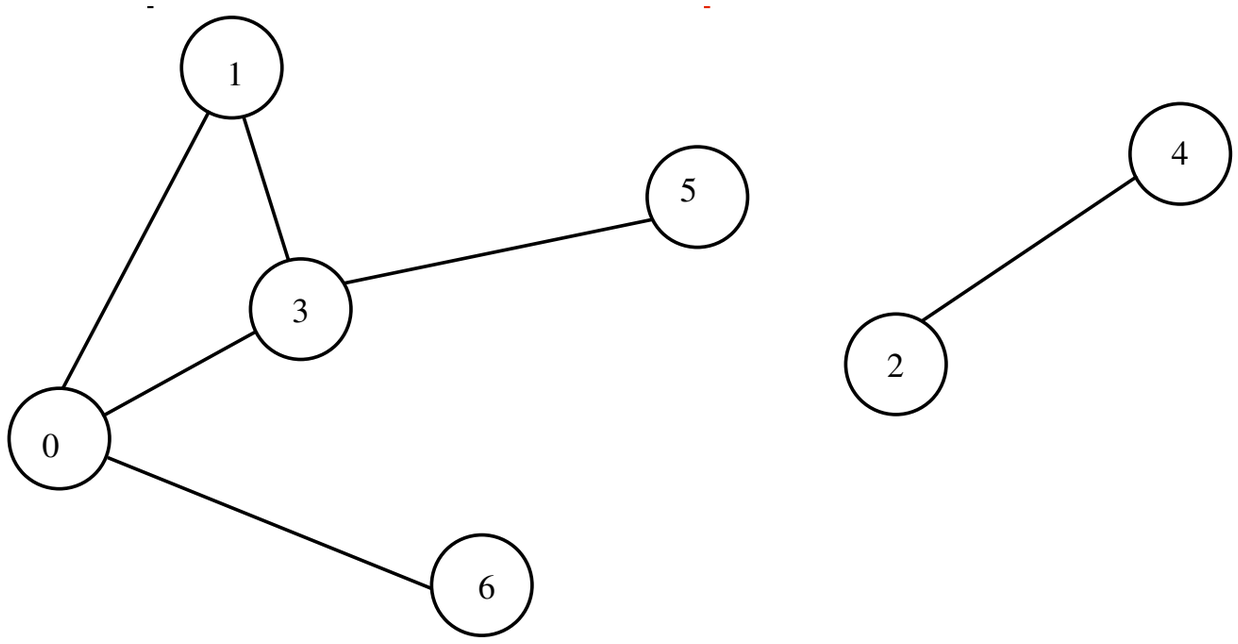
Gráficas

- El estudio del desempeño en algoritmos con gráficas es particularmente retador porque:
  - El costo de un algoritmo depende no solo de las propiedades del conjunto de elementos sino también de las propiedades del conjunto de conexiones.
  - Es difícil hacer modelos precisos de problemas por gráficas.

Gráficas: definiciones y propiedades

- Una gráfica *G* es un par de conjuntos $G = (V, E)$.
- *V* es un conjunto de *n* objetos arbitrarios $u, v, w \in V$ llamados vértices o nodos.
- *E* es un conjunto de aristas, ejes o arcos *m*.
- Las aristas son típicamente pares de vértices, $\{u, v\} \in E$ definiendo una relación entre el conjunto *V* con si mismo: $E \subseteq V \times V$.
- En una gráfica no dirigida los ejes son pares no ordenados o solo conjuntos que contienen dos vértices.
- En una gráfica dirigida o digráfica los ejes son pares ordenados de vértices (están dirigidos).

Gráficas: definiciones y propiedades



- Vértices: 0,1,2,3,4,5,6
- Aristas:

1-3 1-0 0-3
3-5 0-6 2-4

Gráficas: definiciones y propiedades

- Un bucle es una arista reflexiva, donde coinciden el vértice de origen y el vértice de destino: $\{v, v\}$ o $< v, v >$.
- En el caso general, puede ser que haya más de una arista por un par de vértices. En tal caso, la gráfica se llama multigráfica.
- Nos ocuparemos de gráficas simples, donde no hay ejes de un vértice hacia si mismo y hay a lo más un eje de un vértice a cualquier otro.
- Si se asignan pesos o costos a las aristas, la gráfica es ponderada.
- Si se asigna identidad a los vértices o a las aristas, la gráfica es etiquetada.

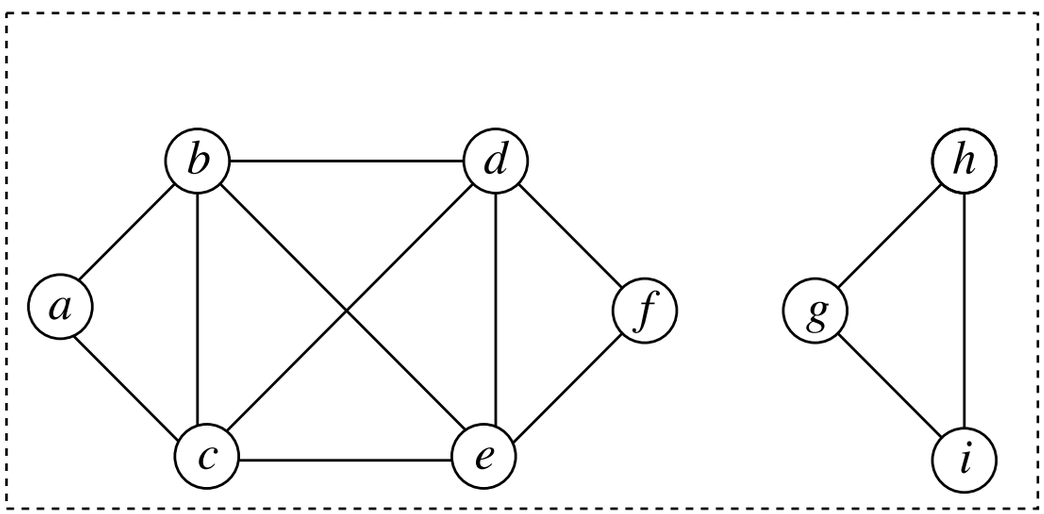
Gráficas: definiciones y propiedades

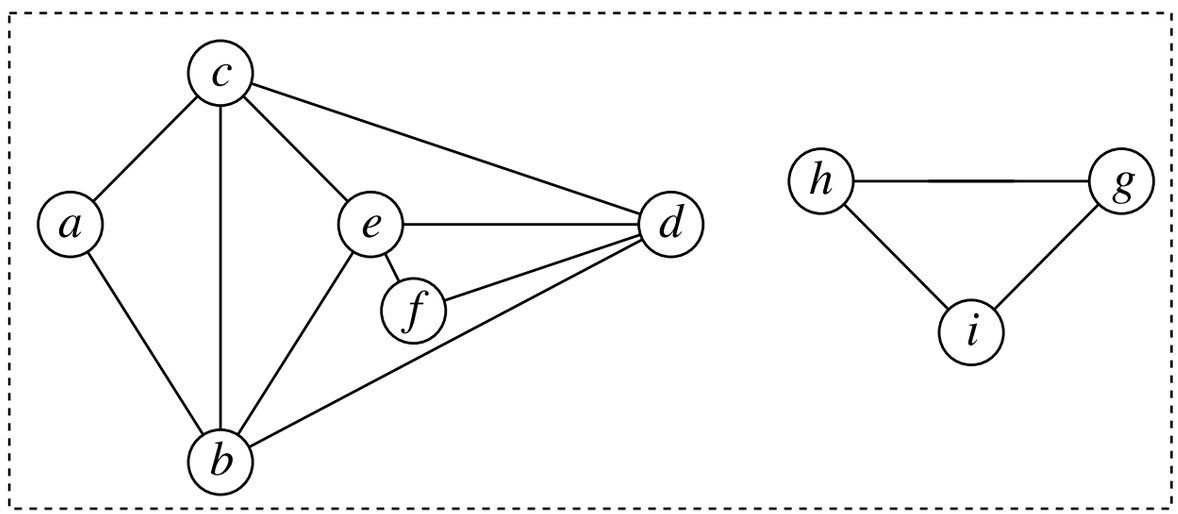
- Usaremos *V* para denotar el número de vértices en una gráfica y *E* para denotar el número de ejes.
- En una gráfica no-dirigida tenemos: $0 \leq E \leq \binom{V}{2}$.
- En una gráfica dirigida tenemos: $0 \leq E \leq V(V-1)$.
- Podemos visualizar las gráficas al mirar una inmersión. La inmersión de una gráfica transforma cada vértice a un punto en el plano y cada eje a una curva o un segmento de recta entre dos vértices.

Gráficas: definiciones y propiedades

- Una gráfica es plana si se puede dibujar en dos dimensiones de tal manera que ninguna arista cruce a otra arista.

- La misma gráfica puede tener varias inmersiones (dibujos) por lo que es importante no confundir la inmersión con la gráfica misma. En particular, una gráfica plana puede tener inmersiones no planas.





Gráficas: definiciones y propiedades

- Dos aristas $\{v_1, v_2\}$ y $\{w_1, w_2\}$ son adyacentes si tienen un vértice común.
- Una arista es incidente a un vértice si ésta lo une al vértice.
- Dos vértices v y w son adyacentes si una arista los une: $\{v, w\} \in E$.
- El grado $\delta(v)$ de un nodo $v \in V$ es el número de aristas de E que inciden en v (el número de vecinos).
- El grado $\Delta(G)$ de una gráfica es el máximo de los grados de sus vértices.

Caminos en gráficas

- Un camino de tamaño n en una gráfica $G=(V,E,\Phi)$ es una secuencia a $n+1$ vértices u_i, para $0 \leq i \leq n$, tales que para todo $1 \leq i \leq n$, existe $e \in E$ tal que $\Phi(e) = (u_{i-1},u_i)$ o $\Phi(e)=(u_i,u_{i-1})$.

- Dicho informalmente, todos los vértices después del primero son adyacentes a su predecesor.

- Camino simple: todos los vértices y las aristas son distintos.

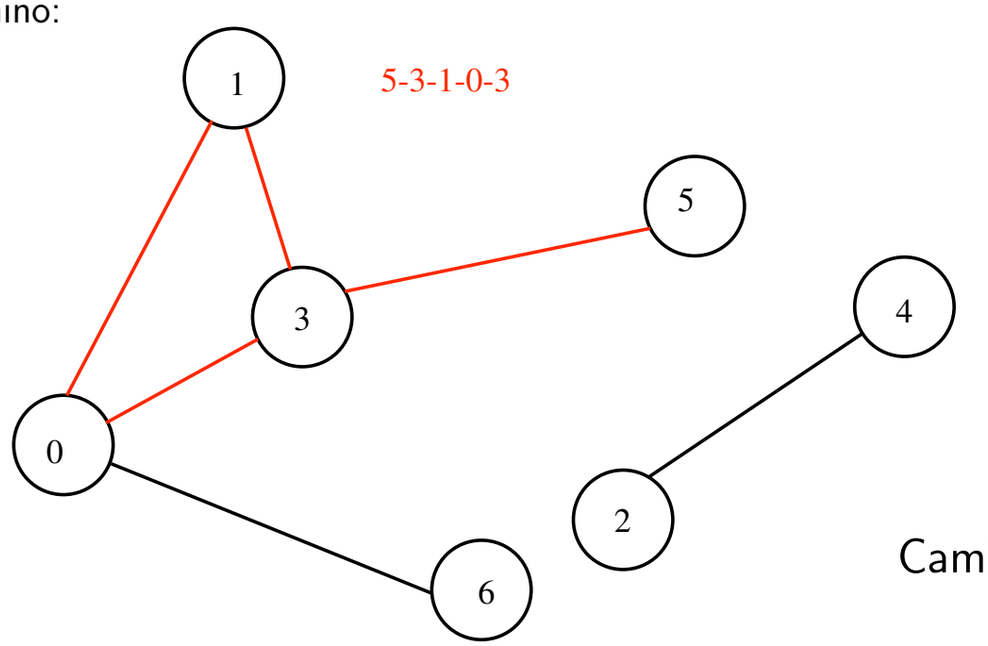
- Cíclo: un camino simple excepto que el primer y el último vértice son iguales.

- Camino cíclico: un camino tal que el primer y el último vértices son iguales.

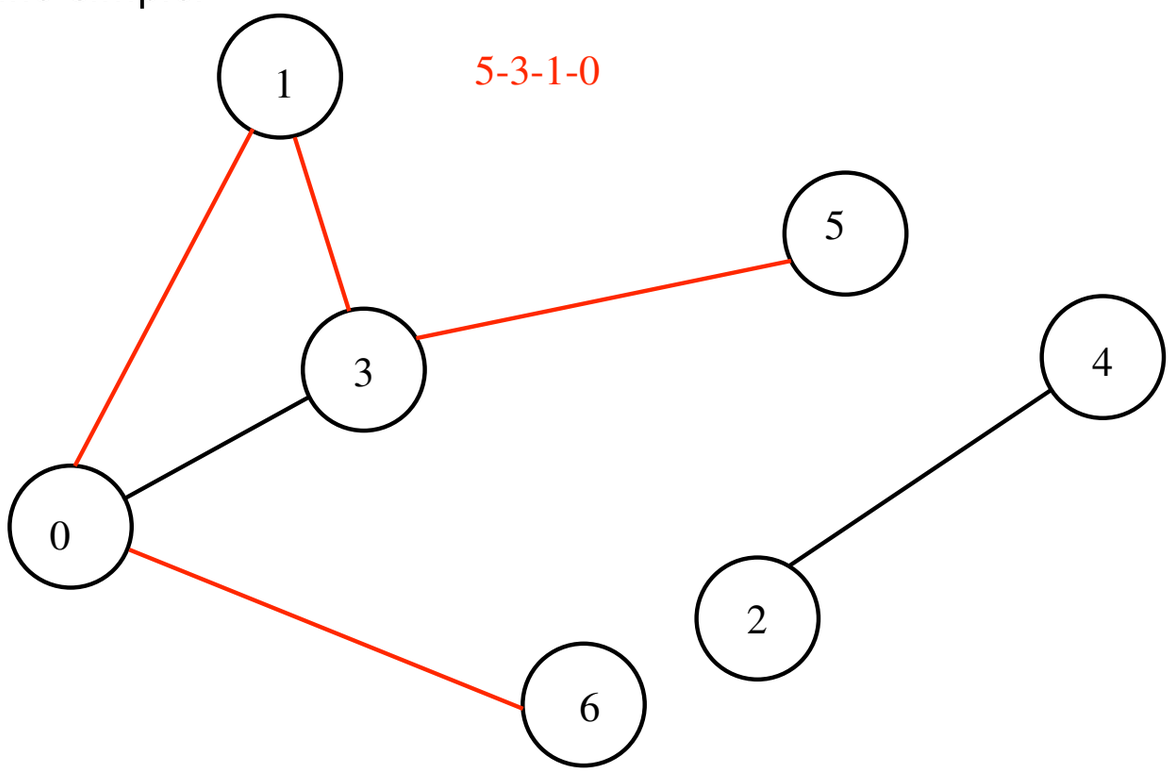
- Tour: camino cíclico que pasa por todos los vértices.

Caminos en gráficas

Camino:

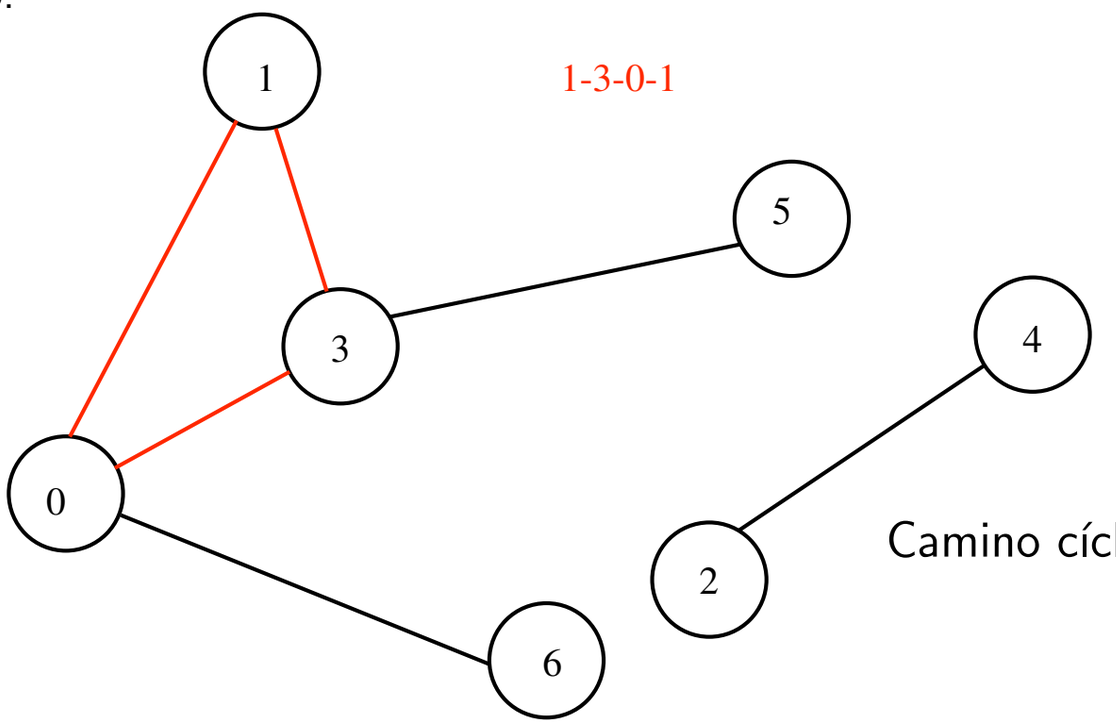


Camino simple:

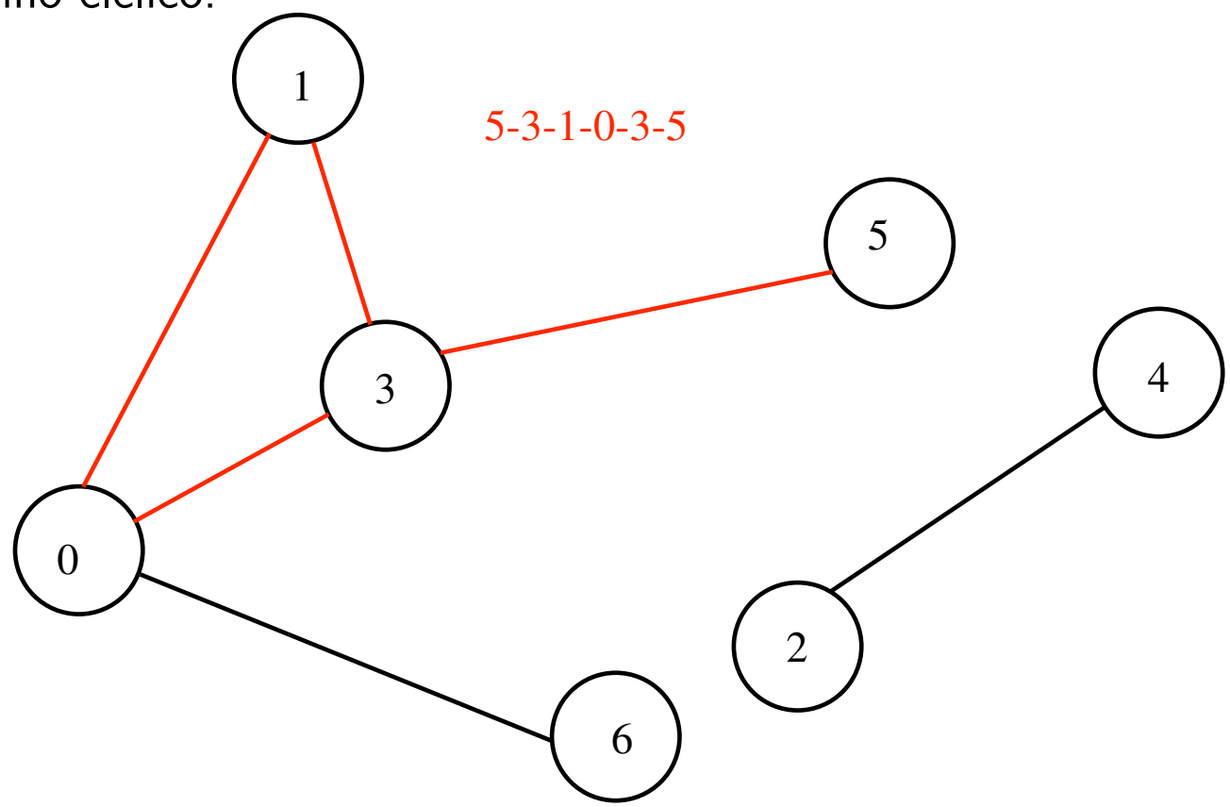


Caminos en gráficas

Ciclo:



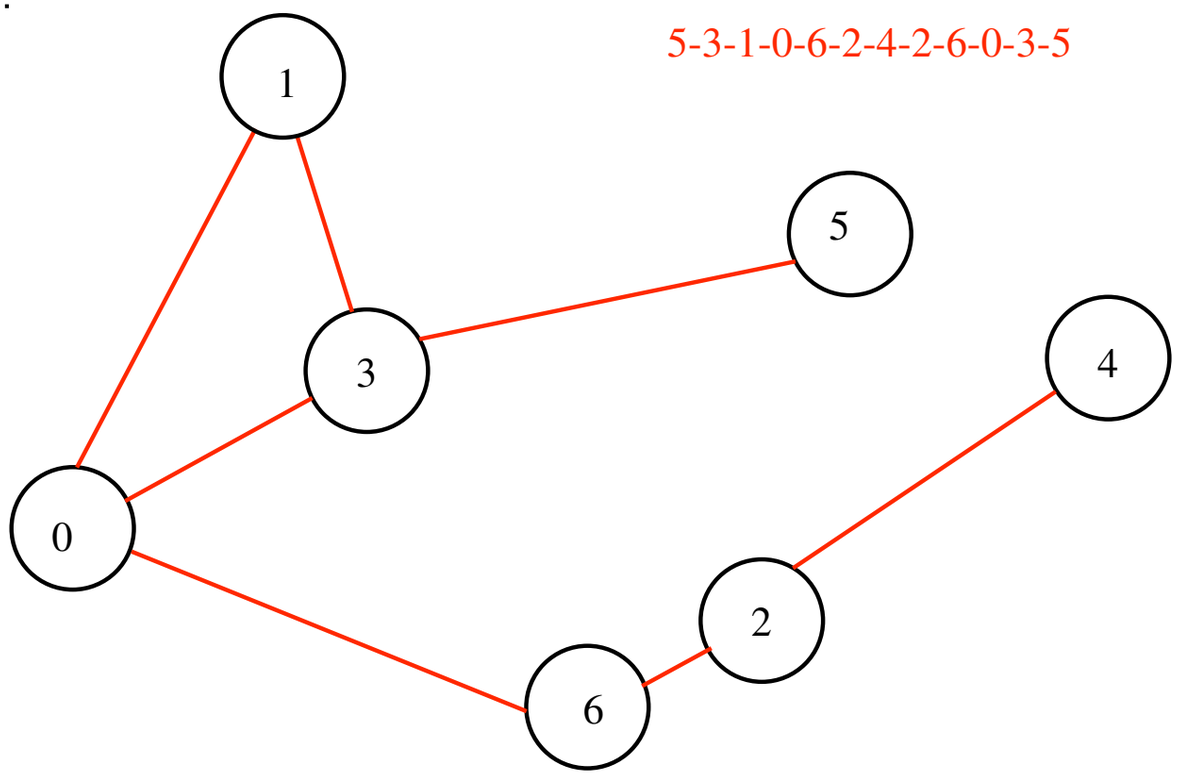
Camino cíclico:



Caminos en gráficas

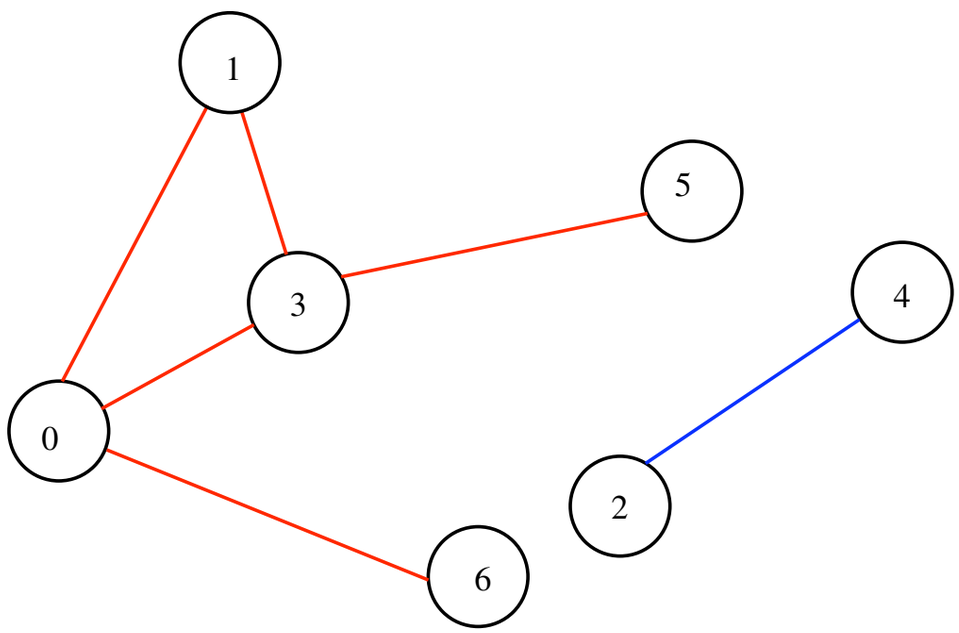
Tour:

5-3-1-0-6-2-4-2-6-0-3-5



Conectividad

- Una gráfica está conectada si existe un camino de cualquier vértice de V a cualquier otro vértice de V.
- Una gráfica desconectada consta de varios componentes conectados, que son subgráficas conectadas.
- Dos vértices están en el mismo componente conectado si y solo si existe un camino entre ellos.



Árboles

- Un ciclo es un camino que empieza y termina en el mismo vértice y tiene al menos un eje.
- Una gráfica es acíclica o un bosque si ningúna subgráfica es un ciclo.
- Los árboles son gráficas especiales que pueden definirse de diferentes formas equivalentes:
  - Un árbol es una gráfica acíclica conectada.
  - Un árbol es un componente conectado de un bosque.
  - Un árbol es una gráfica conectada con a lo más $V-1$ ejes.
  - Un árbol es una gráfica mínimamente conectada; eliminar cualquier eje desconecta al gráfico.

Árboles

- Un árbol es una gráfica acíclica con al menos V-*1* ejes.
- Un árbol es una gráfica acíclica maxima; añadir un eje entre cualquier par de vértices crea un ciclo.

- Un árbol generador (spanning tree) de una gráfica G es una subgráfica que es un árbol y que contiene cada vértice de G.
- Un bosque generador es un una colección de árboles generadores, uno por cada componente conectado de G.

Representación de gráficas

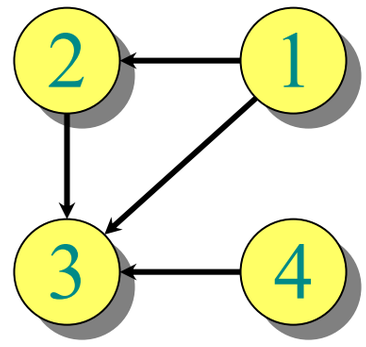
- Aquí se presentan cuatro métodos comunes de representar gráficas.
- Si la gráfica es grande (e.g. infinita), la estructura no se conoce a priori, etc. podemos elegir una representación implícita de la gráfica, donde los ejes y vértices se calculen en línea cuando se necesiten.
- Una representacion explícita es una donde codificamos directamente la estructura de la gráfica en una estructura de datos.

Representación explícita de gráficas

- Hay dos estructuras de datos comunes para la representación explícita de gráficas: las matrices de adyacencia y las listas de adyacencia.
- La matriz de adyacencia de una gráfica $G = (V, E)$ donde $V = \{1, 2, \ldots, n\}$ es la matriz $A[1 \ldots n, 1 \ldots n]$ dada por:

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

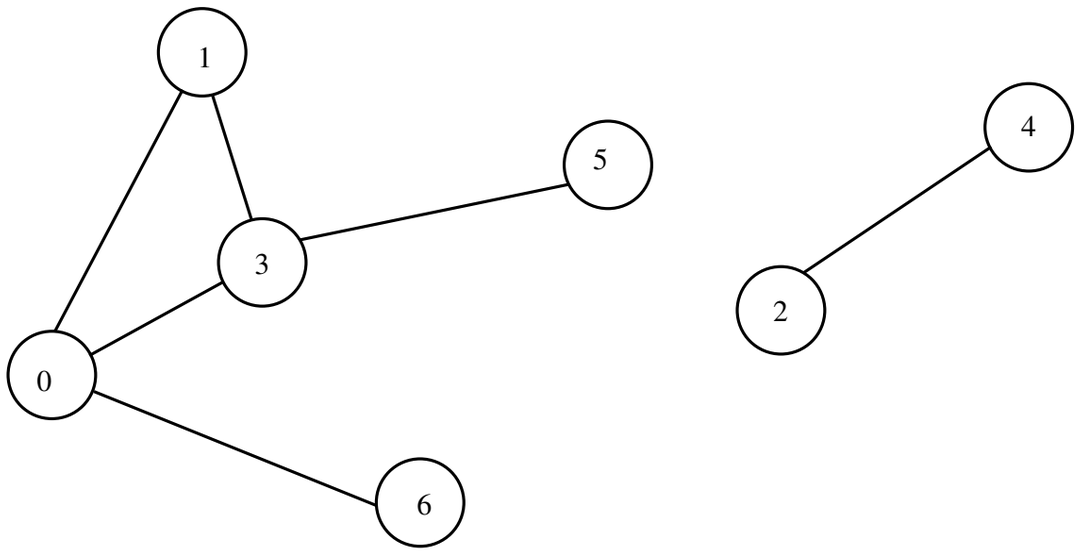


- almacenamiento?

  $\Theta(V^2)$

- representación densa.

Matrices de adyacencia



0	1	0	1	0	0	1
1	0	0	1	0	0	0
0	0	0	0	1	0	0
1	1	0	0	0	1	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
1	0	0	0	0	0	0

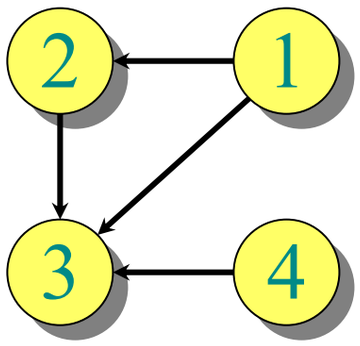
Matrices de adyacencia

$$|V| \times |V|$$

- Un arreglo `Y` tal que `Y[i][j]=1` si `i` y `j` están conectados, cero si no.
  - Matriz de tamaño .
  - En el caso de gráficas no orientados, la matriz Y es simétrica.
  - Representación adecuada para gráficas densas, no adecuada si no (matriz rala)
  - ¿Cuánto tiempo toma verificar si dos vértices están conectados por un eje?
    - ⍬(1) solo verificando la celda indicada de la matriz.
  - ¿Cuánto tiempo toma listar todos los vecinos de un vértice?
    - ⍬(V) para revisar el renglón correspondiente.

Listas de adyacencia

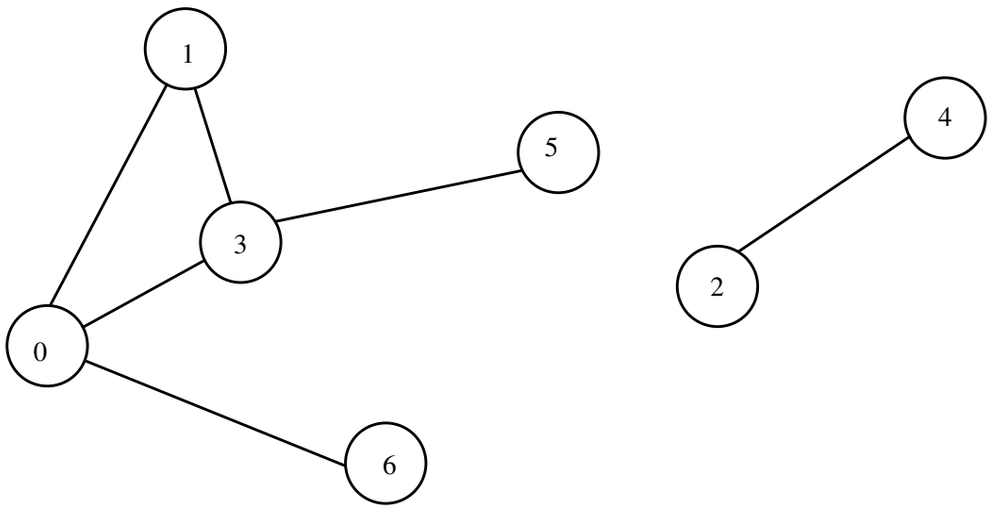
- Una lista de adyacencia de un vértice $v \in V$ es la lista $Adj[v]$ de vértices adyacentes a v.



$$Adj[1] = \{2, 3\}$$
$$Adj[2] = \{3\}$$
$$Adj[3] = \{\}$$
$$Adj[4] = \{3\}$$

- Para gráficas no dirigidas, $|Adj[v]| = degree(v)$.
- Para digráficas, $|Adj[v]| = out\text{-}degree(v)$.

Listas de adyacencia



- 0: $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 6
- 1: $\rightarrow$ 0 $\rightarrow$ 3
- 2: $\rightarrow$ 4
- 3: $\rightarrow$ 0 $\rightarrow$ 1 $\rightarrow$ 5
- 4: $\rightarrow$ 2
- 5: $\rightarrow$ 3
- 6: $\rightarrow$ 0

Listas de adyacencia

- En gráficas no dirigidos, cada eje (u,v) se almacena dos veces, una vez en la lista de vecinos de u y otra en la lista de vecinos de v.
- Para gráficas dirigidas cada eje se almacena una sola vez.
- De cualquier manera el espacio requerido para una lista de adyacencia es O(V+E).
- Listar a los vecinos de un nodo v toma O(1+deg(v)) tiempo.
- ¿A qué otra estructura de datos les recuerda?
  - hashing con encadenamiento.

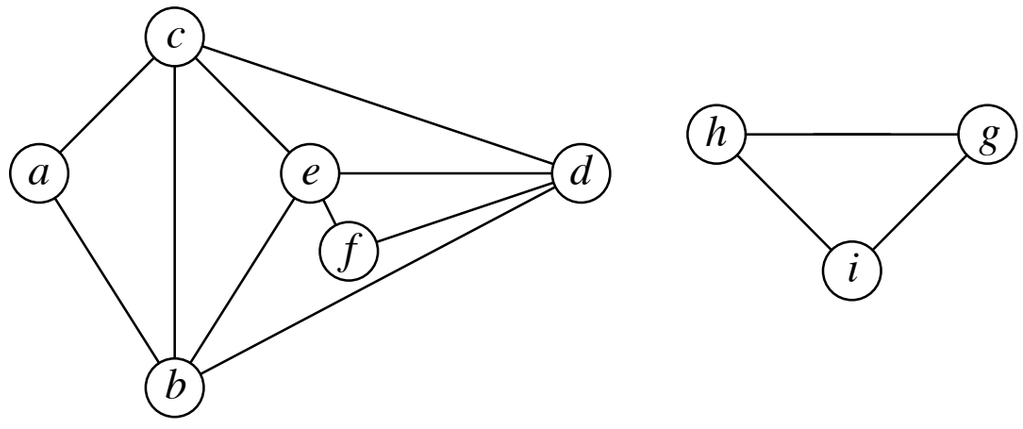
Listas de adyacencia

- Si la estructura de la gráfica es estática (es decir, que no cambia mientras el algoritmo se ejecuta), es común representar las listas de ejes entrantes y salientes como vectores, por eficiencia.
- Para algoritmos más elaborados como gráficas con pesos, se utiliza frecuentemente esta representación.

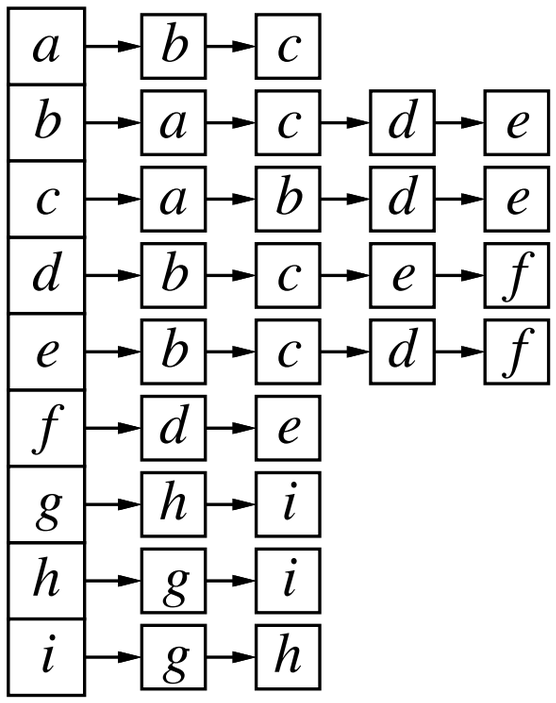
```
public class Edge {
  Vertex x, y;
  double weight;
}

public class Vertex {
  Set<Edge> out;
  Set<Edge> in;
}
```

Representaciones explícitas de grafos



	a	b	c	d	e	f	g	h	i
a	0	1	1	0	0	0	0	0	0
b	1	0	1	1	1	0	0	0	0
c	1	1	0	1	1	0	0	0	0
d	0	1	1	0	1	1	0	0	0
e	0	1	1	1	0	1	0	0	0
f	0	0	0	1	1	0	0	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	1	0	1
i	0	0	0	0	0	0	1	1	0



Representación de gráficas

- Dos maneras estándar de representar una gráfica G=(V,E):
  - listas de adyacencia
  - matrices de adyacencia

- Aplicables a gráficas dirigidas y gráficas no dirigidas.

- La representación con listas de adyacencia se prefiere cuando las gráficas son:
  - ralas, es decir...
  - $|E| << |V^2|$.

- La representación con matrices de adyacencia se prefiere cuando las gráficas son:
  - densas, es decir...
  - $|E| \sim |V^2|$ o cuando queremos hacer qué operación?
  - encontrar conectividad entre nodos.

- En general utilizaremos listas de adyacencia a menos que se especifique lo contrario.

Ejemplo del ADT de una gráfica

```
struct Edge
{
   int v, w;
   Edge( int v0=-1, int w0=-1) : v(v0), w(w0){ }
};
```

```
class GRAPH
{
   private:
      // Codigo que depende de la implementacion
   public:
      GRAPH( int, bool ); // numero de vertices,
                          // dirigido o no dirigido
      ~GRAPH();
      int V() const;
      int E() const;
      bool directed() const;
      int insert( Edge );
      int remove( Edge );
      bool edge( int, int );
      class adjIterator
      {
         public:
            adjIterator(const GRAPH &, int);
            int beg();
            int nxt();
            bool end();
      };
};
```

- Ejemplo de un cliente que procesa gráficas con el ADT anterior.

```
vector<Edge> edges(Graph &G)
{
   int E=0;
   vector<Edge> a(G.E());
   for( int v=0; v < G.V(); v++ ){
      Graph::adjIterator A(G,v);
      for( int w=A.beg(); !A.end(); w=A.nxt()){
         if( G.directed() || v < w)
            a[E++] = Edge(v,w);
      }
   return a;
}
```

- regresa un vector de la STL de C++.

Implementación del ADT con matrices de adyacencia

```
class DenseGRAPH
{
  private:
    int Vcnt, Ecnt;
    bool digraph;
    vector<vector<bool>> adj;

  public:
    DenseGRAPH(int V, bool digraph=false):
adj(V),Vcnt(V),Ecnt(0),digraph(digraph){
      for( int i=0; i < V; i++){
        adj[i].assign(V,false);
      }
    }
    int V() const { return Vcnt; }
    int E() const { return Ecnt; }
    bool directed() const{
       return digraph; }
```

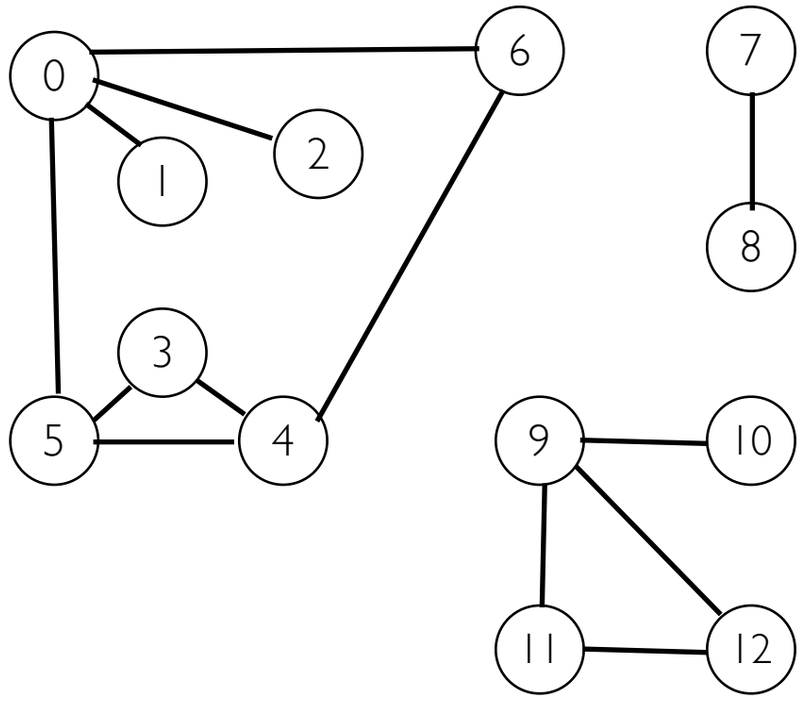
```
void insert(Edge e){
      int v=e.v, w=e.w;
      if( adj[v][w] == false ) Ecnt++;
      adj[v][w] = true;
      if( !digraph ) adj[w][v] = true;
    }
    void remove(Edge e){
      int v=e.v, w=e.w;
      if( adj[v][w] == true ) Ecnt--;
      adj[v][w] = false;
      if( !digraph ) adj[w][v] = false;
    }
    bool edge(int v, int w) const {
      return adj[v][w];
    }
    class adjIterator;
    friend class adjIterator;
};
```

Implementación del ADT con matrices de adyacencia

```
class DenseGRAPH::adjIterator
{
  private:
    const DenseGRAPH &G;
    int i,v;
  public:
    adjIterator( const DenseGRAPH &G, int v) : G(G), v(v), i(-1){ }
    int beg(){
      i=-1;
      return nxt();
    }
    int nxt(){
      for( i++; i < G.V(); i++ )
        if( G.adj[v][i] == true)
          return i;
        return -1;
      }
    bool end(){
      return i >= G.V(); }
};
```

Implementación del ADT con matrices de adyacencia

0	0	1	1	0	0	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0



Implementación del ADT con matrices de adyacencia

- Procesar todos los vértices adyacentes a un vértice dado requiere al menos de un tiempo:
  - ➡ proporcional a V.
- Esta interfaz requiere conocer el número de vértices V al inicio.
- Inicializar todos los campos de la matriz al valor false toma un tiempo:
  - ➡ proporcional a V^2.
- Esta representación:
  - ➡ ¿permite aristas paralelas?
    no
  - ➡ ¿permite self-loops?
    si

- La representación con matriz de adyacencia no es satisfactoria para gráficas ralas grandes, necesitamos al menos V^2 bits para almacenamiento y V^2 pasos para construcción.

Implementación del ADT con listas de adyacencia

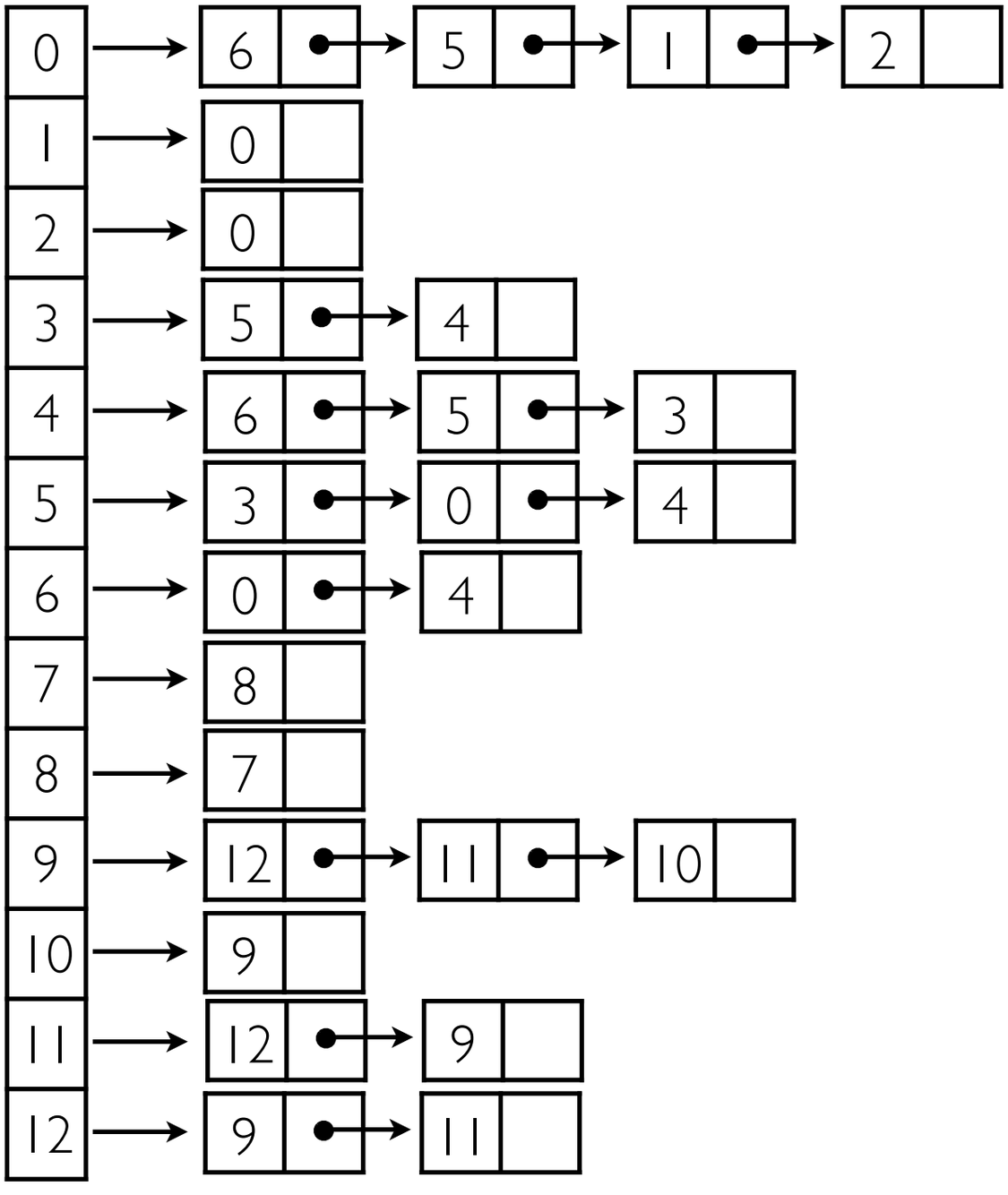
```
class SparseMultiGRAPH
{
  int Vcnt, Ecnt;
  bool digraph;
  struct node {
    int v;
    node *next;
    node( int x, node* t ){
      v = x;
      next = t;
    };
    typedef node* link;
    vector<link> adj;
```

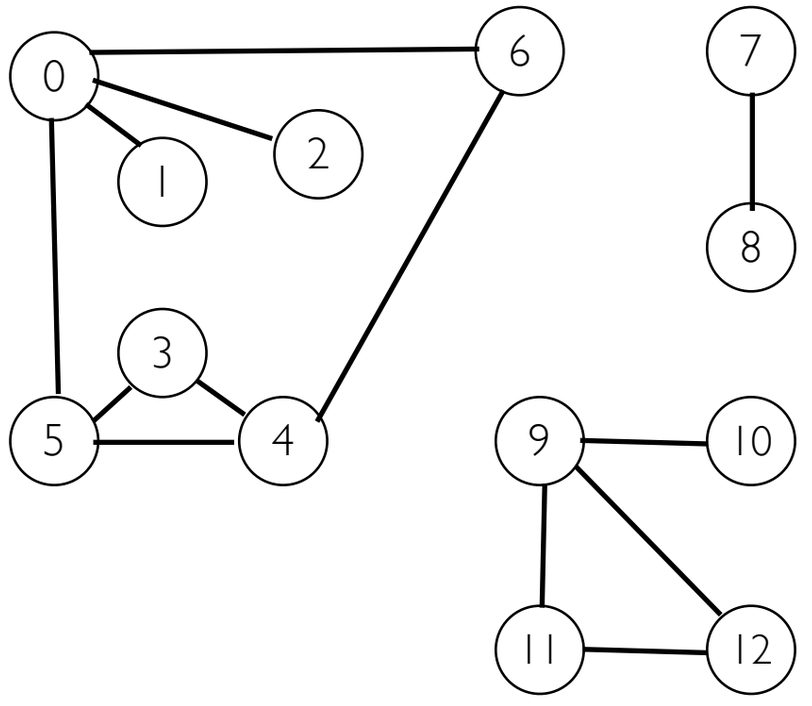
```
public:
    SparseMultiGRAPH( int V, bool digraph=false) :
adj(V), Vcnt(V), Ecnt(0), digraph(digraph){
      adj.assign(V,0);
    }
    int V() const { return Vcnt; }
    int E() const { return Ecnt; }
    bool directed() const { return digraph; }
    void insert( Edge e ){
      int v=e.v, w=e.w;
      adj[v] = new node( w, adj[v]);
      if(!digraph) adj[w] = new node( v, adj[w]);
      Ecnt++;
    }
    void remove(Edge e);
    bool edge( int v, int w) const;
    class adjIterator;
    friend class adjIterator;
};
```

Implementación del ADT con listas de adyacencia

```
class SparseMultiGRAPH::adjIterator
{
     const SparseMultiGRAPH &G;
     int v;
     link t;
   public:
     adjIterator( const SparseMultiGRAPH &G, int v) : G(G), v(v){ t=0; }
     int beg(){
       t = G.adj[v];
       return t?t->v:-1;
     }
     int nxt(){
       if(t) t=t->next;
       return t?t->v:-1;
     }
     bool end(){
       return t==0;
     }
};
```

Implementación del ADT con listas de adyacencia





Implementación del ADT con listas de adyacencia

- Es la representación más utilizada para gráficas que no son densas.
- Una arista se puede agregar en tiempo constante.
- El espacio total es proporcional al número de vértices más el número de aristas.
- Al usar la representación de listas ligadas hay que utilizar el contenedor de la STL o recordar incluir un destructor y un constructor por copia.
- Esta representación ...
  - ¿permite aristas paralelas? si
  - ¿permite self-loops? si
- La principal desventaja es que requiere un tiempo proporcional a V para determinar la existencia de una arista.

Costos en el peor caso para diferentes implementaciones

	arreglo de aristas	matriz de adyacencia	listas de adyacencia
espacio	E	V^2	$V+E$
inicialización	1	V^2	V
copia	E	V^2	$E+V$
destrucción	1	V	E
insertar arista	1	1	1
eliminar arista	E	1	V
¿nodo aislado?	E	V	1
¿camino de u a v?	$E \lg V$	V^2	$V+E$