

# Grafos

- Representaciones
- Las interfaces **Grafo**, **Vertice** y **Arista**
- Implementaciones de la interface **Grafo**
  - Con lista de adyacencia
  - Con matriz de adyacencia
- Recorrido de grafo: DFS (Depth First Search)
  - Implementación en JAVA del dfs
  - Ejemplo 1: recuperar todos los caminos de un costo dado.

# Grafos

## Representaciones

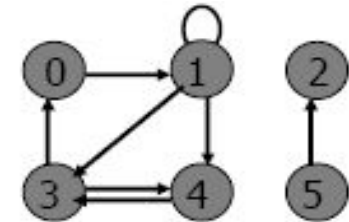
### (1) Matriz de adyacencia

Un grafo  $G=(V,A)$  se representa como una matriz de booleanos de  $|V| \times |V|$  donde:

$G[u,v]$   $\left\{ \begin{array}{l} \text{true si } (u, v) \in A \\ \text{false en cualquier otro caso} \end{array} \right.$

	0	1	2	3	4	5
0	false	true	false	false	false	false
1	false	true	false	true	true	false
2	false	false	false	false	false	false
3	true	false	false	false	true	false
4	false	false	false	true	false	false
5	false	false	true	false	false	false

$G=(V,A)$

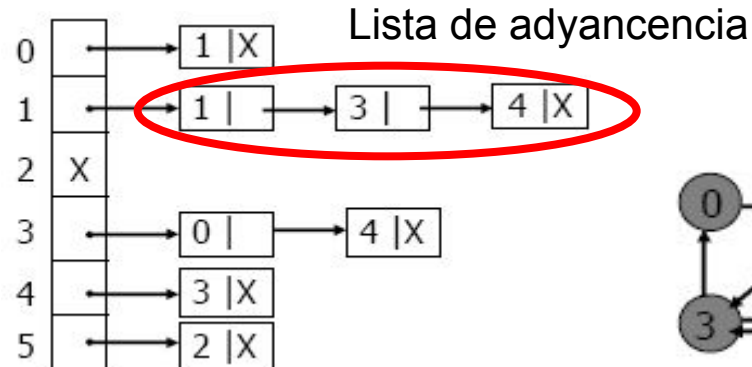


### (2) Lista de adyacencia

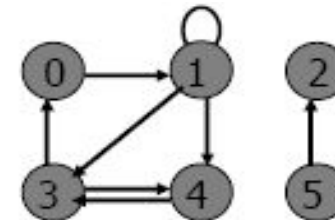
Un grafo  $G=(V,A)$  se representa como un arreglo/lista de tamaño  $|V|$  de vértices.

**Posición  $i$**   $\rightarrow$  puntero a una lista enlazada de elementos, **lista de adyacencia**.

Los elementos de la lista son los vértices adyacentes a  $i$ .

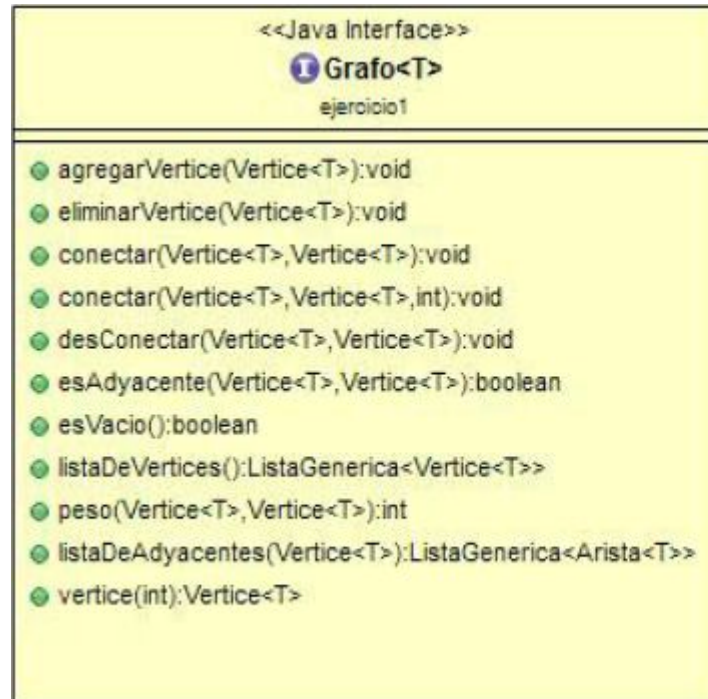


$G=(V,A)$



# Grafos

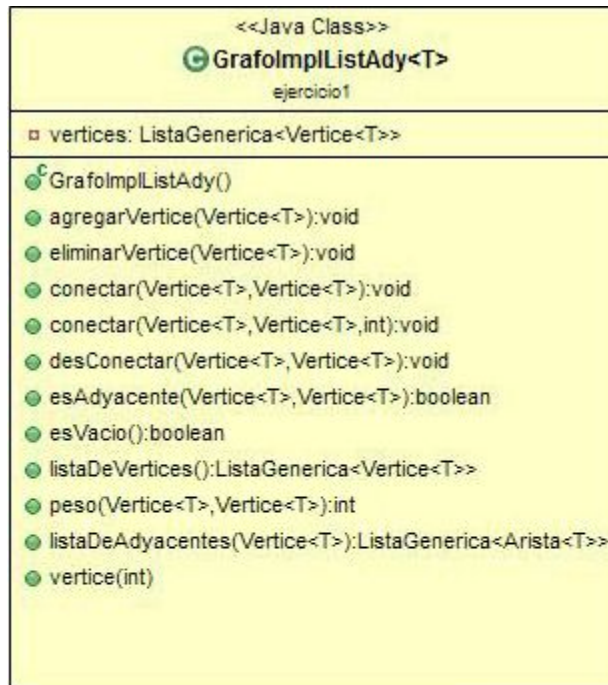
## La interfaces que definen Grafos



Definir Grafos en JAVA con interfaces nos independiza de las implementaciones. Podríamos definir una implementación basada en matriz de adyacencia, otra en listas de adyacencia

# Grafos implementados con Listas de Adyacencia

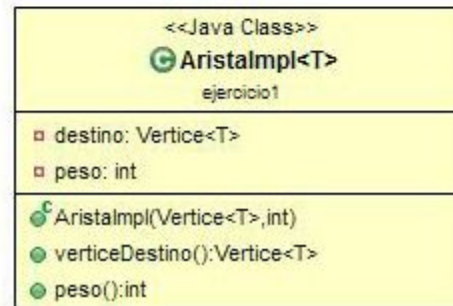
Implementa la  
interface Grafo<T>



Implementa la  
interface Vertice<T>

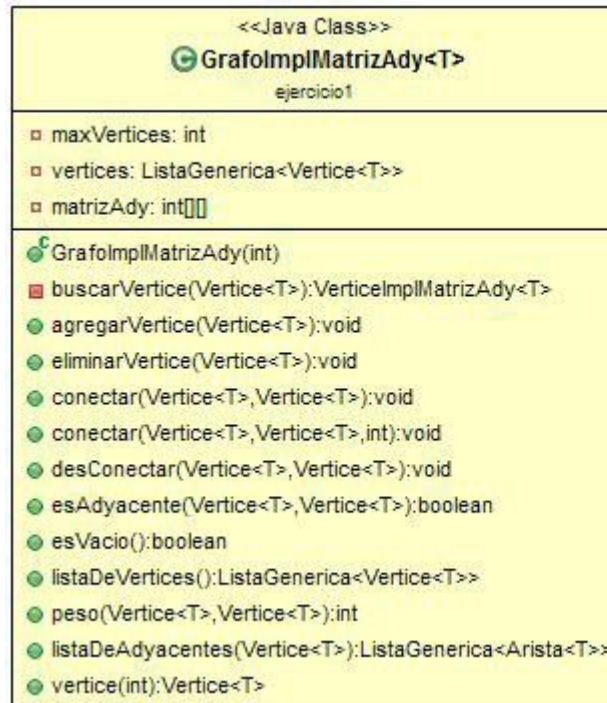


Implementa la  
interface Arista<T>

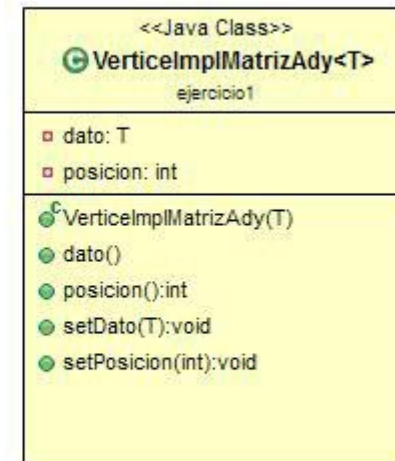


# Grafos implementados matriz de adyacencia

Implementa la  
interface Grafo<T>



Implementa la  
interface Vertice<T>



# Grafos

## Clase que implementa la interface Vertice

### (con Listas de Adyacencia)

```
package ejercicio1;

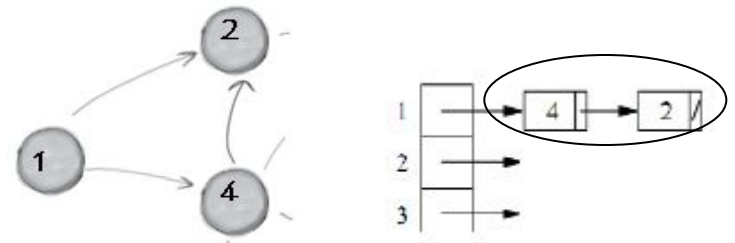
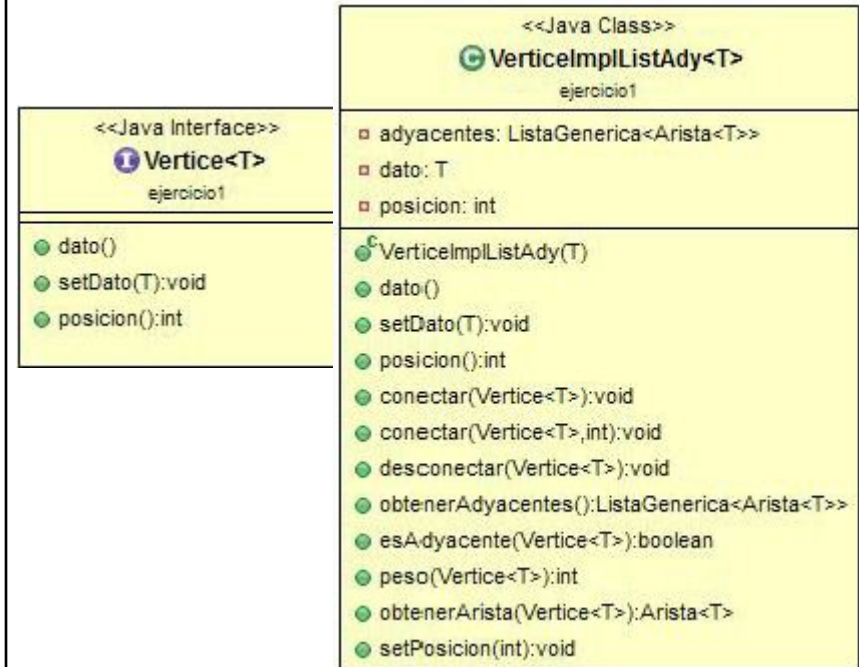
public class VerticeImplListAdy<T> implements Vertice<T> {
    private T dato;
    private int posicion;
    private ListaGenerica<Arista<T>> adyacentes;

    public VerticeImplListAdy(T d) {
        dato = d;
        adyacentes = new ListaEnlazadaGenerica<Arista<T>>();
    }

    public int posicion() {
        return posicion;
    }

    public void conectar(Vertice<T> v) {
        conectar(v, 1);
    }

    public void conectar(Vertice<T> v, int peso) {
        Arista a = new AristaImpl(v, peso);
        if (!adyacentes.incluye(a))
            adyacentes.agregarFinal(a);
    }
}
```



**Vértice:** tiene un dato y una lista de adyacentes. En realidad se tiene una lista de aristas, donde cada una tiene un vértice destino.



# Grafos

## La clase que implementa a la interface Arista (con Listas de Adyacencia)

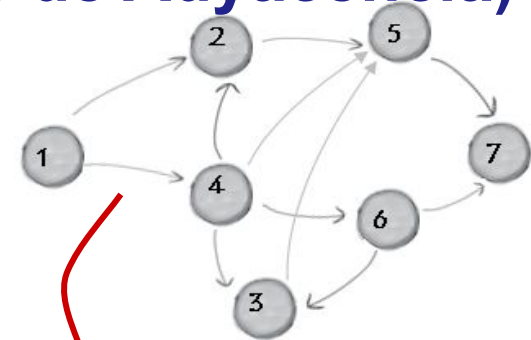
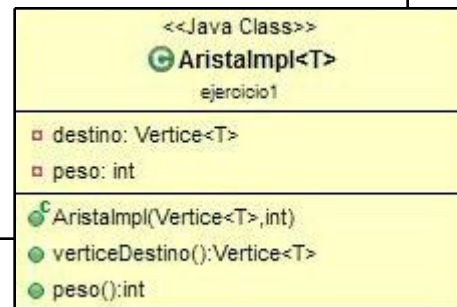
```
package ejercicio1;
```

```
public class AristaImpl<T> implements Arista<T> {  
    private Vertice<T> destino;  
    private int peso;
```

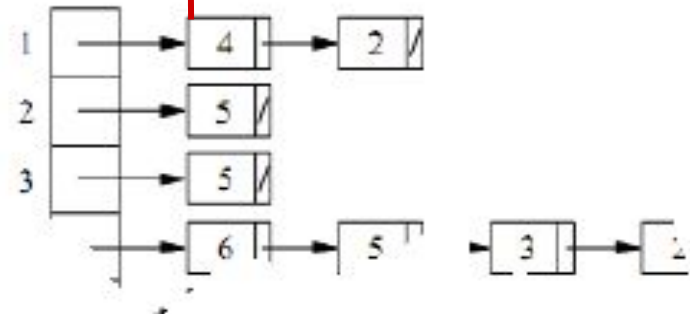
```
    public AristaImpl(Vertice<T> dest, int p){  
        destino = dest;  
        peso = p;  
    }
```

```
    public Vertice<T> verticeDestino() {  
        return destino;  
    }
```

```
    public int peso() {  
        return peso;  
    }  
}
```



**Arista:** una arista siempre tiene el destino y podría tener un peso.

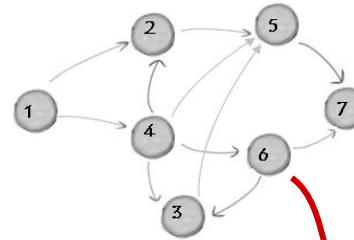


Podría llamarse **AristaImplListaAdy** porque que sólo se usa para Lista de Adyacencias

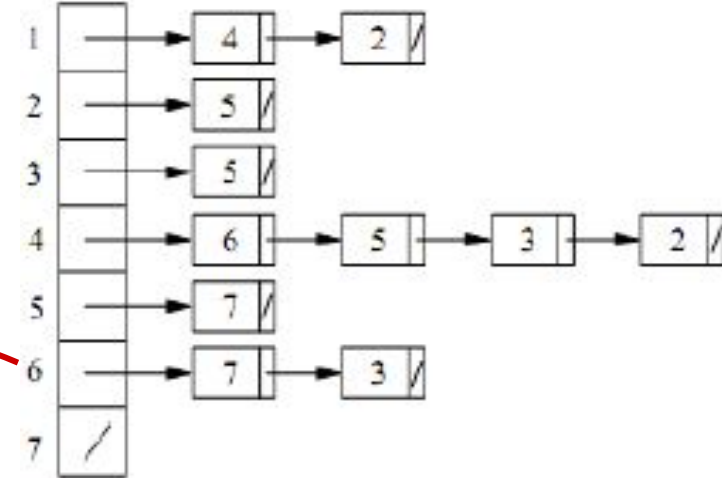
# Grafos

## La clase que implementa a la interface Grafo

(con Listas de Adyacencia)



vertices



```
public class GrafoImplListAdy<T> implements Grafo<T> {
    private ListaGenerica< VerticeImplListAdy<T> > vertices = new
        ListaEnlazadaGenerica<VerticeImplListAdy<T>>();

    public void agregarVertice(Vertice<T> v) {
        if (!vertices.incluye(v)) {
            v.setPosicion(vertices.tamano());
            vertices.agregarFinal(v);
        }
    }

    public void conectar(Vertice<T> origen, Vertice<T> destino) {
        origen.conectar(destino);
    }

    public void conectar(Vertice<T> origen, Vertice<T> destino, int peso) {
        origen.conectar(destino,peso);
    }
}
```



# Grafos

## DFS (Depth First Search)

El DFS es un algoritmo de recorrido de grafos en profundidad. Generalización del recorrido preorden de un árbol.

Esquema recursivo

Dado  $G = (V, A)$

1. Marcar todos los vértices como no visitados.
2. Elegir vértice  $u$  (no visitado) como punto de partida.
3. Marcar  $u$  como visitado.
4. Para todo  $v$  adyacente a  $u$ ,  $(u,v) \in A$ , si  $v$  no ha sido visitado, repetir recursivamente (3) y (4) para  $v$ .

## ¿Cuándo finaliza el recorrido?

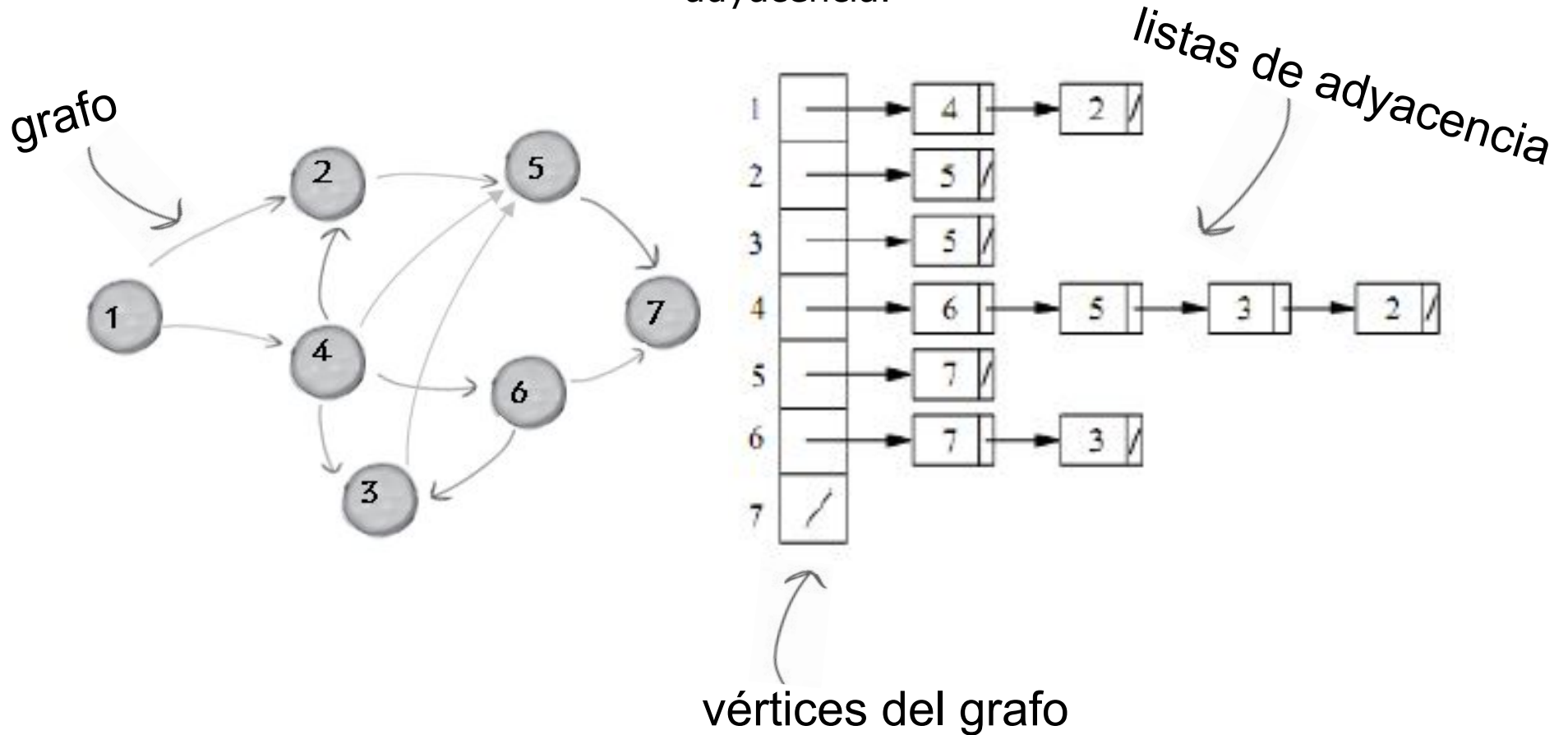
Finalizar cuando se visitaron todos los nodos alcanzables desde  $u$ .

Si desde  $u$  no fueran alcanzables todos los nodos del grafo: volver a (2), elegir un nuevo vértice de partida  $u$  no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

# Grafos

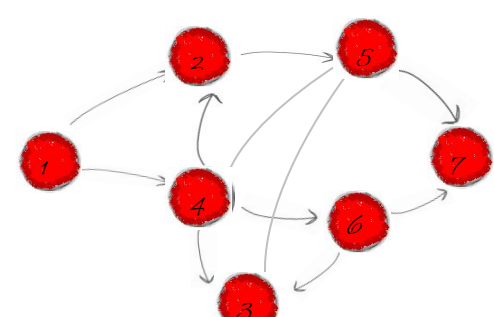
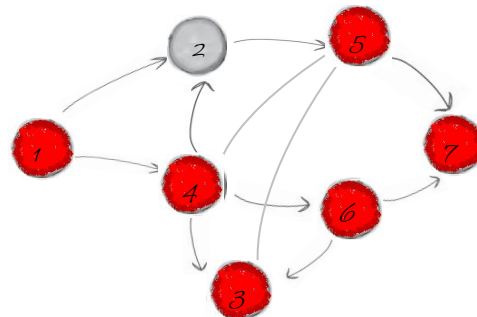
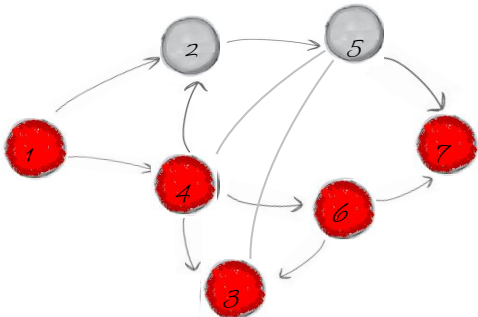
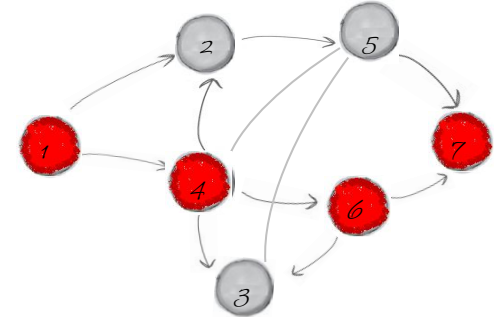
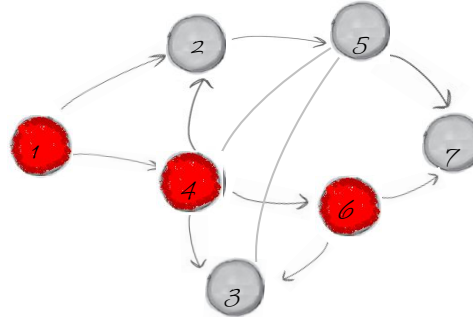
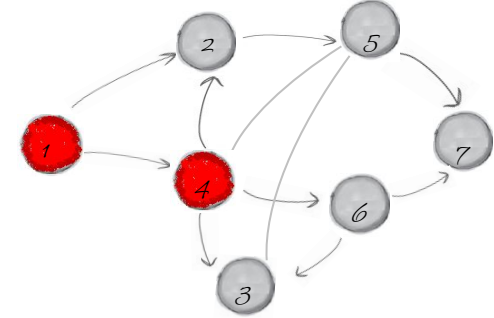
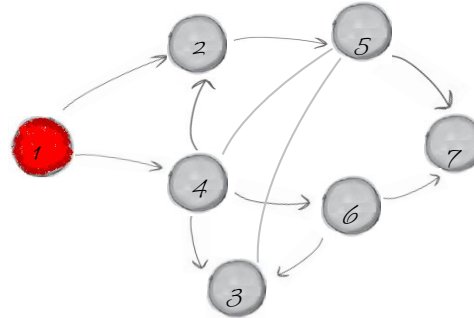
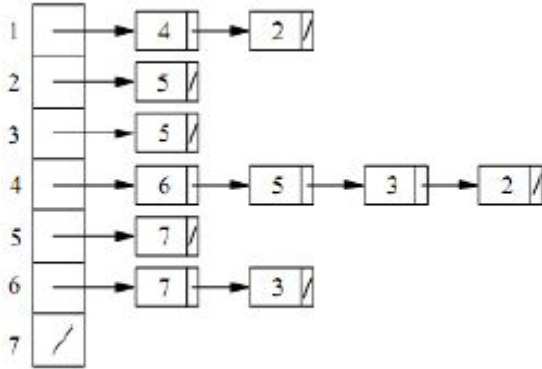
## DFS (Depth First Search)

El recorrido del DFS depende del orden en que aparecen los vértices en las listas de adyacencia.



# Grafos

## DFS (Depth First Search)



# Grafos

## DFS (Depth First Search)

```
public class Recorridos<T> {

    public void dfs(Grafo<T> grafo){
        boolean[] marca = new boolean[grafo.listaDeVertices().tamano()];
        for(int i=0; i<grafo.listaDeVertices().tamano();i++){
            if (!marca[i]) // si no está marcado
                this.dfs(i, grafo, marca);
        }
    }

    private void dfs(int i,Grafo<T> grafo, boolean[] marca){
        marca[i] = true;
        Arista <T> arista=null;
        int j=0;
        Vertice<T> v = grafo.listaDeVertices().elemento(i);
        System.out.println(v);
        ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);
        ady.comenzar();
        while(!ady.fin()){
            arista=ady.proximo();
            j = arista.getVerticeDestino().getPosicion();
            if(!marca[j])
                this.dfs(j, grafo, marca);
        }
    }
}
```

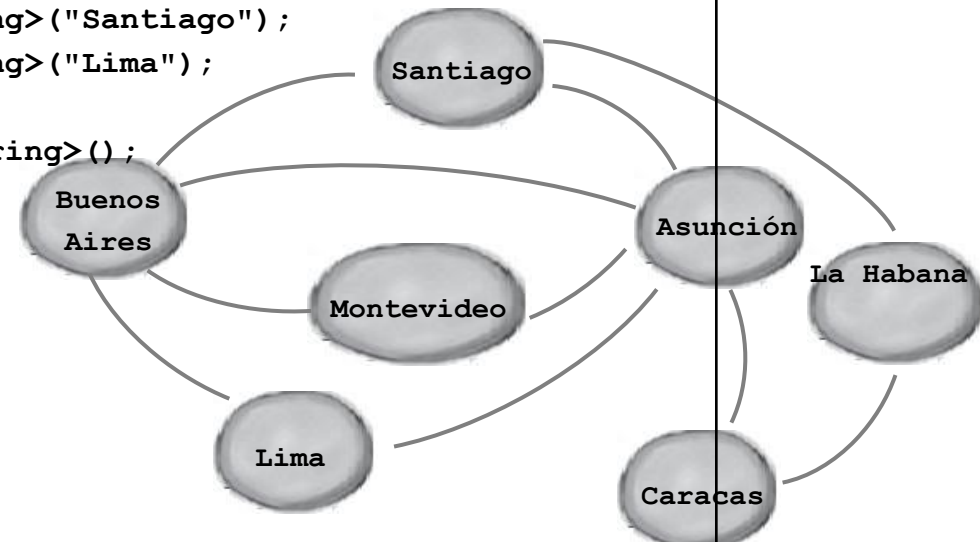
# Grafos

## DFS (Depth First Search)

### Uso del DFS

```
public class RecorridosTest {

    public static void main(String[] args) {
        Vertice<String> v1 = new VerticeImplListAdy<String>("Buenos Aires");
        Vertice<String> v2 = new VerticeImplListAdy<String>("Santiago");
        Vertice<String> v3 = new VerticeImplListAdy<String>("Lima");
        //TODO: completar los vértices
        Grafo<String> ciudades = new GrafoImplListAdy<String>();
        ciudades.agregarVertice(v1);
        ciudades.agregarVertice(v2);
        ciudades.agregarVertice(v3);
        //TODO: agregar los vértices al grafo
        ciudades.conectar(v1, v2);
        ciudades.conectar(v1, v3);
        ciudades.conectar(v1, v4);
        ciudades.conectar(v1, v5);
        ciudades.conectar(v3, v5);
        //TODO: completar las aristas
        Recorridos<String> r = new Recorridos<String>();
        System.out.println("--- Se imprime el GRAFO con DFS ---");
        r.dfs(ciudades);
    }
}
```



```
Console
<terminated> RecorridosTest [Java Application] C:\Program Ixe (30/05/2012)
--- Se imprime el GRAFO con DFS ---
Buenos Aires
Lima
Asuncion
Montevideo
Santiago
La Habana
Caracas
```

# Grafos

## DFS (Depth First Search)

```
public class Recorridos<T> {
    public ListaEnlazadaGenerica<Vertice<T>> dfs(Grafo<T> grafo) {
        boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()];
        ListaEnlazadaGenerica<Vertice<T>> lis = new ListaEnlazadaGenerica<Vertice<T>>();
        for(int i=0; i<grafo.listaDeVertices().tamanio();i++){
            if (!marca[i])
                this.dfs(i, grafo, lis, marca);
        }
        return lis;
    }
    private void dfs(int i,Grafo<T> grafo,ListaEnlazadaGenerica<Vertice<T>> lis,boolean[] marca) {
        marca[i] = true;
        int j=0;
        Arista<T> arista=null;
        Vertice<T> v = grafo.listaDeVertices().elemento(i);
        lis.agregar(v, lis.tamanio());
        ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);
        ady.comenzar();
        while(!ady.fin()){
            arista= ady.proximo();
            j= arista.getVerticeDestino().getPosicion();
            if (!marca[j]){
                this.dfs(j, grafo, lis, marca);
            }
        }
    }
}
```

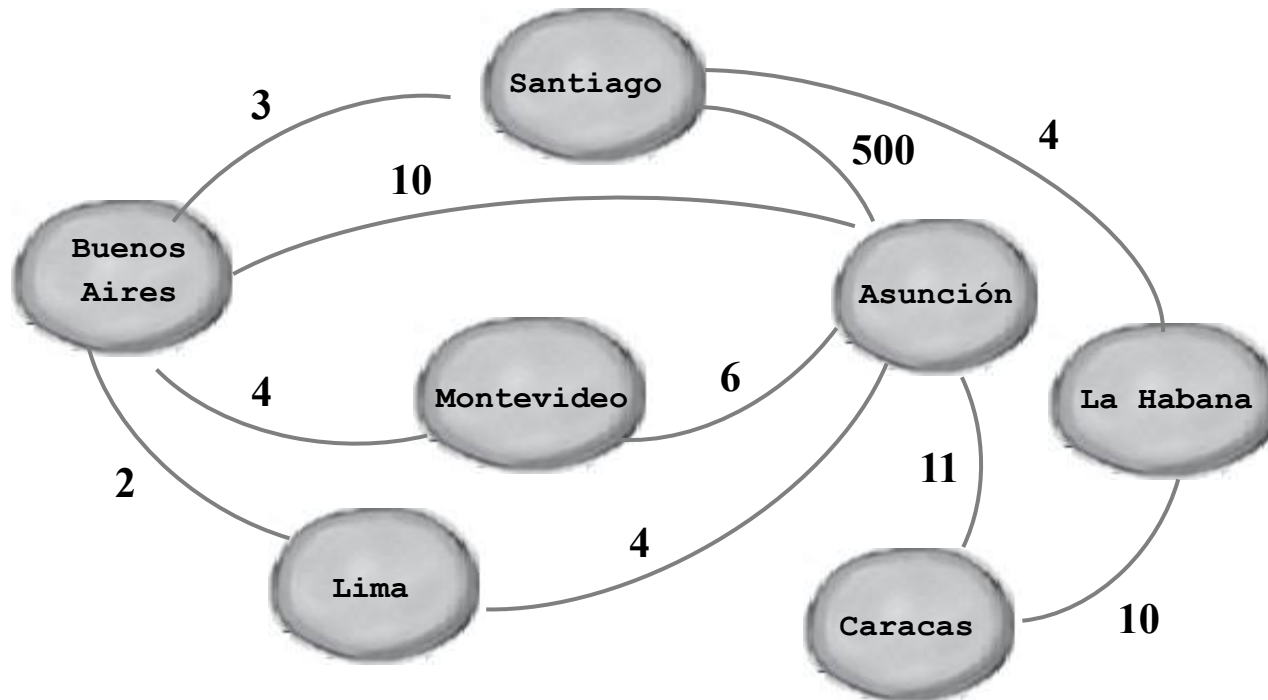
DFS que guarda vértices visitados en una lista



# Ejemplo 1: recuperar todos los caminos de un costo dado

Dado un Grafo orientado y valorado positivamente, como por ejemplo el que muestra la figura, implemente un método que retorne una **lista con todos los caminos cuyo costo total sea igual a 10**. Se considera **costo total del camino** a la suma de los costos de las aristas que forman parte del camino, desde un vértice origen a un vértice destino.

Se recomienda implementar un método público que invoque a un método recursivo privado.



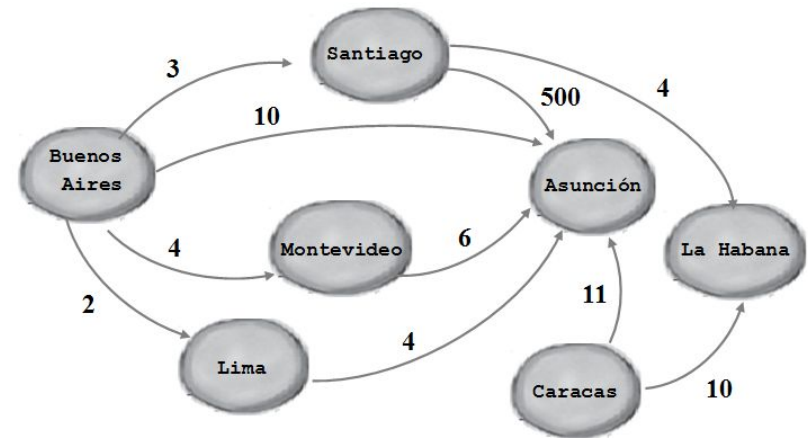
# Ejemplo 1

```
public class Recorridos {  
    public ListaGenerica<ListaGenerica<Vertice<T>>> dfsConCosto(Grafo<T> grafo) {  
        boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()];  
        ListaGenerica<Vertice<T>> lis = null;  
        ListaGenerica<ListaGenerica<Vertice<T>>> recorridos =  
            new ListaGenericaEnlazada<ListaGenericaEnlazada<Vertice<T>>>();  
  
        int costo = 0;  
        for(int i=0; i<grafo.listaDeVertices().tamanio();i++){  
            lis = new ListaGenericaEnlazada<Vertice<T>>();  
            lis.add(grafo.listaDeVertices().elemento(i));  
            marca[i]=true;  
            this.dfsConCosto(i, grafo, lis, marca, costo, recorridos);  
            marca[i]=false;  
        }  
        return recorridos;  
    }  
  
    private void dfsConCosto(int i, Grafo<T> grafo, ListaGenerica<Vertice<T>> lis,  
        boolean[] marca, int costo, ListaGenerica<ListaGenerica<Vertice<T>>> recorridos) {  
        //siguiente diapo  
    }  
}
```

```

public class Recorridos {
    private void dfsConCosto(int i, Grafo<T> grafo, ListaGenerica<Vertice<T>> lis,
        boolean[] marca, int costo, ListaGenerica<ListaGenerica<Vertice<T>>> recorridos) {
        Vertice<T> vDestino = null; int p=0,j=0;
        Vertice<T> v = grafo.listaDeVertices().elemento(i);
        ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);
        ady.comenzar();
        while(!ady.fin()){
            Arista<T> arista = ady.proximo();
            j = arista.verticeDestino().posicion();
            if(!marca[j]){
                p = arista.peso();
                vDestino = arista.verticeDestino();
                costo = costo+p;
                lis.agregarFinal(vDestino);
                marca[j] = true;
                if (costo==10){
                    recorridos.add((lis.copia())); // se crea una copia de la lista y se guarda esa copia
                }
                else this.dfsConCosto(j, grafo, lis, marca, costo, recorridos);
                costo=costo-p;
                lis.eliminar(vDestino);
                marca[j]= false;
            }
        }
    }
}

```



```

<terminated> RecorridosTest [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (31/05/2012 08:26:11)

Invoco a DFS con Costos:
[[Buenos Aires, Asuncion], [Buenos Aires, Montevideo, Asuncion], [Caracas, La Habana]]

```

```
public class Recorridos {
    private void dfsConCosto(int i, Grafo<T> grafo, ListaGenerica<Vertice<T>> lis,
        boolean[] marca, int costo, ListaGenerica<ListaGenerica<Vertice<T>>> recorridos) {
        Vertice<T> vDestino = null; int p=0,j=0;
        Vertice<T> v = grafo.listaDeVertices().elemento(i);
        ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);
        ady.comenzar();
        while(!ady.fin()){
            Arista<T> arista = ady.proximo();
            j = arista.verticeDestino().posicion();
            if(!marca[j]){
                p = arista.peso();
                if ((costo+p) <= 10) {
                    vDestino = arista.verticeDestino();
                    lis.agregarFinal(vDestino);
                    marca[j] = true;
                    if ((costo+p)==10)
                        recorridos.add(lis.copia());
                    else
                        this.dfsConCosto(j, grafo, lis, marca, costo+p, recorridos);
                    lis.eliminar(vDestino);
                    marca[j]= false;
                }
            }
        }
    }
}
```

