Grafos

- Representaciones: matriz de adyacencia y lista de adyacencia
- Implementaciones en JAVA:
  - Las interfaces: **Grafo, Vertice y Arista**
  - Implementaciones de la interface **Grafo**
    - Con lista de adyacencia
    - Con matriz de adyacencia
- Recorrido de grafo:
    - Recorrido en profundidad: DFS (Depth First Search)
    - Recorrido en amplitud: BFS (Breadth First Search)

Grafos
Representaciones

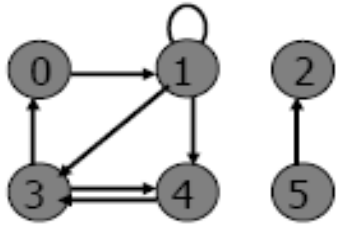
(1) Matriz de adyacencia

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

G[u,v] ⎧ **true** si (u, v) Є A
⎩ **false** en cualquier otro ca

	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)



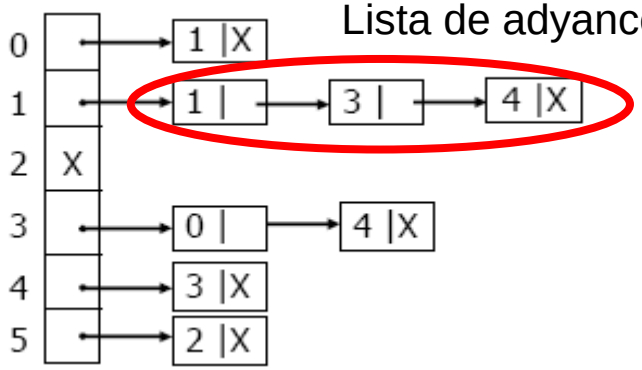
(2) Lista de adyacencia

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

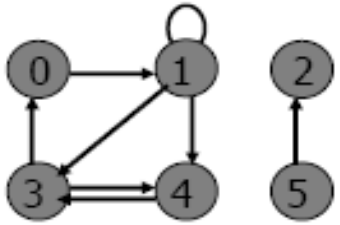
Posición i → puntero a una lista enlazada de elementos, **lista de adyacencia**.

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

Lista de adyancencia

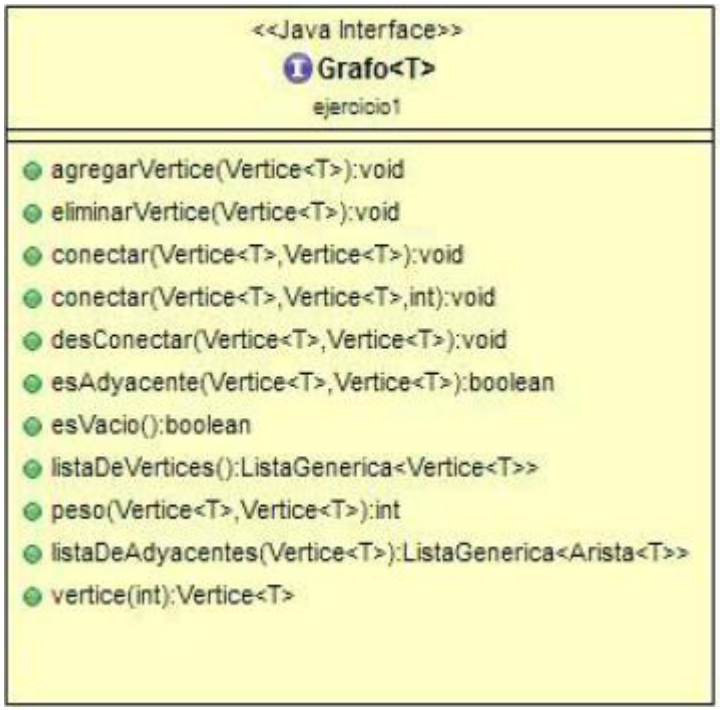


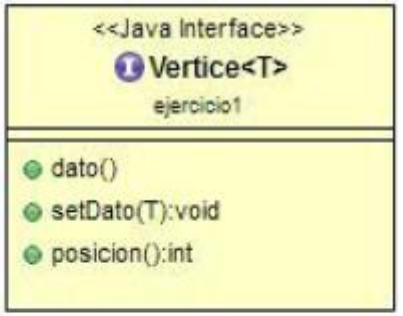
G=(V,A)



Grafos

La interfaces Grafo, Vertice y Arista





Interfaces secundarias

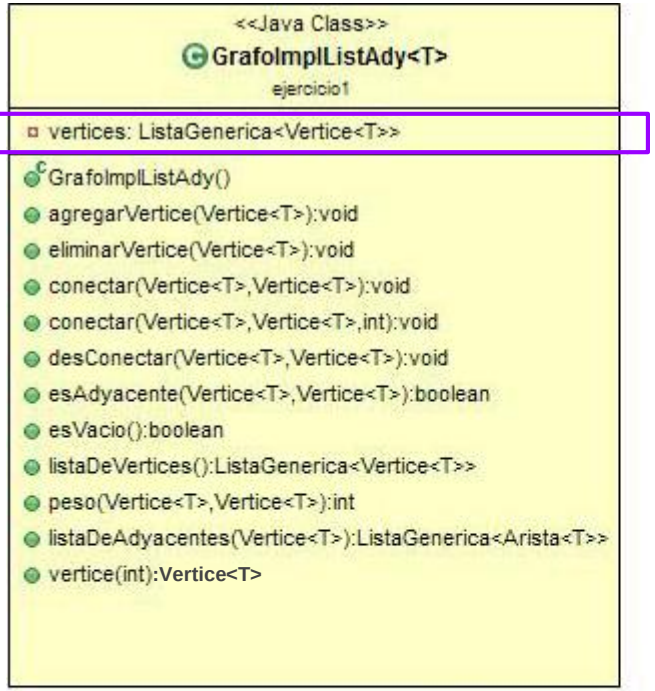
En las **interfaces** definimos el **comportamiento del Grafo o sus operaciones**: **¿qué puede hacer un Grafo?**

Las **interfaces genéricas** nos independiza de las **implementaciones concretas**.

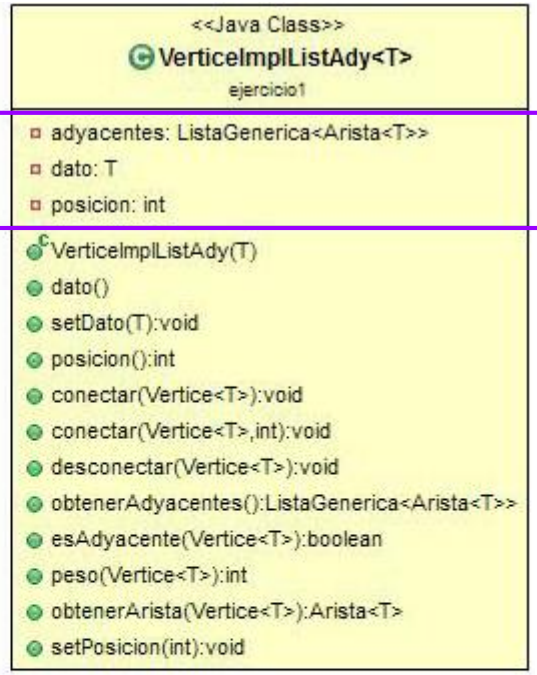
En las **implementaciones concretas** definimos **¿cómo se implementan cada una de las operaciones del grafo y cómo se representará?.** Podríamos definir una implementación basada en **matriz de adyacencia** y otra en **listas de adyacencia.**

Grafos

Implementación con Listas de Adyacencia

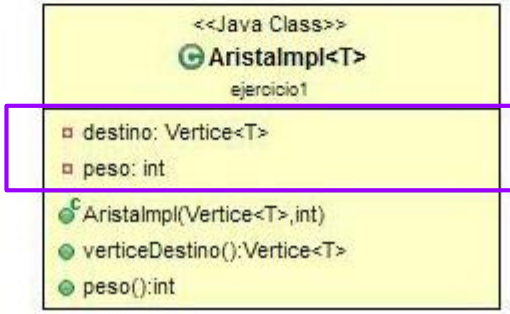


Implementa la interface Grafo<T>



Implementa la interface Vertice<T>

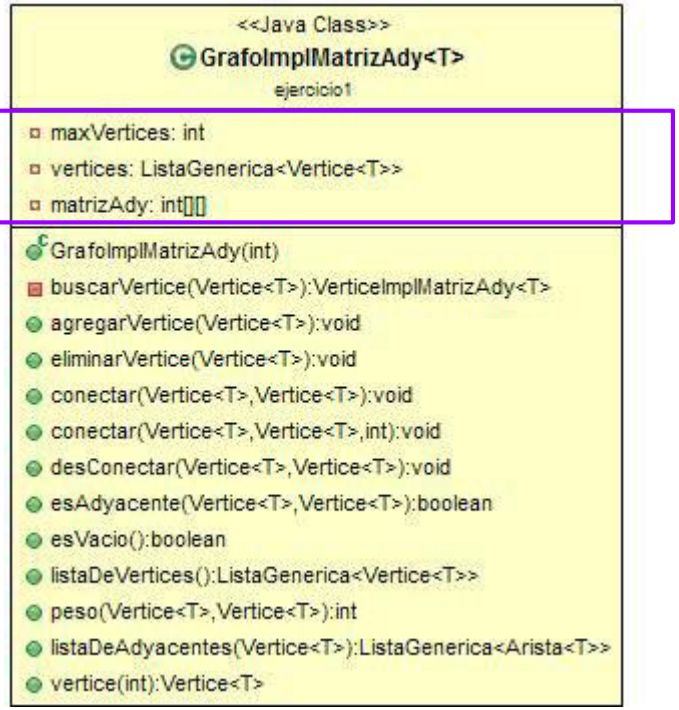
¿Dónde se almacena el Grafo?



Implementa la interface Arista<T>

Grafos
Implementación con matriz de adyacencia

Implementa la interface Grafo<T>



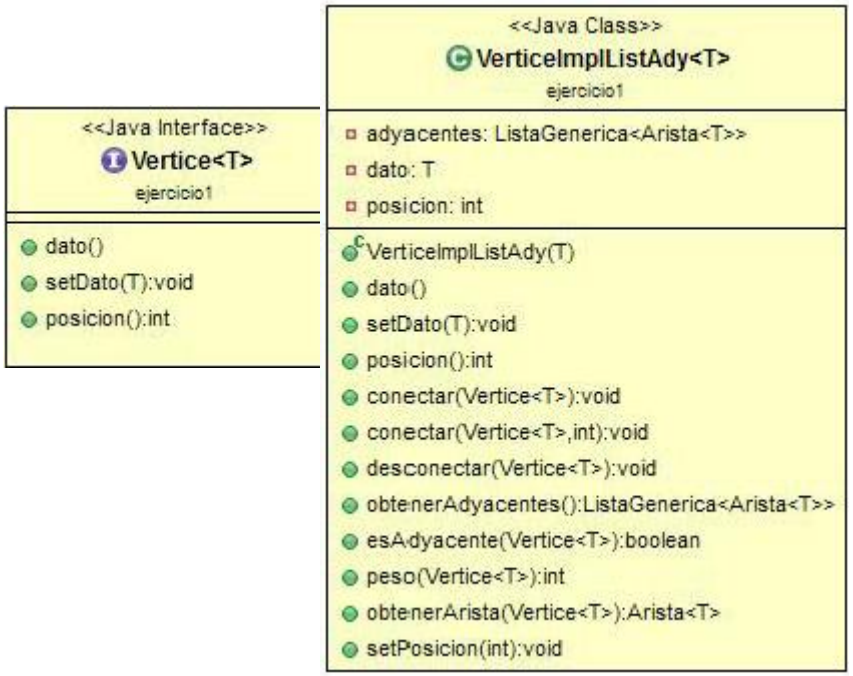
Implementa la interface Vertice<T>

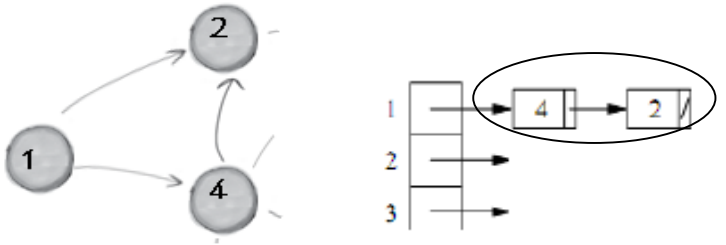
¿Dónde se almacena el Grafo?

Grafos

La 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);
  }
 }
}
```



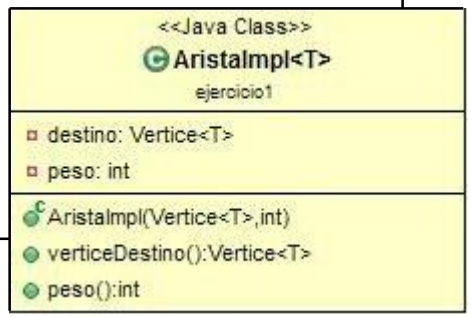


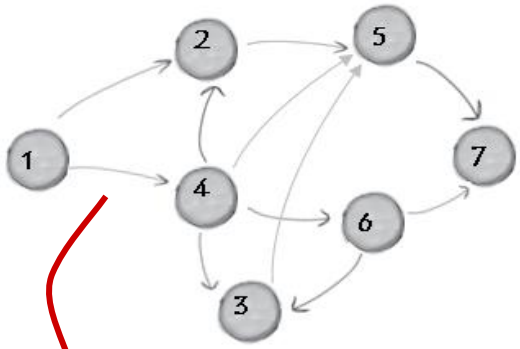
Vértice: tiene un dato y una lista de adyacentes. En realidad se tiene una lista de aristas, donde cada nodo contiene el vértice destino.

Grafos

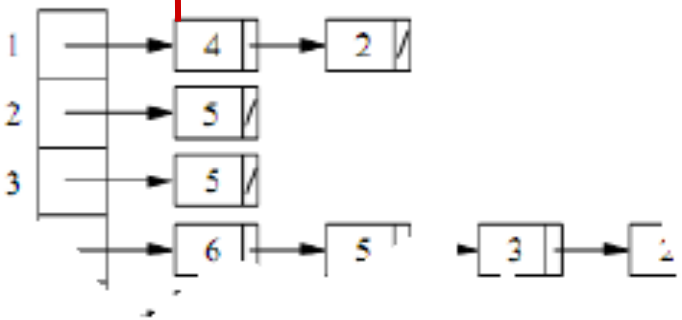
La clase que implementa a la interface `Arista` (con Listas de Adyacencia)

```
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.

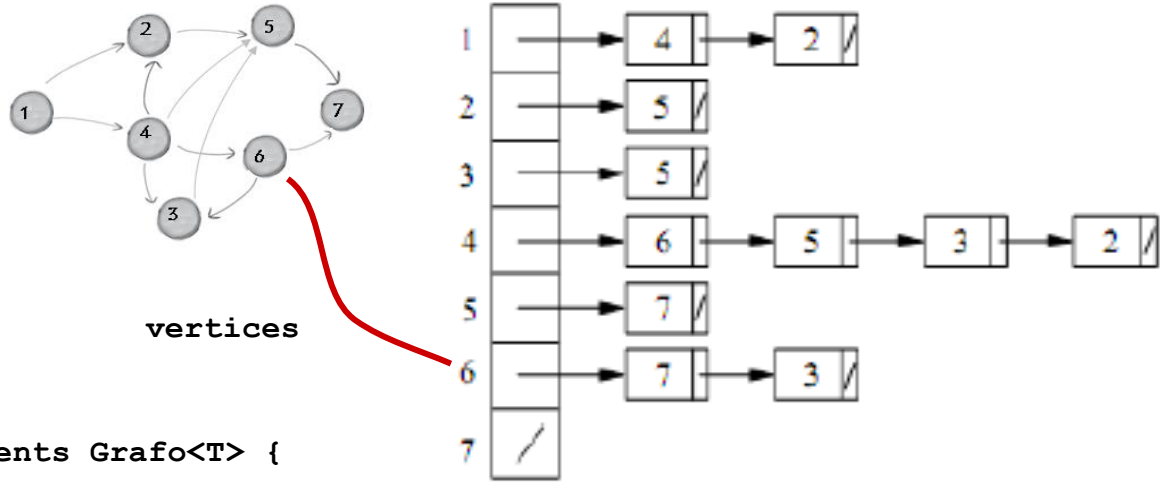


Podría llamarse **`AristaImplListaAdy`** porque que sólo se usa para Lista de Adyacencias.

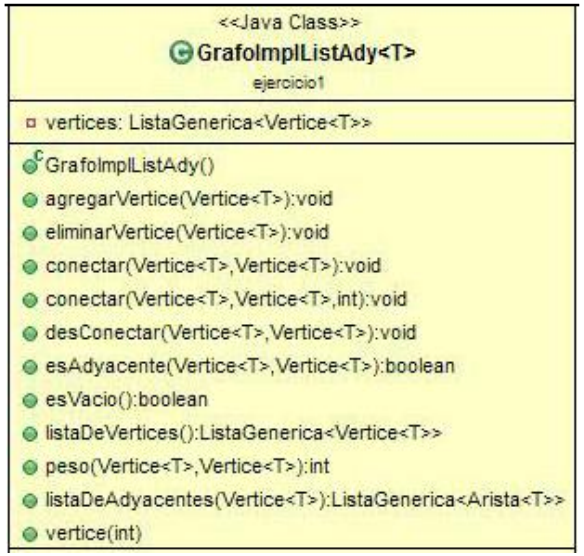
Grafos

La clase que implementa a la interface `Grafo`

(con Listas de Adyacencia)




```
package ejercicio1;

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.tamanio());
       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);
 }
}
```



Recorrido en profundidad de Grafos
Depth First Search (DFS)

El **DFS** es **equivalente** al recorrido **preorden de un árbol.**

El **DFS** explora las aristas del grafo de manera que **se visitan los vértices adyacentes al recién visitado**, con lo que se consigue **profundizar** en las ramas del grafo.

Es un recorrido recursivo.

Dado G = (V , A)

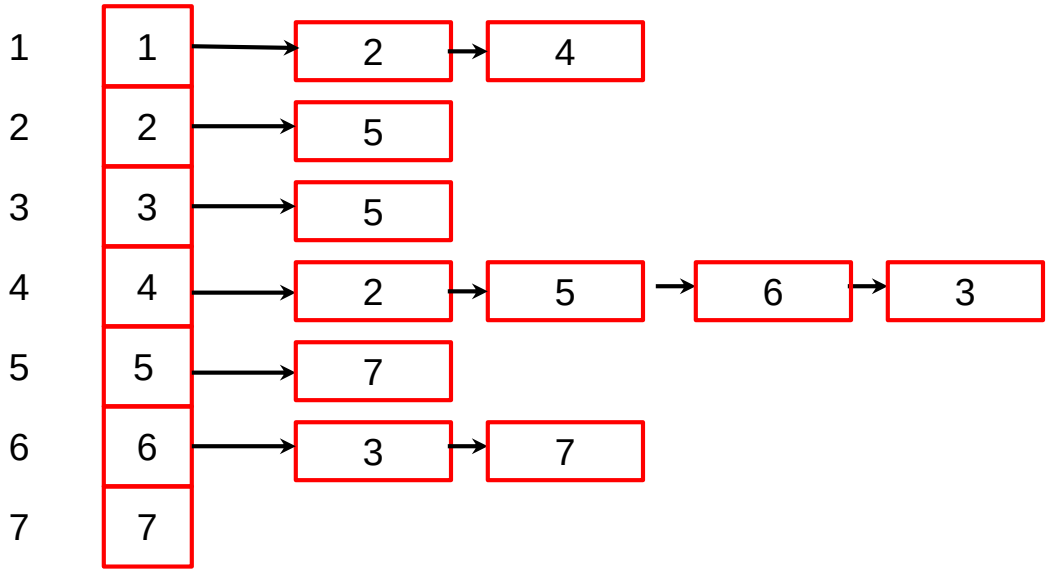
1. Marcar todos los vértices como no visitados.
2. Elegir vértice **u** (no visitado) como **punto de partida.**
3. Marcar **u** como visitado.
4. Para todo **v** adyacente a **u**, (u,v) Є A, si v no ha sido visitado, repetir recursivamente (3) y (4) para **v**.

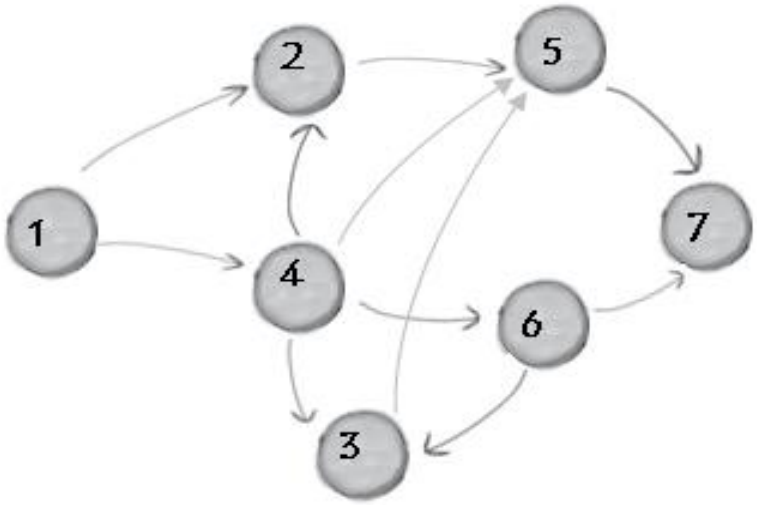
¿Cuándo finaliza el recorrido?

Finaliza 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**.

Recorrido en profundidad de Grafos
Depth First Search (DFS)





Es necesario **registrar los nodos visitados** para evitar recorrerlos varias veces.

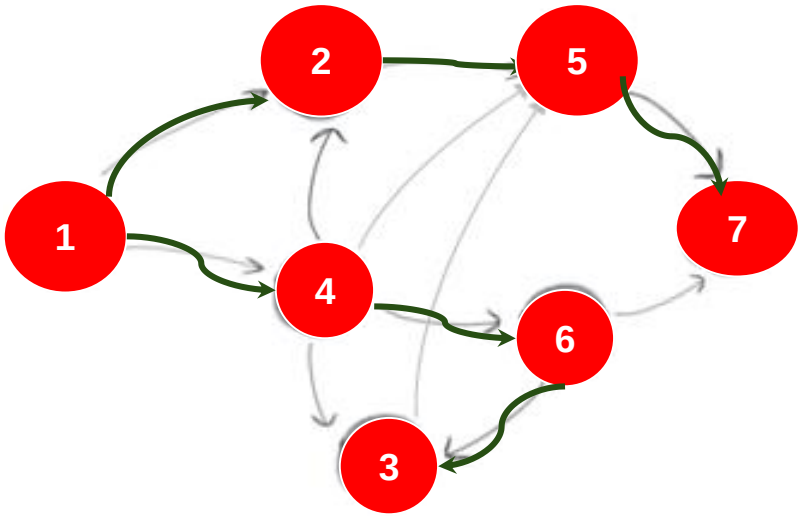
El **recorrido no es único:** depende del vértice inicial y del orden de visita de los vértices adyacentes.

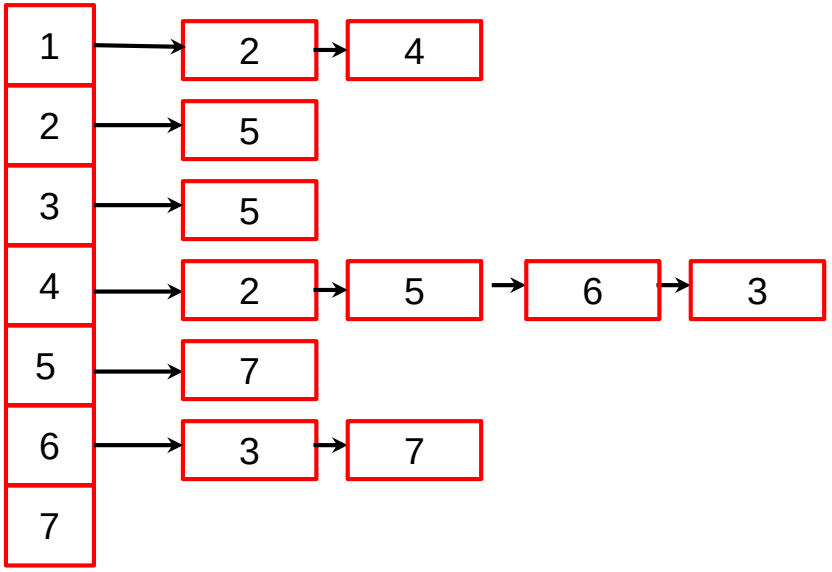
```
public class Recorridos<T> {

 public void dfs(Grafo<T> grafo){
   boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()];
   for(int i=0; i<grafo.listaDeVertices().tamanio();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);
    }
 }
}
```

Veamos cómo funciona el DFS en el ejemplo

Vértice de partida:1

Vértice - Lista de Adyacentes		Marca						
		1	2	3	4	5	6	7
		F	F	F	F	F	F	F
1	2 4	T	F	F	F	F	F	F
2	5	T	T	F	F	F	F	F
5	7	T	T	F	F	T	F	F
7		T	T	F	F	T	F	T
4	2 5 6 3	T	T	F	T	T	F	T
6	3 7	T	T	F	T	T	T	T
3	5	T	T	T	T	T	T	T





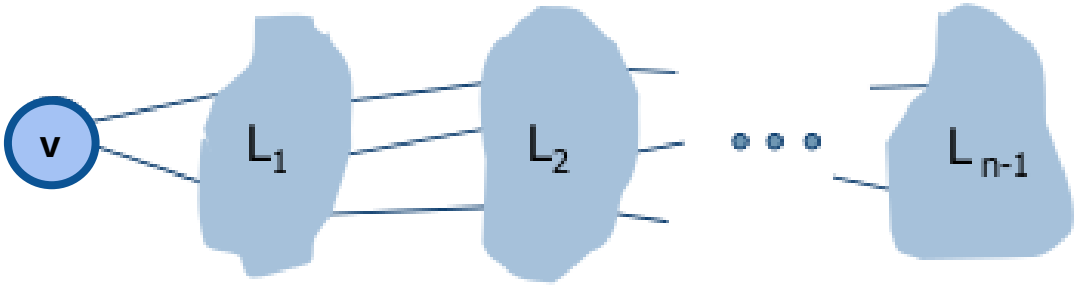
Recorrido DFS: 1 2 5 7 4 6 3

Recorrido en amplitud de Grafos
Breadth First Search (BFS)

Este recorrido es **equivalente al recorrido por niveles de un árbol.**

La estrategia es la siguiente:

- Partir de algún **vértice v**, **visitar v**, después **visitar cada uno de los vértices adyacentes a v**.
- **Repetir** el proceso para **cada nodo adyacente a v**, siguiendo el orden en que fueron visitados.
- **Si desde v no fueran alcanzables todos los nodos del grafo**: elegir un **nuevo vértice de partida no visitado**, y repetir el proceso hasta que se hayan **recorrido todos los vértices**.

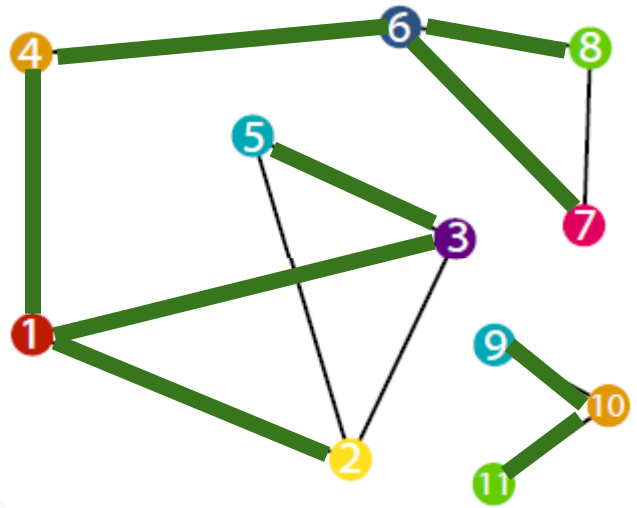


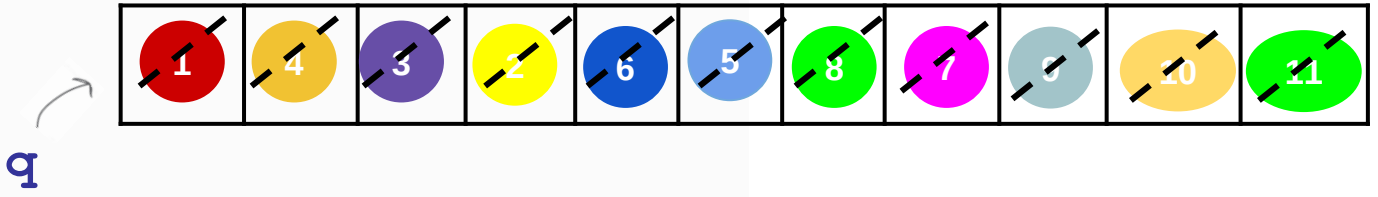
Exploración desde **v por niveles:**

L_0 = {v}
L_1 = Nodos adyacentes a L_0.
L_2 = Nodos adyacentes a los vértices de L_1 que no están en L_0 ni L_1.
L_{l+1} = Nodos adyacentes a los vértices de L_i que no ningún nivel anterior.

```
public class Recorridos {

 public void bfs(Grafo<T> grafo) {
   boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()];
   for (int i = 0; i < marca.length; i++) {
     if (!marca[i])
        this.bfs(i+1, grafo, marca);  //las listas empiezan en la pos 1
   }
 }

 private void bfs (int i, Grafo<T> grafo, boolean[] marca) {
  //siguiente diapo
 }
}
```

```
public class Recorridos {

 private void bfs(int i, Grafo<T> grafo, boolean[] marca) {
  ListaGenerica<Arista<T>> ady = null;
  ColaGenerica<Vertice<T>> q = new ColaGenerica<Vertice<T>>();
  q.encolar(grafo.listaDeVertices().elemento(i));
  marca[i] = true;
  while (!q.esVacia()) {
    Vertice<T> v = q.desencolar();
    System.out.println(v);
    ady = grafo.listaDeAdyacentes(v);
    ady.comenzar();
    while (!ady.fin()) {
       Arista<T> arista = ady.proximo();
       int j = arista.getDestino().posicion();
       if (!marca[j]) {
           Vertice<T> w = arista.getDestino();
           marca[j] = true;
           q.encolar(w);
       }
    }
  }
 }

}
```

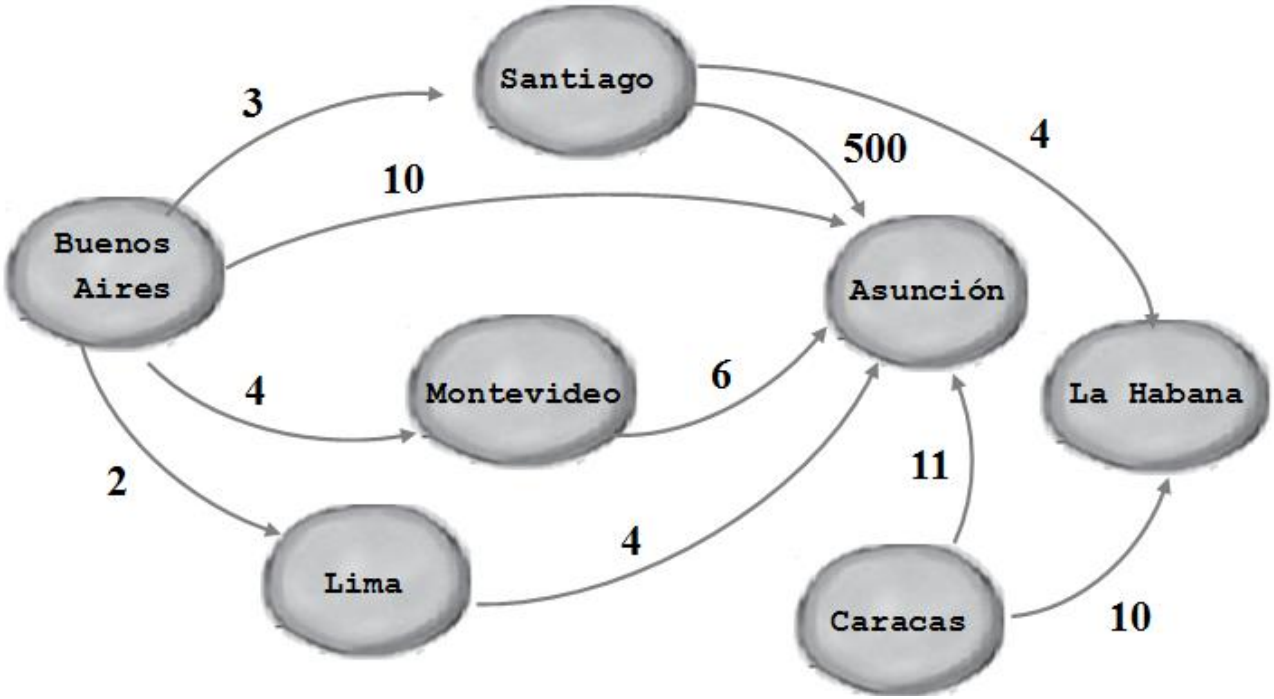




Ejercicio de Parcial

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.

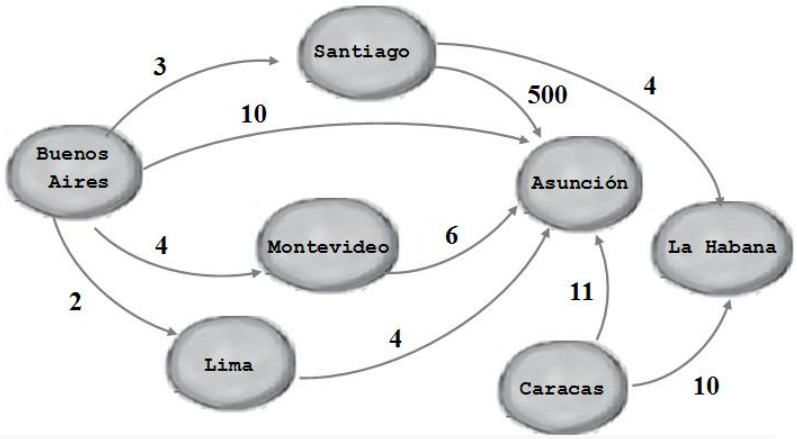


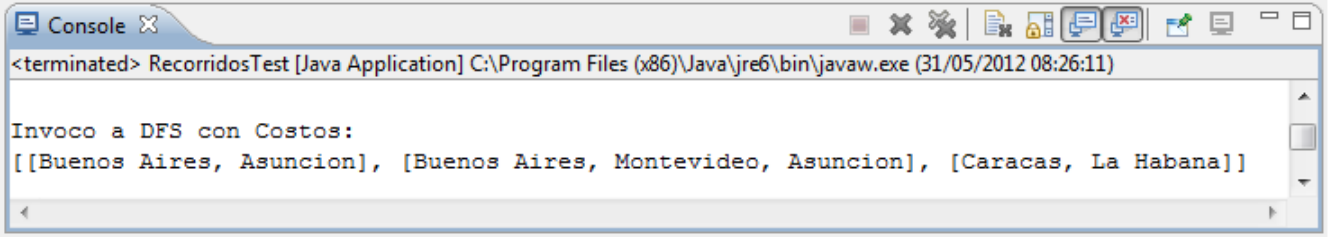
Ejercicio de Parcial (1/2)

```
public class Recorridos {
  public ListaGenerica<ListaGenerica<Vertice<T>>> dfsConCosto(Grafo<T> grafo){
   boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()+1];
   ListaGenerica<Vertice<T>> lis = null;
   ListaGenerica<ListaGenerica<Vertice<T>>> recorridos =
                              new ListaGenericaEnlazada<ListaGenericaEnlazada<Vertice<T>>>();
   int costo = 0;
   for(int i=1; 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) {
  ...
 }
}
```

Ejercicio de Parcial (2/2)

```
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.getVerticeDestino().getPosicion();
      if(!marca[j]){
         p = arista.getPeso();
         if ((costo+p) <= 10) {
            vDestino = arista.getVerticeDestino();
            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;
         }
      }
    }
 }
}
```



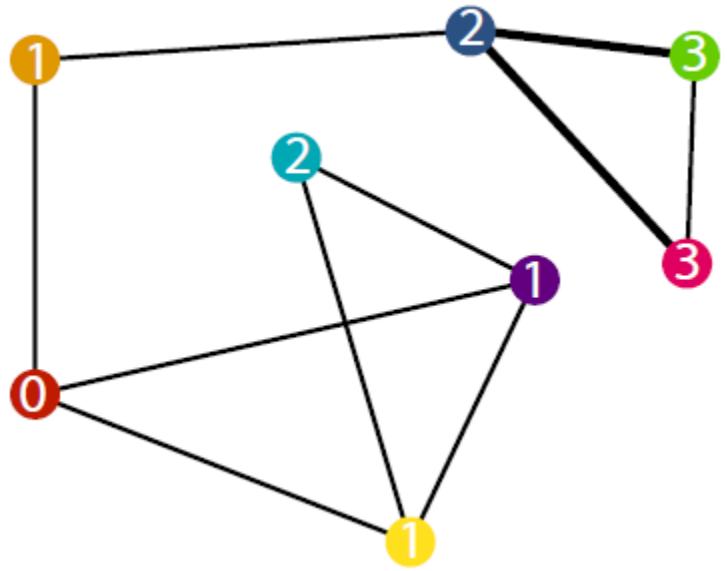


Ejemplo: Virus de computadoras

Un poderoso e inteligente **virus de computadora** infecta cualquier computadora en **1 minuto**, logrando infectar toda la red de una empresa con cientos de computadoras.

Dado un **grafo que representa las conexiones entre las computadoras de la empresa**, y una **computadora infectada**, escriba un programa en Java que permita **determinar el tiempo que demora el virus en infectar el resto de las computadoras.**

Todas las computadoras pueden ser infectadas, no todas las computadoras tienen conexión directa entre sí, y un mismo virus puede infectar un grupo de computadoras al mismo tiempo sin importar la cantidad.



```
public class BFSVirus {
 public int calcularTiempoInfeccion(Grafo<String> g, Vertice<String> inicial) {
  int n = g.listaDeVertices().tamanio();
  ColaGenerica<Vertice<String>> cola = new ColaGenerica<Vertice<String>>();
  int distancias[] = new int[n+1];    //no se usa la posicion 0
  int maxDist = 0; int nuevaDist = 0;
  for (int i = 0; i<=n; ++i) {
      distancias[i] = Integer.MAX_VALUE;
  }
  distancias[inicial.posicion()] = 0;
  cola.encolar(inicial);
  while (!cola.esVacia()) {
     Vertice<String> v = cola.desencolar();
     nuevaDist = distancias[v.posicion()] + 1;
     ListaGenerica<Arista<String>> adyacentes = v.obtenerAdyacentes();
     adyacentes.comenzar();
     while (!adyacentes.fin()) {
        Arista<String> a = adyacentes.proximo();
        Vertice<String> w = a.getDestino();
        int pos = w.posicion();
        if (distancias[pos]== Integer.MAX_VALUE) {
            distancias[pos] = nuevaDist;
            if (nuevaDist > maxDist)
               maxDist = nuevaDist;
            cola.encolar(w);
        }
     }
    }
    return maxDist;
 }}
```