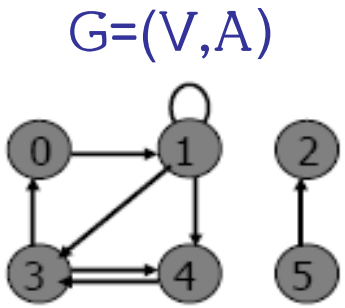
Grafos
Representaciones

(1) Matriz de adyacencia

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

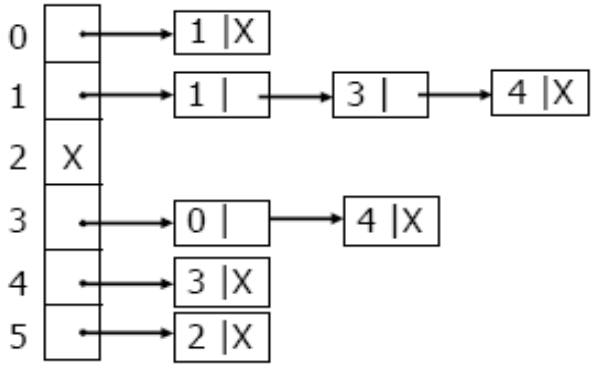
G[u,v] = true si (u, v) Є A
G[u,v] = false si (u, v) Є A

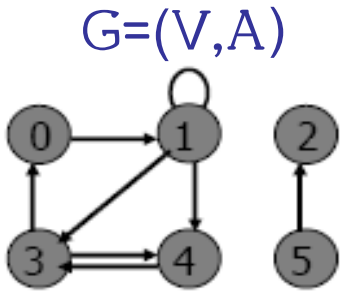
	0	1	2	3	4	5
0	false	true	false	false	false	false
1	false	true	false	true	true	false
2	false	false	false	false	false	false
3	true	false	false	false	true	false
4	false	false	false	true	false	false
5	false	false	true	false	false	false



(2) Lista de adyacencia

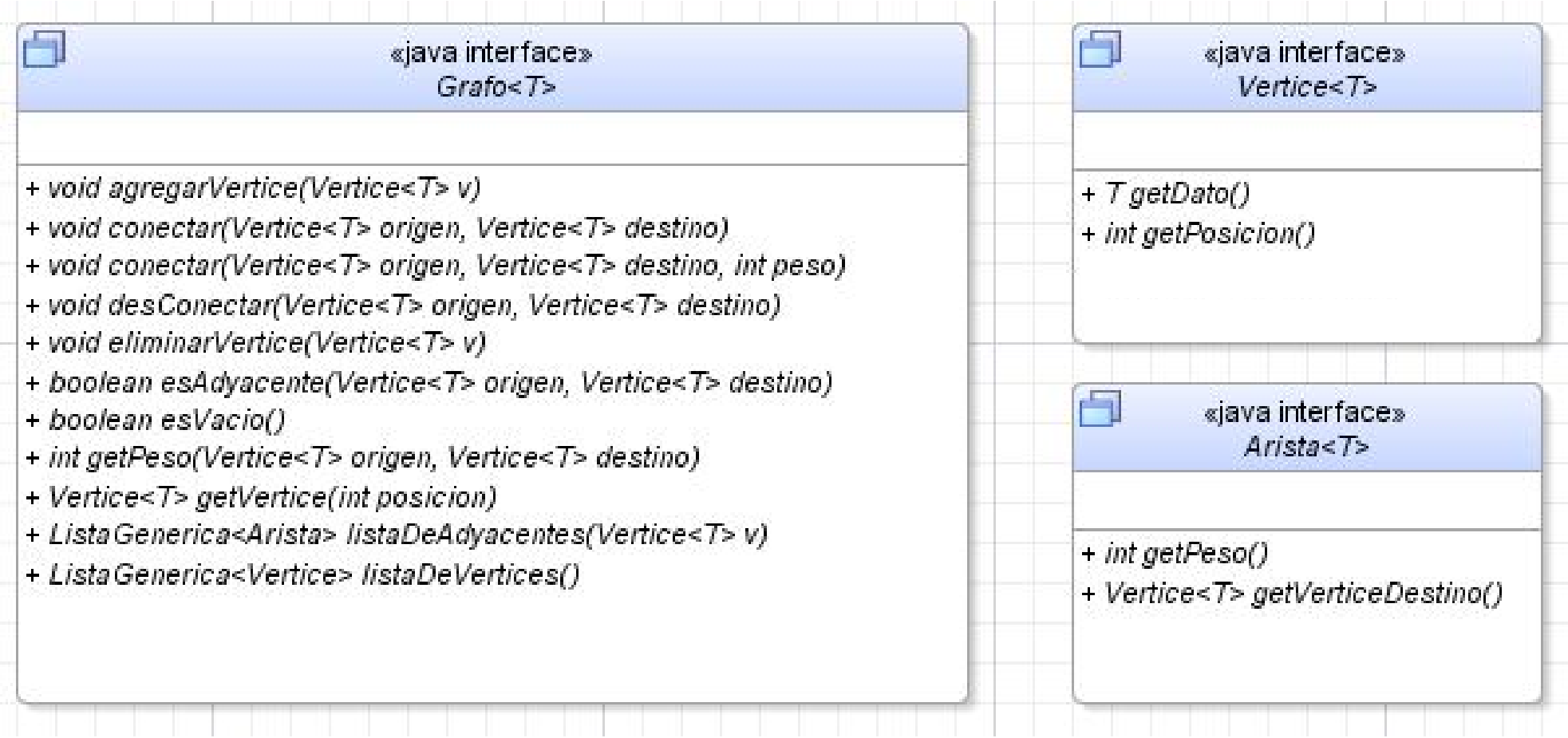
Un grafo G=(V,A) se representa como un arreglo/lista de |V| de vértices donde:



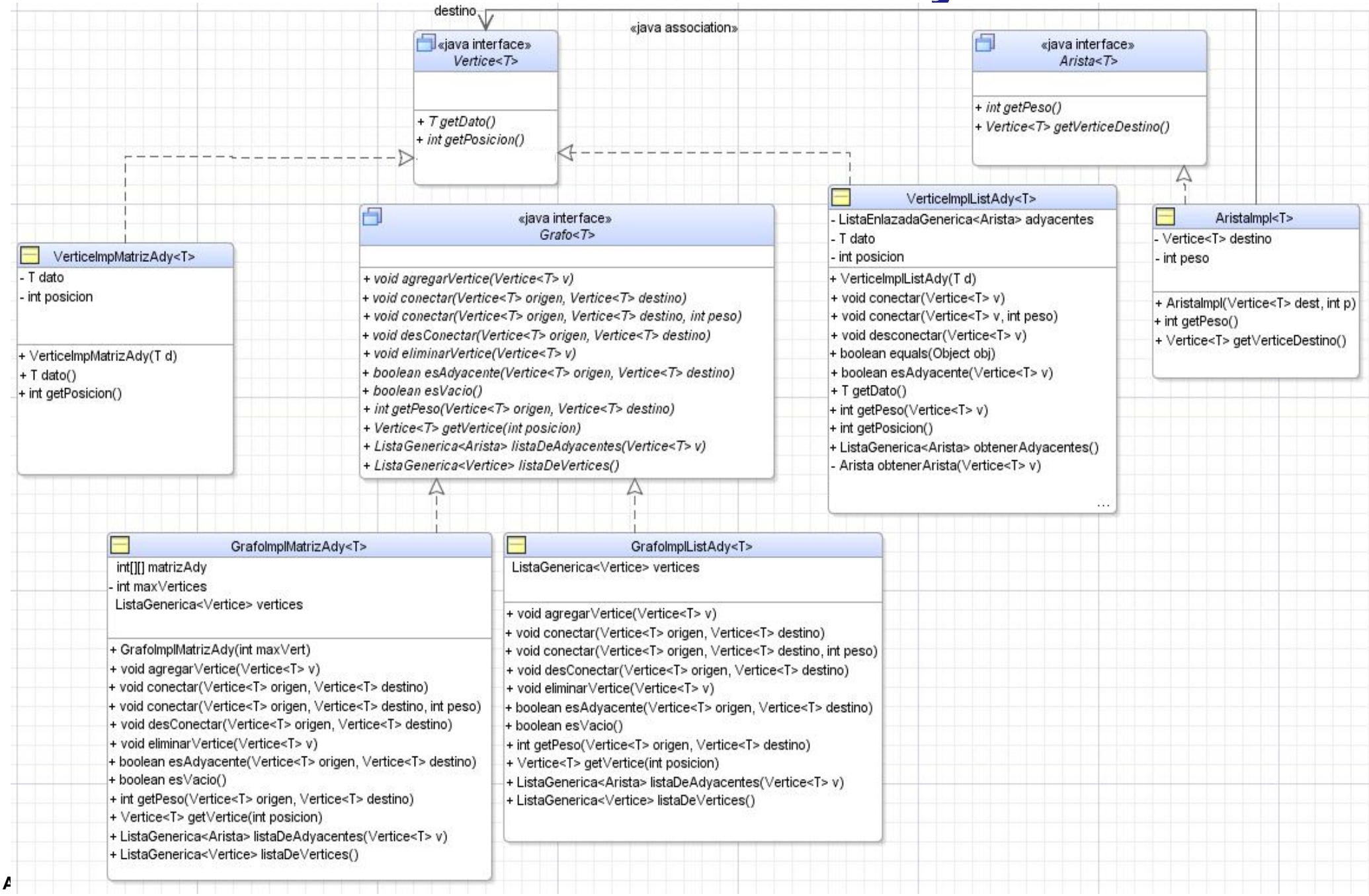


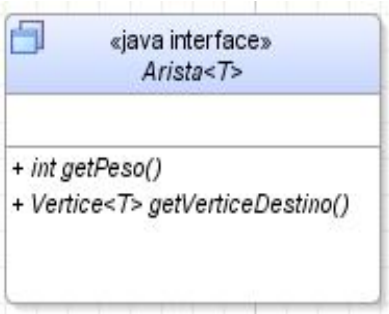
Grafos

La interfaces para definir Grafos



Grafos - La interfaces y clases

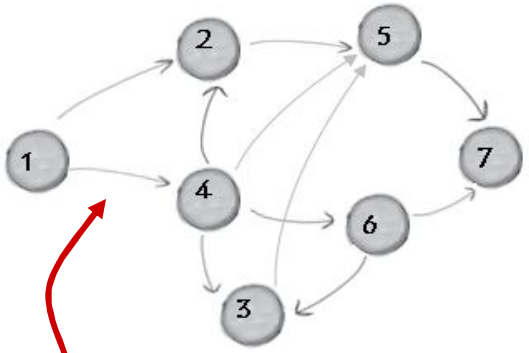




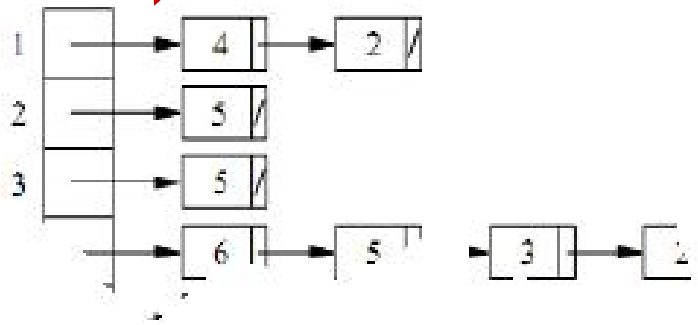
Grafos

La clase que implementa a la interface Arista

```
package estructuras.grafo;

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;
 }

 @Override
 public Vertice<T> getVerticeDestino() {
   return destino;
 }

 @Override
 public int getPeso() {
   return peso;
 }
}
```



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

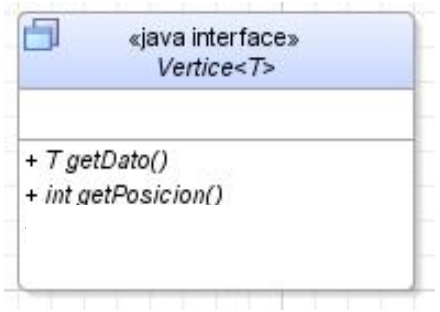


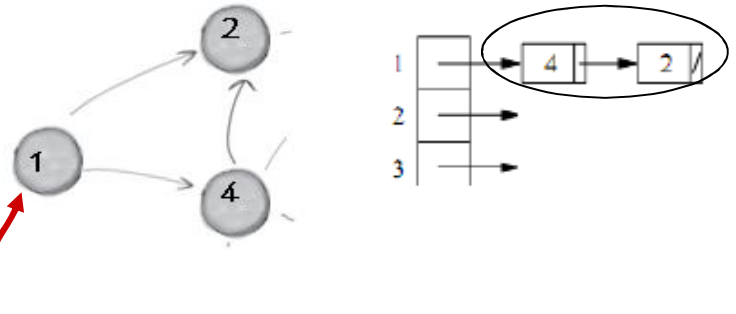
Podría llamarse **`AritaImplListaAdy`** porque que solo se usa para Lista de Adyacencias

Grafos

Clase que implementa la interface `Vertice` (con Listas de Adyacencia)

```
package estructuras.grafo;
public class VerticeImplListAdy<T> implements Vertice<T> {
 private T dato;
 private int posicion;
 private ListaEnlazadaGenerica<Arista<T>> adyacentes;

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

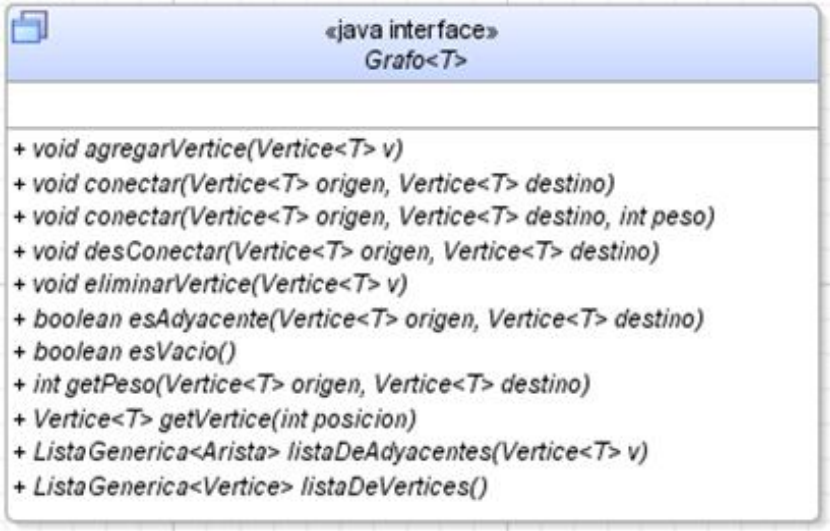


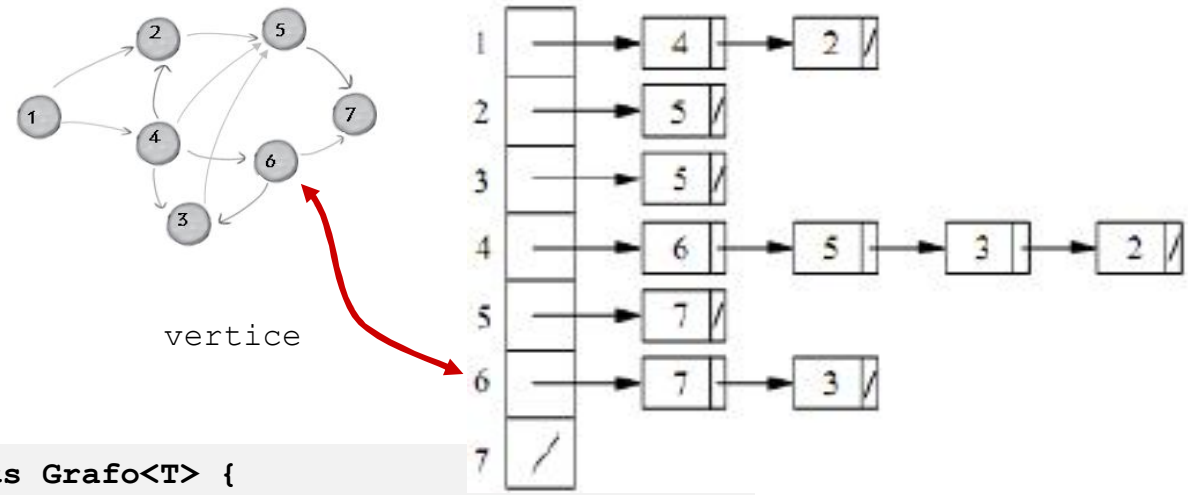


Vértice: tiene un dato y una lista de adyacentes. En realidad se tiene una lista de aristas, donde cada una tiene un vértice destino.

Grafos

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






```
public class GrafoImplListAdy<T> implements Grafo<T> {
ListaGenerica<Vertice<T>> vertices = new ListaEnlazadaGenerica<Vertice<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);
 }
 . . .
}
```

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) ∈ A, si v no ha sido visitado, repetir recursivamente (3) y (4) para v.

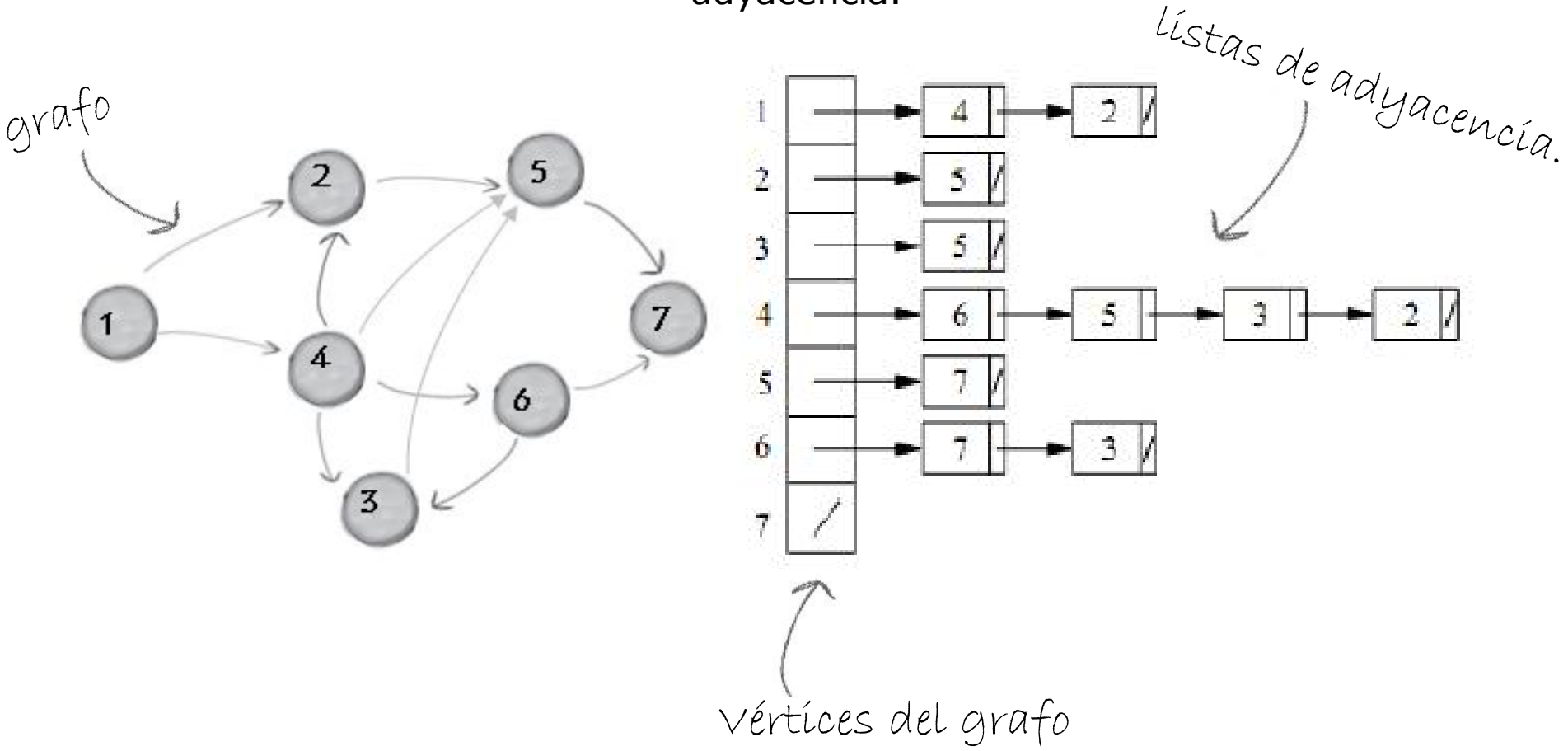
¿Cuándo finaliza el recorrido?

Finalizar cuando se hayan visitado 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 v no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

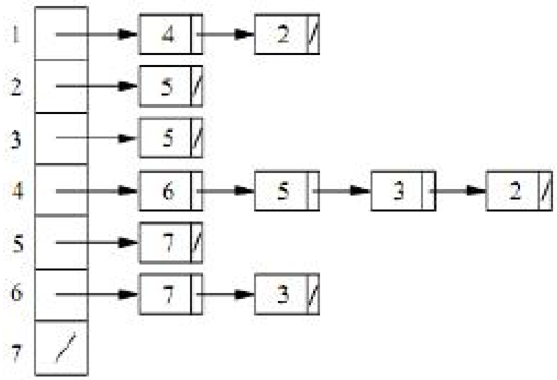
Grafos
DFS (Depth First Search)

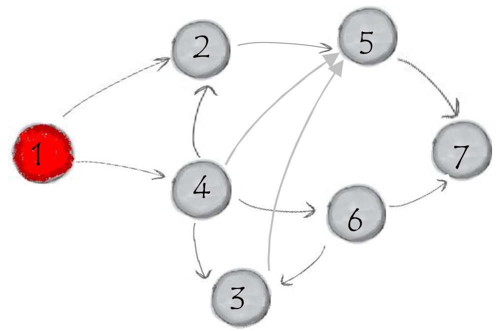
El recorrido del DFS depende del orden en que aparecen los vértices en las listas de adyacencia.

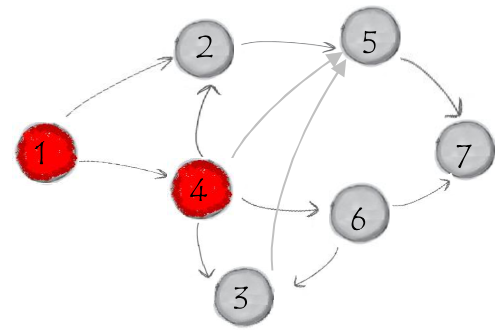


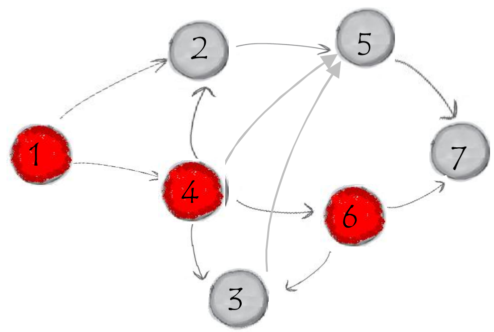
Grafos

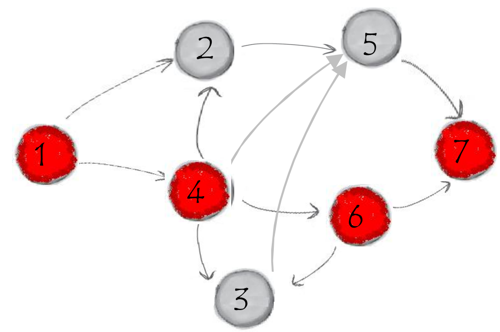
DFS (Depth First Search)

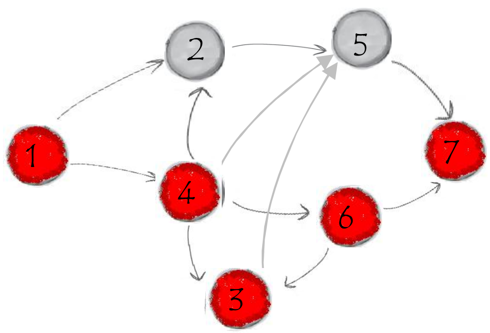


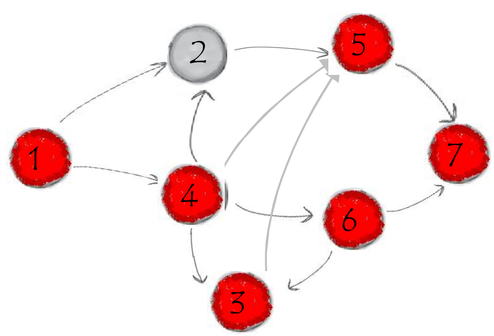


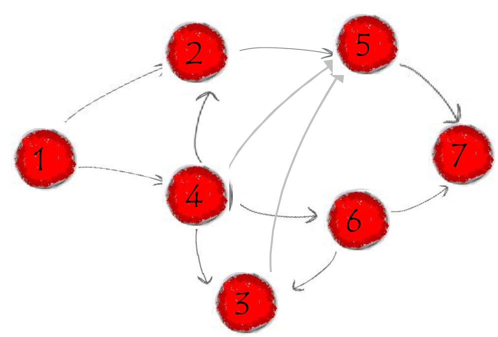












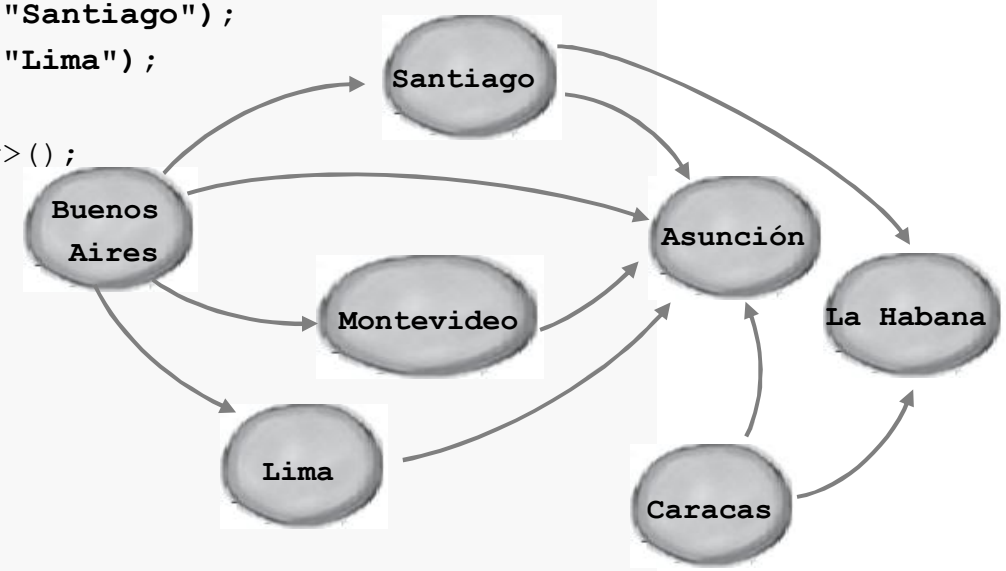
Grafos
DFS (Depth First Search)

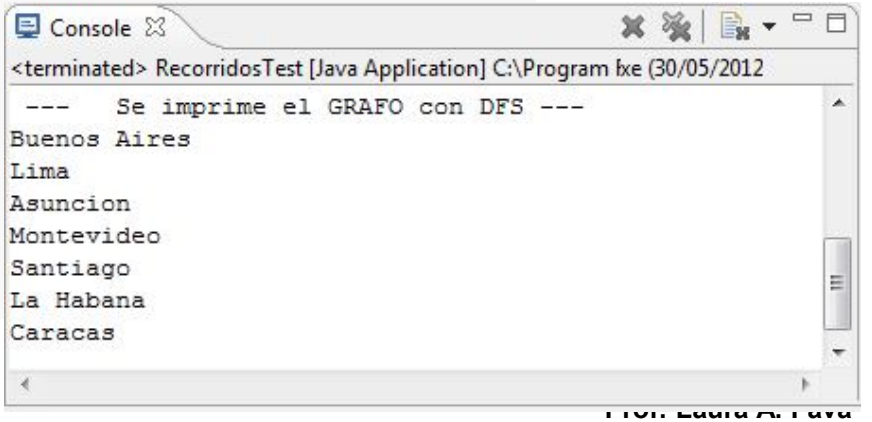
```
public class Recorridos<T> {

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

 private void dfs(int i,Grafo<T> grafo, boolean[] marca){
    marca[i] = true;
    Vertice<T> v = grafo.listaDeVertices().elemento(i);
    System.out.println(v);
    ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);
    ady.comenzar();
    while(!ady.fin()){
       int j = ady.proximo().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");
   . . .
  Grafo<String> ciudades = new GrafoImplListAdy<String>();
  ciudades.agregarVertice(v1);
  ciudades.agregarVertice(v2);
  ciudades.agregarVertice(v3);
   . . .
  ciudades.conectar(v1, v2);
  ciudades.conectar(v1, v3);
  ciudades.conectar(v1, v4);
  ciudades.conectar(v1, v5);
  ciudades.conectar(v3, v5);
   . . .
  Recorridos<String> r = new Recorridos<String>();
  System.out.println("--- Se imprime el GRAFO con DFS ---");
  r.dfs(ciudades);
 }
}
```





Grafos
DFS (Depth First Search)

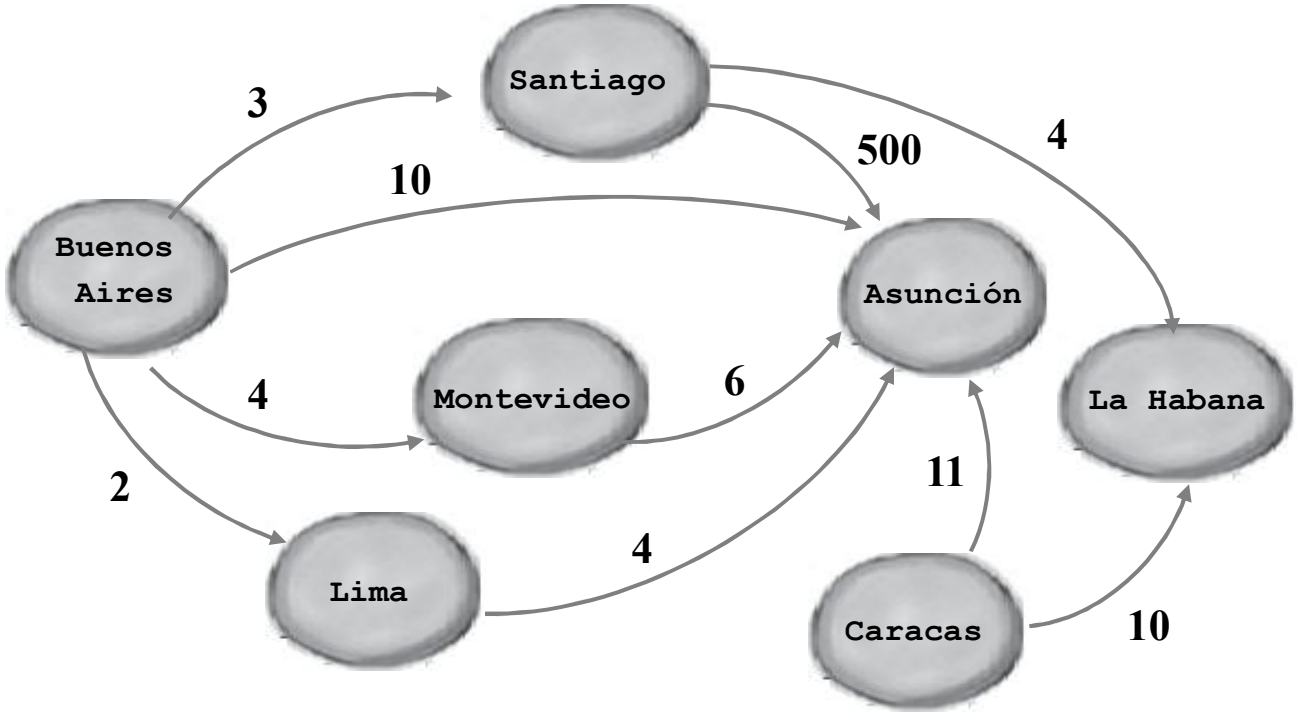
```
public class Recorridos<T> {
 public ListaEnlazadaGenerica<Vertice<T>> dfs(Grafo<T> grafo){
   boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()];
   ListaEnlazadaGenerica<Vertice<T>> lis = new ListaEnlazadaGenerica<Vertice<T>>();
   for(int i=0; i<grafo.listaDeVertices().tamanio();i++){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

DFS que guarda vértices visitados en una lista

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.

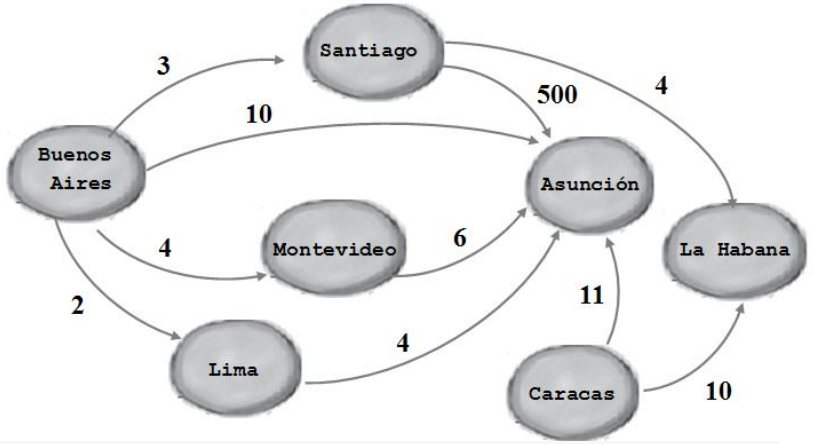


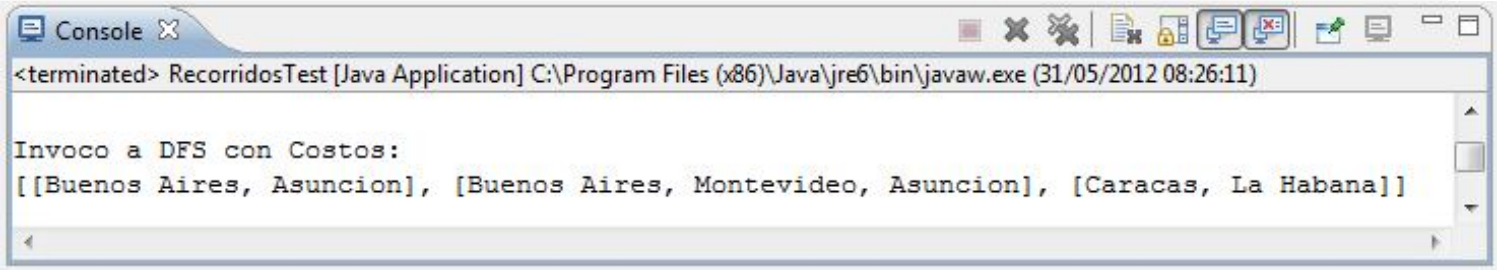
Ejercicio de Parcial (1/3)

```
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) {
  ...
 }
}
```

Ejercicio de Parcial (2/3)

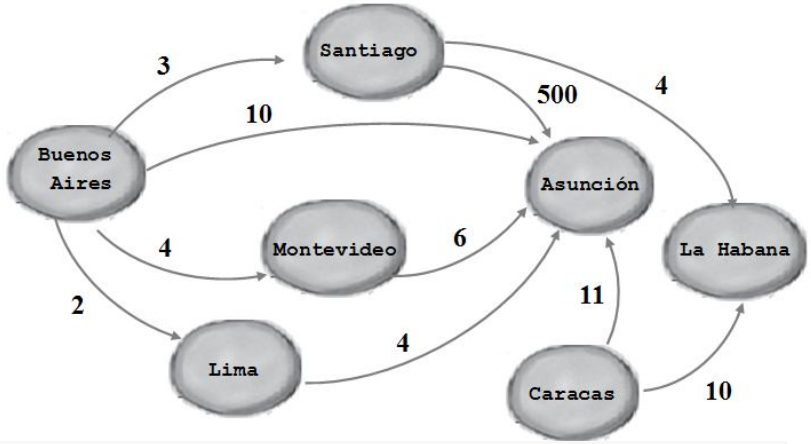
```
public class Recorridos {
 private void dfsConCosto(int i, Grafo<T> grafo, ListaGenerica<Vertice<T>> lis,
        boolean[] marca, int costo, ListaGenerica<ListaGenerica<Vertice<T>>> recorridos) {
  Vertice<T> vDestino = null; int p=0,j=0;
  Vertice<T> v = grafo.listaDeVertices().elemento(i);
  ListaGenerica<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();
      vDestino = arista.getVerticeDestino();
      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
      }
      this.dfsConCosto(j, grafo, lis, marca, costo, recorridos);
      costo=costo-p;
      lis.eliminar(vDestino);
      marca[j]= false;
    }
  }

 }
}
```





Ejercicio de Parcial (3/3)

```
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;
          }
       }
     }
 }
}
```



Console

<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]]
```

Grafos
BFS (Breath First Search)

Este algoritmo es la generalización del 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.

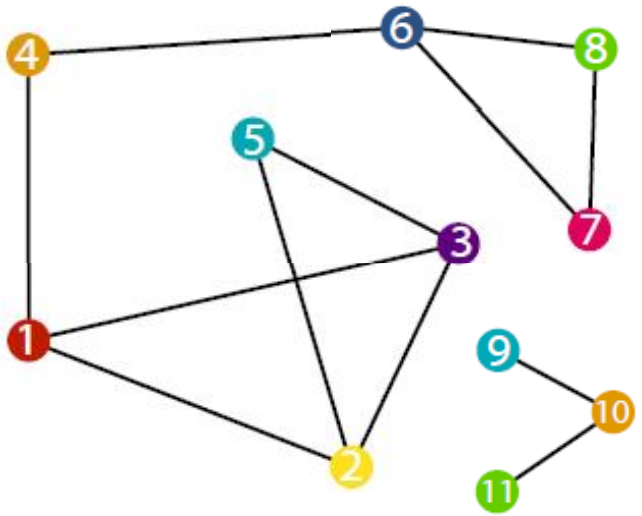
Si desde v no fueran alcanzables todos los nodos del grafo: elegir un nuevo vértice de partida no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

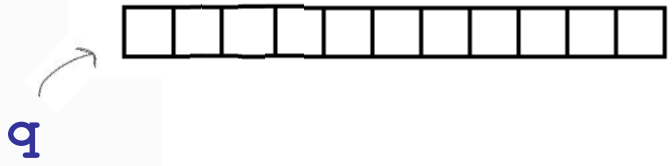
```
public class Recorridos {
 public void bfs(Grafo<T> grafo) {
   boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()];
   for (int i = 0; i < marca.length; i++) {
     if (!marca[i])
        this.bfs(i+1, grafo, marca);  //las listas empiezan en la pos 1
   }
 }
 private void bfs (int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

Grafos

BFS (Breath First Search)

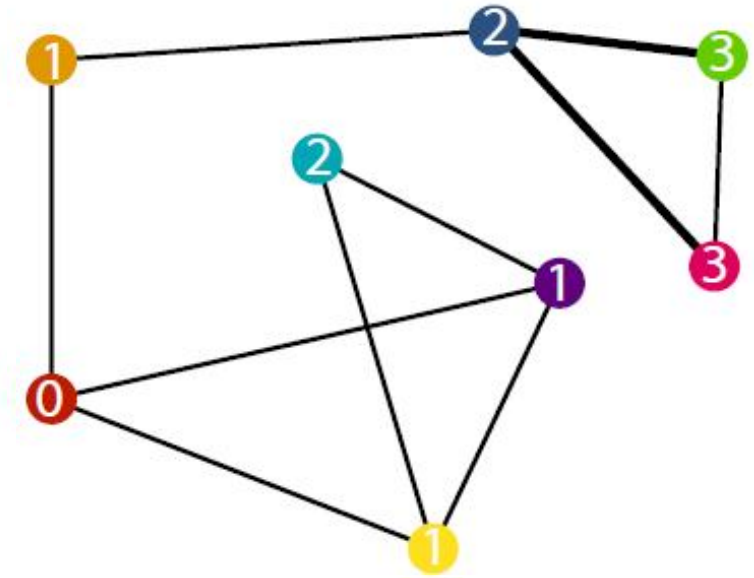
```
public class Recorridos {
  . . .
 private void bfs(int i, Grafo<T> grafo, boolean[] marca) {
  ListaGenerica<Arista<T>> ady = null;
  ColaGenerica<Vertice<T>> q = new ColaGenerica<Vertice<T>>();
  q.encolar(grafo.listaDeVertices().elemento(i));
  marca[i] = true;
  while (!q.esVacia()) {
    Vertice<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.getVerticeDestino().getPosicion();
       if (!marca[j]) {
           Vertice<T> w = arista.getVerticeDestino();
           marca[j] = true;
           q.encolar(w);
       }
    }
   }
  }
}
```





Ejercicio de Parcial

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 ya infectada, escriba un programa en Java que permita determinar el tiempo que demora el virus en infectar el resto de las computadoras. Asuma que todas las computadoras pueden ser infectadas, no todas las computadoras tienen conexión directa entre si, y un mismo virus puede infectar un grupo de computadoras al mismo tiempo sin importar la cantidad.



Ejercicio de Parcial

```
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 usaria la posicion 0
  int maxDist = 0; int nuevaDist = 0;
  for (int i = 0; i<=n; ++i) {
      distancias[i] = Integer.MAX_VALUE;
  }
  distancias[inicial.getPosicion()] = 0;
  cola.encolar(inicial);
  while (!cola.esVacia()) {
     Vertice<String> v = cola.desencolar();
     nuevaDist = distancias[v.getPosicion()] + 1;
     ListaGenerica<Arista<String>> adyacentes = v.obtenerAdyacentes();
     adyacentes.comenzar();
     while (!adyacentes.fin()) {
        Arista<String> a = adyacentes.proximo();
        Vertice<String> w = a.getVerticeDestino();
        int pos = w.getPosicion();
        if (distancias[pos]== Integer.MAX_VALUE) {
            distancias[pos] = nuevaDist;
            if (nuevaDist > maxDist)
               maxDist = nuevaDist;
            cola.encolar(w);
        }
     }
    }
    return maxDist;
 }}
```