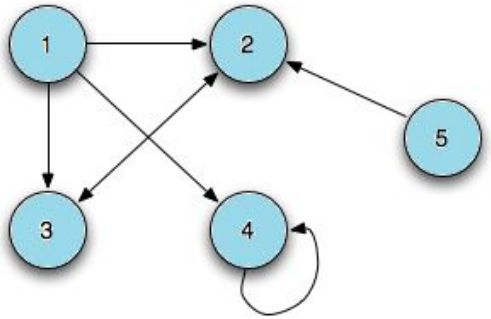
Grafos en JAVA
Terminología

Un grafo es un conjunto de nodos que mantienen relaciones entre ellos. Podemos definir a un grafo como un par de conjuntos finitos G=(V,A) donde V es el conjunto finito de vértices y A es el conjunto finito de Aristas.

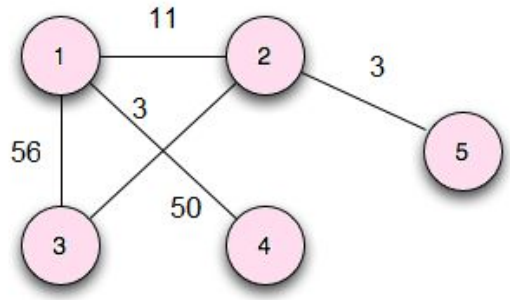
Por lo general, los **vértices** son nodos de procesamiento o estructuras que contienen algún tipo de información mientras que las **aristas** determinan las relaciones entre nodos. Las artistas también pueden tener algún tipo de información asociada (distancia, peso, etc.) en cuyo caso es un **grafo pesado**.

En cualquiera de los dos grafos, las relaciones las determinan las **aristas**.



Si se utilizan flechas para conectar los nodos decimos que el **grafo es dirigido** (también llamado **digrafo**).

Grafo no pesado



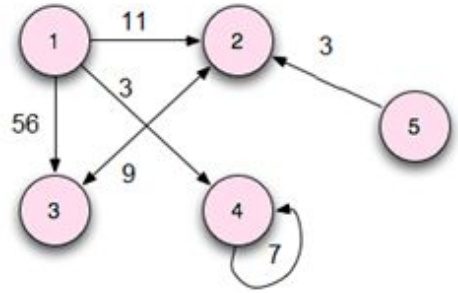
Si la conexión entre los vértices no tiene dirección, el **grafo es no dirigido**.

Grafo pesado

Grafos en JAVA
Representaciones

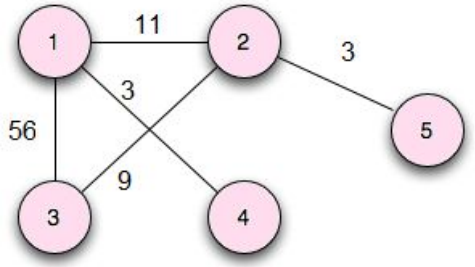
Los métodos más comunes para representar grafos son matriz de adyacencias y lista de adyacencias.

- **Matriz de adyacencias**: el grafo se representa como una matriz de |V| x |V|, con valores enteros (o de otro tipo de dato).



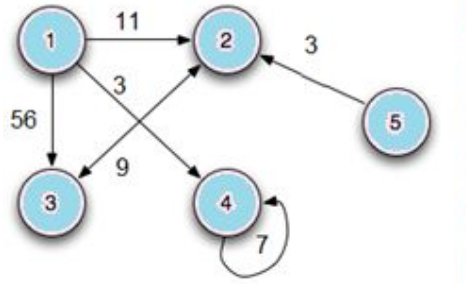
	1	2	3	4	5
1	0	11	56	3	0
2	0	0	9	0	0
3	0	9	0	0	0
4	0	0	0	7	0
5	0	3	0	0	0

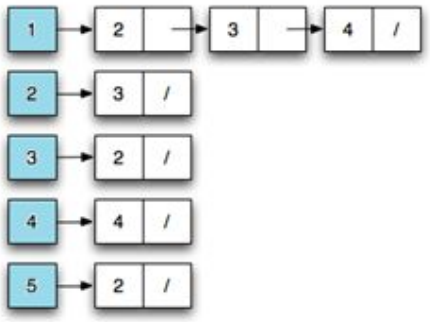
Matriz simétrica

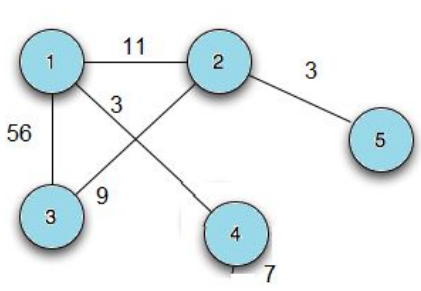


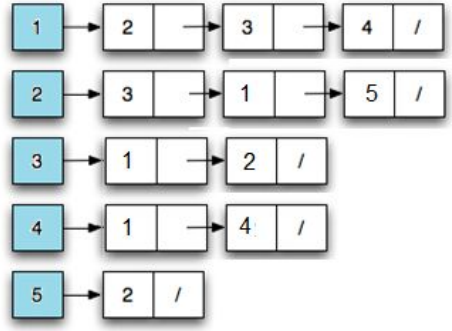
	1	2	3	4	5
1	0	11	56	3	0
2	11	0	9	0	3
3	56	9	0	0	0
4	3	0	0	0	0
5	0	3	0	0	0

- **Lista de adyacencias**: el grafo G=(V,A) se representa como un arreglo/lista de |V| de vértices.

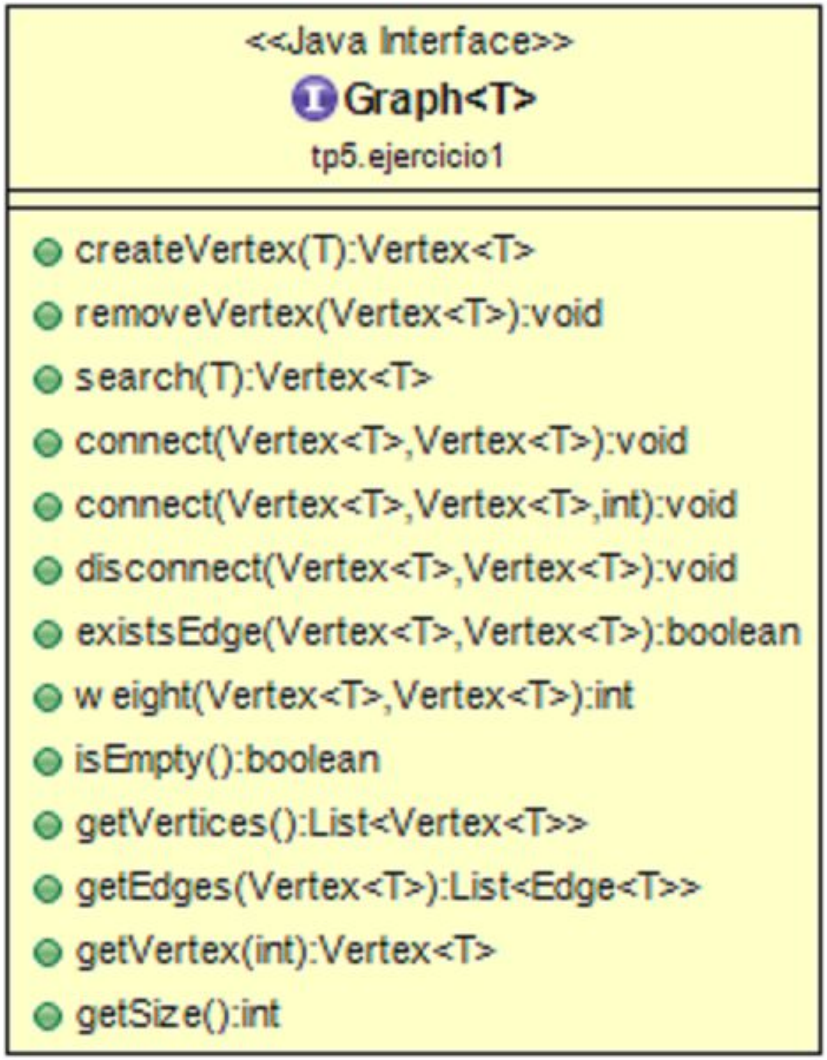


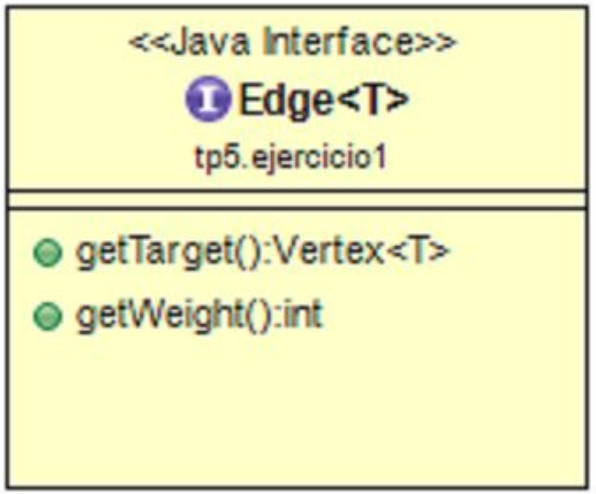


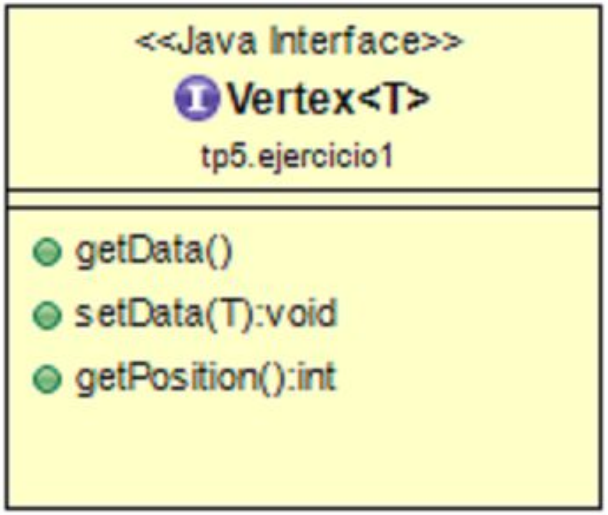




Grafos en JAVA
Interfaces para definir Grafos

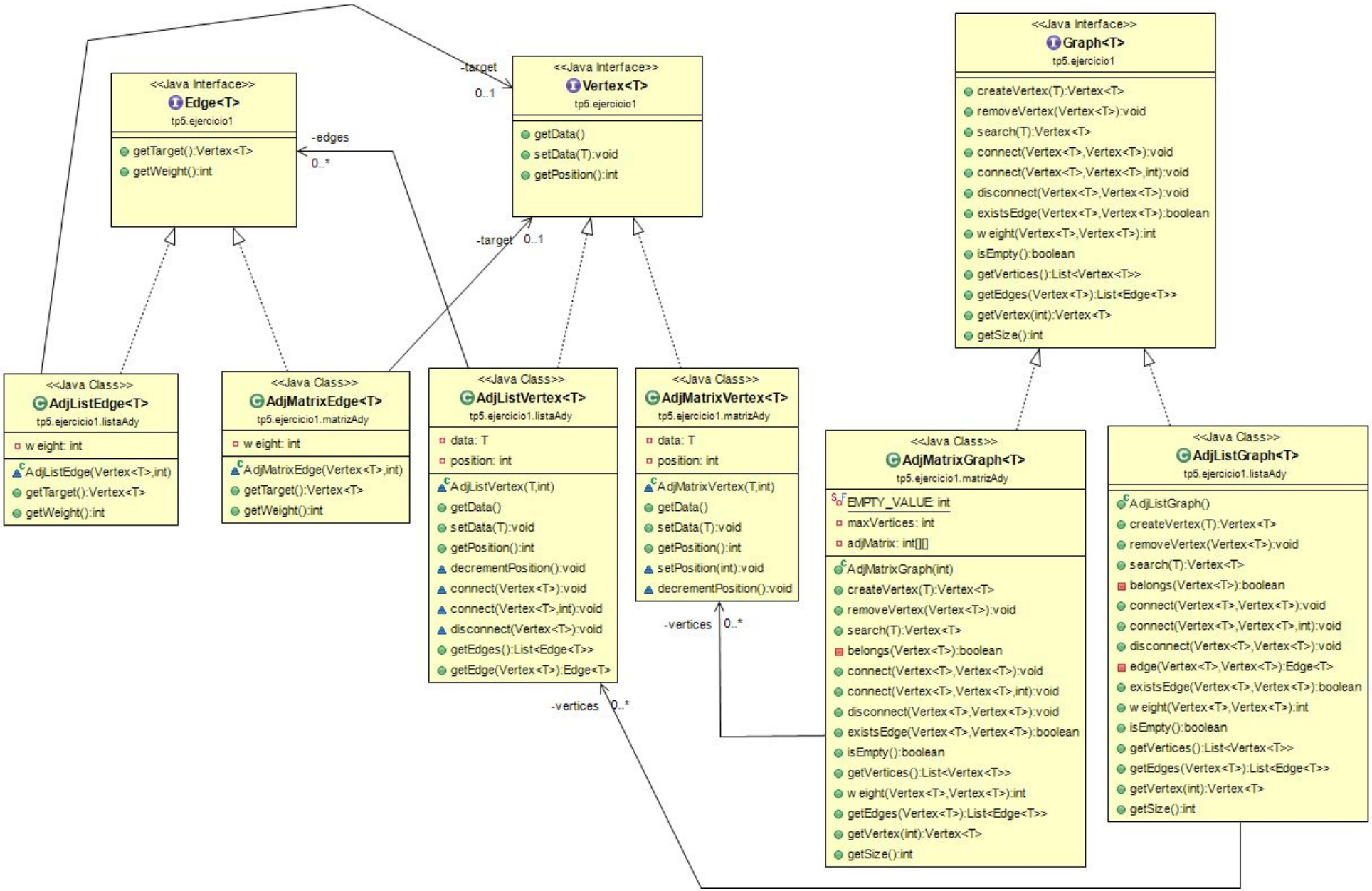






Profesores: Laura Fava - Pablo Iuliano

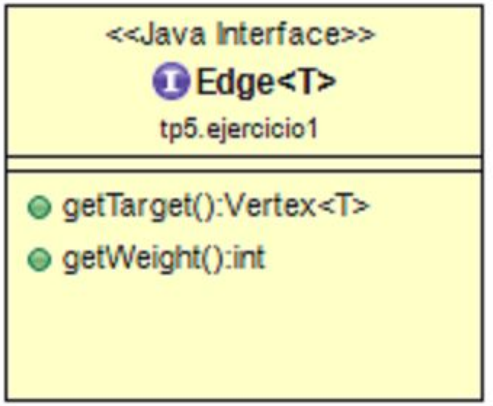
Grafos - La interfaces y clases



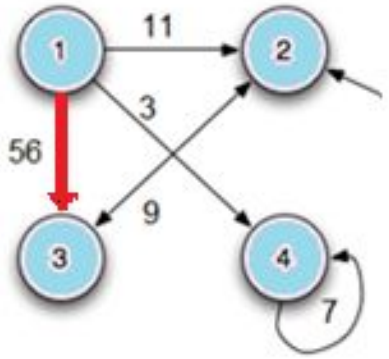
Grafos en JAVA

La clase que implementa a la interface **Edge**

```
package tp5.ejercicio1;

public class AdjListEdge<T> implements Edge<T> {
    private Vertex<T> target;
    private int weight;

    public AdjListEdge(Vertex<T> target, int weight){
        this.target = target;
        this.weight = weight;
    }

    @Override
    public Vertex<T> getTarget() {
        return target;
    }

    @Override
    public int getWeight() {
        return weight;
    }
}
```



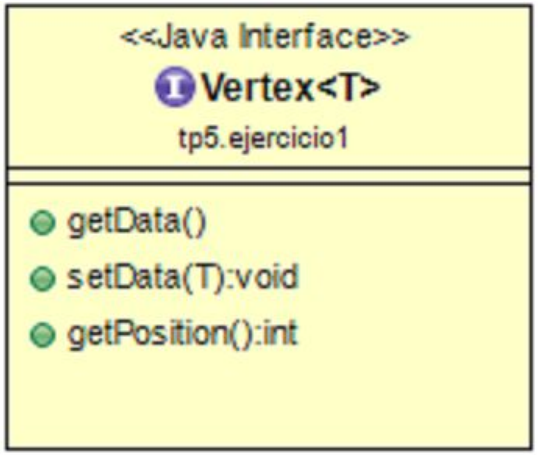
Una arista siempre tiene el destino y podría tener un peso.



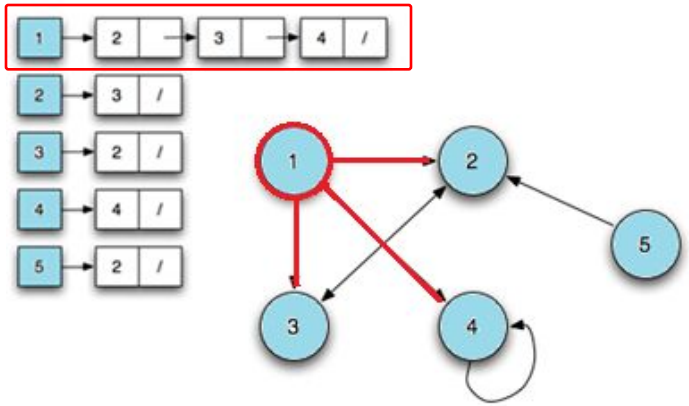
Grafos con Lista de Adyacencias

Clase que implementa la interface **`Vertex`**

```
public class AdjListVertex<T> implements Vertex<T> {
	private T data;
	private int position;
	private List<Edge<T>> edges;

	@Override
	public T getData() {
		return this.data;
	}

	@Override
	public void setData(T data) {
		this.data = data;
	}

	@Override
	public int getPosition() {
		return this.position;
	}

	public List<Edge<T>> getEdges() {
		return this.edges;
	}

	public Edge<T> getEdge(Vertex<T> destination) {
		for (Edge<T> edge : this.edges) {
			if (edge.getTarget() == destination) {
				return edge;
			}
		}
		return null;
	}
	. . .
}
```



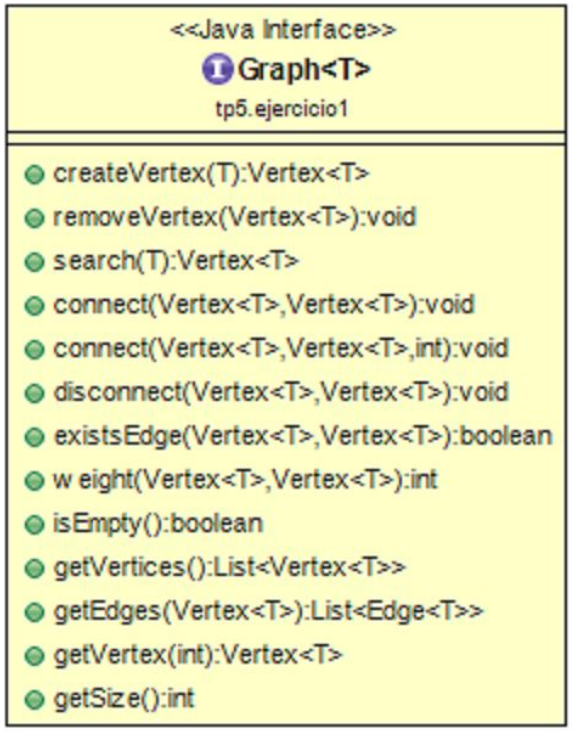
Un vértice tiene un dato y una lista de adyacentes. La lista es de aristas, donde cada una tiene un vértice destino.

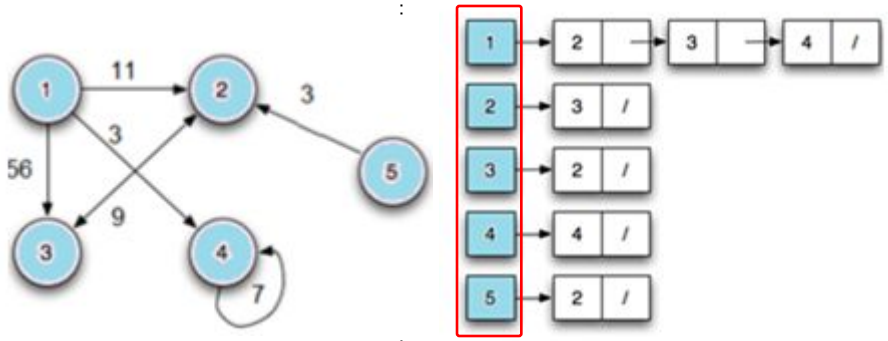


Grafos con Lista de Adyacencias

Clase que implementa la interface `Graph`

```
public class AdjListGraph<T> implements Graph<T> {

    private List<AdjListVertex<T>> vertices;

    public AdjListGraph() {
      this.vertices = new ArrayList<>();
    }

      public Vertex<T> createVertex(T data) {
            int newPos = this.vertices.size();
            AdjListVertex<T> vertex = new AdjListVertex<>(data, newPos);
            this.vertices.add(vertex);
            return vertex;
      }

      public void removeVertex(Vertex<T> vertex) {
            int position = vertex.getPosition();
            if (this.vertices.get(position) != vertex) {
                  // el vértice no corresponde al grafo
                  return;
            }
            this.vertices.remove(position);
            for (;position < this.vertices.size(); position++) {
                  this.vertices.get(position).decrementPosition();
            }
            for (Vertex<T> other : this.vertices) {
                  this.disconnect(other, vertex);
            }
      }

      public Vertex<T> search(T data) {
            for (Vertex<T> vertex : this.vertices) {
                  if (vertex.getData().equals(data)) {
                        return vertex;
                  }
            }
            return null;
      }
```



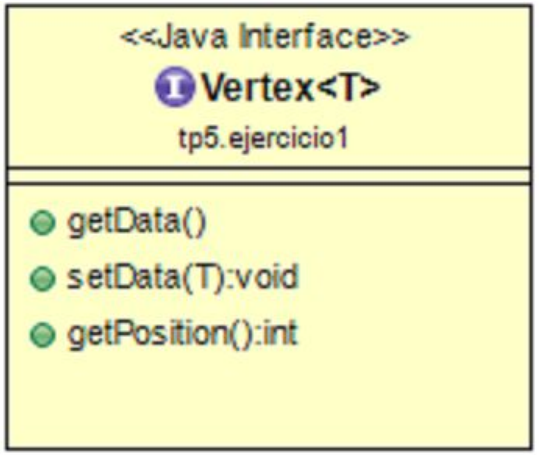


Un Grafo tiene una lista de vértices

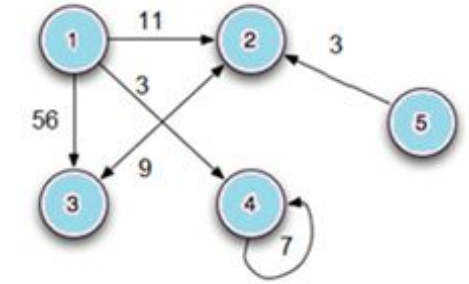
Grafos con Matriz de Adyacencias

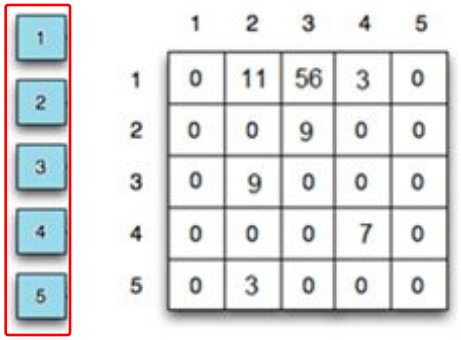
Clase que implementa la interface `Vertex`

```
public class AdjMatrixVertex<T> implements Vertex<T> {
      private T data;
      private int position;

      AdjMatrixVertex(T data, int position) {
            this.data = data;
            this.position = position;
      }

      public T getData() {
            return this.data;
      }

      public void setData(T data) {
            this.data = data;
      }

      public int getPosition() {
            return position;
      }

      void setPosition(int position) {
            this.position = position;
      }

      void decrementPosition() {
            this.position--;
      }
}
```



Un vértice tiene un dato y la posición dentro de la lista.



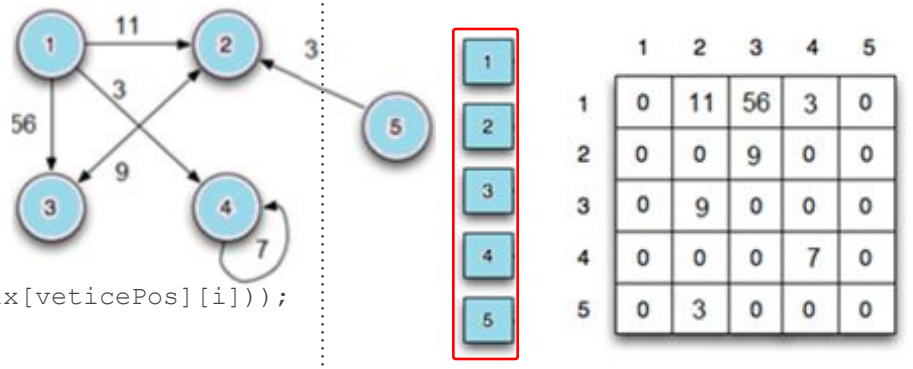


	1	2	3	4	5
1	0	11	56	3	0
2	0	0	9	0	0
3	0	9	0	0	0
4	0	0	0	7	0
5	0	3	0	0	0

Grafos con Matriz de Adyacencias

Clase que implementa la interface `Graph`

```
public class AdjMatrixGraph<T> implements Graph<T> {
    private static final int EMPTY_VALUE = 0;
    private int maxVertices;
    private List<AdjMatrixVertex<T>> vertices;
    private int[][] adjMatrix;

    public AdjMatrixGraph(int maxVert) {
        maxVertices = maxVert;
        vertices = new ArrayList<>();
        adjMatrix = new int[maxVertices][maxVertices];
        for (int i = 1; i < maxVertices; i++) {
            for (int j = 0; j < maxVertices; j++) {
            adjMatrix[i][j] = EMPTY_VALUE;
            }
        }
    }

    public Vertex<T> createVertex(T data) {
      if (vertices.size() == maxVertices) {
            // se llego al máximo
            return null;
      }
      AdjMatrixVertex<T> vertice = new AdjMatrixVertex<>(data, vertices.size());
      vertices.add(vertice);
      return vertice;
    }

    public List<Edge<T>> getEdges(Vertex<T> v) {
        List<Edge<T>> ady = new ArrayList<Edge<T>>();
        int veticePos = v.getPosition();
        for (int i = 0; i < vertices.size(); i++) {
            if (adjMatrix[veticePos][i] != EMPTY_VALUE) {
                ady.add(new AdjMatrixEdge<T>(vertices.get(i), adjMatrix[veticePos][i]));
            }
        }
        return ady;
    }
```



Un `Grafo` tiene una lista de `vértices` y la matriz

Agenda - Grafos

Recorridos

- en profundidad: **DFS** (Depth First Search)
- en amplitud: **BFS** (Breath First Search)

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.

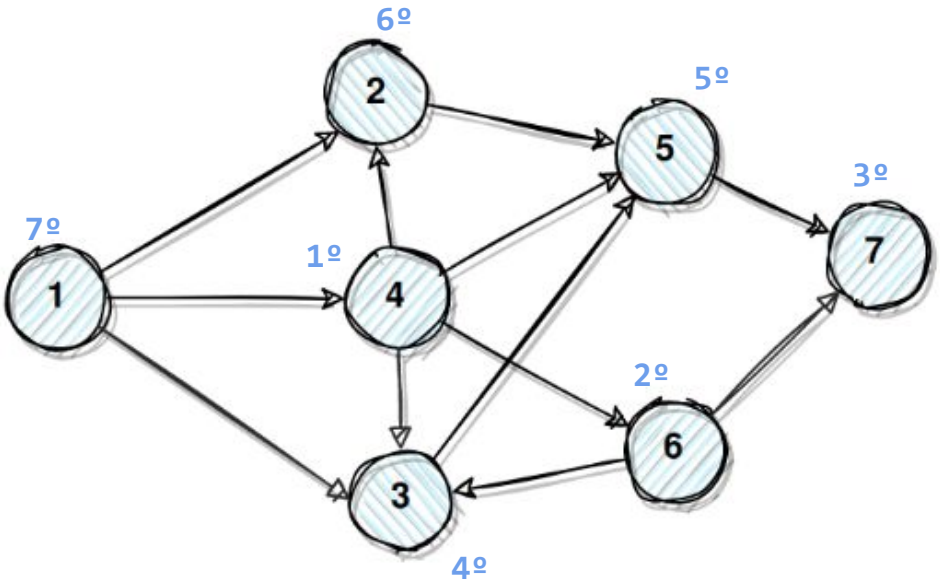
¿Cuándo finaliza el recorrido?

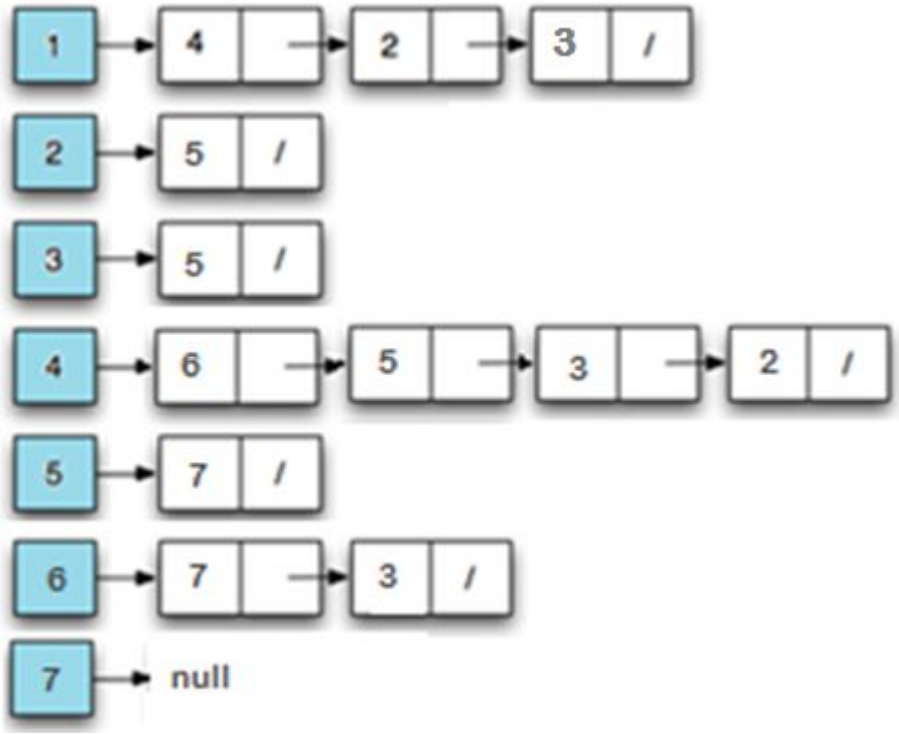
Finalizar cuando se hayan visitado todos los nodos alcanzables desde u.

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.





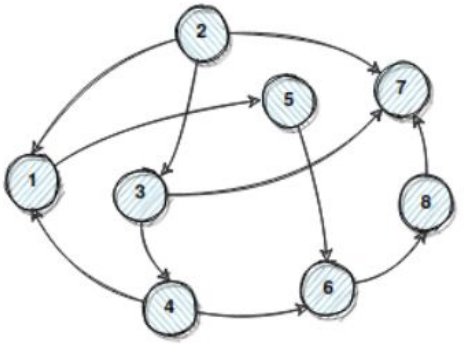
Grafos

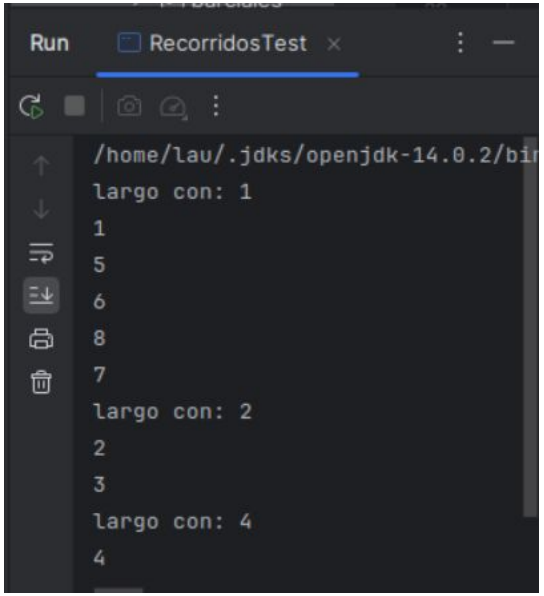
DFS (Depth First Search)

```
public class Recorridos<T> {
  public void dfs(Graph<T> grafo) {
    boolean[] marca = new boolean[grafo.getSize()];
    for (int i = 0; i < grafo.getSize(); i++) {
       if (!marca[i]) {
         System.out.println("largo con: "+grafo.getVertex(i).getData());
         dfs(i, grafo, marca);
       }
     }
   }
   private  void dfs(int i, Graph<T> grafo, boolean[] marca) {
     marca[i] = true;
     Vertex<T> v = grafo.getVertex(i);
     System.out.println(v);
     List<Edge<T>> adyacentes = grafo.getEdges(v);  //adyacentes
     for (Edge<T> e: adyacentes){
          int j = e.getTarget().getPosition();
          if (!marca[j]) {
               dfs(j, grafo, marca);
          }
     }
   }
}
```

equivalente

```
Iterator<Edge<T>> it = adyacentes.iterator();
while (it.hasNext()) {
     int j = it.next().getTarget().getPosition();
     if (!marca[j]){
        dfs(j, grafo, marca);
     }
}
```

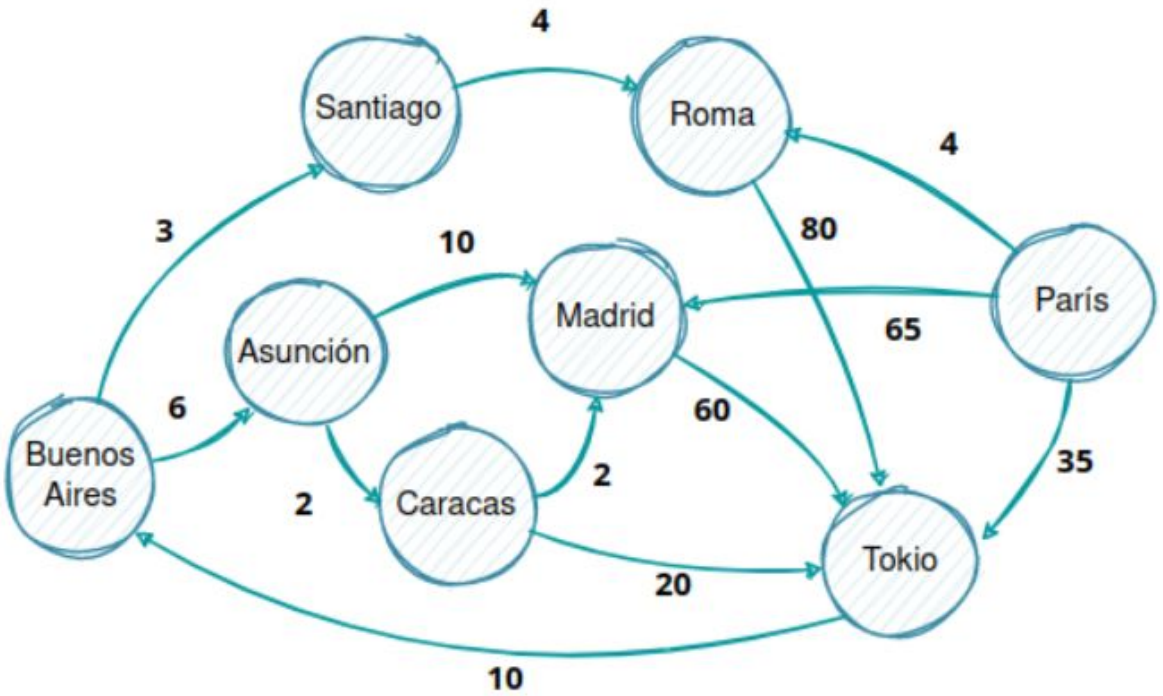




Encontrar los vuelos con un costo determinado

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.

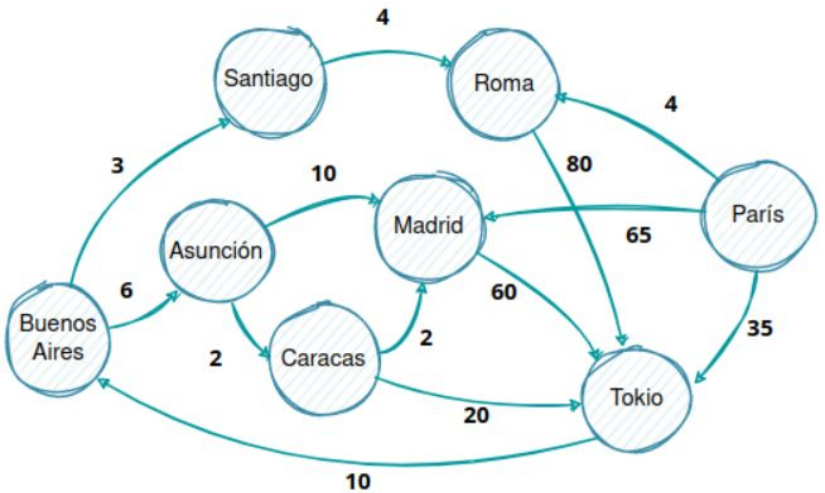


Encontrar los vuelos con un costo determinado

```
public class Vuelos<T> {

        public LinkedList<LinkedList<String>> dfsConCosto(Graph<T> grafo) {
            boolean[] marca = new boolean[grafo.getSize()];
            LinkedList<String> lis = null;
            LinkedList<LinkedList<String>> recorridos = new LinkedList<LinkedList<String>>();
            int costo = 0;
            for (int i=0; i<grafo.getSize(); i++) {
                lis = new LinkedList<>();
                lis.add(grafo.getVertex(i).getData().toString());
                marca[i] = true;
                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) {
   ...
 }
}
```

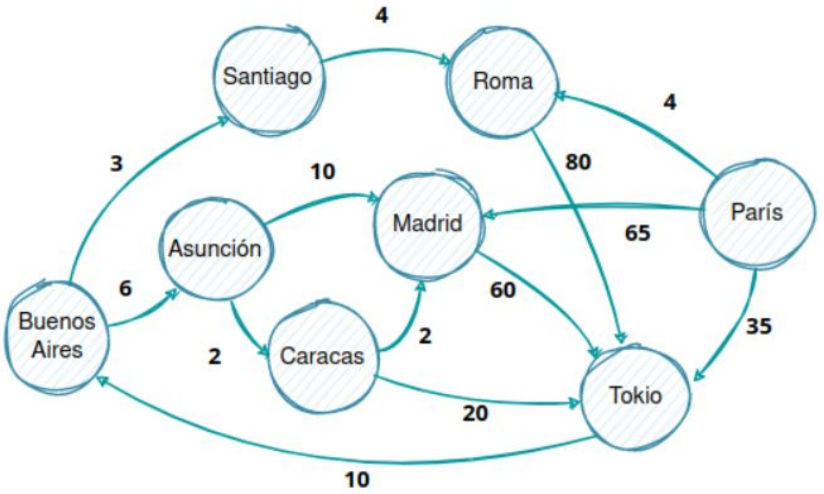
Encontrar los vuelos con un costo determinado

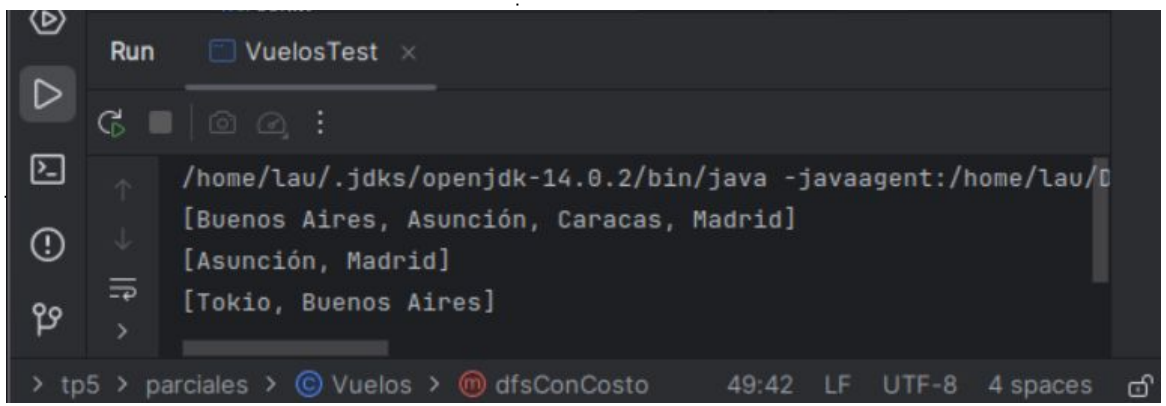
```
private void dfsConCosto(int i, Graph<T> grafo, List<String> lis,boolean[] marca, int costo,
                         LinkedList<LinkedList<String>> recorridos) {
   Vertex<T> vDestino = null;
   int p = 0, j = 0;
   Vertex<T> v = grafo.getVertex(i);
   List<Edge<T>> ady = grafo.getEdges(v);
   for (Edge<T> e: ady) {
       j= e.getTarget().getPosition();
       if (!marca[j]) {
           p = e.getWeight();
           if ((costo + p) <= 10) {
               vDestino = e.getTarget();
               lis.add(vDestino.getData().toString());
               marca[j] = true;
               if ((costo+p) == 10)
                  recorridos.add(new LinkedList<String>(lis));
               else
                  dfsConCosto(j, grafo, lis, marca, costo + p, recorridos);
               lis.remove(vDestino.getData().toString());
               marca[j] = false;
           }
       }

   }
}
```



Encontrar los vuelos con un costo determinado

```
public class VuelosTest {

public static void main(String[] args) {
  Graph<String> ciudades = new AdjListGraph<>();
  Vertex<String> v1 = ciudades.createVertex("Buenos Aires");
  Vertex<String> v2 = ciudades.createVertex("Santiago");
  Vertex<String> v3 = ciudades.createVertex("Asunción");
  Vertex<String> v4 = ciudades.createVertex("Caracas");
  Vertex<String> v5 = ciudades.createVertex("Madrid");
  . . .
  ciudades.connect(v1, v2, 3);  //"Buenos Aires", "Santiago"
  ciudades.connect(v1, v3, 6);  //"Buenos Aires", "Asunción"
  ciudades.connect(v2, v7, 4);  //"Santiago", "Roma"
  . . .
  Vuelos<String> vuelos = new Vuelos();
  LinkedList<LinkedList<String>> lista = vuelos.dfsConCosto(ciudades);
  for (LinkedList<String> e: lista){
     System.out.println(e);
  }
}
```





Grafos
BFS (Breath First Search)

Este algoritmo es la generalización del recorrido por niveles de un árbol. La estrategia es la siguiente:

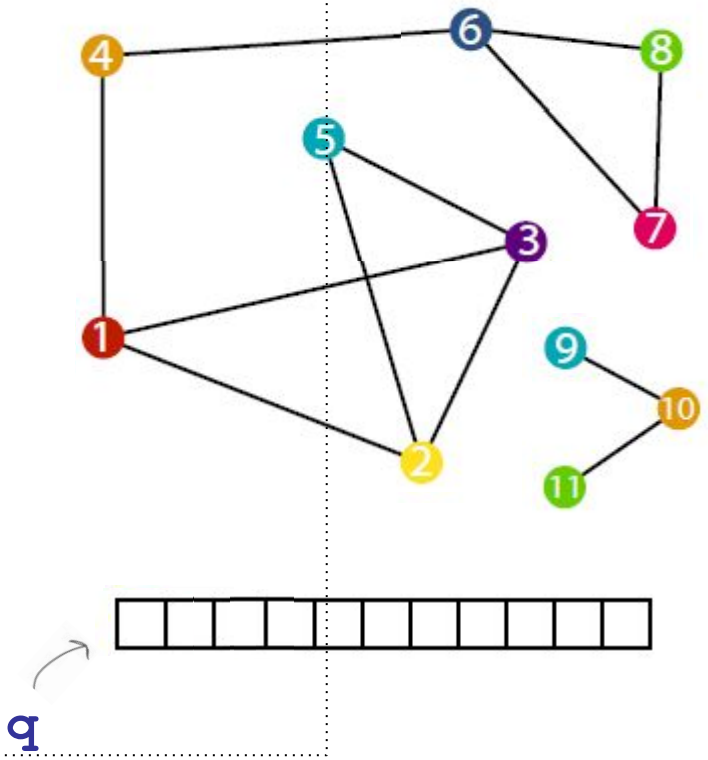
- Partir de algún vértice v, visitar v, después visitar cada uno de los vértices adyacentes a v.
- Repetir el proceso para cada nodo adyacente a v, siguiendo el orden en que fueron visitados.

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

Grafos
BFS (Breath First Search)

```
private void bfs(int i, Graph<T> grafo, boolean[] marca) {
    Queue<Vertex<T>> q = new Queue<Vertex<T>>();
    q.enqueue(grafo.getVertex(i));
    marca[i] = true;
    while (!q.isEmpty()) {
        Vertex<T> w = q.dequeue();
        System.out.println(w);
        // para todos los vecinos de w:
        List<Edge<T>> adyacentes = grafo.getEdges(w);
        for (Edge<T> e: adyacentes) {
            int j = e.getTarget().getPosition();
            if (!marca[j]) {
                marca[j] = true;
                //Vertex<T> v = e.getTarget();
                q.enqueue(e.getTarget());
            }
        }

    }
}
```



Ejercicio de Parcial

Tiempo de infección de una red

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 sí, y un mismo virus puede infectar un grupo de computadoras al mismo tiempo sin importar la cantidad.

Ejercicio de Parcial

Tiempo de infección de una red

```
public static int calcularTiempoInfeccion(Graph<String> g, Vertex<String> inicial) {
   int tiempo = 0;
   boolean visitados[] = new boolean[g.getSize()];
   Queue<Vertex<String>> cola = new Queue<Vertex<String>>();
   visitados[inicial.getPosition()] = true;
   cola.enqueue(inicial);
   cola.enqueue(null);
   while (!cola.isEmpty()) {
       Vertex<String> v = cola.dequeue();
       if (v != null) {
           List<Edge<String>> adyacentes = v.getEdges();
           for (Edge<String> e : adyacentes) {
               Vertex<String> w = e.getTarget();
               if (!visitados[w.getPosition()]) {
                   visitados[w.getPosition()] = true;
                   cola.dequeue();
               }
           }
       } else if (!cola.isEmpty()) {
           tiempo++;
           cola.enqueue(null);
       }
   }
   return tiempo;
}
```