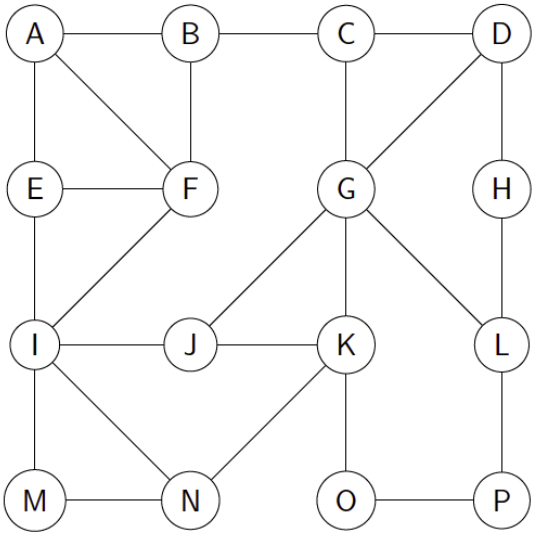
TEMA 9: GRAFOS

Un **grafo** es una forma de representar las relaciones que existen entre pares de objetos. Es un conjunto de objetos, llamados **vértices (vertices)**, y una colección de **aristas (edges)**, donde cada arista conecta dos vértices.

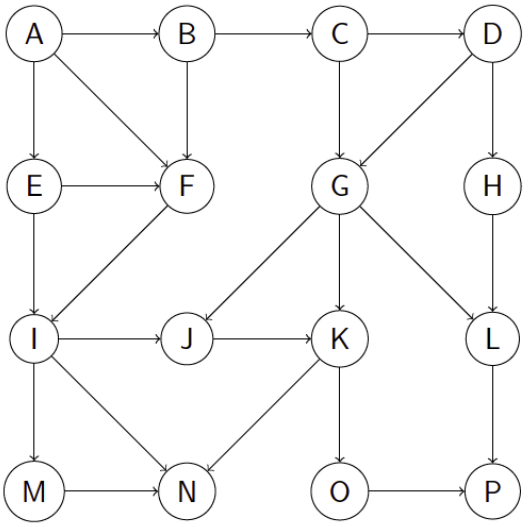
Podemos verlo como un conjunto de vértices V = {u, v, w, x…} y una colección de aristas E = [(u, v), (u, x), …]. Los grafos son de aplicación en múltiples dominios: mapas, transporte, instalaciones eléctricas, redes de computadores, conexiones en redes sociales, …

Las aristas que conectan los vértices (o **nodos**) de un grafo pueden ser de **dos tipos**:

- **Aristas no dirigidas**: Decimos que una arista es no dirigida cuando el par (u, v) no está ordenado. La arista te lleva de u a v y de v a u. El par (u, v) será lo mismo que el par (v, u)
- **Aristas dirigidas**: Decimos que una arista es dirigida cuando el par (u, v) está ordenado. La arista únicamente te lleva de u a v, pero no de v a u. Si todas las aristas de un grafo son aristas no dirigidas, decimos que el grafo es no dirigido. Si hay alguna arista dirigida, el grafo es un grafo dirigido



Grafo no dirigido

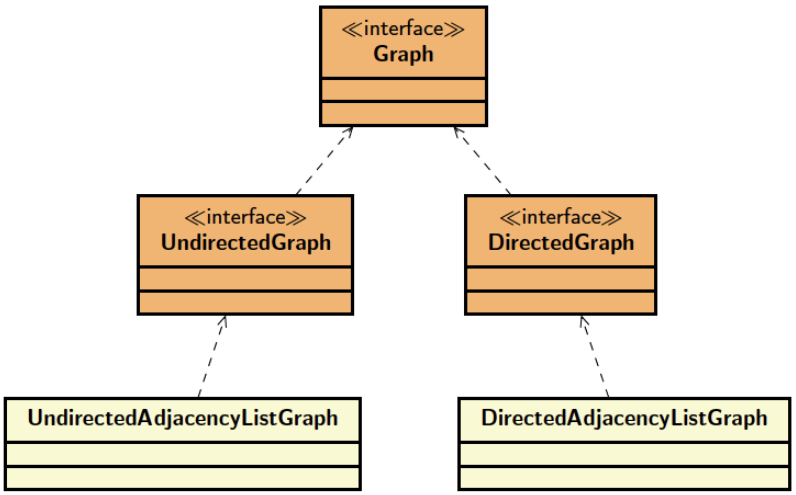


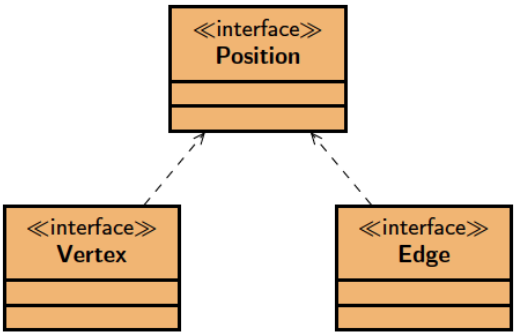
Grafo dirigido

Definiciones:

- Dos vértices son **adyacentes** (**adjacent**) si hay una arista que los conecta
- El **origen** y **destino** son los vértices inicial y final de una arista dirigida
- Un nodo puede tener **aristas salientes** (**outgoing edges**) que tienen como origen el nodo y **aristas entrantes** (**incoming edges**), que tienen el nodo como destino
- El grado de un nodo es el número de aristas que entran y salen del nodo. Podemos distinguir entre el **grado entrante** y el **grado saliente**
- Un **camino** (**path**) es una secuencia de vértices y aristas que empieza en y acaba en un vértice, de forma que cada arista del camino es adyacente con su vértice anterior y su vértice siguiente del camino
- Un **ciclo** (**cycle**) es un camino cuyo primer y último nodo son el mismo
- Un **camino simple** es un camino que no repite vértices (**no contiene ciclos**)
- Un **bosque** es un grafo sin ciclos

Jerarquía de clases:





Interfaz Graph<V,E>

```
public interface Graph <V, E> {
  public int size();
  public boolean isEmpty();
  public int numVertices();
  public int numEdges();
  public int degree(Vertex<V> v) throws IllegalArgumentException;
  public Iterable<Vertex<V>> vertices();
  public Iterable<Edge<E>> edges();
  public V set(Vertex<V> p, V o) throws IllegalArgumentException;
  public E set(Edge<E> p, E o) throws IllegalArguemntExceptio;
  public Vertex<V> insertVertex(V o);
  public V removeVertex(Vertex<V> v) throws IllegalArgumentException;
  public E removeEdge(Edge<E> e) throws IllegalArgumentException;
}
```

- Un grafo dispone de dos genéricos `<V,E>` para almacenar información en los vértices (`V`) y en las aristas (E)
- Los métodos `size`, `isEmpty`, `numVertices` y `numEdges` permiten consultar el número elementos del grafo
- `degree` devuelve el número de aristas incidentes en un vértice
- `vertices` y `edges` permiten recorrer los vértices y las aristas que componen el grafo mediante un Iterable
- `insertVertex` permite insertar un vértice. La inserción de las aristas se deja para las clases especializadas para grafos dirigidos y no dirigidos
- `removeVertex` borran un vértice y las aristas que llegan y salen de él
- `removeEdge` borra una arista, pero no los vértices que la formaban
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

Interfaz UndirectedGraph<V,E>

```
public interface UndirectedGraph <V,E> extends Graph <V,E> {
  public Iterable<Vertex<V>> endVertices(Edge<E> e)
    throws IllegalArgumentException;
  public Edge<E> insertUndirectedEdge(Vertex<V> u, Vertex<V> v, E o)
    throws IllegalArgumentException;
  public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
    throws IllegalArgumentException;
  public boolean areAdjacent(Vertex<V> u, Vertex<V> v)
    throws IllegalArgumentException;
  public Iterable<Edge<E>> edges(Vertex<V> v)
    throws IllegalArgumentException;
}
```

- `UndirectedGraph<V,E>` define el interfaz de un grafo no dirigido
- `insertUndirectedEdge` permite crear aristas conectando los nodos u y v y asociar a la arista un objeto
- Dado un nodo y una arista, el método `opposite` devuelve el nodo que está "al otro lado de la arista"
- El método `areAdjacent` permite saber si dos nodos están conectados mediante alguna arista (**son adyacentes**)
- `edges` (sobrecargado) recibe un vértice y devuelve todas las aristas incidentes en él
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

Minimum Spanning Tree (MST): es un árbol que contiene todos los vértices y un subconjunto de aristas, donde la suma de los pesos de las aristas es mínimo. No tienen por qué ser únicos.

Algoritmo de Prim para construir un MST:

(1) G es un grafo con pesos, y T es el árbol que vamos a construir
(2) Elige cualquier vértice v∈G como la raíz del árbol
(3) Considera todas las aristas e_0, ..., e_n que conectan vértices dentro del árbol con un vértice que todavía no está incluído en él. Elige la arista de *peso mínimo* (**¡OJO! de forma que no haga un ciclo**) y añade la arista junto con sus vértices al árbol T.
(4) Repite paso 3 hasta que todos los vértices estén dentro del árbol T

Este algoritmo es **óptimo** ya que logra construir un árbol donde la suma de los pesos de las aristas es **mínima**.

¿Cómo podemos elegir la arista con el peso mínimo, empezando en un vértice dentro del árbol y terminando en un vértice todavía no en el árbol?

- Usamos una estructura de datos para guardar las aristas que empiezan en un vértice dentro del árbol, y terminando fuera.
- Ordenamos estas aristas según sus pesos.
- Elegimos la arista con el peso mínimo, y lo borramos. Después, quizás, añadimos más aristas a la estructura de datos (si terminan fuera del árbol).

¿Cuál es una estructura de datos buena para tener aristas ordenadas, borrando siempre la arista de peso mínimo e insertando nuevas aristas? Una **cola de prioridad** ya que borrar el elemento mínimo e insertar una nueva arista tiene coste **O(log n)**

Interfaz DirectedGraph<V,E>

```
public interface DirectedGraph <V,E> extends Graph <V,E> {
  public Vertex<V> startVertex(Edge<E> e) throws IllegalArgumentException;
  public Vertex<V> endVertex(Edge<E> e) throws IllegalArgumentException;
  public Edge<E> insertDirectedEdge(Vertex<V> from , Vertex<V> to , E o)
    throws IllegalArgumentException ;
  public Iterable<Edge<E>> outgoingEdges(Vertex<V> v)
    throws IllegalArgumentException;
  public Iterable<Edge<E>> incomingEdges(Vertex<V> v)
    throws IllegalArgumentException;
  public int inDegree(Vertex<V> v) throws IllegalArgumentException;
  public int outDegree(Vertex<V> v) throws IllegalArgumentException;
}
```

- `DirectedGraph<V,E>` define el interfaz de un grafo dirigido
- `insertDirectedEdge` permite crear aristas dirigidas conectando los nodos `from` y `to` y asociar a la arista un objeto
- Dada una arista, los métodos `startVertex` y `endVertex` permiten obtener los nodos participantes en una arista
- Los métodos `outgoingEdges` y `incomingEdges` permiten obtener las aristas entrantes y salientes de un vértice
- `inDegree`, `outDegree` devuelven el número de aristas entrantes o salientes de un vértice
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

¿Para qué tipo de problemas se usan **grafos** en la informática? Uno de los más usados y conocidos es el **Problema del Viajante** (**Travelling Salesman**) en el que, dado un mapa (**un grafo**) con ciudades, y las distancias entre ellos, y una ciudad inicial, devuelve la ruta más corta que visita todas las ciudades y vuelve al ciudad inicial. Para encontrar la solución a este problema y dado que puede llegar a ser muy complejo utilizamos el **Algoritmo de Dijkstra** que encuentra un camino óptimo desde un punto inicial (x,y) a cualquier otro punto del mapa.