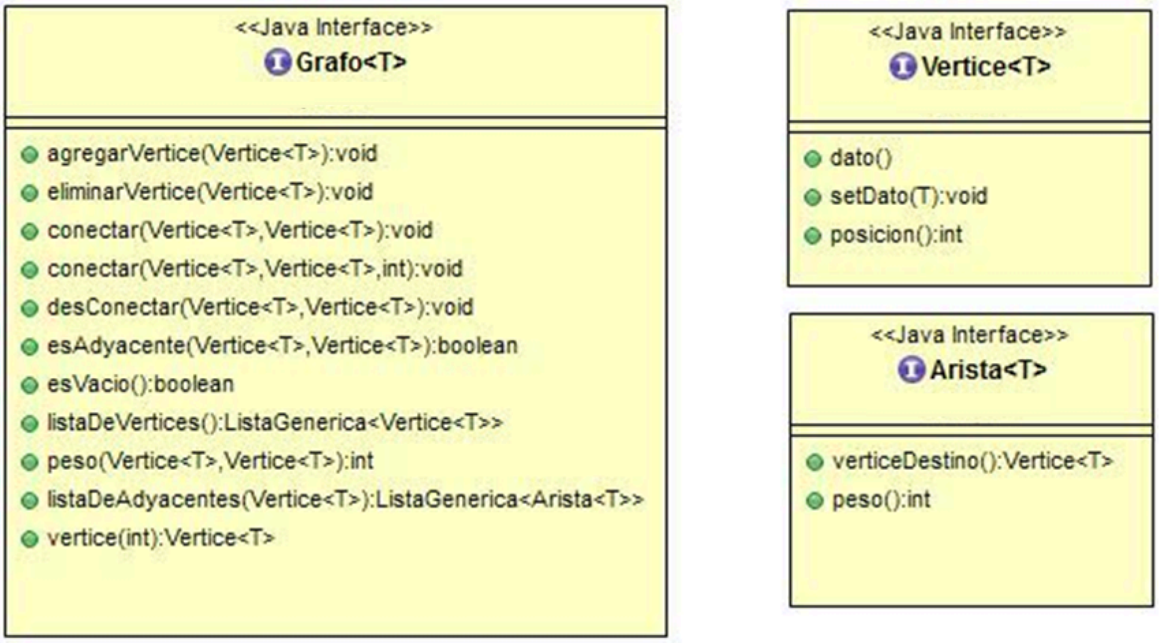
Práctica 6
Grafos
Recorridos, orden topológico

Importante: Descargue el material en archivo .zip disponible del Campus. Se recomienda continuar trabajando dentro del proyecto AyED y generar paquetes y subpaquetes para esta práctica. El archivo .zip contiene las clases que deberá importar al proyecto dentro del IDE con el cual trabaja habitualmente.

1. Considere la siguiente especificación **Grafo**



Interface Grafo

- El método **agregarVertice(Vertice<T> v)** //Agrega un vértice al Grafo. Verifica que el vértice no exista en el Grafo.
- El método **eliminarVertice(Vertice<T> v)** // Elimina el vértice del Grafo. En caso que el vértice tenga conexiones con otros vértices, se eliminan todas sus conexiones.
- El método **conectar(Vertice<T> origen, Vertice<T> destino)** //Conecta el vértice *origen* con el vértice *destino*. Verifica que ambos vértices existan, caso contrario no realiza ninguna conexión.
- El método **conectar(Vertice<T> origen, Vertice<T> destino, int peso)** // Conecta el vértice *origen* con el vértice *destino* con *peso*. Verifica que ambos vértices existan, caso contrario no realiza ninguna conexión.
- El método **desConectar(Vertice<T> origen, Vertice<T> destino)** //Desconecta el vértice *origen* con el *destino*. Verifica que ambos vértices y la conexión *origen* -->*destino* existan, caso contrario no realiza ninguna desconexión. En caso de existir la conexión *destino* -->*origen*, ésta permanece sin cambios.
- El método **esAdyacente(Vertice<T> origen, Vertice<T> destino): boolean** // Retorna true si *origen* es adyacente a *destino*. False en caso contrario.
- El método **esVacio(): boolean** // Retorna true en caso que el grafo no contenga ningún vértice. False en caso contrario.

- El método **listaDeVertices(): ListaGenerica<Vertice<T>>** //Retorna la lista con todos los vértices del grafo.
- El método **peso(Vertice<T> origen, Vertice<T> destino): int //**Retorna el peso de la conexión *origen* -->*destino* . Si no existiera la conexión retorna 0.
- El método **listaDeAdyacentes(Vertice<T> v)): ListaGenerica<Arista**> // Retorna la lista de adyacentes de un vértice.
- El método **vertice(int posicion): Vertice<T>** // Retorna el vértice dada su posición.

Interface Vértice

- El método **dato(): T** // Retorna el dato del vértice.
- El método **setDato(T d)** // Setea el dato del vértice
- El método **posicion(): int** // Retorna la posición del vértice.

Interface Arista

- El método **verticeDestino(): Vertice<T>** // Retorna el vértice destino de la arista.
- El método **peso(): int** // Retorna el peso de la arista

a) Incorpore y analice las interfaces **Grafo**, **Vértice** y **Arista** de acuerdo a la especificación que se detalló, ubicada dentro del paquete tp06.

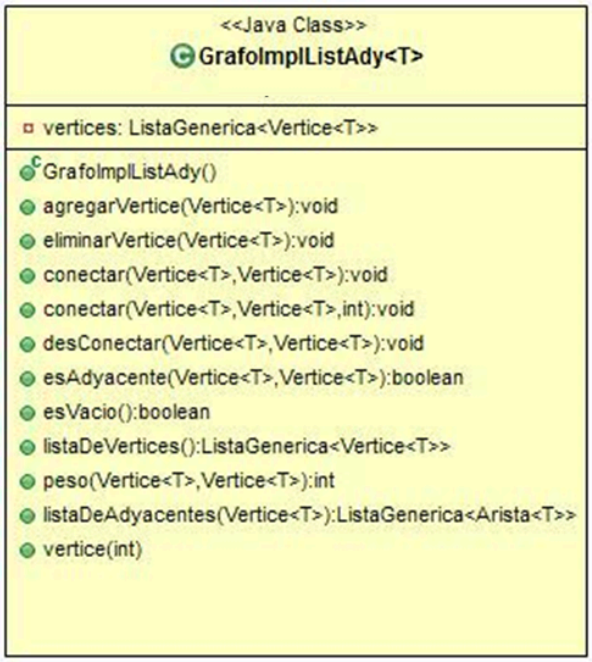
b) Escriba una clase llamada **GrafoImplMatrizAdy** que implemente la interface **Grafo**, y una clase llamada **VerticeImplMatrizAdy** que implemente la interface **Vértice**.

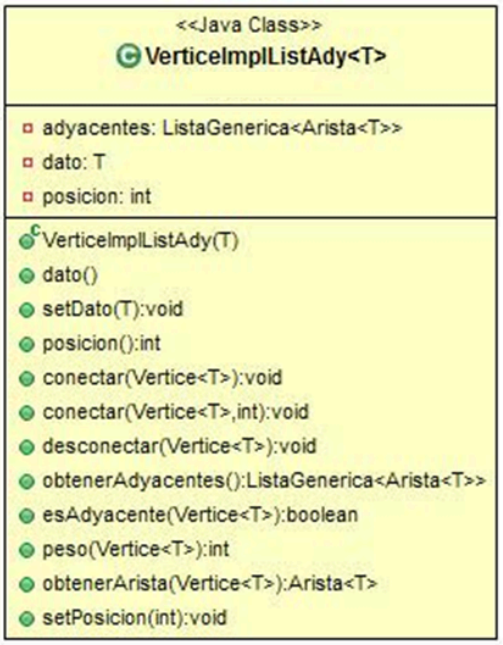
c) Escriba una clase llamada **GrafoImplListAdy** que implemente la interface **Grafo**, y una clase llamada **VerticeImplListAdy** que implemente la interface **Vértice**.

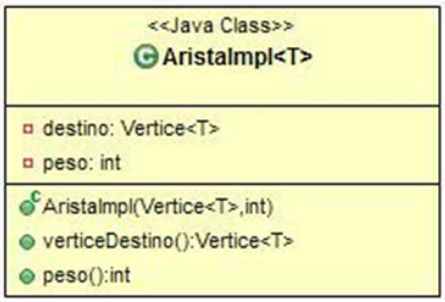
d) Escriba una clase llamada **AristaImpl** que implemente la interface **Arista.** ¿Es posible utilizar la interface y clases que implementan la misma tanto para grafos ponderados como no ponderados? Analice el comportamiento de los métodos que componen la misma.

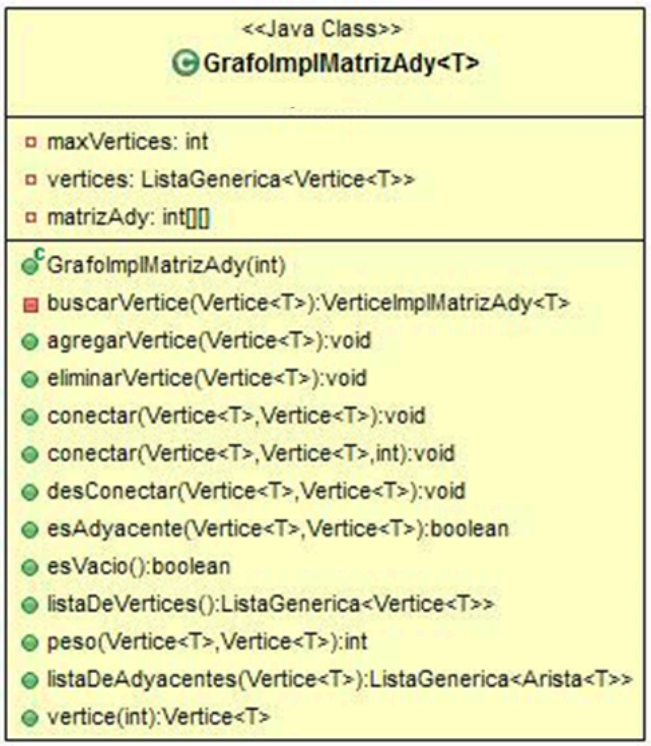
e) Analice qué métodos cambiarían el comportamiento en el caso de utilizarse para modelar grafos dirigidos.

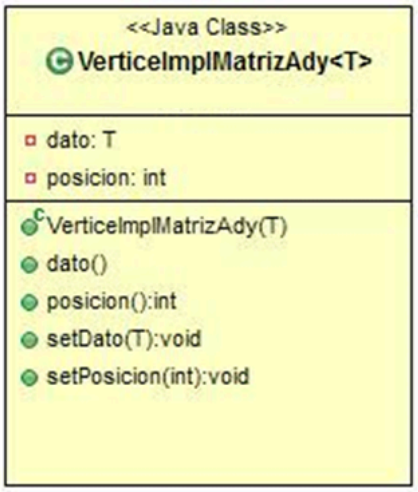
Nota: la clase **ListaGenerica** es la utilizada previamente, está provista en el tp01 y es la misma implementación que en prácticas anteriores.


<<Java Class>>
GrafoImplListAdy<T>
vertices: ListaGenerica<Vertice<T>>
GrafoImplListAdy()
agregarVertice(Vertice<T>):void
eliminarVertice(Vertice<T>):void
conectar(Vertice<T>,Vertice<T>):void
conectar(Vertice<T>,Vertice<T>,int):void
desConectar(Vertice<T>,Vertice<T>):void
esAdyacente(Vertice<T>,Vertice<T>):boolean
esVacio():boolean
listaDeVertices():ListaGenerica<Vertice<T>>
peso(Vertice<T>,Vertice<T>):int
listaDeAdyacentes(Vertice<T>):ListaGenerica<Arista<T>>
vertice(int)


<<Java Class>>
VerticeImplListAdy<T>
adyacentes: ListaGenerica<Arista<T>>
dato: T
posicion: int
VerticeImplListAdy(T)
dato()
setDato(T):void
posicion():int
conectar(Vertice<T>):void
conectar(Vertice<T>,int):void
desconectar(Vertice<T>):void
obtenerAdyacentes():ListaGenerica<Arista<T>>
esAdyacente(Vertice<T>):boolean
peso(Vertice<T>):int
obtenerArista(Vertice<T>):Arista<T>
setPosicion(int):void


<<Java Class>>
AristaImpl<T>
destino: Vertice<T>
peso: int
AristaImpl(Vertice<T>,int)
verticeDestino():Vertice<T>
peso():int


<<Java Class>>
GrafoImplMatrizAdy<T>
maxVertices: int
vertices: ListaGenerica<Vertice<T>>
matrizAdy: int[][]
GrafoImplMatrizAdy(int)
buscarVertice(Vertice<T>):VerticeImplMatrizAdy<T>
agregarVertice(Vertice<T>):void
eliminarVertice(Vertice<T>):void
conectar(Vertice<T>,Vertice<T>):void
conectar(Vertice<T>,Vertice<T>,int):void
desConectar(Vertice<T>,Vertice<T>):void
esAdyacente(Vertice<T>,Vertice<T>):boolean
esVacio():boolean
listaDeVertices():ListaGenerica<Vertice<T>>
peso(Vertice<T>,Vertice<T>):int
listaDeAdyacentes(Vertice<T>):ListaGenerica<Arista<T>>
vertice(int):Vertice<T>


<<Java Class>>
VerticeImplMatrizAdy<T>
dato: T
posicion: int
VerticeImplMatrizAdy(T)
dato()
posicion():int
setDato(T):void
setPosicion(int):void

2.

a. Implemente en JAVA una clase llamada **Recorridos** ubicada dentro del paquete **tp06** cumpliendo la siguiente especificación:

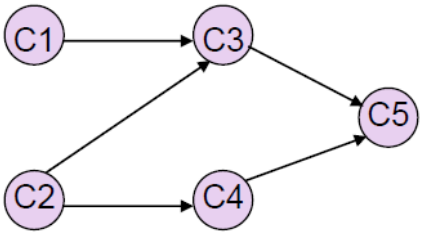
dfs(Grafo<T> grafo): ListaGenerica <T> // Retorna una lista de vértices con el recorrido en profundidad del *grafo* recibido como parámetro.

bfs(Grafo<T> grafo): ListaGenerica <T>// Retorna una lista de vértices con el recorrido en amplitud del *grafo* recibido como parámetro.

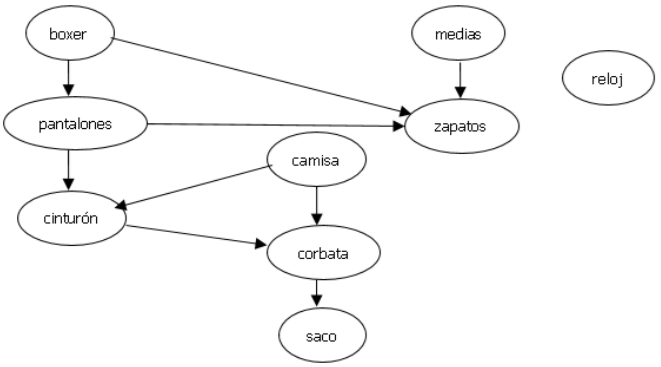
b. Estimar los órdenes de ejecución de los métodos anteriores.

3. La organización topológica (o "sort topológico") de un grafo dirigido acíclico (DAG) es un proceso de asignación de un orden lineal a los vértices del DAG de modo que si existe una arista (v,w) en el DAG, entonces v aparece antes de w en dicho ordenamiento lineal.
Por ejemplo, sea el siguiente DAG, posibles organizaciones topológicas son las siguientes:

- C1, C2, C4, C3 y C5
- C2, C4, C1, C3 y C5
- C1, C2, C3, C4 y C5

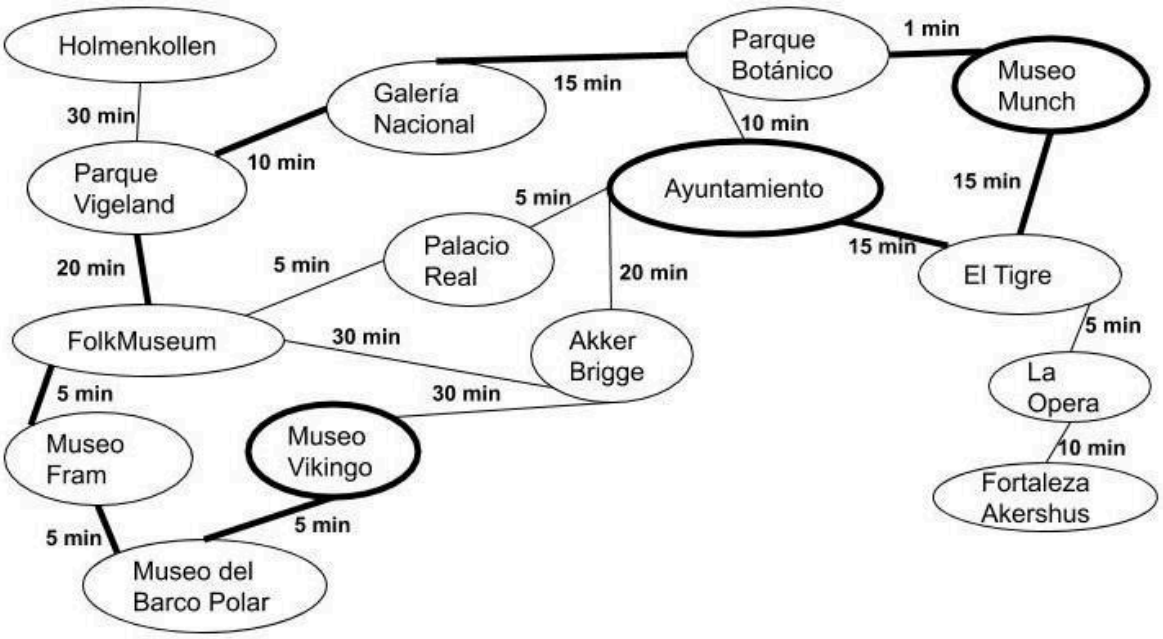


El siguiente DAG surge cuando el Profesor Miguel se viste a la mañana. El profesor debe ponerse ciertas prendas antes que otras. Por ejemplo, las medias antes que los zapatos. Otras prendas pueden ponerse en cualquier orden. Por ejemplo, las medias y los pantalones. Una arista dirigida (v,w) en el DAG indica que la prenda v debe ser puesta antes que la prenda w. Enumere algunos posibles órdenes topológicos que se pueden obtener a partir del DAG previo.



4. Se quiere realizar un paseo en bicicleta por lugares emblemáticos de Oslo. Para esto se cuenta con un grafo de bicisendas. Partiendo desde el "Ayuntamiento" hasta un lugar destino en menos de X minutos, sin pasar por un conjunto de lugares que están restringidos.
Escriba una clase llamada **VisitaOslo** e implemente su método:

ListaGenerica<String> paseoEnBici(Grafo<String> lugares, String destino, int maxTiempo, ListaGenerica<String> lugaresRestringidos)

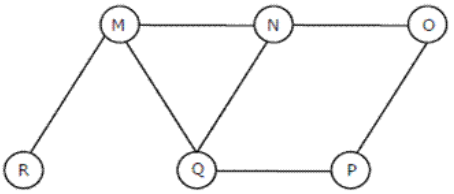


En este ejemplo, para llegar desde **Ayuntamiento** a **Museo Vikingo**, sin pasar por: {"Akker Brigge", "Palacio Real"} y en no más de 120 minutos, el camino marcado en negrita cumple las condiciones.
Notas:

- El "Ayuntamiento" debe ser buscado antes de comenzar el recorrido para encontrar un camino.
- De no existir camino posible, se debe retornar una lista vacía.
- Debe retornar el **primer camino** que encuentre que cumple las restricciones.
- Ejemplos de posibles caminos a retornar, sin pasar por "Akker Brigge" y "Palacio Real" en no más de 120 min (maxTiempo)
    - **Ayuntamiento, El Tigre, Museo Munch, Parque Botánico, Galería Nacional, Parque Vigeland, FolkMuseum, Museo Fram, Museo del Barco Polar, Museo Vikingo. El recorrido se hace en 91 minutos.**

- **Ayuntamiento, Parque Botánico, Galería Nacional, Parque Vigeland, FolkMuseum, Museo Fram, Museo del Barco Polar, Museo Vikingo. El recorrido se hace en 70 minutos.**

5. ¿Cuál de los siguientes es un recorrido BFS válido para el grafo de la figura?



(a) MNOPQR **(b)** NQMPOR **(c)** QMNPRO **(d)** QMNPOR

6. Teniendo en cuenta las dos representaciones de grafos: Matriz de Adyacencias y Lista de Adyacencias.

a. Bajo qué condiciones usaría una Matriz de Adyacencias en lugar de una Lista de Adyacencias para representar un grafo. Y una Lista de Adyacencias en lugar de una Matriz de Adyacencias. **Fundamentar**.
b. ¿En función de qué parámetros resulta apropiado realizar la estimación del orden de ejecución para algoritmos sobre grafos densos? ¿Y para algoritmos sobre grafos dispersos? **Fundamentar**.
c. Si representamos un grafo no dirigido usando una Matriz de Adyacencias, ¿cómo sería la matriz resultante? **Fundamentar**.

7.

a. Responda las siguientes preguntas considerando un grafo no dirigido de n vértices. **Fundamentar**.
    i. ¿Cuál es el mínimo número de aristas que puede tener si se exige que el grafo sea conexo?
    ii. ¿Cuál es el número de aristas que puede tener si se exige que el grafo sea completo? (Un grafo es completo si hay una arista entre cada par de vértices.)
b. En un grafo dirigido y que no tiene aristas que vayan de un nodo a sí mismo, ¿Cuál es el mayor número de aristas que puede tener? **Fundamentar**.

8.

a. ¿Cuál es la diferencia entre un grafo y un árbol?

b. Indicar para cada uno de los siguientes casos si la relación se representa a través de un grafo no dirigido o de un grafo dirigido (digrafo).

i) Vértices: países. **Aristas:** es limítrofe.

ii) Vértices: usuarios de Instagram. **Aristas:** es seguidor.

iii) Vértices: dispositivos en una red de computadoras. **Aristas:** conectividad.

iv) Vértices: aeropuertos. **Aristas:** existe vuelo.

v) Vertices: países que jugaron en Qatar 2022. **Aristas:** le ganó a.