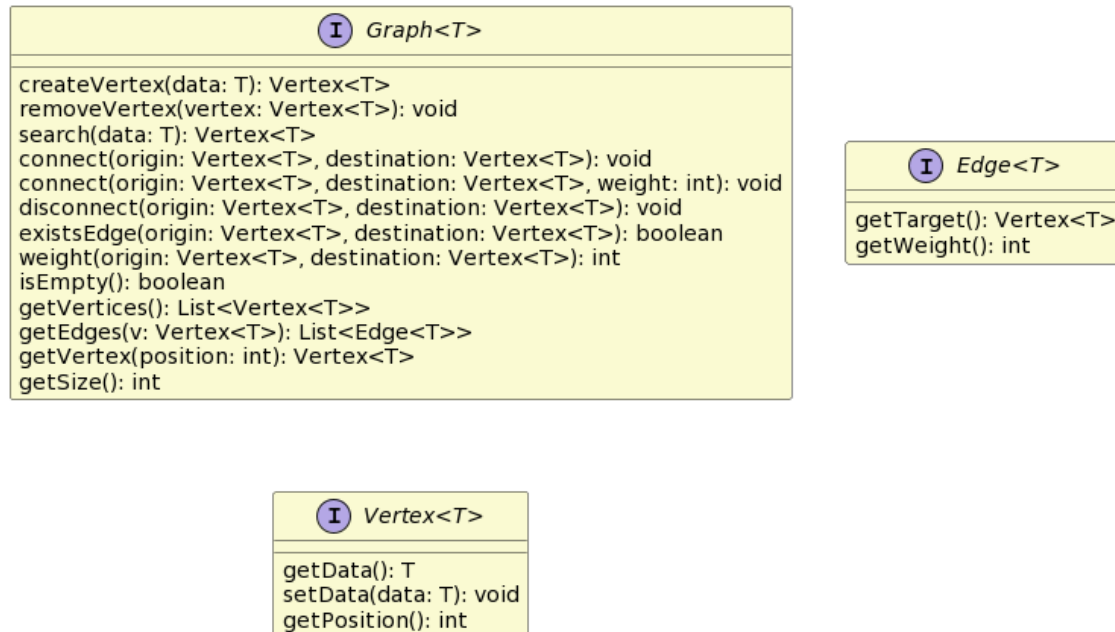


Trabajo Práctico N° 5: Grafos.

Ejercicio 1.

Sea la siguiente especificación de un Grafo:



Interface Graph<T>

Con los siguientes métodos:

- ***public Vertex<T> createVertex(T data)***
Crea un vértice con el dato recibido y lo retorna.
- ***public void removeVertex(Vertex<T> vertex)***
Elimina el vértice del Grafo.
En caso de que el vértice esté relacionado con otros, estas relaciones también se eliminan.
- ***public Vertex<T> search(T data)***
Busca y retorna el primer vértice cuyo dato es igual al parámetro recibido.
Retorna null si no existe tal.
- ***public void connect(Vertex<T> origin, Vertex<T> destination)***
Conecta el vértice origen con el vértice destino.
Verifica que ambos vértices existan, caso contrario no realiza ninguna conexión.
- ***public void connect(Vertex<T> origin, Vertex<T> destination, int weight)***
Conecta el vértice origen con el vértice destino con peso.
Verifica que ambos vértices existan, caso contrario no realiza ninguna conexión.
- ***public void disconnect(Vertex<T> origin, Vertex<T> destination)***
Desconecta el vértice origen con el destino.
Verifica que ambos vértices existan, caso contrario no realiza ninguna desconexión.

En caso de existir la conexión destino \rightarrow origen, ésta permanecerá sin cambios.

- ***public boolean existsEdge(Vertex<T> origin, Vertex<T> destination)***

Retorna true si existe una arista entre el vértice origen y el destino.

- ***public boolean isEmpty()***

Retorna si el grafo no contiene datos (sin vértices creados).

- ***public List<Vertex<T>> getVertices()***

Retorna la lista de vértices.

- ***public int weight(Vertex<T> origin, Vertex<T> destination)***

Retorna el peso entre dos vértices.

En caso de no existir la arista, retorna 0.

- ***public List<Edge<T>> getEdges(Vertex<T> v)***

Retorna la lista de adyacentes al vértice recibido.

- ***public Vertex<T> getVertex(int position)***

Obtiene el vértice para la posición recibida.

- ***public int getSize()***

Retorna la cantidad de vértices del grafo.

Interface Vertex<T>

- ***public T getData()***

Retorna el dato del vértice.

- ***public void setData(T data)***

Reemplaza el dato del vértice.

- ***public int getPosition()***

Retorna la posición del vértice en el grafo.

Interface Edge<T>

- ***public Vertex<T> target()***

Retorna el vértice destino de la arista.

- ***public int getWeight()***

Retorna el peso de la arista.

(a) Definir las interfaces *Graph*, *Vertex* y *Edge* de acuerdo a la especificación que se detalló previamente, dentro del paquete *ejercicio1*.

(b) Definir las clases necesarias para implementar grafos con matriz de adyacencia, utilizando las interfaces dadas.

(c) Definir las clases necesarias para implementar grafos con listas de adyacentes, utilizando las interfaces dadas.

(d) Dada la interfaz *Edge* previa, ¿es posible utilizarla para implementar grafos ponderados como no ponderados? Analizar el comportamiento de los métodos que componen la misma.

Dada la interfaz *Edge* previa, es posible utilizarla para implementar tanto grafos ponderados como no ponderados.

En cuanto al comportamiento de los métodos que componen la misma:

- *public Vertex<T> target()*: Este método es válido tanto para grafos ponderados como no ponderados.
- *public int getWeight()*: Este método pierde significado semántico en grafos no ponderados, aunque se puede seguir usando asignando un valor fijo (como 1) a todas las aristas.

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

Los métodos relacionados con la creación de vértices, búsqueda, datos o estructura básica no cambian. Los métodos que cambiarían de comportamiento en el caso de utilizarse para modelar grafos dirigidos son:

- ***public void removeVertex(Vertex<T> vertex):***
Elimina las aristas salientes y entrantes.
- ***public void connect(Vertex<T> origin, Vertex<T> destination):***
Agrega sólo la arista *origin* \rightarrow *destination*.
- ***public void connect(Vertex<T> origin, Vertex<T> destination, int weight):***
Agrega sólo la arista *origin* \rightarrow *destination*.
- ***public void disconnect(Vertex<T> origin, Vertex<T> destination):***
Elimina sólo la arista *origin* \rightarrow *destination* y se aclara que, si existe *destination* \rightarrow *origin*, se mantiene.
- ***public boolean existsEdge(Vertex<T> origin, Vertex<T> destination):***
Retorna *true* sólo si existe la arista *origin* \rightarrow *destination*, no al revés.
- ***public int weight(Vertex<T> origin, Vertex<T> destination):***
Retorna sólo el peso de la arista *origin* \rightarrow *destination*.
- ***public List<Edge<T>> getEdges(Vertex<T> v):***
Retorna sólo las aristas salientes desde *v*.

(f) *Importar las clases dadas por la cátedra y compararlas contra la implementación realizada en los incisos (b) y (c).*

Ejercicio 2.

(a) Implementar en JAVA una clase llamada *Recorridos* ubicada dentro del paquete *ejercicio2* cumpliendo la siguiente especificación:

dfs(Graph<T> grafo): List<T>

// Retorna una lista con los datos de los vértices, con el recorrido en profundidad del grafo recibido como parámetro.

bfs(Graph<T> grafo): List<T>

// Retorna una lista con los datos de vértices, con el recorrido en amplitud del grafo recibido como parámetro.

(b) Estimar el orden de ejecución de los métodos anteriores.

El orden de ejecución de los métodos anteriores es $O(|V|+|E|)$, donde V es el número de vértices y E es el número de aristas.

Ejercicio 3.

C Mapa
mapaCiudades: Graph<String> devolverCamino(ciudad1: String, ciudad2: String): List<String> devolverCaminoExeptuando(ciudad1: String, ciudad2: String, ciudades: List<String>): List<String> caminoMasCorto(ciudad1: String, ciudad2: String): List<String> caminoSinCargarCombustible(ciudad1: String, ciudad2: String, tanqueAuto: int): List<String> caminoConMenorCargaDeCombustible(ciudad1: String, ciudad2: String, tanqueAuto: int): List<String>

1. devolverCamino(String ciudad1, String ciudad2): List<String>

Retorna la lista de ciudades que se deben atravesar para ir de ciudad1 a ciudad2 en caso de que se pueda llegar, sino retorna la lista vacía. (Sin tener en cuenta el combustible).

2. devolverCaminoExeptuando(String ciudad1, String ciudad2, List<String> ciudades): List<String>

Retorna la lista de ciudades que forman un camino desde ciudad1 a ciudad2, sin pasar por las ciudades que están contenidas en la lista ciudades pasada por parámetro. Si no existe camino, retorna la lista vacía. (Sin tener en cuenta el combustible).

3. caminoMasCorto(String ciudad1, String ciudad2): List<String>

Retorna la lista de ciudades que forman el camino más corto para llegar de ciudad1 a ciudad2. Si no existe camino, retorna la lista vacía. (Las rutas poseen la distancia).

4. caminoSinCargarCombustible(String ciudad1, String ciudad2, int tanqueAuto): List<String>

Retorna la lista de ciudades que forman un camino para llegar de ciudad1 a ciudad2. El auto no debe quedarse sin combustible y no puede cargar. Si no existe camino, retorna la lista vacía.

5. caminoConMenorCargaDeCombustible(String ciudad1, String ciudad2, int tanqueAuto): List<String>

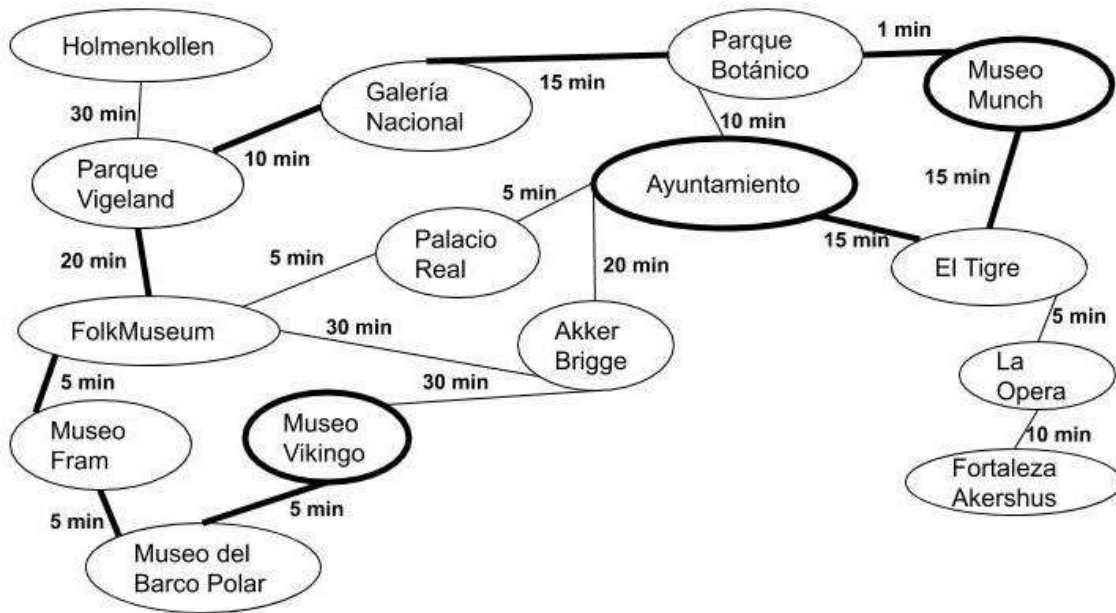
Retorna la lista de ciudades que forman un camino para llegar de ciudad1 a ciudad2 teniendo en cuenta que el auto debe cargar la menor cantidad de veces. El auto no se debe quedar sin combustible en medio de una ruta, además puede completar su tanque al llegar a cualquier ciudad. Si no existe camino, retorna la lista vacía.

Ejercicio 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.

Escribir una clase llamada *VisitaOslo* e implementar su método:

paseoEnBici(Graph<String> lugares, String destino, int maxTiempo, List<String> lugaresRestringidos): List<String>.



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.
- Se debe retornar el primer camino que se encuentre que cumpla 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.

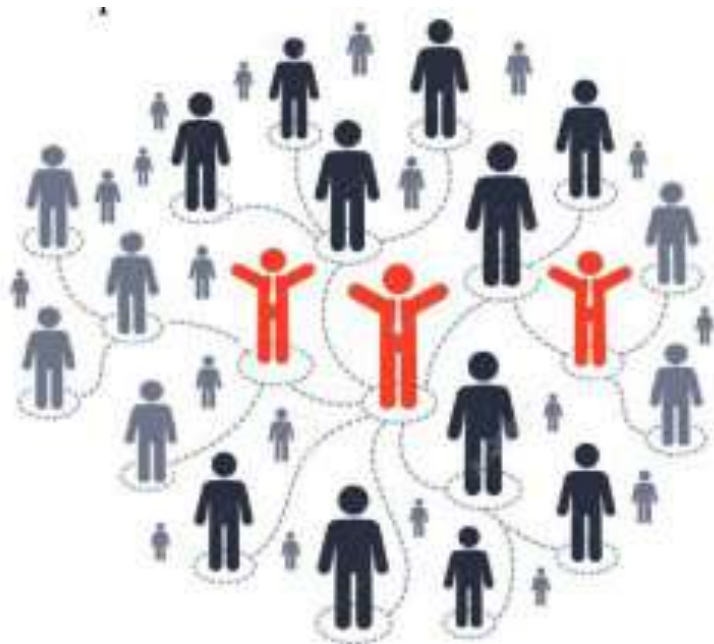
Ejercicio 5.

El Banco Itaú se suma a las campañas “QUEDATE EN CASA” lanzando un programa para acercar el sueldo a los jubilados hasta sus domicilios. Para ello, el banco cuenta con información que permite definir un grafo de personas donde la persona puede ser un jubilado o un empleado del banco que llevará el dinero.

Se necesita armar la cartera de jubilados para cada empleado repartidor del banco, incluyendo, en cada lista, los jubilados que vivan un radio cercano a su casa y no hayan percibido la jubilación del mes.

Para ello, implementar un algoritmo que, dado un Grafo<Persona>, retorne una lista de jubilados que se encuentren a una distancia menor a un valor dado del empleado Itaú (grado de separación del empleado Itaú).

El método recibirá un Grafo<Persona>, un empleado y un grado de separación/distancia y debe retornar una lista de hasta 40 jubilados que no hayan percibido la jubilación del mes y se encuentren a una distancia menor a lo recibido como parámetro.



En este grafo simple, donde los empleados del banco están en color rojo, si se desea retornar los jubilados hasta distancia 2, se debería retornar los jubilados en color negro. La persona conoce si es empleado o jubilado, el nombre y el domicilio.

Ejercicio 6.

Un día, Caperucita Roja decide ir desde su casa hasta la de su abuelita, recolectando frutos del bosque del camino y tratando de hacer el paseo de la manera más segura posible. La casa de Caperucita está en un claro del extremo oeste del bosque, la casa de su abuelita en un claro del extremo este y, dentro del bosque, entre ambas, hay algunos otros claros.

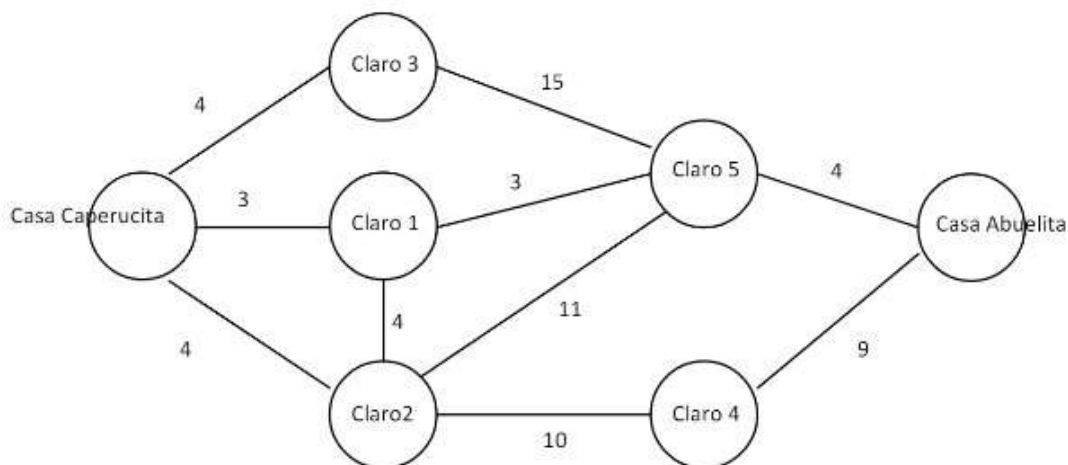
El bosque está representado por un grafo, donde los vértices representan los claros (identificados por un String) y las aristas los senderos que los unen. Cada arista informa la cantidad de árboles frutales que hay en el sendero. Caperucita sabe que el lobo es muy goloso y le gustan mucho las frutas, entonces, para no ser capturada por el lobo, desea encontrar todos los caminos que no pasen por los senderos con cantidad de frutales ≥ 5 y lleguen a la casa de la abuelita.

La tarea es definir una clase llamada *BuscadorDeCaminos*, con una variable de instancia llamada "bosque" de tipo *Graph*, que representa el bosque descrito e implementar un método de instancia con la siguiente firma:

```
public List<List <String>> recorridosMasSeguro(),
```

que devuelva un listado con **TODOS** los caminos que cumplen con las condiciones mencionadas anteriormente.

NOTA: La casa de Caperucita debe ser buscada antes de comenzar a buscar el recorrido.



Ejercicio 7.

- (a) *Implementar, nuevamente, el Ejercicio 3.3 haciendo uso del algoritmo de Dijkstra.*
- (b) *Implementar, nuevamente, el Ejercicio 3.3 haciendo uso del algoritmo de Floyd.*
- (c) *Comparar el tiempo de ejecución de las tres implementaciones.*

Al comparar el tiempo de ejecución de las tres implementaciones, se observa que la implementación que hace uso del algoritmo de *Dijkstra* es la menos eficiente, seguida de la primera implementación, mientras que la más eficiente es la que hace uso del algoritmo de *Floyd*.