UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos
Cursada 2018

Práctica 6
Grafos

Ejercicio 1

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: países. Aristas: principales mercados de exportación.
    iii. Vértices: dispositivos en una red de computadoras. Aristas: conectividad.
    iv. Vértices: variables de un programa. Aristas: relación "usa". (Decimos que la variable **x** usa la variable **y**, si **y** aparece del lado derecho de una expresión y **x** aparece del lado izquierdo, por ejemplo **x = y**).

Ejercicio 2

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 máximo número de aristas que puede tener si se exige que el grafo sea acíclico?
    iii. ¿Cuál es el número de aristas que puede tener si se exige que el grafo sea conexo y acíclico?
    iv. ¿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**.

Ejercicio 3

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

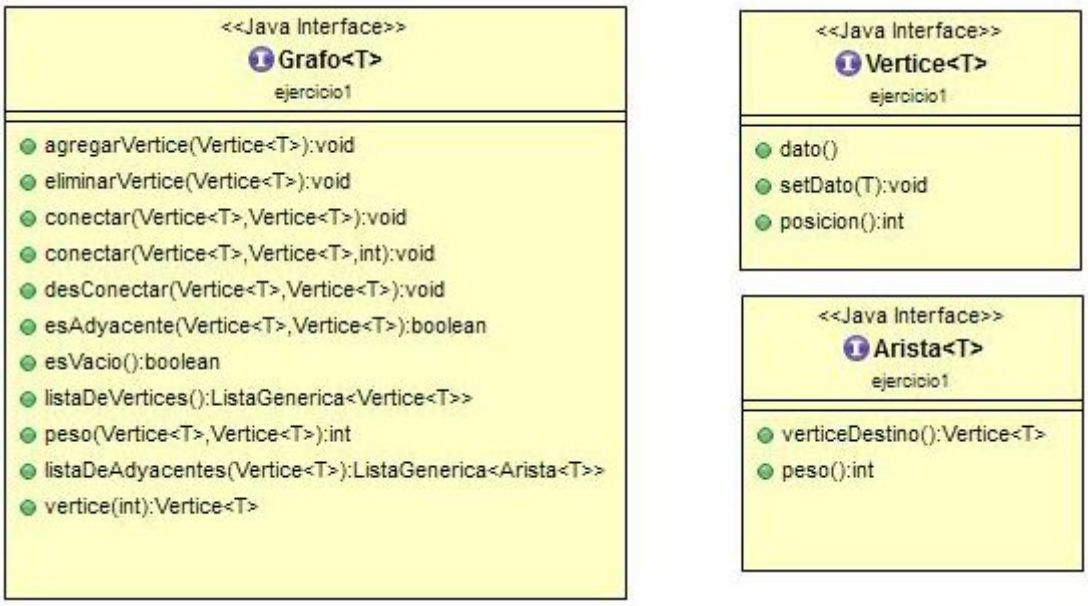
a. Bajo qué condiciones usaría una Matriz de Adyacencias en lugar de una Lista de Adyacencias para representar un grafo. Y una Lista de Adyacencias en lugar de una Matriz de Adyacencias. **Fundamentar**.
b. ¿En función de qué parámetros resulta apropiado realizar la estimación del orden de ejecución 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**.

Ejercicio 4

Sea la siguiente **especificación** de un Grafo:

UNLP. Facultad de Informática.

Algoritmos y Estructuras de Datos
Cursada 2018



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) Defina las interfaces **Grafo**, **Vértice** y **Arista** de acuerdo a la especificación que se detalló, ubicada dentro del paquete ejercicio1.

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**.

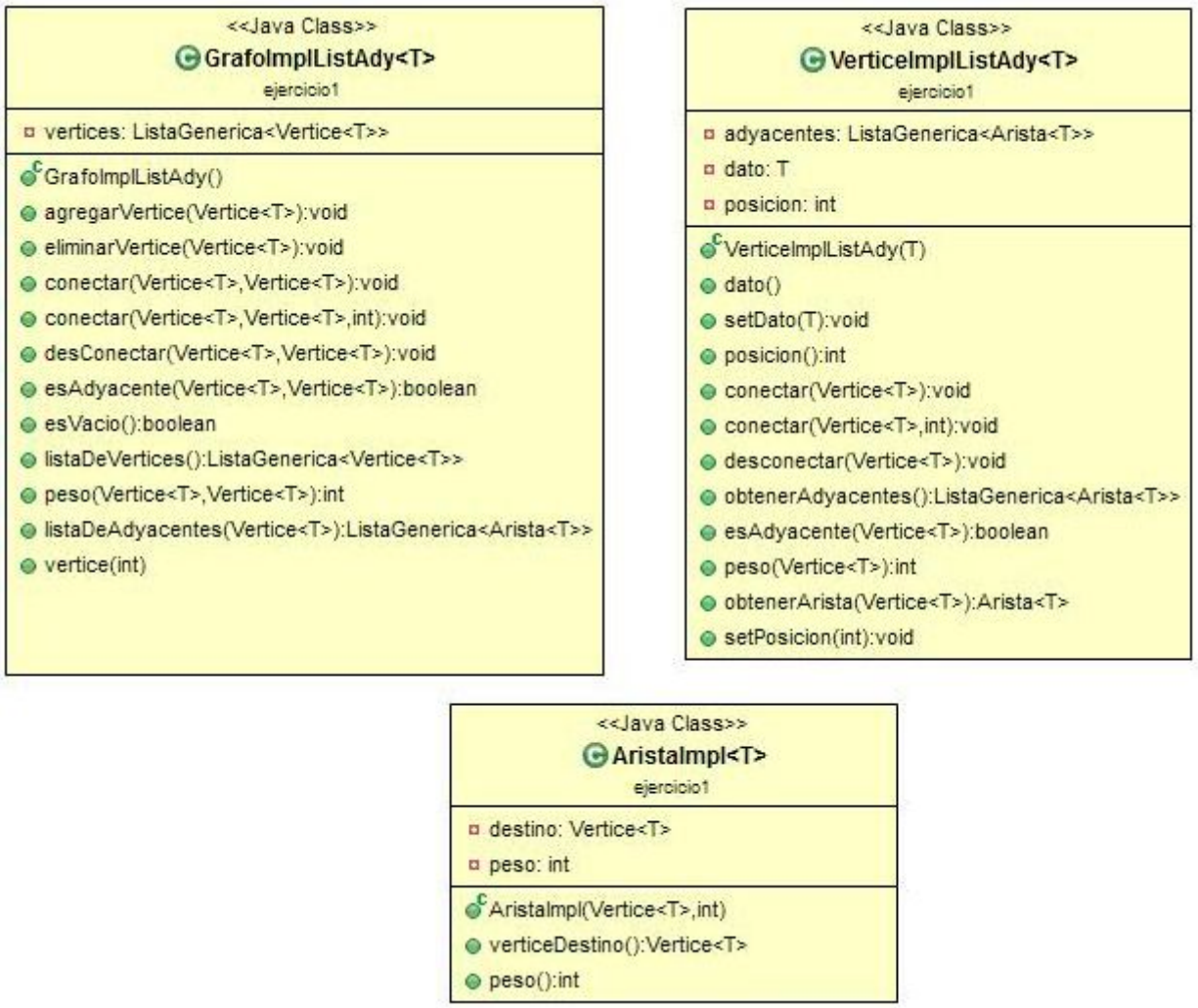
d) Escriba una clase llamada **AristaImpl** que implemente la interface **Arista.** Es posible utilizar la interface y clases que implementan la misma tanto para grafos ponderados como no ponderados? Analice el comportamiento de los métodos que componen la misma.

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

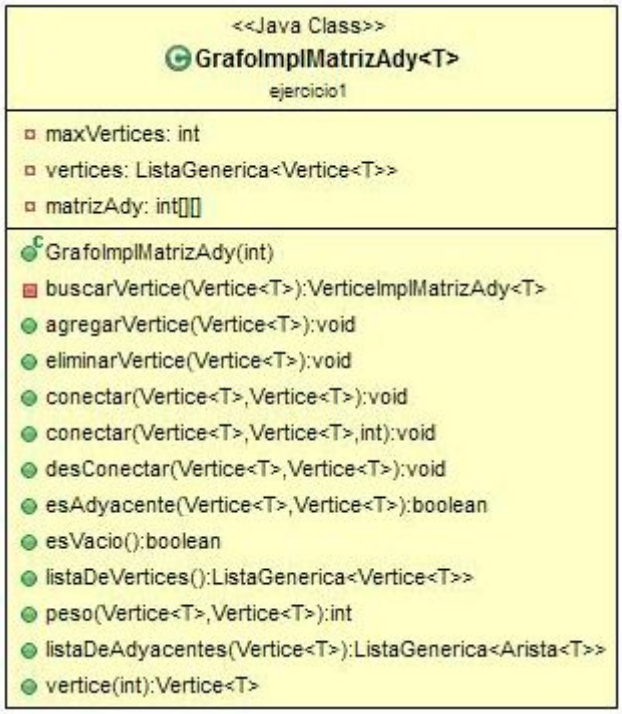
Nota: la clase **ListaGenerica** es la utilizada previamente.

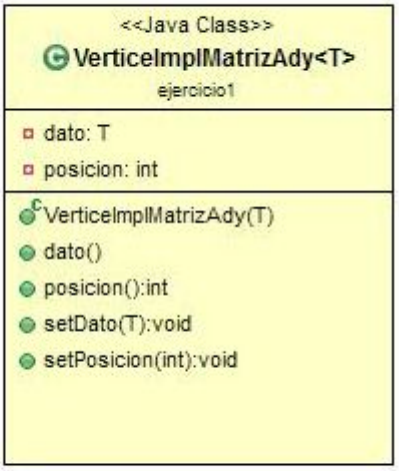
UNLP. Facultad de Informática.

Algoritmos y Estructuras de Datos
Cursada 2018



UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos
Cursada 2018





Ejercicio 5

a. Implemente en JAVA una clase llamada **Recorridos** ubicada dentro del paquete **ejercicio5** cumpliendo la siguiente especificación:

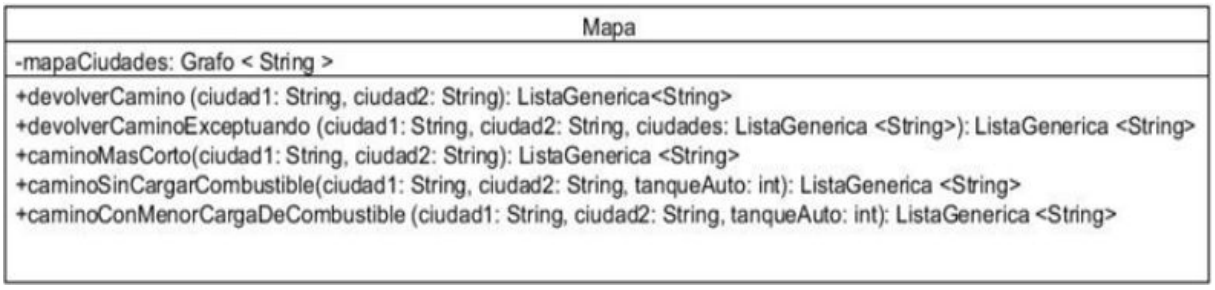
dfs(Grafo<T> grafo): ListaGenerica <T> // Retorna una lista de vértices con el recorrido en profundidad del *grafo* recibido como parámetro.

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.

UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos
Cursada 2018

Ejercicio 6



- El método **devolverCamino (String ciudad1, String ciudad2): ListaGenerica<String>** // Retorna la lista de ciudades que se deben atravesar para ir de *ciudad1* a *ciudad2* en caso que se pueda llegar, si no retorna la lista vacía. (Sin tener en cuenta el combustible).
- El método **devolverCaminoExceptuando (String ciudad1, String ciudad2, ListaGenerica<String> ciudades): ListaGenerica<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).
- El método **caminoMasCorto(String ciudad1, String ciudad2): ListaGenerica<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). (Sin tener en cuenta el combustible).
- El método **caminoSinCargarCombustible(String ciudad1, String ciudad2, int tanqueAuto): ListaGenerica<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.
- El método **caminoConMenorCargaDeCombustible (String ciudad1, String ciudad2, int tanqueAuto): ListaGenerica<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.

UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos
Cursada 2018

Ejercicio 7 – Grados de Separación

En nuestro interconectado mundo se especula que dos personas cualesquiera están relacionadas entre sí a lo sumo por 6 grados de separación. En este problema, debemos realizar un método para encontrar el **máximo grado de separación** en una red de personas, donde una arista entre dos personas representa la relación de conocimiento entre ellas, la cual es simétrica.

Entre dos personas, el grado de separación es el mínimo número de relaciones que son necesarias para conectarse entre ellas.

Si en la red hay dos personas que no están conectadas por una cadena de relaciones, el grado de separación entre ellas se considerará igual a -1.

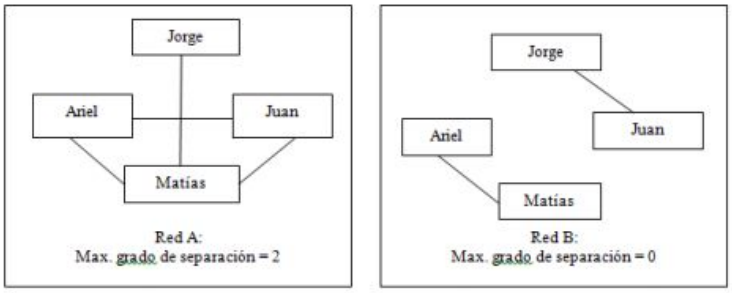
En una red, el **máximo grado de separación** es el mayor grado de separación entre dos personas cualesquiera de la red.

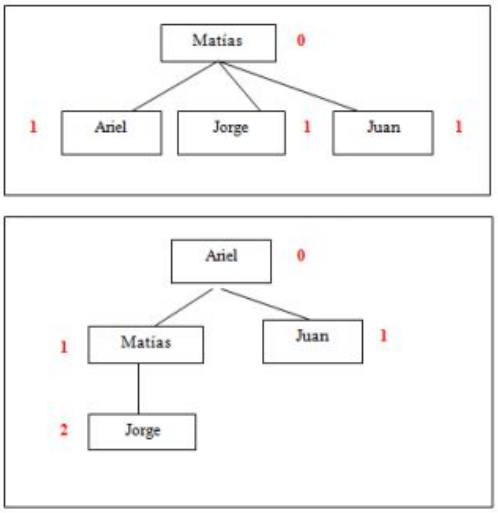
Implemente en JAVA una clase llamada **GradosDeSeparacion** ubicada dentro del paquete **ejercicio7**, cumpliendo la siguiente especificación:

- **maximoGradoDeSeparacion(Grafo<String> grafo) : int** // Retorna el máximo grado de separación del grafo recibido como parámetro. Si en el grafo hubiera dos personas cualesquiera que no están conectadas por una cadena de relaciones entonces se retorna 0.

UNLP. Facultad de Informática.

Algoritmos y Estructuras de Datos
Cursada 2018





UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos
Cursada 2018

Ejercicio 8

"El Paso City", años 20. Las mafias controlan varios sitios y calles de la ciudad. El intendente que debe desplazarse diariamente en su auto desde su residencia a la municipalidad, está seriamente amenazado.

Ud. debe ayudar al intendente encontrando la ruta más segura para realizar su traslado diario implementando en Java un método que recibe como parámetro la ciudad, y retorne la ruta que pase por el menor número de calles y sitios controlados por la mafia. (En caso de existir más de una ruta con retornar alguna de ellas alcanzará).

La ciudad se describe como un conjunto de n sitios y varias calles bidireccionales que unen
esos sitios. Cada sitio tiene la información si está controlado por la mafia o no. Lo mismo sucede con cada una de las calles de la ciudad.