Mini Social Networking App

Basic Procedures

You must:

- Have a style (indentation, good variable names, etc.)
- Comment your code well in JavaDoc style (no need to overdo it, just do it well)
- Have code that compiles with the command: **javac *.java** in your user directory
- Have code that runs with the commands: java **Driver2**

You may:

- Add additional methods and variables, however these methods **must be private**.
- Use built in Java classes and interfaces such as , Queue, Iterator, ArrayList, Stack, HashMap, etc.
- Add **@SuppressWarnings("unchecked")** for equals() method of the Vertex class.

You may NOT:

- **Make your program part of a package. You will lose some points if you violate this.**
- Add additional public methods or variables
- Alter any method signatures defined in this document of the template code (if any). Note: "throws" is part of the method signature in Java, don't add/remove these.
- Add **@SuppressWarnings** to any methods unless they are private helper methods for use with a method specified in the "You may" section above.
- Add any additional libraries/packages which require downloading from the Internet.

Overview

In this task, we are going to develop a mini social networking application called, SConnect. A social networking service or SNS (sometimes called a social networking site) is an online platform, which people use to build social networks or social relationships with other people who share similar personal or career content, interests, activities, backgrounds or real-life connections. Most social networking apps use graph data structure as one of the underlying data structures.

In this task, you will build a simple social networking app that uses graph data structure. The following is a simple demo of the system.

SConnect m = new SConnect();

```
Profile profile1 = new Profile();
profile1.setName("John", "Doe");
profile1.setStatus("My name is John.");

Profile profile2 = new Profile();
profile2.setName("Jane", "Doe");
profile2.setStatus("My name is Jane.");

Profile profile3 = new Profile();
profile3.setName("John", "Smith");
profile3.setStatus("My name is John.");

Profile profile4 = new Profile();
profile4.setName("Jane", "Smith");
profile4.setStatus("My name is Jane.");

m.addUser(profile1);
m.addUser(profile2);
m.addUser(profile3);
m.addUser(profile4);

m.createFriendship(profile1, profile2);
m.createFriendship(profile2, profile3);
m.createFriendship(profile1, profile4);
m.createFriendship(profile4, profile2);
m.createFriendship(profile3, profile4);

m.traverse(profile1);
```

Example Run (Command Line)

- java Driver2

```
Name: John Doe
        Status: My name is John.
        Number of friend profiles: 2
Friends:
        Jane Doe
        Jane Smith

Name: Jane Doe
        Status: My name is Jane.
        Number of friend profiles: 3
Friends:
        John Doe
        John Smith
        Jane Smith

Name: Jane Smith
        Status: My name is Jane.
        Number of friend profiles: 3
Friends:
        John Doe
        Jane Doe
        John Smith

Name: John Smith
        Status: My name is John.
        Number of friend profiles: 2
Friends:
        Jane Doe
        Jane Smith
```

Implementation/Classes

This project will be built using a number of classes representing the component pieces of the SConnect. We are going to use built-in java classes and interfaces extensively for the implementation of the project. Below are the descriptions of these classes and interfaces.

INTERFACE (GraphInterface): GraphInterface.java

This interface represents all the functionalities of the graph data structure you are going to use in this task. The graph that we are going to use to implement SConnect is **undirected graph**.

Operations

+ addVertex(T vertexLabel): boolean - Adds a given vertex to this graph. If vertexLabel is null, it returns false.

+removeVertex(T vertexLabel): VertexInterface<T> - Removes a vertex with the given vertexLabel from this graph and returns the removed vertex. If vertex does not exist, it will return null.

+addEdge(T begin, T end, double edgeWeight): boolean- Adds a weighted edge between two given distinct vertices that are currently in this graph. The desired edge must not already be in the graph. Note that the graph is undirected graph.

+addEdge(T begin, T end):boolean - Adds an unweighted edge between two given distinct vertices that are currently in this graph. The desired edge must not already be in the graph.

+removeEdge(T begin, T end, double edgeWeight): boolean- Removes a weighted edge between two given distinct vertices that are currently in this graph. The desired edge must already be in the graph. It returns true if the removal is successful, false otherwise.

+removeEdge(T begin, T end):boolean - Removes an unweighted edge between two given distinct vertices that are currently in this graph. The desired edge must already be in the graph. It returns true if the removal is successful, false otherwise

+hasEdge(T begin, T end):boolean - Sees whether an undirected edge exists between two given vertices.

+getNumberOfVertices (): int - This method returns the number of Vertices in this graph.

+getNumberOfEdges (): int - This method returns the number of undirected Edges in this graph.

+isEmpty():boolean - This method returns true, if this graph is empty, false otherwise.

+getVertices():List<VertexInterface<T>> - This method returns the list of all vertices in the graph. If the graph is empty, it returns null.

+clear():void – clears the graph.

+getBreadthFirstTraversal(T origin): Queue<T> - Performs a breadth-first traversal of a graph and returns the queue that contains the result. Empty queue can be returned.

+getShortestPath(T origin, T destination, Stack<T> path): int - returns the shortest distance between the origin and destination. If a path does not exist, it returns the maximum integer (to simulate infinity).

INTERFACE (VertexInterface): VertexInterface.java

This interface represents all the functionalities of a Vertex in a graph.

Operations

+getLabel(): T - Gets this vertex's label.

+getNumberOfNeighbors():int - Returns the number of neighbors of this vertex.

+visit(): void - Marks this vertex as visited.

+unvisit(): void - Removes this vertex's visited mark.

+isVisited():boolean - Returns true if the vertex is visited, false otherwise.

+connect(VertexInterface<T> endVertex, double edgeWeight): Boolean - Connects this vertex and endVertex with a weighted edge. The two vertices cannot be the same, and must not already have this edge between them. Two vertices are equal (same)if their labels are equal (same). Returns true if the connection is successful, false otherwise.

+connect(VertexInterface<T> endVertex): boolean- Connects this vertex and endVertex with a unweighted edge. The two vertices cannot be the same, and must not already have this edge between them. Two vertices are equal (same)if their labels are equal (same). Returns true if the connection is successful, false otherwise.

+disconnect(VertexInterface<T> endVertex, double edgeWeight): Boolean - Disconnects this vertex from a given vertex with a weighted edge, i.e., removes the edge. The Edge should exist in order to be disconnected. Returns true if the disconnection is successful, false otherwise.

+disconnect(VertexInterface<T> endVertex): boolean- Disconnects this vertex from a given vertex with an unweighted edge. The Edge should exist in order to be disconnected. Returns true if the disconnection is successful, false otherwise.

+getNeighborIterator():Iterator<VertexInterface<T>> – creates an iterator of this vertex's neighbors by following all edges that begin at this vertex.

+hasNeighbor():boolean – Sees whether this vertex has at least one neighbor.

+getUnvisitedNeighbor():VertexInterface<T> – Gets an unvisited neighbor, if any, of this vertex.

+setPredecessor(VertexInterface<T> predecessor):void – Records the previous vertex on a path to this vertex.

+getPredecessor():VertexInterface<T> – Gets the recorded predecessor of this vertex.

+hasPredecessor():boolean– Sees whether a predecessor was recorded for this vertex.

+setCost(double newCost): void– Records the cost of a path to this vertex.

+getCost(): double– Returns the cost of a path to this vertex.

Note that these interfaces should be generic, i.e., it should be able to accommodate any type of object. The SConnect is just an example application that uses these interfaces.

CLASS (Vertex) Vertex.java

This generic class implements the VertexInterface. The class has the following attributes:

- o **label: T** – Represents the label of the vertex.
- o **visited: boolean** - Stores if the vertex is visited or not, true if visited.
- o **previousVertex: VertexInterface<T>** - on path to this vertex
- o **cost: double** - of path to this vertex
- o **edgeList: List<Edge>** - list of edges to neighbors. Note that there is an Edge class used.

+ Vertex(T vertexLabel): constructor - initializes label to the given value, visited → false, cost → 0.0, previousVertex →null, and the edgeList to a default list.

The Big-O of your methods' implementations **Should be** as follows:

Method Name	Big-O	Remark
getLabel()	O(1)	
visit()	O(1)	
unvisit()	O(1)	
isVisited()	O(1)	
getNumberOfNeighbors()	O(n)	Where n is the number of neighbors(vertices)
connect(endVertex, edgeWeight)	O(n)	
connect(endVertex)	O(n)	
disconnect(endVertex)	O(n)	
disconnect(endVertex, edgeWeight)	O(n)	
getNeighborIterator()	O(n)	
hasNeighbor()	O(1)	
getUnvisitedNeighbor()	O(n)	
setPredecessor(predecessor)	O(1)	
getPredecessor()	O(1)	
hasPredecessor()	O(1)	
setCost(double newCost)	O(1)	
getCost()	O(1)	

CLASS (Graph) Graph.java

This generic class implements the GraphInterface. The implementation will provide an undirected graph. The class has attribute, which contains the collection of vertices the graph has. This attribute is a collection of key value pair, where key is the label of the vertex and the value is the vertex. Note that this attribute can be any data structure that implements the Map interface.

- **vertices** - A dictionary of key (Vertex label), value (Vertex) pair.

+Graph(): constructor - initializes the graph with an empty graph structure.

The Big-O of your methods' implementations Should be as follows:

Method Name	Big-O	Remark
addVertex(T vertexLabel)	O(1)	
removeVertex(T vertexLabel)	O(n^2)	Where n is the number of neighbors(vertices)
addEdge(T begin, T end, double edgeWeight)	O(n)	
addEdge(T begin, T end)	O(n)	
removeEdge(T begin, T end, double edgeWeight)	O(n)	
removeEdge(T begin, T end)	O(n)	
hasEdge(T begin, T end)	O(n)	
getNumberOfVertices ()	O(1)	
getNumberOfEdges ()	O(1)	
isEmpty()	O(1)	
getVertices()	O(n)	
clear()	O(1)	
getBreadthFirstTraversal(T origin)	O(|V|+|E|)	Where |V| is number of vertices and |E| number of edges.
getShortestPath(T origin, T destination, Stack<T> path)	O(|V|+|E|)	

CLASS (Profile): Profile.java

This class represents the profiles of the users of the SConnect. It has the following attributes:

- name – a String value that represents the full name of the user.
- status – a String that the user uses to specify their status.
- friendProfiles – an arraylist of profiles that stores friends of the user.

It also has the following behaviors:

+Profile(): constructor - initializes all the String attributes to empty strings and a default arraylist.

+Profile(name, status, friendProfiles): constructor - initializes the attributes with the accepted valued.

+Profile(name, status): constructor - initializes the attributes with the accepted valued and the last attribute with a default arraylist object.

+setName(firstName, lastName) - the setter method for the name attribute that accepts the first and last name of the user and set the name attribute with firstName +" " +lastName (Note the space between the two names).

+getName(): - the getter method for the name attribute.

+setStatus(status) and getStatus(): - the setter and getter methods for the status attribute.

+ toString(): String – returns the a string that represents the profile of the user. It displays the string in the following format.

"Name: " + name + "\n\tStatus: " + status + \n\tNumber of friend profiles: " + friend's number + "\n"

+display():void - displays the profile and the friends profiles. Take a look at the sample run to see the format of display.

+getFriendProfiles(): - the getter method for the friendProfiles attribute.

+addFriend(Profile user): void - add a new friend to the friends list.

+unFriend(Profile user): boolean - removes an existing friend from the list of friends. returns true if the removal of the profile is successful, false otherwise.

Method Name	Big-O	Remark
setName(firstName, lastName)	O(1)	
getName()	O(1)	
setStatus()	O(1)	
getStatus()	O(1)	
getFriendProfiles()	O(1)	
addFriend(Profile user)	O(n)	Where n is number of friends
unFriend(Profile user)	O(n)	
toString()	O(1)	
Display()	O(n)	

CLASS (SConnect): SConnect.java

This class is the main class of the social networking app. It has a graph of profiles that are connected together to create the social network of users. Each user is represented by a profile object. It also has the following behaviors:

+ SConnect(): constructor - initializes the social networking app.

+ addUser(Profile p):void - Adds a new user to the social network.

+ removeUser(Profile p):Profile - Removes an existing user from the social network. If the user does not exist, it returns null.

+ createFriendship(Profile a, Profile b):boolean - Creates a friendship between two users on SConnect. If the friendship is created successfully, it returns true, false otherwise.

+ removeFriendship(Profile a, Profile b):boolean removes a friendship between two users on SConnect. If the friendship is discontinued successfully, it returns true, false otherwise.

+hasFriendship(Profile a, Profile b): boolean- Returns true if there is friendship between Profiles a and b, false otherwise.

+traverse(Profile startPoint): void - this method displays each profile's information and friends, starting from the startPoint profile. See the sample run on the format of the display.

+exists(Profile user): boolean- this returns true if a user with the given profile exists in SConnect, false otherwise.

+friendSuggestion(Profile user): List<Profile>- Returns a list of Profiles, who are friends with one or more of the profile's friends (but not currently the profile's friend). It returns null, if the user does not exist or if it does not have any friend suggestions. Take a look at the sample run for an example.

+friendshipDistance(Profile a, Profile b): int- Returns the friendship distance between two profiles. A friendship distance is simply how many profiles away the two profiles are. For example, if a and b are friends their friendship distance is 1. If they have a common friend but they are not friends, their friendship distance is 2. If either of the profiles are not in the social networking app, the method returns -1.

The Big-O of your methods' implementations should be as follows:

Method Name	Big-O	Remark
hasFriendship(Profile a, Profile b)	O(n)	Where n is the number of profiles on SConnect
traverse(Profile startPoint)	**O(n+E)**	Where E is the number of distinct friendships in the networking app
addUser(Profile p)	**O(1)**	
removeUser(Proile p)	**O(n^2)**	
createFriendship(Profile a, Profile b)	**O(n)**	
exists(Profile user)	**O(1)**	
friendSuggestion(Profile user)	**O(n^2)**	
friendshipDistance(Profile a, Profile b)	**O(n+E)**	
removeFriendship(Profile a, Profile b)	**O(n)**	

Requirements

An overview of the requirements are listed below.

- **Implementing the classes** - You will need to implement required classes. The following are list of Java classes and interfaces you may need to use in the project:
  - HashMap
  - Iterator
  - Queue
  - LinkedList/ArrayDeque
  - List
  - ArrayList
  - Stack
- All attributes should be declared as **private** or **protected**.
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods. Check
provided classes for example JavaDoc comments.
- **Big-O** - Template files provided to you contains instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.

Testing

The main method provided in the template file contains useful code to test your project as you work. You can use command like "**java Driver2**" to run the testing defined in **main()**. JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade will be based on automatic grading using test cases made from the specifications provided (for all the classes: Veretex, Graph, Profile, SConnect). Some of the screenshots of the expected output for the command **java Driver2** are given at the beginning of this document.