CSCI241, Winter 2020, Lab 3
Check Canvas for due date and time

Unlike the homework assignments for CSCI241, you are **encouraged** to work with your peers in completing the labs. However, each student must have their own submission; you cannot exchange files. If any of this is unclear, please ask for further clarification.

1 Overview and Goals

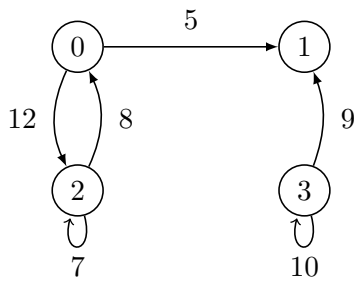
For this lab, you'll implement a Graph class, and you'll write a main routine that adds edges to the graph that are supplied by the user via the command line. Unlike lab 2, this is a linked list implementation of a graph, which also reasons about weights (edge costs).

There are many ways that you can achieve a linked list implementation of a graph. A robust and efficient implementation would use a hash map, a variety of generics, and try-catch blocks, and not to mention Vertex objects with "next" pointers. However, the implementation that you are being asked for in this lab is bare-bones. You will use the Java LinkedList collection, so you don't have to create a linked list from scratch; that way you can focus on the concepts. **Note that for HW2, you are not allowed to us the LinkedList collection**

A screenshot of the behavior of the program is shown in Figure 1, along with a schematic of the graph that is created.

```
How many vertices? 4
Input from, to, then cost. Type "end" to finish
From vertex 0
To vertex 1
Cost 5
From vertex 0
To vertex 2
Cost 12
From vertex 2
To vertex 0
Cost 8
From vertex 2
To vertex 2
Cost 7
From vertex 3
To vertex 1
Cost 9
From vertex 3
To vertex 3
Cost 10
From vertex end
vertex 0 connected to 2 at cost 12
vertex 0 connected to 1 at cost 5
vertex 2 connected to 2 at cost 7
vertex 2 connected to 0 at cost 8
vertex 3 connected to 3 at cost 10
vertex 3 connected to 1 at cost 9
```

(a)



(b)

Figure 1: Example invocation (a), in which the user is prompted to specify how many vertices are in the graph, followed by prompts for the edges. Once the user specifies "end," the graph's edges are printed. The edges input correspond to the graph in (b).

2 Create a new branch

In your local git (log onto a lab linux computer, and using the terminal, navigate to the directory where you have your code for CSCI241 and where you cloned your git repo), create a new branch named *lab3* – remember to make a new branch from the *master* branch only; refer to lab 1 for a refresher – and then create a new directory, *lab3*. Do all of your work in your lab3 directory. Commit and push often.

The commands that you'd issue are something like the following, assuming starting off in your home directory, and that you have a CSCI241 directory that you've created :

```
cd CSCI241
git checkout -b lab3
mkdir lab3
cd lab3
```

Using your favorite IDE (jGRASP, Atom, or just emacs or vim, for example), create a new file, *WeightedDirGraph.java*, and in it create a new class, also named *WeightedDirGraph*. The start of that file should look like the following:

```
public class WeightedDirGraph{
}
```

Add the lab3 directory to your lab3 branch, and also add the *WeightedDirGraph.java* file. Then commit and push to the remote repo. To do that, using the command line, from the CSCI241 directory, you'd issue the following :

```
git add lab3
git add lab3/WeightedDirGraph.java
git commit -m ‘‘added lab3 branch and directory, and Graph skeleton’’
git push origin lab3
```

3 Code Description, screenshot

For this lab, you will write a *main* routine that prompts the user to input graph edges via the command line. The high-level skeleton of your program is the following :

- An `Edge` class*, whose properties are `from`, `to`, and `cost`. Whenever a new edge is created, it is constructed to contain that information.
- A `Graph` class*, whose properties are vertices, and edges in the form of a LinkedList of `Edge` objects. The Graph class also has `addEdge` and `printGraph` functions.
- A `main()` function, that prompts the user to input the count of vertices in the graph, followed by the from (vertex), to (vertex), and cost combinations. Once a user is done providing the graph's details, details of the graph are printed to the screen, via invoking the function `printGraph`.

* You may write your code so that you have separate .java files for the `Edge` and `Graph` classes, but in these instructions, the `Edge` and `Graph` declarations are included in the *WeightedDirGraph.java* file.

The entirety of the skeleton of the code (indicating the exact function signatures that you need), is the following:

```
public class WeightedDirGraph{

  static class Edge{
    public Edge(int from, int to, int cost) {}
  }

  static class Graph{
    public Graph(int numVertices){}
    public void addEdge(int from, int to, int cost){}
    public void printGraph(){}
  }

  public static void main(String[] args) throws IOException {}

}
```

In this program, the values/IDs of the vertices (0, 1, 2, and 3 in Figure 1) are not provided explicitly, but instead correspond to the index values of the array in the Graph that contain the LinkedList objects. For example, if there are 3 vertices in the graph, their IDs would be 0, 1, and 2.

4 File header, Import Statements

The top of your *WeightedDirGraph.java* file should contain your name, date, and a short description of your program. Also include the following imports:

```
  import java.util.LinkedList;
  import java.io.BufferedReader;
  import java.io.IOException;
  import java.io.InputStreamReader;
```

A BufferedReader and InputStreamReader are needed so that your program can prompt for the user's input.

5 The `Edge` class

The `Edge` class should have three properties (instance variables). They are `from`, `to` and `cost`, all of type `int`. The constructor for `Edge` should have three parameters, which are `from`, `to`, and `cost`. Use the arguments that are passed to the constructor to assign the values of the edge properties.

6 The `Graph` class

The `Graph` class is a bit more complex than the `Edge` class, but that's because it contains several functions, in addition to the constructor. The `Graph` class should have 2 instance variables,

`vertices` of type `int`, which is the count of vertices, and `adjList`, which is the array of LinkedLists, composed of `Edge` objects. The `Graph` class variables, and constructor signature, should be the following:

```
int vertices;
LinkedList<Edge> [] adjList;

// constructor
Graph(int numVertices) {}
```

Write the `Graph` constructor so that the `vertices` field is populated, and the `adjList` array is initalized. Each entry of `adjList` should be initialized to be a new `LinkedList` made up of `Edge` objects.

6.1 The `addEdge` function

In the class `Graph`, implement the function `addEdge`, with the following signature:

```
public void addEdge(int from, int to, int cost) {}
```

Write the `addEdge` function so that it creates a new `Edge` object using the arguments that are passed into the variables `from`, `to`, and `cost`. The just-created `Edge` object should be added to the `from` index of `adjList`, using the `addFirst` function that is part of the LinkedList java collection.

6.2 The `printGraph` function

In the class `Graph`, implement the function `printGraph`, with the following signature:

```
public void printGraph(){}
```

The function should iterate over the `adjList`, and print all edge IDs, as well as the costs of each edge. See Figure 1 for the formatting that you should implement.

7 The `main` function

The `main` function should do the following :

- Instantiate a buffered reader
- Prompt the user how many vertices there exist in the graph
- Create an object of type `Graph`, with as many vertices as the user specifies
- Prompt the user to provide from, to and cost numbers. After each input, that edge (and its cost) should be inserted into the graph, using the `addEdge` function.
- If a user inputs *end* for the `from` prompt, the proram should stop prompting the user
- Print the contents of the graph by invoking the `printGraph` function.

8 Compiling and Running

To compile and run your program (which you should do incrementally), you'd issue the following from the command line:

```
javac WeightedDirGraph.java
java WeightedDirGraph
```

Note that depending on which version of Java you are running, you most likely will receive a warning, due to the use of generics and improper casting. That is a topic that you'll learn about when you take CSCI345. For the time being, the warning is okay.

9 Pushing to your git

Add, commit, and push your .java file only, to your lab3 branch. **By convention, compiled code, nor object files, are NEVER (except in special, rare cases, and this is not one of them!) pushed to a git repo**.

```
git add WeightedDirGraph.java
git commit -m "Added this-and-that code"
git push origin lab3
```

Rubric

The *WeightedDirGraph.java* file has been pushed to the lab3 branch, which contains the lab3 directory. Class files (.class) nor anyting other than the .java file, are not allowed to be pushed to the git	2
The asked-for functions (with EXACT signatures, return types, etc.) have been declared.	5
Each of the two classes `Graph` and `Edge`, and asked-for functions, implement the asked-for functionality.	20
The `main` function correctly prompts the user for the edges of the graph (including costs), and the graph's edges are printed after a user inputs *end*.	5
The program compiles via `javac WeightedDirGraph.java`, and upon invocation, exhibits the correct behavior.	5
The code is commented – including your name, date, brief description, etc. at the top of the .java file, variable names are appropriately named, and good coding practices are followed.	3
Total	40 points

To confirm that you've correctly pushed files to your repo, after you push, open your browser and navigate to the gitlab for CS, and double check that all of the files, as well as branch and directory, are there.

Note : It is a good idea to NOT modify the git repo using the web interface. If you do add files, remove files, etc., via the web interface (insteada of via the command line), you'll need to `pull` the changes when logged in to your linux account.