Dr. Hasmik Gharibyan
CSC/CPE 349

Project5: Directed Graphs

Deadlines: demo Part1 and Part2 in person by the end of the lab hour on Friday, 11/30/18.

Part 1 of 4 (25 points):

Note: some *LinkedList* class methods you may need: *add, remove, contains* to add, delete and search for the given item, *size()* to get the length of the list, and *get(int x)* to get the element at *x* index (indexing starts at 0).

1. Define a class *DiGraph* to implement a directed graph as an array of Adjacency Linked Lists.
Vertices are numbered by integers **0 through N-1** (N is the number of vertices in the graph).
DiGraph class contains one *private* instance variable, one constructor and 5 *public* methods:

a) One **private** instance variable: this is an **array** of linked lists (**use Java's *LinkedList* class**).

b) A ***constructor*** with one *int* type parameter for N. This constructor creates and initializes the instance variable-array.

c) A public method ***addEdge(int from, int to)*** where two parameters identify vertices representing the edge that needs to be added to the graph (*to* vertex is added as *from* vertex's neighbor).
Important: the edge should **not** be added if it already exists: **needs to be checked** before adding.
Attention: vertex-numbers are given in **natural** numbering (starting with 1) so you should "turn" them to Java-indexing to reflect correct connection. No need for validity check.

d) A public method ***deleteEdge(int from, int to)*** where two parameters identify vertices representing the edge that needs to be deleted from the graph (*to* vertex is removed as *from* vertex's neighbor).
Note: nothing is done if such edge doesn't exist (no error message or exception).
Attention: vertex-numbers are given in **natural** numbering (starting with 1) so you should "turn" them to Java-indexing to reflect correct connection. No need for validity check.

e) A public method ***edgeCount()***: computes and returns the number of edges in the graph.

f) A public method ***vertexCount()***: returns the number of vertices in the graph (it's the array's length)

g) A public method ***print()*** that outputs the graph in the following format: for each vertex *i* (from 0 to N-1) outputs a line: ***i* is connected to: *x1*, …, *xk*** where *x1,…, xk* are vertices that are adjacent to *i*.
Example: for some 5-vertex graph your output may look like this:

1* is connected to: *2, 4
***2* is connected to:**
3* is connected to: *2, 4, 1
4* is connected to: *3
5* is connected to: *3, 1

Attention: numbering of vertices in the output should be **natural**, i.e. **starting with 1**.

Important: Separate two neighbors with a comma; **do NOT** put comma after the last neighbor

2. Define an application class *DiGraphTest.* In *main* method of this class you need to do the following:

1. Define a scanner object for keyboard input (i.e. connect it to *System.in*). You must make a scanner for **System.in** only once here and use it for all **keyboard** inputs in this *main* method. **DON'T** re-define this scanner or create a new scanner for ***System.in*** – **it will cause problems in grading.**
2. Prompt the user to enter the number of vertices.
3. Input the number of vertices and define an object of *DiGraph* class.
4. Output a menu to the user, listing the above mentioned 5 operations that your *DiGraph* class provides. For example, your menu may look like this:

Choose one of the following operations:
- add edge (enter a)
- delete edge (enter d)
- edge count (enter e)
- vertex count (enter v)
- print graph (enter p)
- Quit (enter q)

Important: must use *a, d, e, v, p, q* letters for menu choices (do NOT change these letters)

5. For as long as the user does not choose to quit, do the following:
   - Prompt the user to enter a menu choice (do **NOT** print the menu here; the menu should be output **only once,** in step 3 above, **before** entering this loop).
   - Input user's choice and analyze it (you are recommended to use a *switch* statement with *default* case which will be executed for invalid menu choices). Arrange the execution of the requested operation; if input values are necessary (for *add edge* and *delete edge* menu choices), prompt the user to enter two integers and input them (no need for validity check). For every request, after executing it, **output feedback**: e.g. if you are adding/deleting an edge, output a message saying for example that the edge (x, y) was added/deleted; if you are giving some returned answer (e.g. number of edges), precede it with an informative message (e.g. "Number of edges is: "); or before printing the graph, output a sentence "The graph is the following:"

   Note: in all communications with the **user** the vertex numbering is **natural**, i.e. **starting with 1**.

3. Edit, compile and execute *DiGraphTest* class, test it thoroughly. When running the program, first provide the graph by inputting N (number of vertices) and then **one by one adding vertices** (with 'a' menu option). Extensively test the functionality of your program: request different menu choices and check their execution; then add and delete edges to change the graph and repeat the testing for the modified graph. Finish testing when you are absolutely sure all the functionality of the program is correct.

Part 2 of 4 (15 points):

1. Make additions in the *DiGraph* class to include the implementation of the Topological Sort algorithm including a supporting routine for computing vertex *indegrees*.

Add the following content to the *DiGraph* class's definition.

a) A ***private*** method ***indegrees***: returns an array of integers representing the indegrees of all vertices in the graph: the i-th integer in the resulting array is the indegree of the i-th vertex.

b) A *public* method ***topSort***: returns an array containing the list of topologically sorted vertices (values in the array should represent **natural** vertex-numbers, i.e. **starting with 1**).
Important: If the graph is **cyclic**, this method must throw *IllegalArgumentException* type exception (read the *note* on top of the last page of your *Topological Sort* lecture handout).

Attention: In *topSort* you need a regular queue (**not** a priority queue). To implement a queue, in Java you can define an object of ***LinkedList*** class (the list is for integers). Your list will function like a regular queue if you always add an element to the end of the list (*addLast* method) and delete an element from the front of the list (*removeFirst* method).

2. Make additions to the *DiGraphTest* class's *main* method to incorporate one additional service.

Add one **new** menu-choice for outputting the topologically sorted list of vertices of the graph and arrange the execution of that **new** service.

Attention: 1. Use ***t*** letter for the new menu choice (***t*** - **t**opological sort).
2. Numbering of vertices in the output should be **natural** (**starts with 1**).
3. The list should be output on **one** line, with vertex-numbers separated by **commas.**
Important: there should **NOT** be a comma after the last vertex.

3. Edit, compile and execute *DiGraphTest* program, testing the new service very thoroughly.

DEMO: you will get demo instructions indicating how to run your program and with what input. Run your program according to these instructions, make sure all answers/outputs are correct, and then put your name on the board. I will approach you to check your work.

Attention: 1. **Demo early** (as soon as you finish the assignment) to avoid last-day commotion.
2. Make sure **both partners are in attendance** at the time of the demo.