2018-2 Data Structures Final Examination (Programming Part)

This part will carry 40% of the total final exam grade.

You have limited access to (a) lecture notes, either at eTL or in printed form and (b) uploaded codes in eTL. You may not refer to any personal codes saved in your account and/or machine, but you can use your submitted lab and/or assignment and/or midterm codes. Otherwise, accessing the Internet (including access to the Java API documents) is strictly prohibited.

Occasionally save your work. Proctors are not responsible for any damages that may incur to your written codes.

Write everything, including comments, in English. Using any other language than English might involve compile errors. Note that you cannot get any points if your code does not compile.

Do not change the format of input and output. You will get severe penalties if you do not follow the input and/or output specification.

If a specific problem/task includes constraints on `import`s, you need to follow those.

Remove any statements in your code that include `package`. In most cases, they will trigger compile errors.

Upload your work in eTL. For questions that lets you write something, type your answer in the description part inside the submission screen. For codes, you should submit only a single TAR or ZIP file containing those files:
[1] Problem 1: LZW.java .
[2] Problem 2: MathTree.java .
[3] Problem 3: GraphAdjacency.java .
Do not include any subdirectories or any other files inside your archive, so that we can see your source codes right after untar/unzipping it.

Proctor will open a Q&A session about 10 minutes after the exam begins. Try to read and understand the questions before that time.

You need to finish your work no later than 8:15PM. Submission due is 8:20PM. No late submission is allowed.

Grading will be done in Linux environment using Java (OpenJDK) 11, identical to that inside the lab machines. Keep that in mind when writing code in other environments.

You may not proceed before the proctor allows you to do so.

Problem 1

Write a Java class `LZW` (in LZW.java) that performs decoding of a text file.

- This class gets single command line parameter: location of a text file. Write a `main(String[] ar)` method that handles those parameters.
Graders will run something like java LZW ./encoded.txt .
- Run `decodeLZW()` method inside `main(String[] ar)` that performs LZW decompression. Input file consists of a single line, where a list of codewords exists. Each codeword is separated by a comma(‘,’). You may assume that the original content contains only lower-case alphabets. Thus, the initial code table is as follows:

code	0	1	2	...	25
key	a	b	c	...	z

Decoded text should be printed out to the console. For example:

cat ./encoded.txt
```
0,1,26,26,27,27,29,32,32
```
java LZW ./encoded.txt
```
abababbabaabbabbaabba
```

- You may refer to the notes of the Lecture 12 for details of LZW algorithms.
- You do not need to consider erroneous parameters and/or inputs.

Problem 2

Write a Java class `MathTree` (in MathTree.java) that converts (a) an infix expression to postfix version and (b) a postfix expression to infix version.

- Your program starts with reading inpostfix.txt. Each line contains the type of an expression (either i (infix) or p (postfix)), followed by the actual expression. When an in(post)fix expression is given, you need to convert it to the post(in)fix version.
- Each expression uses any of the four operators: +, -, * and /. It uses 1 to 26 variables in upper-case alphabets.
- When the conversion is complete, your program should print out the result to outfix.txt in this format:

postfix/infix[blank]expression

for example:

postfix A B * C +

- You need to write `main(String[] ar)`, `toInfix()` and `toPostfix()` methods.

cat ./inpostfix.txt

```
p A B C * +
i A * B + C
p E D + A B - * C G + /
```

java MathTree

(Nothing is output to the console.)

cat ./outfix.txt

```
infix A + B * C
postfix A B * C +
infix (E + D) * (A - B) / (C + G)
```

- You may refer to the notes of the Lecture 13 for details of infix/postfix notations and trees.
- You do not need to consider erroneous parameters and/or inputs.

Problem 3

Write a Java class `GraphAdjacency` (in GraphAdjacency.java) that loads a graph from a text file and constructs (a) an adjacency matrix and (b) an adjacency list representations of the graph.

- This class gets one command line parameter: location of a text file. Write a `main(String[] ar)` method that handles this parameter.
- Input text file contains two parts. First part is a single line, with `numVertex`, `numEdge` and `isDirected` (0 if undirected, 1 if directed) separated by a comma(‘,’). Second part consists of `numEdge` number of lines, where each line specifies a single edge. A vertex has its identifier from 1 to `numVertex`. An edge is denoted by three elements `from`, `to` and `weight` separated by comma(‘,’)s. Two example text files:

3,3,0 1,2,4 3,2,5 1,3,2	This is an undirected graph with 3 vertices and 3 edges. An edge from vertex 1 to 2 (2 to 1) exists, with its weight 4. An edge from vertex 3 to 2 (2 to 3) exists, with its weight 5. An edge from vertex 1 to 3 (3 to 1) exists, with its weight 2.	3,3,1 1,2,4 3,2,5 1,3,2	This is a directed graph with 3 vertices and 3 edges. An edge from vertex 1 to 2 (NOT 2 to 1) exists, with its weight 4. An edge from vertex 3 to 2 (NOT 2 to 3) exists, with its weight 5. An edge from vertex 1 to 3 (NOT 3 to 1) exists, with its weight 2.

All of `numVertex`, `numEdge` and `weight` will not exceed range of `int`.
- Write two functions `toAdjacencyMatrix()` and `toAdjacencyList()` and invoke them inside `main(String[] ar)` method. An adjacency matrix is constructed in either `ArrayList<ArrayList<int> >` or `Vector<Vector<int> >`, whereas an adjacency list of a vertex is considered as either `ArrayList<int>` or `Vector<int>`.
- After invoking those two functions, display the constructed representations to the console. This should also be done in `main(String[] ar)` method.
- Afterwards, get an user’s input from the console inside `main(String[] ar)` method using `Scanner`. Until the user enters `q`, you should continuously get inputs from the console. When two numbers are given as an input, determine whether an edge exists. If it exists, then print its weight, otherwise print a warning. Two example cases:

cat ./und.txt 3,3,0 1,2,4 3,2,5 1,3,2 **java GraphAdjacency ./und.txt** Matrix 0,1,1 1,0,1 1,1,0 List 1- 2,3	**cat ./dir.txt** 3,3,1 1,2,4 3,2,5 1,3,2 **java GraphAdjacency ./dir.txt** Matrix 0,1,1 0,0,0 0,1,0 List 1- 2,3

2- 1,3 3- 1,2 Input: **1** Input: **2** An edge exists. Weight: 4 Input: **3** Input: **1** An edge exists. Weight: 2 Input: **2** Input: **2** No edge exists. Input: **q**	2- none 3- 2 Input: **1** Input: **2** An edge exists. Weight: 4 Input: **3** Input: **1** No edge exists. Input: **2** Input: **2** No edge exists. Input: **q**

- You may refer to the notes of the Lecture 19 for graph representations.
- You do not need to consider erroneous parameters and/or inputs.

Challenge Problem

This is not a mandatory task. Proceed if you have extra time upon completion of all three problems.

Extend class `LZW` (in LZW.java) to support encoding of a text file.

- Write `encodeLZW()` method that performs LZW compression. For maintaining dictionaries, you may use `HashMap<K, V>`.
- You may assume the same criteria as those mentioned in **Problem 1**.