Stacks and Queues:
Implementation of Abstract Data Types and applications

Part A

You are given the interfaces StringStack and StringQueue that declare the basic methods for a stack and a FIFO queue, with elements of type String. Create an implementation of the ADT StringStack ʁand StringQueue, i.e. write 2 classes implementing the 2 interfaces.

Implementation Instructions:

- Your classes should be called StringStackImpl and StringeueImpl.
- The implementation for the 2 interfaces should be done using a singly-linked list.
- By performing an element insertion or extraction operation (i.e. executing the push and pop methods on the stack, and put and get in the FIFO queue) should be completed in O(1) time, i.e. in time independent of the of the number of objects in the queue. Similarly, the size interval should be executed in O(1).
- When the stack or queue is empty, the read inputs from the structure should throw exception of type NoSuchElementException. The NoSuchElementException exception belongs to the core library of Java. Import it from the java.util. Do not build your own exception.
- You can use the singly-linked list presented in the tutorial in the course, or you can write your own list from scratch, or use only Node-type counterexamples within the stack/queue class. To get the best possible practise, we suggest that you write your own classes from scratch(you certainly won't lose any units, but you can use whatever you have seen in the laboratory).
- Optional: you can write your implementation using generics to allow you to handle stacks and queues with any type of object.
- You are not allowed to use existing implementations of list, stack, queue type structures from Java library (e.g. Vector, ArrayList, etc.)

Part B

Using the implementation of the stack from Part A, write a client program that will traverse a maze with the goal of finding the exit. Your program should take as input a .txt file, which should contain the maze in the form of a character table of dimensions n × m, where n, m, integers. The matrix may contain only the 0, 1, and E characters, where E is found only in a point in the matrix denotes the entrance to the labyrinth. When you enter the maze, you can move horizontally or vertically (but not diagonally) to any direction containing 0 (see the example below). If you reach the border of the matrix (first or last line and first or last column), and find 0, then you have reached an exit of the maze. It is possible to have multiple exits in the maze (or none). Your program should print the coordinates of the output that it found, and if there is no way out, it should print a message to that effect.

Example: The input would be a file in the form of the following example:

```
9 7
0 3
1 1 1 E 1 1 1
1 1 1 0 1 1 1
1 0 0 0 1 0 1
1 0 1 0 1 0 0
1 1 1 0 1 1 1
1 0 0 0 0 0 1
1 0 1 1 1 0 1
1 0 1 1 0 0 1
0 1 1 1 0 1 1
```

The first line indicates the dimensions of the labyrinth (here n = 9, m = 7). The second line are the coordinates of the entry point, where here they are on line 0 and in column 3 (we assume that we denote the lines from 0 to n - 1 and the columns from 0 to m - 1). In this example, the exploration will unfold moving downwards. If you turn left (as we look at the table), you will see that you will reach you'll soon reach a dead end and you'll have to turn back. Eventually, by continuing to search, an exit that you can is at coordinates (8, 4).

Implementation Instructions:

- Your program should be called Thiseas.java.
- You should make use of the stack implementation from Part A. Use of the stack is recommended to enable you to implement the search for output with backtracking. Consider what you need to do when you reach a dead end, and how you can continue the search.
- To test the orderliness of your program, it is advisable to make a number of different labyrinths with different characteristics (e.g. with multiple outputs, with no output, with larger dimensions, etc.) and run your code with these inputs.
- It is permissible, if you define the labyrinth into a character table, to make subsequent changes on the elements of the matrix (e.g. if you want to use another chart to show that you have already returned to some position during the duration of the programme). Beyond that, however, you should use the stack in Part A accordingly.
- Your work will be tested on inputs of the above form. If there is any error in the data input data, the program should terminate by printing a corresponding message (e.g. if the rows are columns read in the first input row do not match the table read afterwards or if there is no E in the maze, etc.).
- You must give the entire path for the .txt input file as the argument, e.g., if you run it from line command line, the execution of the program will be as follows:

```
> java Thiseas path_to_file/filename.txt
```

Part C

The implementation of the StringQueue interface with a singly-linked list in part A, must use 2 variables as pointers at the head and tail, so that you can correctly import and export data, as we saw in the example. The task in part C is to build a new implementation of the FIFO queue, using only one of these indexes. Hint: Use a cycle list instead of a singly-linked list.

Implementation Instructions:

- Your class should be called StringQueueWithOnePointer.java.
- The import and export operations should be done in O(1) and in general all the instructions given for Part A, apply here as well.