CS 300
Data Structures
Homework 1

Assigned: October 16, 2017
Due: October 25, 2017 at 11:55pm

PLEASE NOTE:

- **SOLUTIONS HAVE TO BE YOUR OWN. NO COLLABORATION OR COOPERATION AMONG STUDENTS IS PERMITTED.**
- **10% PENALTY WILL BE INCURRED FOR EACH DAY OF OVERTIME. SUBMISSIONS THAT ARE LATE MORE THAN 3 DAYS WILL NOT GET ANY CREDITS.**
- **SUBMISSIONS WILL BE MADE TO THE SUCORSE SERVER. NO OTHER METHOD OF SUBMISSION WILL BE ACCEPTED.**

In this homework, you will learn to navigate through a maze using stacks. You may naturally ask, “What do stacks have to do with mazes?” Well, read on!

A maze consists of a series of rooms connected with corridors. You will be given an entry point to enter the maze. Your task is to come out of the maze from the other end. The maze will be represented with a two-dimensional *n* by *m* array of integers where 0s indicate regions that you can move through (rooms, corridors, whatever) and 1s indicate walls. You will navigate through the maze by starting at the given entry point. As you are moving, if you get to a point on the boundary other than the entry point, then you are done. Otherwise, you should remember all your suitable neighbors (left, right, up and down), by pushing their coordinates on to a stack (unless you have good memory!) If you find yourself completely blocked, then pop the top of the stack. You are then magically “teleported” back at some point that you can move from. You can continue searching for other neighbors at this point. Obviously, we have left a lot of the details out. Feels free to make any decisions on any details you feel are appropriate.

The coordinate system of the maze is such that top-left corner will be point (0, 0) **with coordinates increasing as you move right and down**. Note that we use the first coordinate (x) to indicate the rows going down and the second coordinate (y) to indicate the columns going right.

The size of the maze, the coordinates of the start point and maze structure will be read from the standard input. The first number in the input indicates the number of rows and the second one indicates the number of columns. The second pair of numbers will give you the coordinates of the start or entry point. After that, a row-by-row representation of the structure of the maze will be given. For instance the input

```
10 10
0 4
1 1 1 1 0 1 1 1 1 1
1 0 0 1 0 1 0 0 0 1
1 0 0 0 0 0 0 1 0 1
1 0 1 1 1 1 0 1 0 1
1 0 1 0 0 1 0 1 0 1
1 0 1 1 1 1 0 0 0 1
1 1 1 0 1 0 0 1 0 1
0 0 1 0 1 0 1 1 0 0
1 0 1 0 0 0 0 0 0 1
1 0 1 1 1 1 1 1 1 1
```

describes a maze of 10 rows by 10 columns whose start point is at coordinates (x=0, y=4).

Your code will first print the input data to the standard output as shown above and will then produce the following output showing path from the entrance point to an exit point in terms of pairs of numbers, giving coordinates of the points on the path.

For the example above, your code should produce the following output (for the path highlighted with boldface above) after the input is printed:

```
The solution to the puzzle is:
0 4
1 4
2 4
2 5
2 6
3 6
4 6
5 6
5 7
5 8
6 8
7 8
7 9
```

There **can** be more than one path from entrance to exit. You need to output just one of the correct paths. Obviously, the path should not contain any repetitions. You can assume that the puzzles that will be given will always have a solution.

Here is one solution that we suggest that you may use. You keep a stack of maze coordinates which means that you will need to define a very simple class `MazeCoordinate` to store coordinate data. You will need methods to construct a `MazeCoordinate` from given x and y coordinates, get the x and y coordinates of a

`MazeCoordinate`, find the list of neighbors of a `MazeCoordinate` given the boundary of the puzzle (note that a `MazeCoordinate` can have at most 4 neighbors). We are sure that you can easily write this class with any additional methods you feel are necessary. **Furthermore, we ask you to implement the stack data structure by using linked lists**. **You need to use your own implementation of linked lists.**

Here is a crude description of how you can proceed: You start by pushing the entry point to the stack to kick off the process. Then, you will go on as follows: If you have not already found an exit point, pop a point from the stack and mark it as **visited** (so you should figure out a way to remember if you have visited a maze point). Check if this point is a boundary point and NOT the same as the entry point. If so, then you have a solution. You can print the path you have been keeping track of (you will need a separate stack to keep track of the path) and exit. If not create a list of all the neighbors with a 0 which have NOT been visited before and push them in some order to the stack and repeat.

For the example above, initially the stack will contain (0, 4); you pop that and mark it as visited. You then push (1, 4) as the only neighbor that has a 0 and is not visited. You then pop (1, 4) mark it as visited and push (2, 4) as the only neighbor that has a 0 and is not visited. You then pop (2, 4) and mark it as visited and push (2, 3) and (2, 5) as the neighbors that have a 0 and are not visited. You proceed in this manner. If you find you have no suitable neighbors (for instance if you ever come to (5, 1) there are no suitable neighbors), you can not push anything to the stack since you are stuck. But that is OK, you just pop a new point from the stack (since you are guaranteed to have a solution, there will always be an alternate point on the stack). Eventually (7, 9) should be popped from the stack and will be noted to be a boundary cell different from the input point and the solution will be found. One thing that is not detailed here is how you keep track of the actual path and how to print in the right order. You should figure these out. I very strongly suggest you solve the maze above (or some smaller one) by hand, but simulating the stack so you understand the whole process.

Your code should be submitted to SUCOURSE at the deadline given on the first page. You should follow the following steps:

- Name the folder containing your source files as *XXXXX-NameLastname-hw4* where *XXXXX* is your student number. Make sure you do NOT use any Turkish characters in the folder name. You should remove any folders containing executables (Debug or Release), since they take up too much space.

- Use the Winzip or something equivalent to compress your folder to a compressed file named, for example, 0*5432-AliMehmetoglu-hw4.zip*. After you compress, please make sure it uncompresses properly and reproduces your folder exactly

- You then submit this compressed file in accordance with the deadlines above.

Your homework will be graded in the following way:

- If the source code does NOT compile you get 0 points and that is it.
- We will run 5 tests on your homework and check the results. The test mazes will range from small (say 5 by 5) to medium size (say 15 by 15) mazes. Each correct test will earn you 18 points. **Note that your outputs should be in the exact same format we described above. Any deviation will cause the test to be considered to fail.**
- The style of your OWN code will be graded on clarity, commenting, etc. This will cover 10 points. You will however NOT get this credit if your program fails in ALL the tests we perform. A nice looking but useless program is just a useless program not matter how nice it is.

Good luck!