CS 171: Introduction to Computer Science II
Assignment #4: Maze Traversal
(Due: Thursday, March 1st at 11:59 PM on Canvas)

Goals

This assignment practices the use of Queues and Stacks. You will use links to store and construct paths and implement a path finding algorithm to solve the maze search problem.

Problem and Requirement

You are to implement two versions of a path finding algorithm for finding a path through a square maze: one version will use a stack, and the other will use a queue to store a list of positions to explore as the search algorithm proceeds. A maze is a square space with an entrance at the upper left-hand corner, an exit at the lower right-hand corner, and some internal walls. Your algorithm will find a path through a given maze, starting from the entrance and finishing at the exit that does not go through any walls (a path must go around walls).

Your program will take a command line argument, the filename (e.g. maze1.txt) for the maze description. (To input a command line argument in Eclipse or IntelliJ, you can click "Run – "Run Configurations – "Arguments" and then type in the argument in the "Program Arguments box.) Your program next prints the maze to stdout, then tries to find a path through the maze. If a path is found, the positions in the path are printed to stdout with a success message. Otherwise, an error message is printed to the screen. Your program will print the paths that result from executing both implementations of the path finding algorithm.

The maze will be represented as an $N \times N$ array of 1s and 0s; if `maze[i][j] == 1` then there is an internal wall in that position of the maze. Otherwise, there is no wall. The search algorithm will start at `maze[0][0]` and find a path to `maze[N-1][N-1]`. A path is represented by a sequence of `[i][j]` position coordinates, starting with `[0][0]` and ending with `[N-1][N-1]`. From `position[i][j]` in the path, the next position in the path can only be the position to the left, right, up or down from `position[i][j]`; a path cannot move diagonally from one position to another. For example, the following is the array representation of a 10×10 maze:

```
ENTER --> 0    1    1    1    0    0    0    0    0    0
          0    0    0    1    0    0    0    1    0    0
          0    1    0    1    1    0    0    1    0    0
          0    1    0    0    1    0    1    1    0    0
          0    1    0    0    1    0    1    1    0    0
          1    1    1    0    1    0    1    0    0    0
          0    0    1    0    0    0    1    0    1    1
          0    0    1    0    0    0    1    0    1    1
          0    1    1    0    1    0    1    0    0    0
          0    0    0    0    1    0    1    1    0    0 -->  EXIT
```

The following path is possible through this maze:

```
Path: ([0][0], [1][0], [1][1], [1][2], [2][2], [3][2], [3][3], [4][3], [5][3],
[6][3], [6][4], [6][5], [5][5], [4][5], [3][5], [2][5], [2][6], [1][6], [0][6],
[0][7], [0][8], [1][8], [2][8], [3][8], [4][8], [5][8], [5][7], [6][7], [7][7],
[8][7], [8][8], [8][9], [9][9])
ENTER  --> X    1    1    1    0    0    X----X----X    0
            |                           |         |
           X----X----X    1    0    0   X    1    X    0
                 |                       |         |
            0    1    X    1    1   X----X    1    X    0
                 |              |              |
            0    1   X----X    1    X    1    1    X    0
                          |         |              |
            0    1    0    X    1    X    1    1    X    0
                     |              |              |
            1    1    1    X    1    X    1   X----X    0
                     |              |        |
            0    0    1   X----X----X    1    X    1    1
                                         |
            0    0    1    0    0    0    1    X    1    1
                                             |
            0    1    1    0    1    0    1   X----X----X
                                                       |
            0    0    0    0    1    0    1    1    0    X --> EXIT
```

Program Operation

Your program will perform the following actions:

1. The name of the maze file from command line.

2. Use the provided method readMaze to read the maze into the array representation and print the maze to stdout.

3. Next, your program will search for a path from the maze entrance point to the exit point using both versions of the path searching algorithm: `stackSearch` and `queueSearch`.

4. For both algorithms, your program will either print an error message (if there is no path through the maze) or will print:

   (a) The path as a list of `[i][j]` positions, starting at the entrance point and ending at the maze exitpoint.

   (b) The maze with the path coordinates indicated by 'X' characters.

The Path Finding Algorithm

You should implement the following algorithm for finding a path (some of the details are left for you to fill in):

In one version of the algorithm, you will use a queue of Position objects to enqueue neighbor elements and to dequeue the next element at each step. The other version will use a stack to push neighbor elements and to pop the next element at each step. Once you implement one version of

---

Create a search list for positions yet to explore, add the entrance position, (0,0) to the search

while the list is not empty **do**

    Remove the next position from the search list

    if it is the exit position `[n-1][n-1]` **then**

        a path is found, construct the path and return the path

    else

        mark the position as visited, add all valid up, down, left, or right neighbor positions to the search list

    end if

end while

if the list is empty and the method has not returned **then**

    there is no path

end if

---

the algorithm, you can implement the second version by simply copying the code to a new method and replacing the data structure use and the calls to enqueue/dequeue with push/pop.

To construct the path, you need to keep track of the previous position (parent) that leads to each position on the explored paths. The Position class has been defined for you, which contains the position information and a reference to its previous (parent) Position on a path. This is similar to the Node class in the LinkedList examples we have learned which contains a data item and a reference to its next Node. When adding a neighbor position of the current position to the search list, you need to create a new Position object or the neighbor position to be added with a reference to its parent – note that the current position is in fact the parent of the neighbor position. When the exit position is reached or a path is found, you can follow the previous (parent) links from the exit position to construct the path.

Think carefully about when a neighbor is valid and how you can ensure that your implementation terminates.

Important note for grading When adding, positions to the stack/queue, follow this order:

Down, up, right, left.

So if you start at grid[0][0] = (0,0), you would add:

(1,0) (-1,0) (0,1) (0,-1) Of course you need to make sure these are valid.

Data Structures

You can use `java.util.ArrayDeque` class for storing the search list. See the lecture notes and the Java APIf or examples of usage and documentation. The maze is stored as a 2-dimensional array of char values defined as `char[][]`. You can see how to declare, initialize, and use 2-dimensional arrays in the provided method `readMaze`.

Getting started

1. Download the starter code on Piazza and understand it. It has the following methods that are already implemented:

- `char[][] readMaze(String filename)`: reads maze from file, returns 2-dimensional array with each entry containing a 0 or a 1(wall)

- `printMaze(char[][] maze)`: prints a visual representation of the 2D maze array to stdout

- `main()`: reads and prints a maze, finds a path using `stackSearch` (to be implemented) and `queueSearch` (to be implemented), prints the path using `printPathmethod` (to be implemented)and prints the maze for each version of the algorithm. **DO NOT** change the `main` method.

2. Fill in your implementation for the `stackSearch`, `queueSearch` and `printPath` methods.

3. Test your program with the provided mazes (`maze1.txt`, `maze2.txt`, `maze3.txt`, `maze4.txt`)

4. Think about the following question (you do not need to submit the answers)

   - Why do both versions of the algorithm work?

   - Why do they sometimes output different paths for some of the mazes?

Submission Instructions

The programming assignment must be saved in your cs171 folder. From a terminal, type the commands

```
>cd cs171
>/home/cs171003/turnin PathFinder.java hw4
```

Grading

- If your program does not compile, you will get 0 points

- Correct usage of `ArrayDeque` to implement the `stackSearch` algorithm: 25 points

- Correct usage of `ArrayDeque` to implement the `queueSearch` algorithm: 25 points

- `stackSearch` algorithm correctly finds the path or returns `null` if no path exists: 15 points

- `queueSearch` algorithm correctly finds the path or returns `null` if no path exists: 15 points

- The path to exit (if any) is correctly marked in maze and printed: 15 points

- Coding style: 5 points

Honor Code

The assignment is governed by the College Honor Code and Departmental Policy. Please remember to have the following comment included at the top of the file.

```
/*
THIS CODE WAS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING
CODE WRITTEN BY OTHER STUDENTS. _Your_Name_Here_
*/
```