Rat in a Maze

Introduction

A **maze** is a rectangular area with an entrance and an exit. The interior of the maze contains walls or obstacles that one cannot walk through. In our mazes these obstacles are placed along rows and columns that are parallel to the rectangular boundary of the maze. The entrance is at the upper-left corner, and the exit is at the lower-right corner.

Suppose that the maze is to be modeled as an $\boldsymbol{n} \times \boldsymbol{m}$ matrix with position $(\boldsymbol{i}, \boldsymbol{j})$ of the matrix representing the entrance and position $(\boldsymbol{k}, \boldsymbol{l})$ representing the exit. Each maze position is described by its row and column intersection. The matrix has a # if and only if there is an obstacle at the corresponding maze position. Otherwise, there is a dot at this matrix position. The following grid of sharp signs (#) and dots (·) is a double-subscripted array representation of a maze.

The *rat in a maze* problem is to find a path from the entrance to the exit of a maze. A path is a sequence of positions, none of which is blocked, and such that each (other than the first) is the north, south, east, or west neighbor of the preceding position.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Problem Statement

Design and implement an application **solveMaze.java** to solve the rat in a maze problem. You may assume that the mazes for which your program is to work are square (i.e., $m = n$) and are sufficiently small that the entire maze can be represented in the memory of the target computer. Your program must use the **LinkedStack.java** class that we have developed in class.

Design Remarks

- In the preceding double-subscript array, the sharp signs (#) represent the walls of the maze and the dots (.) represent squares in the possible paths through the maze. Moves can only be made to a location in the array that contains a dot (.).
- There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming there is an exit). If there is not an exit, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the maze to the right. As long as you do not remove your hand from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.
- Another useful algorithm that relates to programming is as follows. Whenever you have options to move in more than one direction, push all but one of the options onto the stack and then go in

the direction you did not push. When you run into a dead end, walk backwards to your last option (i.e. pop the stack) and proceed from there.

- Below is another algorithm (**FindPath Algorithm**) to find a path from $(1,1)$ to (m, m):

```
Algorithm: FindPath
// Find a path from (1,1) to the exit (m,m).
// Initialize wall of obstacles around the maze

// Initialize variables to keep track of our current
// position in the maze
Position here;
here.row = 1;
here.col = 1;

// Prevent return to entrance
maze[1][1] = 'X';

// Search for a path to the exit
while( not at exit) do {
     find a neighbor to move to;
     if( there is such a neighbor)      {
         add position here to path stack;
         // Move to and block neighbor
         here = neighbor;
         maze[here.row][here.col] = 'X';
     }
     else      {   // Cannot move forward, backup
         if( path empty) return false;
         back up to position here which is at top of path stack;
     }
     return true;
}
```

Program Input

There is no input for this program. Your program randomly generates a maze and then traverses the generated maze. This can be done in two steps:

- You write a **mazeGenerator** method that randomly produces a maze. The method should also provide the starting and ending locations of the maze.
- Then you write a **mazeTraverse** method to walk through the maze.

Program Output

- As method **mazeTraverse** attempts to locate the exit from the maze, it should place the character **X** in each square in the path.
- Your program should display the maze after each move so the user can watch as the maze is solved. This should look like an animated picture.

Submission Instructions

When you are satisfied with your implementation, and after testing it thoroughly, submit the **solveMaze.java** file via Blackboard Learn.