Project: A-Maze-ING Stack and Queue

In this project, you will implement the Stack interface using a LinkedList to generate mazes and the Queue interface using a LinkedList to solve mazes. You will do so without using the java.util library, which means, you will have to do ALL the implementation work yourself.

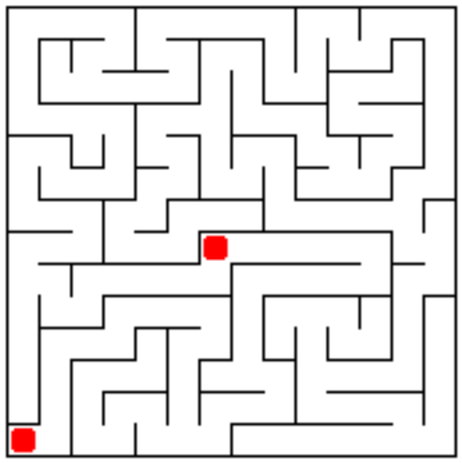
Due Data: The project is due on Tuesday, December 5th at 2355

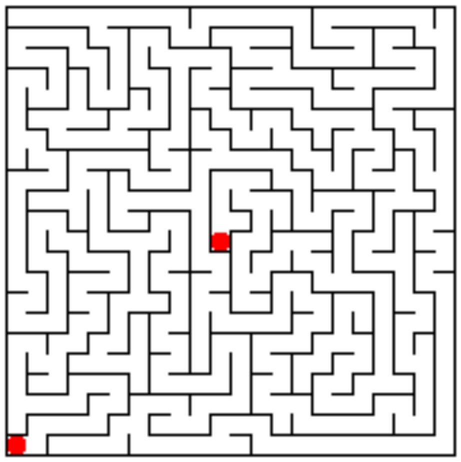
Project Presentation Day: On Wednesday, December 6th during class time.

Supporting files: All supporting files are uploaded into project directory. Extract stdlib.jar file using following command then update Maze.java in the same directory.

```
jar xf jar-file
```

Project Description: Generate a perfect maze like the following two





Write a program Maze.java that takes a command-line argument n, and generates a random n-by-n perfect maze. A maze is *perfect* if it has exactly one path between every pair of points in the maze, i.e., no inaccessible locations, no cycles, and no open spaces. Here's a nice algorithm to generate such mazes. Consider an n-by-n grid of cells, each of which initially has a wall between it and its four neighboring cells. For each cell (x, y), maintain a variable north[x][y] that is true if there is wall separating (x, y) and (x, y + 1). We have analogous variables east[x][y], south[x][y], and west[x][y] for the corresponding walls. Note that if there is a wall to the north of (x, y) then north[x][y] = south[x][y+1] = true.

Construct the maze by knocking down some of the walls as follows:

a. Start at the lower level cell (1, 1).
b. Find a neighbor at random that you haven't yet been to.
c. If you find one, move there, knocking down the wall. If you don't find one, go back to the previous cell.
d. Repeat steps ii. and iii. until you've been to every cell in the grid.

Hint: maintain an (n+2)-by-(n+2) grid of cells to avoid tedious special cases.

Getting out of the maze: Given an n-by-n maze (like the one created in the previous exercise), write a program to find a path from the start cell (1, 1) to the finish cell (n, n), if it exists. To find a solution to the maze, run the following algorithm, starting from (1, 1) and stopping if we reach cell (n, n).

Solving a Maze with a Queue: The process of solving a maze with queue is a lot like "hunting" out the end point from the start point. To do that, we must ensure that we test all possible paths. The queue will allow us to track which spaces we should visit next.

The basic routine works as follows:

- Initialize a queue with the start index (0,0)
- Loop Until: queue is empty
  - Dequeue current index and mark it as visited
  - If current index is the finish point
    - Break! We've found the end
  - Enqueue all indexes for reachable and unvisited neighbors of the current index
  - Continue the loop.

Looking over this routine, imagine the queue, full of spaces. Each time you dequeue space you are searching around for more spaces to search out next. If you find any, you add them to the queue to eventually be analyzed later.

Because of the FIFO nature of a queue, the resulting search pattern is a lot like a water rushing out from the start space, through all possible path in the maze until the end is reached.

Drawing a Maze with a Stack: The process of drawing a maze is very similar to solving a maze, but instead the goal is to explore all spaces, removing walls along the way, such that there exist a path between any two spaces through a series of reachable spaces. That also means that there will exist a path from the from the start to the end.

The algorithm to ensure all reachability will work as follows:

- Initialize a stack with the start index (0,0)
- Loop Until: stack is empty
    - pop current index off the stack and mark it as visited
    - If current index has any unvisited neighbors
        - Choose a random unvisited neighbor index
        - Remove the wall between the chosen neighbor index and the current index
        - push the current index on the stack
        - push the randomly choose neighbor index on the stack
    - Continue the loop.

Looking over this routine, you should be able to invision how this procedure will dive through the maze randomly, like a snake, until the snake reaches a dead end (a space without any unlisted neighbors). The exploration of the snake would then have to backtrack until there is a space with unvisited neighbors and explore from there. Eventually, the snake will have explored the entire maze, at which point, you're done.