BIRZEIT UNIVERSITY

Department of Computer Science

COMP232/242/2321 (First Semester – Fall 2016/2017)

Project#2 Due Date: 30/11/2016

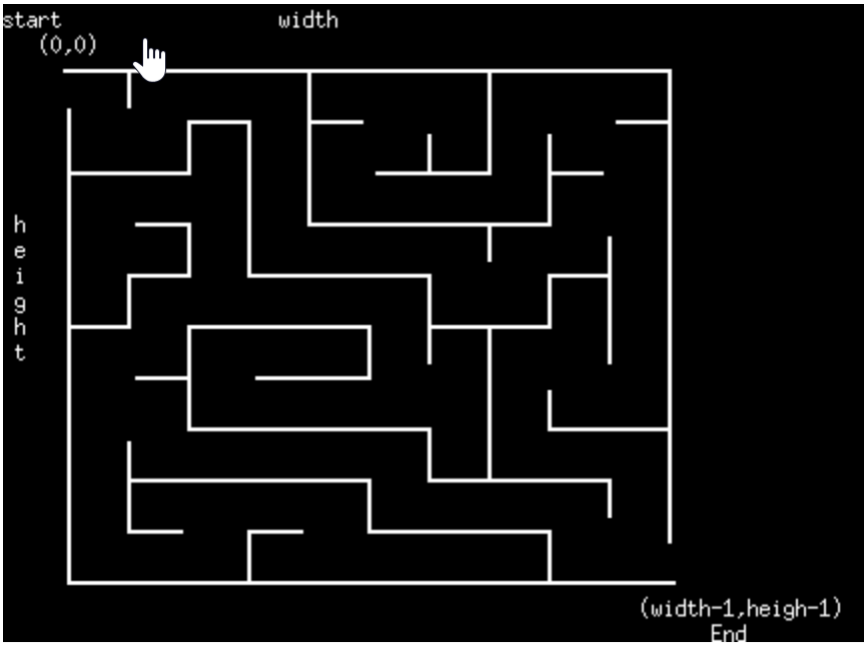
In this project you will implement the Stack and Queue interface using a LinkedList to both solve and generate mazes. Amazing! You will do so without using the java.util library or any other C library, which means, you'll have to do ALL the implementation work yourself.

Description

Solving and Generating Mazes

You will be asked to solve and generate mazes using a queue and a stack, respectively. These processes are independent of the implementation choices for the queue and stack.

Let's consider a maze as a 2-D array of spaces, where between each space there is a wall or not. The start point of the maze will be the upper left and the finish point is the lower right.



Notice that the maze has a width and height, and we index spaces in the maze with (w,h). The top left index is (0,0), and the bottom right index is (width-1,height-1). The maze will always start in the upper left and finish in the bottom right. We will describe two spaces as "reachable" if they are adjacent and do not have a wall between them.

Solving a Maze with a Queue

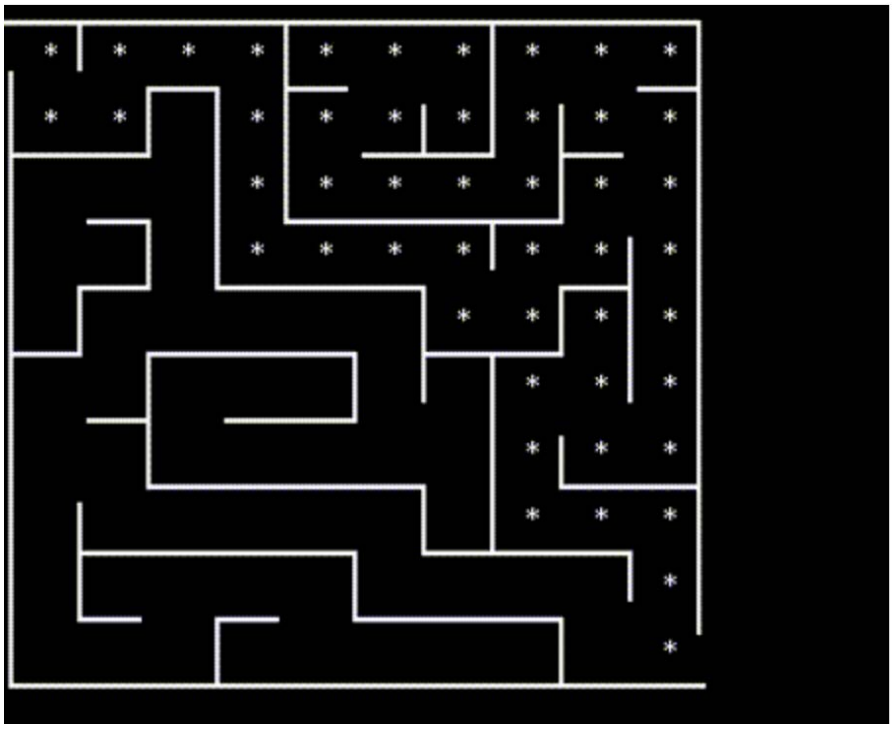
The process of solving a maze with queue is a lot like "hunting" out the end point from the start point. To do that, we must ensure that we test all possible paths. The queue will allow us to track which spaces we should visit next.

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. This search pattern is also referred to as Breadth First Search (or BFS).



For consistency, you should always enqueue the indexes into the queue in the order that they are provided via the **getReachableUnvisited()** neighbors method. That is enqueue index 0 of the returned list first, then the next index, and so on.

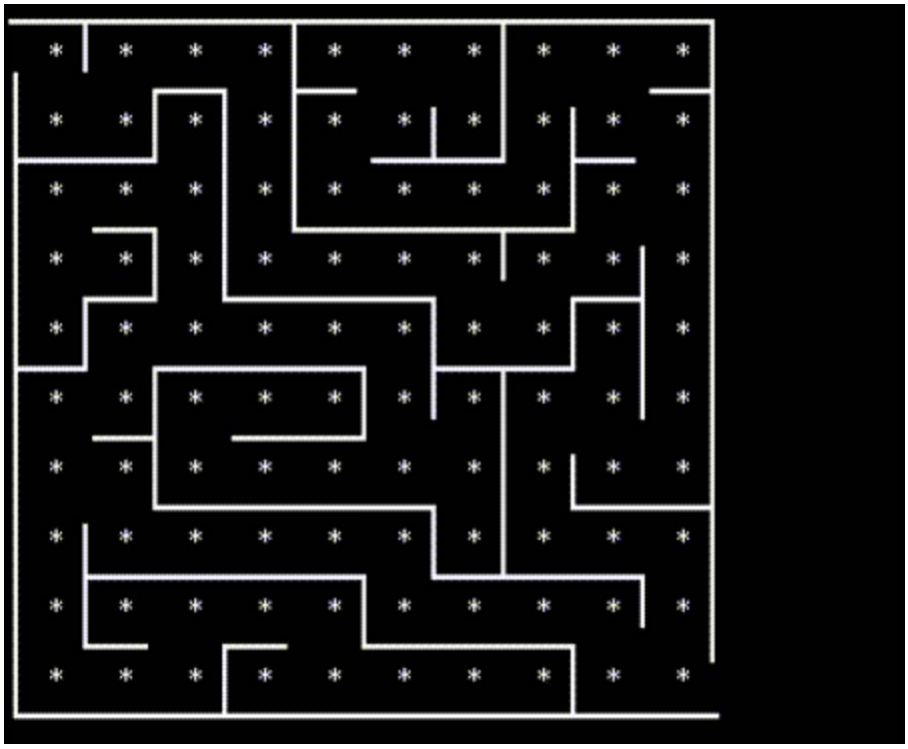
Drawing a Maze with a Stack

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 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 envision 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. This search pattern is also referred to as Depth First Search (or DFS)



Hint: https://www.usna.edu/Users/cs/aviv/classes/ic312/f16/project/01/project.html

Grading policy and general notes on the projects:

Your application should have all functionalities working properly. Twenty marks will be graded for the functionality of the project;

1. The following notes will make up the remaining 10 marks of the grade:
    a. There has to be adequate documentation and comments in the code (i.e., functions, loops, etc.);
    b. Your code should follow the code convention (i.e., spaces, indentations, etc.); and
    c. Your application should contain a menu to allow the user to select which option (s)he would like to run.

Good luck!