EECS 268: Spring 2009
Laboratory 6: Stacks and Queues

Due: 11:59:59 pm the day before your lab meets the week of March 30th

Lecture Topics: Stacks, Queues
Lab Topics: Tree Traversal

1. Introduction

The main purpose of this lab is to implement a stack and queue. The Stack and Queue implementation should utilize an underlying List ADT to reduce code duplication. The implemented data structures will be utilized in the provided source code to solve a maze using two traversal algorithms. This lab will provide you with experience of implementing and utilizing different abstract data types (ADTs) in program developments.

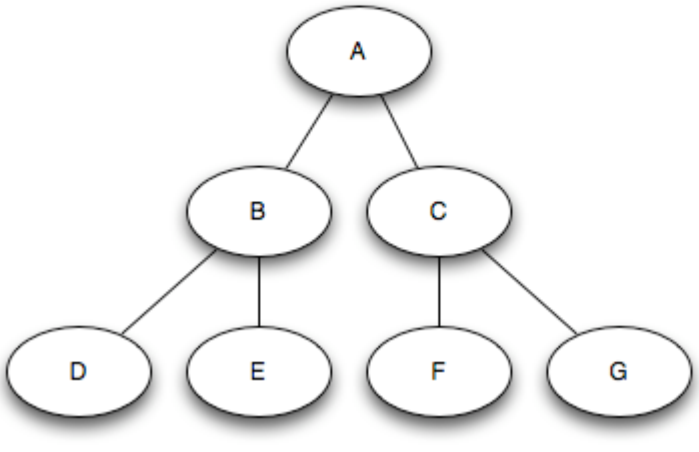
The main function and Maze class will be provided to you. Your job will be to implement the Stack, Queue, and List classes as well as the corresponding Exception classes necessary for exception handling.

The process of solving a maze can be thought of as a tree-like structure, where at each position in the maze the possible location of the exit could be down any path leading from that position. For example, if you were in a maze, at a particular junction where you could go left or right, then the exit could be found by following either the left path or the right path. Thus, in visualizing a solution, this would be a particular node in the tree with two branches, one for the left path and one for the right path.

Without any more information as to which path is a better one to explore first, there are two general strategies for exploring paths to find a solution to a tree-like search such as this: breadth-first search and depth-first search.

1.1 Breadth-first search

Breadth-first search (BFS) starts at the root of the search tree and considers all the children of this root as possible solutions first. When there are no more children of the root to examine, the algorithm considers the first child's children, then the second child's children, and so on, until a solution is found. Thus it moves along the breadth of the tree first, moving to the next level in the tree only when all of the nodes in the current level have been exhausted. BFS is guaranteed to find a solution if one exists. Given the example tree below:



A BFS algorithm would search in the following order: A is the root so will be examined first, then: B, C, D, E, F, G.

To implement this search strategy, one can use a Queue data structure. Using this structure, we can enqueue children of the current node being examined, but as it is a FIFO structure, if another neighbor-node has still not been examined, it will be examined before the children of the current node.

An informal algorithm would be:

```
Enqueue root node
Repeat while  solution not found:
      Dequeue a node and examine it
      If it is a solution
            End loop, solution has been found
      Else
            Enqueue any children of the node that have not been examined
      If Dequeue fails (queue is empty)
            End loop, solution cannot be found
End loop
```

So with our example tree, A would be enqueued first, then it would be dequeued and examined. If not the solution, then its children, B and C would be enqueued. Then the next iteration of the loop, B would be dequeued, examined, and its children, D and E, would be added to the queue. The next node to be examined would then be C, with its children, F and G, added to the end of the queue. And so on.

1.2 Depth-first search

Another general strategy is depth-first search (DFS). Here, the algorithm traverses down each possible branch to its end before considering another path. So at each node, if the node is not a solution, and the node has children, follow down the left-most child until you run out of children. At that point, move back up the tree and consider another path. If we were to consider the sample tree above, A DFS strategy would start at the root node A, and then consider B. As B has children it would then move on to D, and then E. only then would it continue to C. So the full search would be A, B, D, E, C, F, G.

A stack data structure is used to implement this style of search. At each level, a node is examined. If not the solution, then its children are pushed to the stack (in reverse order so that the left-most child would be pushed last). As the data structure is FILO, the last node added to the stack will be the next to be examined by the system.

An informal algorithm would be:

```
Push root node
Repeat while solution not found:
      Pop a node and examine it
      If it is a solution
            End loop, solution has been found
      Else
            Push any children of the node that have not been examined
      If Pop fails (stack is empty)
            End loop, solution cannot be found
End loop
```

1.2 BFS vs. DFS

BFS is always guaranteed to find a solution. And it will always be the most optimal (least number of nodes to get there). But BFS has a high memory requirement – the memory requirement *m* is an exponential function of the branching factor *b* and the depth of the solution *d* – $m = b^d$.

DFS is not guaranteed to find a solution. For example, if a particular branch has no end, but does not lead to a solution, then the solution will never be found. If a solution is found, it is not guaranteed to be optimal. However, DFS is much more efficient in terms of space than BFS. $M = b*h$. Where *h* is the total depth of the search space.

Fortunately for you, the implementation of these algorithms are provided in the Maze class. You just need to implement the underlying data structures.

2. Input & Output

2.1 Command line I/O
From the command line, the user will specify the input file containing any number of mazes and the output file name.

```
$ ./lab6 <input_file> <output_file>
```

Each maze will indicate the number of rows and columns in the maze, and then define the structure of the maze with X's representing walls or invalid positions and O's representing open or valid positions.

For each maze, the original maze will be output. If a valid solution to the maze is found, for both the BFS and DFS solutions, the number of moves to find the solution will be output as well as the maximum size of the data structure used in the algorithm. The solution in the maze will be marked by A's in place of the O's in the original maze

2.2 Sample Input File

An example of the maze input file would be:

```
10 16
XXXXXXXOXXXXXXXX
XOXOOOXOXXOOOOOX
XOXOXOOOOOOXOXXX
XOXOXXXOXOXXOOOX
XOOOXOXOXOOXOXXX
XOXOXOXOXXOXOOOX
XOXOOOXOOXOXOXOX
XOXOXOXXXXOXOXOX
XOOOXOOOOOOOOXOX
XXXXXXXXXXXXXXOX

10 16
XXXXXXXXOXXXXXXX
XOOOOOXXOXOOOXOX
XXXOXOOOOOOXOXOX
XOOOXXOXOXXXOXOX
XXXOXOOXOXOXOOOX
XOOOXOXXOXOXOXOX
XOXOXOXOOXOOOXOX
XOXOXOXXXXOXOXOX
XOXOOOOOOOOXOOOX
```

```
XOXXXXXXXXXXXXX
```

(Coloration added for effect. Gray indicating invalid positions and white valid ones.)

2.3 Sample Output File

```
Maze 1:
XXXXXXXOXXXXXXXX
XOXOOOXOXXOOOOOX
XOXOXOOOOOOXOXXX
XOXOXXXOXOXXOOOX
XOOOXOXOXOOXOXXX
XOXOXOXOXXOXOOOX
XOXOOOXOOXOXOXOX
XOXOXOXXXXOXOXOX
XOOOXOOOOOOOOXOX
XXXXXXXXXXXXXXOX

BFS Solution: 19 moves
Max Queue Size: 9
XXXXXXXAXXXXXXXX
XOXOOOXAXXAAAOOX
XOXOXOOAAAAXAXXX
XOXOXXXOXOXXAOOX
XOOOXOXOXOOXAXXX
XOXOXOXOXXOXAAAX
XOXOOOXOOXOXOXAX
XOXOXOXXXXOXOXAX
XOOOXOOOOOOOOXAX
XXXXXXXXXXXXXXAX

DFS Solution: 23 moves
Max Stack Size: 5
XXXXXXXAXXXXXXXX
XOXOOOXAXXOOOOOX
XOXOXOOAAAOXOXXX
XOXOXXXOXAXXOOOX
XOOOXOXOXAAXOXXX
XOXOXOXOXXAXAAAX
XOXOOOXOOXAXAXAX
XOXOXOXXXXAXAXAX
XOOOXOOOOOAAAXAX
XXXXXXXXXXXXXXAX

Maze 2:
XXXXXXXXOXXXXXXX
XOOOOOXXOXOOOXOX
XXXOXOOOOOOXOXOX
XOOOXXOXOXXXOXOX
XXXOXOOXOXOXOOOX
XOOOXOXXOXOXOXOX
XOXOXOXOOXOOOXOX
```

```
XOXOXOXXXXOXOXOX
XOXOOOOOOOOXOOOX
XOXXXXXXXXXXXXXX

BFS Solution: 19 moves
Max Queue Size: 8
XXXXXXXXAXXXXXXX
XOOAAAXXAXOOOXOX
XXXAXAAAAOOXOXOX
XOOAXXOXOXXXOXOX
XXXAXOOXOXOXOOOX
XAAAXOXXOXOXOXOX
XAXOXOXOOXOOOXOX
XAXOXOXXXXOXOXOX
XAXOOOOOOOOXOOOX
XAXXXXXXXXXXXXXX

DFS Solution: 33 moves
Max Stack Size: 6
XXXXXXXXAXXXXXXX
XOOOOOXXAXAAAXOX
XXXOXOOOAAAXAXOX
XOOOXXOXOXXXAXOX
XXXOXOOXOXOXAOOX
XAAAXOXXOXOXAXOX
XAXAXOXOOXAAAXOX
XAXAXOXXXXAXOXOX
XAXAAAAAAAAXOOOX
XAXXXXXXXXXXXXXX
```

(Coloration added for effect. Blue represents solution.)

Again, this code has already been provided for you.

3. Grading Criteria

30	**Implementation of List class**
20	**Implementation of Queue class**
20	**Implementation of Stack class**
10	**Exception Handling**
10	**Code style & output formatting**
10	**Documentation**