Solving a Maze

Here is an algorithm for finding a path from the entrance to the exit of a maze. This algorithm, like many others, is based on the idea of a *worklist* that holds individual steps towards the solution. For our problem the worklist holds squares of the maze that we know how to reach from the entrance.

We'll actually code two solutions -- one where the worklist is a stack an one were it is a queue. But the algorithm is the same regardless of the structure used for the worklist.

We will start the worklist with the entrance square for the maze as its only entrance.

We will also use a *marking* system, marking squares that have been completely explored. Each time we remove a square from the worklist we mark it and then add all of its unmarked neighbors to the worklist. Note that it is possible to add a square to the worklist that is already in the worklist, because we don't mark a node until it is removed from the list. We only need to explore a node once, so if we take a marked square from the worklist we ignore it.

We will also be adding *path edges* from nodes in the worklist back to the nodes that put them there. Eventually these path edges will form a path from the goal back to the entrance. When we reverse it this path will be our solution to the maze.

Here is the algorithm:

- Start the worklist with the entrance square
- At each step:
  - If the worklist is empty there is no solution
  - If the worklist isn't empty take node p from it.
  - If p is marked ignore it; take another square.
  - Mark p.
  - Let L be the list of p's unmarked neighbors.
  - For each q in L:
    - If q is the goal you found a path.
    - Otherwise add q to the worklist with a path edge from q back to p.

For example, consider the following maze, where E indicates the entrance and G (for Goal) indicates the exit:

	0	**1**	**2**	**3**	**4**
0		■	E	■	
1					
2	■			■	
3	■			■	G

Assuming that we use a stack for the worklist and push children of a square on it in the order West, South, East, North, we get the following sequence of states for our worklist:

	0	**1**	**2**	**3**	**4**
0		■	E	■	
1					
2	■			■	
3	■			■	G

E = Entrance, G = Exit

$$\begin{tikzcd}
(0, 2) & (1, 2) \arrow[l] & (1, 3) \arrow[l] & (1, 4) \arrow[l] & (0, 4) \arrow[l] & (2, 4) \arrow[dl] & \textbf{(3, 4)EXIT} \arrow[l] \\
 & & (2, 2) \arrow[ul] & (2, 2) \arrow[l] & (2, 4) \arrow[ul] & (2, 2) \arrow[dl] & (2, 2) \arrow[l] \\
 & & (1, 1) \arrow[uul] & (1, 1) \arrow[l] & (2, 2) \arrow[ul] & (1, 1) \arrow[dl] & (1, 1) \arrow[l] \\
 & & & & (1, 1) \arrow[ul] & &
\end{tikzcd}$$

Following the path edges we get the path from the exit to the entrance:

(3, 4) (2, 4) (1, 4) (1, 3) (1, 2) (0, 2)

Reversing this gives a path from the entrance to the exit, which is the solution to the maze that we seek.

Note that a solution to the maze is a path -- a sequence of squares from the entrance to the exit where each square is a neighbor of the previous one. Our algorithm will generate such a path if there is one, regardless of whether we use a stack or a queue for the worklist. Changing the data structure changes the order in which we add nodes to the worklist, but either structure will eventually get us a path if there is one.

Here is what you need to do for Lab 3:

a) Create a Square class to represent one square of the maze, and a Maze class that holds the maze as a 2D array. The Square class needs to keep track of its (row, column) position in the maze. The lab document suggests using Java's Point class for this. If you don't like translating between Point's x and y coordinates and your maze's rows and columns, just use your own row and column variables.

b) Implement Stack<E> using ArrayList<E> for storage, and Queue<E> using a linked structure. There is a StackADT interface and a QueueADT interface to list the methods you need. Test your implementations carefully.

c) Create an abstract MazeSolver class that solves the maze in terms of abstract methods for interacting with the worklist. The key method here is called step( ). This performs one step of the algorithm, taking one square from the worklist, marking it, and adding its neighbors to the worklist. There is also a method solve( ) that calls step( ) until the exit is reached or the worklist becomes empty.

d) MazeSolver is an abstract class. Create two concrete subclasses MazeSolveStack and MazeSolverQueue that instantiate the abstract methods of MazeSolver. This is quite easy; the real works comes in MazeSolver.

Marking

There are several sets of markings that we use for squares in the maze. Initially there are 4 types of squares: Start, Exit, Empty and Wall. For the empty squares there are several additional markers: squares that are unexplored, or on the worklist, or completely explored. Finally, when a solution has been found we mark the squares that are on the path from the start to the exit. The Square class has a toString( ) method that needs to change as the the square's status changes. You also need to be able to tell easily the status of a square. You can handle this any way you want, but one relatively easy approach is to have a field *kind* that has different values for Wall and Empty, and a field *marker* that has different values for Unexplored, OnWorklist, and Explored. The mark for being on the final path only needs to be applied in the toString( ) method -- by the time you know a square is on the final path you are no longer running the algorithm that looks for a soluton.