Data Structures and Object Oriented Programming

Homework #4

ESCAPE THE MAZE!

Based on CSCI 151 lab assignment from Oberlin college and Prof. Alvarado's writeup.
Project 4 is due May 3rd 2017, at 23:59:59. **This project has mileston submission. Please read the Submission section carefully for more details.**

Grading: 100 points

Submit your files using Vocareum. Make sure that you follow directions on how to submit files precisely. Read the submission report after you submitted your files. Failed submissions will receive a 0.

Contents

1 Implementing Stack and Queue (based on Alvarado's assignment) 2
1.1 Deque12 and BoundedDequeTester . . . 2
1.2 Implementing the Stack ADT . . . 3
1.3 Implementing the Queue ADT . . . 3

2 Implementing the Maze 4
2.1 Square Class . . . 5
2.2 Maze Class . . . 7
2.3 MazeTester . . . 8

3 Escaping the Maze 8
3.1 Preamble: The Maze Solving Algorithm . . . 8
3.2 The MazeSolver Abstract Class . . . 9
3.3 MazeSolverStack and MazeSolverQueue Class . . . 11
3.4 Testing the MazeSolver Class . . . 11
3.5 How to Trace the Path? . . . 11

4 Magic! 12

5 Style 12

6 Submission 13

7 Appendix 13
7.1 Passing command-line arguments to main . . . 13

Overview of the project

In this assignment you will implement data structures that provide an implementation for three abstract data types: A double-ended queue using a circular array, a stack and a queue. In addition, you will use these data structures in a method that implements a search in a Maze. Also, along the way you will get more experience with implementing Java interfaces, writing JUnit test cases, and using the Adapter design pattern.

1 Implementing Stack and Queue (based on Alvarado's assignment)

1.1 Deque12 and BoundedDequeTester

BoundedDequeTester.java

Read the documentation in the source code file for the **BoundedDeque**$< E >$ interface. Understand the responsibilities of each method required by the interface. Sketch a test plan for a class that implements this interface. Define a class named BoundedDequeTester that implements your test plan, using an Deque12 object as a test fixture.

The class you will be testing is Deque12, which implements BoundedDeque, and which you will be defining as part of this assignment. You should have at least one test for every method, but it is better form to create multiple tests for some methods depending on the condition you are testing. For example, you will probably want a separate test for adding to a full Deque from the test for adding to a non-full Deque.

As a practical matter, you do not need to completely write your BoundedDequeTester program before starting to define Deque12. In fact, an iterative test-driven development process can work well: write some tests in BoundedDequeTester, and implement the functionality in Deque12 that will be tested by those tests, test that functionality with BoundedDequeTester, write more tests, etc. The end result will (hopefully!) be the same: A good tester, and a good implementation of the class being tested.

As for previous (and future) assignments, make sure that your BoundedDequeTester does not depend on anything not specified by the documentation of the BoundedDeque interface and the Deque12 class. For example, do not make use of any other constructors or instance or static methods or variables that the ones required in the documentation.

Deque12.java

Define a generic class **Deque12**$< E >$ that implements the BoundedDeque$< E >$ interface. Besides the requirements for the methods and constructor documented in that interface, for this assignment there is the additional requirement that this implementation must use a circular array to hold its elements. (Note also that, as for other assignments, your Deque12 should not define any public methods or constructors other than those in the interface specification.)

A circular array can be rather tricky to implement correctly! It will be worth your time to sketch it out on paper before you start writing code.

You will of course need to create an array with length equal to the capacity of the Deque12 object. Because raw arrays dont play nicely with generics, you should use an ArrayList object for this purpose. However there are a couple of subtleties to look out for when using an ArrayList instead of an array:

1. Be careful with the difference between the ArrayLists add method and its set method. You will almost certainly want to use set, not add. But this means that you must add capacity items to your ArrayList before you do anything else (i.e. in the constructor), or set will throw an IndexOutOfBounds error.
2. Remember that no matter what size you initialize your ArrayList, it can always be increased, so be careful not to accidentally grow your ArrayList beyond your Bounded-Deques capacity. If you only use set and not add, you should not run into this problem.

You will want to have instance variables for the size of the Deque12 (how many data elements it is currently holding), and to indicate which elements of the array are the current front and back elements. Think about what the values of these instance variables should be if the Deque12 is empty, or has one element, or is full (has size equal to its capacity). And in each case, think carefully about what needs to change for an element to be added or removed from the front or the back. Different solutions are possible, as long as all the design decisions you make are consistent with each other, and with the requirements of the interface you are implementing.

1.2 Implementing the Stack ADT

Read and understand the interface specification in BoundedStack.java and then define a generic class **MyStack**$< E >$ that implements the BoundedStack$< E >$ interface. Once Deque12 is implemented and tested and debugged, defining MyStack is quite easy. We have provided you with some unit tests that you can use to test your implementation. Note that the methods required by the BoundedStack interface are different from, though closely related to, the BoundedDeque interface methods. Use the Adapter design pattern! That is, in your MyStack class, create an instance variable of type BoundedDeque (instantiated to a Deque12) and use it to perform all of the necessary BoundedStack methods. If this class is complicated, you are over-thinking it.

1.3 Implementing the Queue ADT

Read and understand the interface specification in BoundedQueue.java (or view the javadoc page for BoundedQueue) and then define a generic class **MyQueue**$< E >$ that implements the BoundedQueue$< E >$ interface. As with MyStack, if Deque12 is already implemented and tested and debugged, defining MyQueue is quite easy. We have provided you with some unit tests that you can use to test your implementation. Again note that the methods required by the BoundedQueue interface are different from, though closely related to, the BoundedDeque interface methods, and so the Adapter design pattern is applicable here as well.

Please make sure that you test your MyStack and MyQueue thoroughly before moving to the next Part of the Homework. This is VERY important as the last part of the Project is entirely dependent on the successful implementation of the Part 1.

2 Implementing the Maze

You are part of a science research team exploring the vast canal systems on Mars. Due to the harsh surface conditions, all of the exploration is done through the use of remote controlled robots. To explore an area, you drop in a large rover-bot that needs to make its way to an exit. Each rover-bot has a large army of smaller probe-bots that can be used to scout the route.

Unfortunately, the probe-bots have to be manually controlled, and only one group can move at a time, and only explore one new square at a time. These probe-bots travel in small swarms, so groups can be left to explore in different directions at each intersection.

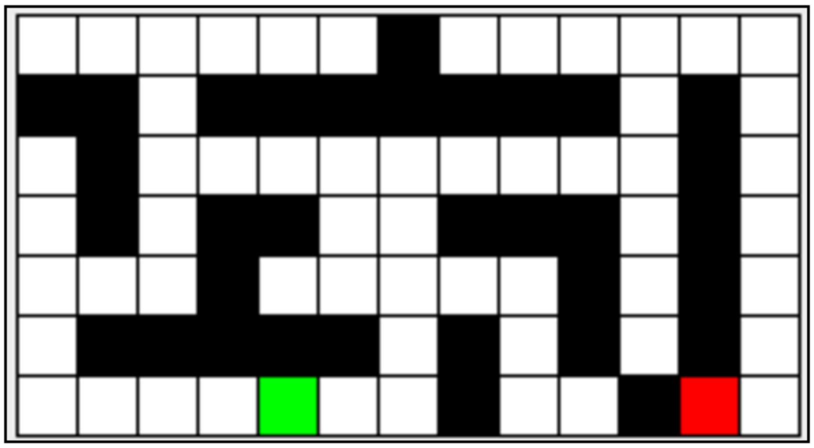
Two of your colleagues (Janice and Marina) have been arguing about the best way to use the probe-bots to scout a route for the rover-bot. Janice thinks that the locations should be scouted in a First-Come, First-Served fashion, while Marina thinks that you should focus on the most recently explored areas first.

Your job is to simulate the various approaches using your nifty CSE12 skills and examine the different approaches suggested by Marina and Janice.

Preamble : Representation of a Maze

First let's talk about a basic Maze. It has walls and pathways, and it has a starting point and an exit (Note: There may also be mazes with only an entry and no exit). Furthermore, one wall is just like another, and any open space (not including start and finish) is also identical. So, we can think of a maze as being made up of individual squares, each square either empty, a wall, the start, or the exit. **You may assume that any maze will have exactly one start and zero or one finish.**

Below is a graphical representation of a maze like the ones given in the starter files. The green box represents the start, the red box the exit, and the black squares the walls.



We represent such a maze with a text file of the following format. The first line of the file contains two integers. The first indicates the number of rows (R), the second, the number of columns (C).

The rest of the file will be R rows of C integers. The value of the integers will be as follows:
0 - an empty space
1 - a wall
2 - the start
3 - the exit

In terms of coordinates, consider the upper left corner to be position [0,0] and the lower right to be [R-1,C-1].

For example, this is the text version of the maze above (start is at [6,4] and exit at [6,11]).

```
7 13
0 0 0 0 0 0 1 0 0 0 0 0 0
1 1 0 1 1 1 1 1 1 1 0 1 0
0 1 0 0 0 0 0 0 0 0 0 1 0
0 1 0 1 1 0 0 1 1 1 0 1 0
0 0 0 1 0 0 0 0 0 1 0 1 0
0 1 1 1 1 1 0 1 0 1 0 1 0
0 0 0 0 2 0 0 1 0 0 1 3 0
```

2.1 Square Class

A Square object represents a single square in the maze. A Square should have a class variable that represents its type (space, wall, start, or exit).

It turns out that later it will be useful if a Square knows where it is positioned in the maze. Therefore, you should give each Square private variables of type int named **row** and **col** to represent it's location within the maze. Also, you will want to mark a Square as you visit it when exploring the Maze. Make a boolean variable **visited** in order to keep track of this information.

Method description for Square class

1. **Constructor**
   public Square(int row, int col, char type) : A constructor that takes the row, col and type of the square as parameters and creates a new Square object.

2. **toString**
   public String toString() : Return the character corresponding to this Square, using the following notations: **(Please use the exact same notations as given below)**

   _ for empty space (0)
   # for wall (1)
   S for Start (2)
   E for Exit (3)

   HINT: Now's a good time to use switch statements

   Later, you will be adding the following notations to this toString() method **after implementing parts of Section 3**. The below symbols are only applied to empty spaces, not start or exit squares. You will learn more about the solver worklist in the next section. Hang on!

   o - is on the solver work list
   . - has been explored
   x - is on the final path to the exit

3. **Accessor methods**

   public int getRow() : Returns row value
   public int getCol() : Returns col value
   public int getType() : Returns the type

4. **Methods to check status of the Square**

public boolean isFinish() Returns true for Exit
public boolean isStart() Returns true for Start
public boolean isValid() Returns true if it isn't a wall or the Start location
public boolean isVisited() Returns true if this Square has been visited

5. **Setters**

   void setVisited() Sets 'visited' flag to true
   void clearVisited() Sets 'visited' flag to false

You will be modifying this class further, after reading through Section 3

2.2 Maze Class

Refer to the Maze class in the starter files, that stores the logical layout of a maze. It should contain (as a class variable) a 2D array of Squares. Initially, this array will be empty and you will use a method to populate it.

As you begin working on **Maze.java** don't forget to also create **MazeTester.java** with JUnit test cases for the various methods of the Maze class.

1. *public Maze()* : A constructor that takes no arguments. NOTE: you can just omit writing a constructor and Java will just use the default values, the real work will be done with the loadMaze() method described next.

2. *boolean loadMaze(String fname)* : Load the maze that is contained in the file named fname. The format of the file is described at the beginning of this Part.

   Good news! This method is already provided in the starter code to make your life easier :)

3. *ArrayList<Square> getNeighbors(Square sq)* : return an ArrayList of the Square neighbors of the parameter Square **sq**. There will be at most four of these (to the North, East, South, and West) and you should list them **in that order**. If the square is on a border, skip over directions that are out of bounds of the maze. Do not add null values to the result instead. Also skip adding neighbor Squares that are not valid.
   For example, in the above example (maze-2), the neighbor of Square (0,0) is (0,1). The neighbors of (3,0) are (2,0) and (4,0).

4. **Accessor methods**
   public Square getStart() - Returns the Start Square in the maze
   public Square getFinish() - Returns the Exit Square in the maze
   public Square[][] getMaze() - Returns the 2D array

5. *void setVisit(row,col)* : Sets the Square at (row,col) as visited.

6. *void clearMaze()* : Clear all visited squares. HINT: One way you might do this is by giving each Square a clear() method too, and then just loop through the squares and asking them to clear themselves.

7. *String toString* : Return a String representation of this Maze in the format given below. This is where it's useful to have a working Square.toString() method.
(This method is provided in the starter code)

2.3 MazeTester

In order to make sure that the Maze is implemented correctly, create a JUnit tester called **MazeTester.java** to test the methods of the Maze class. Use **maze-4** in the tester for testing the methods. Among other things, this test should

a) Load a maze from one of the supplied files
b) Get the neighbors of some specific square (the start square, for example)
c) Assert that (1) there are the correct number of neighbors, and (2) the neighbors are in the correct locations.
d) In the MazeTester that you submit, do not use a local file name/path. Instead refer to the Appendix section to see where to place the Maze file and pass only the name of the file "**maze-4**" to the loadMaze method.

You probably should do this for the corners and border cases, at least. There should also be a test to print out the maze, and to confirm your getStart and getFinish methods return the correct squares.

3 Escaping the Maze

We have implemented the Maze and all the data structures it requires so far. We will now dive into the application section of this homework, where we will use the working Stack and Queue to solve the mazes.

We will be writing the MazeSolver class, who's main functionality is to determine whether a given maze has a valid solution.

3.1 Preamble: The Maze Solving Algorithm

Our maze solving algorithm goes something like this: begin at the start location, and trace along all possible paths to (eventually) visit every reachable square. If at some point you visit the finish Square, it was reachable. If you run out of squares to check, it isn't reachable.

At the start

1. Create an (empty) worklist (Stack/Queue) of locations to explore.
2. Add the start location to it.

Each step thereafter

1. Is the worklist empty? If so, the exit is unreachable; terminate the algorithm.
2. Otherwise, grab the "next" location to explore from the worklist.
3. Does the location correspond to the exit square? If so, the finish was reachable; terminate the algorithm and output the path you found.
4. Otherwise, it is a reachable non-finish location that we haven't explored yet. So, explore it as follows:
    (a) Compute all the adjacent up, right, down, left locations that are inside the maze and aren't walls (We already have a method for this!)
    (b) Add them to the worklist for later exploration **provided they have not previously been added to the worklist.** (Section 3.2, Part 9 explains how to keep track of this information)
5. Also, record the fact that you've explored this location so you won't ever have to explore it again. **Note that a location is considered "explored" once its neighbors have been put on the worklist. The neighbors themselves are not "explored" until they are removed from the worklist and checked for their neighbors.**

This pseudocode is entirely agnostic as to what kind of worklist you use (namely, a stack or a queue). You will need to pick one when you create the worklist, but subsequently everything should work abstractly in terms of the worklist operations.

3.2 The MazeSolver Abstract Class

Create an abstract class **MazeSolver** that will implement the above algorithm, with a general worklist. Its abstract methods will be implemented differently depending on whether the worklist is a stack or a queue. The MazeSolver class should have a non-public class member of type Maze, and should have the following methods:

1. *abstract void makeEmpty()* : Create an empty worklist

2. *abstract boolean isEmpty()* : Return true if the worklist is empty

3. *abstract void add(Square sq)* : Add the given Square to the worklist

4. *abstract Square next()*: Return the "next" item from the worklist

5. *MazeSolver(Maze maze)* : Create a (non-abstract) constructor that takes a Maze as a parameter and stores it in a variable that the children classes can access.

6. *public Maze getMaze()* : Getter for maze

7. *boolean foundExit()* : A non-abstract method that the application program can use to see if a path exists. This method will return true if a path from the start to the exit has been found, false otherwise. You may use a boolean instance variable **foundExit** to keep track of this, and add a setter as well.

8. *public boolean gameOver()* : In addition to foundExit, you may add another boolean instance variable **gameOver** to keep track of the game progress. The method gameOver() returns true when the game ends - either when an exit is found or when the worklist becomes empty. You may terminate the algorithm when gameOver becomes true. Also add a setter to set this variable to true.

9. *String getPath()* : Returns either a string of the solution path as a list of coordinates [i,j] from the start to the exit or a message indicating no such path exists If the maze isn't solved, you should probably return a message indicating such.

10. *Square step()* : Perform one iteration of the algorithm above (i.e., steps 1 through 5) and return the Square that was just explored (and null if no such Square exists). Note that this is not an abstract method, that is, you should implement this method in the MazeSolver class by calling the abstract methods listed above.

    In order to keep track of which squares have previously been added to the worklist, you will "mark" each square that you place in the worklist. Then, before you add a square to the worklist, you should first check that it is not marked (and if it is, refrain from adding it).

    IMPORTANT: **Here is a suggestion for marking a Square**: Have each Square keep track of which Square added it to the worklist (i.e., "Which Square was being explored when this Square was added to the worklist?"). That is, add a new class member **Square previous** to the Square class, which will represent the Square previous to the current one; initialize this variable to null in the constructor/reset method. Then, when a Square is being added to the list for the first time, you will set the previous variable to point to the current Square (the Square that is being explored).

If the previous variable is already non-null, then this Square has already been placed on the list, and you should not do so again.

You can also add a getter and setter to your Square class to help you access the **previous** member. Do not forget to clear it in the Square's clear() method.

11. *void solve()* : Repeatedly call step() until you get to the exit square or the worklist is empty.

3.3 MazeSolverStack and MazeSolverQueue Class

We will now create two child classes of the MazeSolver abstract class - MazeSolverStack and MazeSolverQueue. (Java Refresher: child classes must *extend* parent classes).

- These will not be abstract classes, and so you must implement the MazeSolver's abstract methods.
- Each class should contain as a class variable a worklist of the appropriate type (so, MazeSolverStack should have a class member of type MyStack<Square> and MazeSolverQueue should have one of type MyQueue<Square>). You must add a getter for the worklist in each class.
- Implement the abstract methods is perform the appropriate operations on the stack or queue class member.
- Don't forget to include a call to **super(maze)** as the first line of your constructor.

3.4 Testing the MazeSolver Class

For testing purposes, we have added main( ) methods to both MazeSolverStack and MazeSolverQueue. The main() method gets a maze file from the command-line argument, creates the appropriate type of worklist, calls the solve( ) method to find a solution and then prints the resulting path, if there is one. You can use this main() method to test each of the Solver individually. Refer to the appendix for more information about how to accept command-line arguments. **Please do not modify the main methods given as your submission will be graded based on the output of these main methods**

3.5 How to Trace the Path?

In order to output the solution to the maze in step 3 of the algorithm, you will need to keep track of the path that was followed in your algorithm. This seems to be a difficult proposition; however, you've already done most of the work when you marked your worklist nodes using the previous variable. Yayy!!! Let us explain.

In order to keep from wandering in a circle, you should avoid exploring the same location twice. You only ever explore locations that are placed on your worklist, so you can guarantee that **each location is explored at most once by making sure that each location goes on the worklist at most once.** You've already accomplished this by "marking" each square that you place in the worklist.

You are marking the square by putting an arrow in it that points back to the square from which you added it to the worklist. Now, when you are at the exit, you can just follow the arrows back to the start.

Of course, following the arrows gives you the path in reverse order. If only you had a way to keep track of items such that the Last item in was the First item out, then you could read all the arrows in one pass and write them back out in the correct order........

The Squares on the final path must be markes 'x' on the Maze board. Add another boolean variable **onFinalPath** (and a public setter for it) to the Square class to indicate whether or not the Square is on the final path to the Exit. This boolean can be turned true when you trace the path from the exit to the start.

4 Magic!

You are also given another starter file: MazeApp.java. If everything is working in your Maze and MazeSolver classes, you should be able to run the MazeApp program and get a GUI interface that will allow you to animate the process of finding the solution of the maze.

PLEASE do not modify this file

- The load and quit buttons operate as you might expect.
- The reset button will call the Maze's reset() method and then create a new MazeSolver.
- If you click on the stack button it will toggle between using a Stack or Queue to solve the maze.
- The step button performs a single step of the MazeSolver and start will animate things taking one step per timer delay interval.
- Don't forget to hit Enter after changing values in the textboxes.
- Your Maze's toString method is used to display the maze in the main window, and the getPath() method from MazeSolver is used for the bottom window.

5 Style

Refer to the Coding Style Guidelines on the course website.

6 Submission

For this assignment we have created a milestone that will get you to work right away and give you 5 points (which are part of the 100). These submissions are not required but strongly recommended. You should submit a ”decent quality” code: empty files, a few lines of codes, meaningless code will not count. It is OK to modify your code after the milestone but the change should not be drastic.

- Milestone files (5 points): Saturday, 11:59pm
  1. Deque12.java
  2. BoundedDequeTester.java
  3. MyStack.java
  4. MyQueue.java
- **Final submission**: You will be submitting ALL the files together for the final submission
  1. Deque12.java
  2. BoundedDequeTester.java
  3. MyStack.java
  4. MyQueue.java
  5. Square.java
  6. Maze.java
  7. MazeTester.java
  8. MazeSolver.java
  9. MazeSolverStack.java
  10. MazeSolverQueue.java

7 Appendix

7.1 Passing command-line arguments to main

- Place the file inside your project directory (a level above the src directory)
- Go to Run -> Run Configurations -> (Select the file with the main() method on the left) -> Arguments -> Program Arguments. Add the name of your file here and click Apply.