**CS 251: Project 2**

**Summer 2019**

**Due: Sunday, June 23 by 11:59 PM**

**Overview**

In this project, you will (1) create and solve mazes and (2) implement a binary search tree for storing and searching Scores from Project 1.

**General Guidelines**

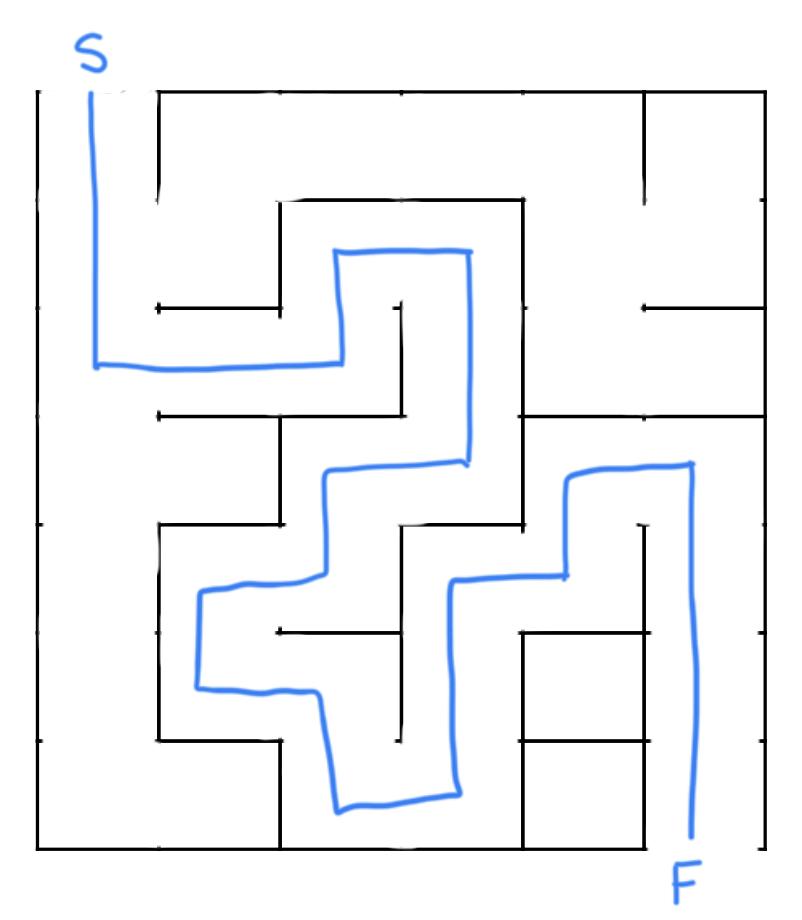
Follow the instructions carefully. Do not use any classes or libraries that are not specifically allowed. If you are unsure, ask. Give yourself plenty of time to submit. Make sure your code compiles and runs on data.cs before submitting.

**Part 1 Specification**

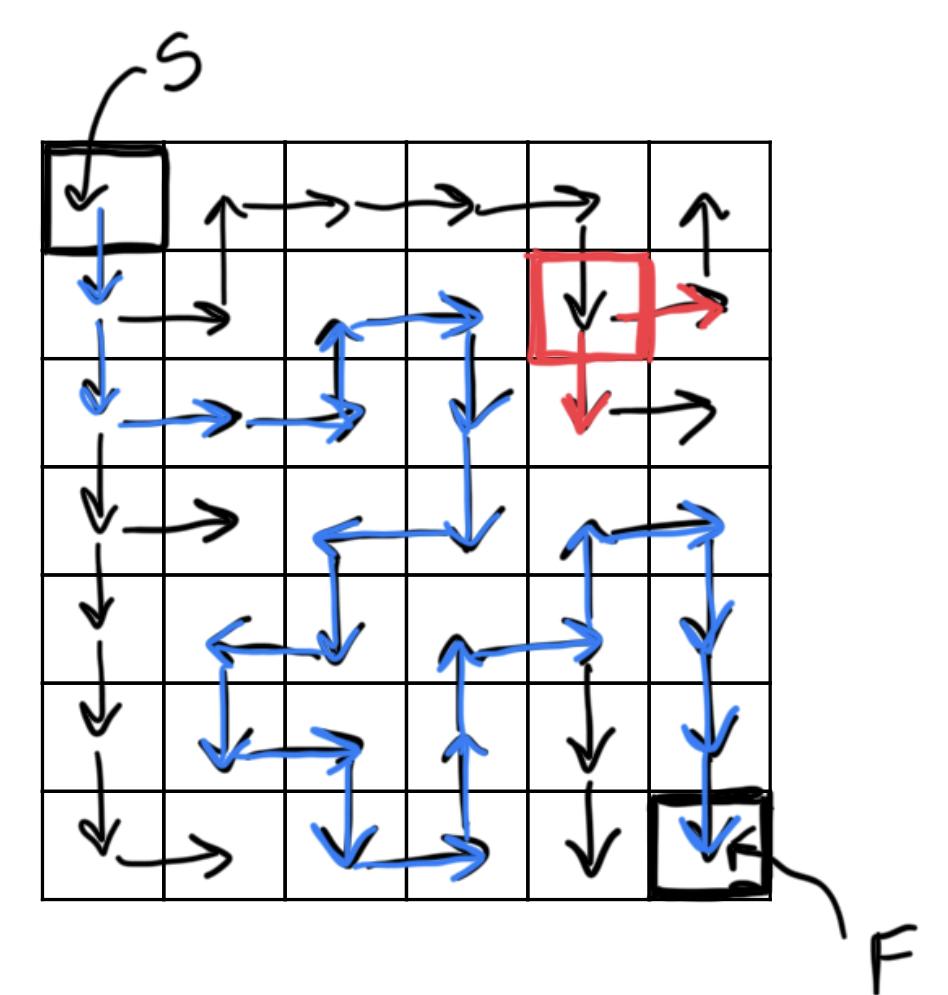
In this part, you will write a program that represents a maze in an abstract way, and then solves the maze. In order to do this, you will use a Queue data structure to represent the maze and a Stack data structure to help solve the maze.

**1.1** **Representing and solving a Maze**

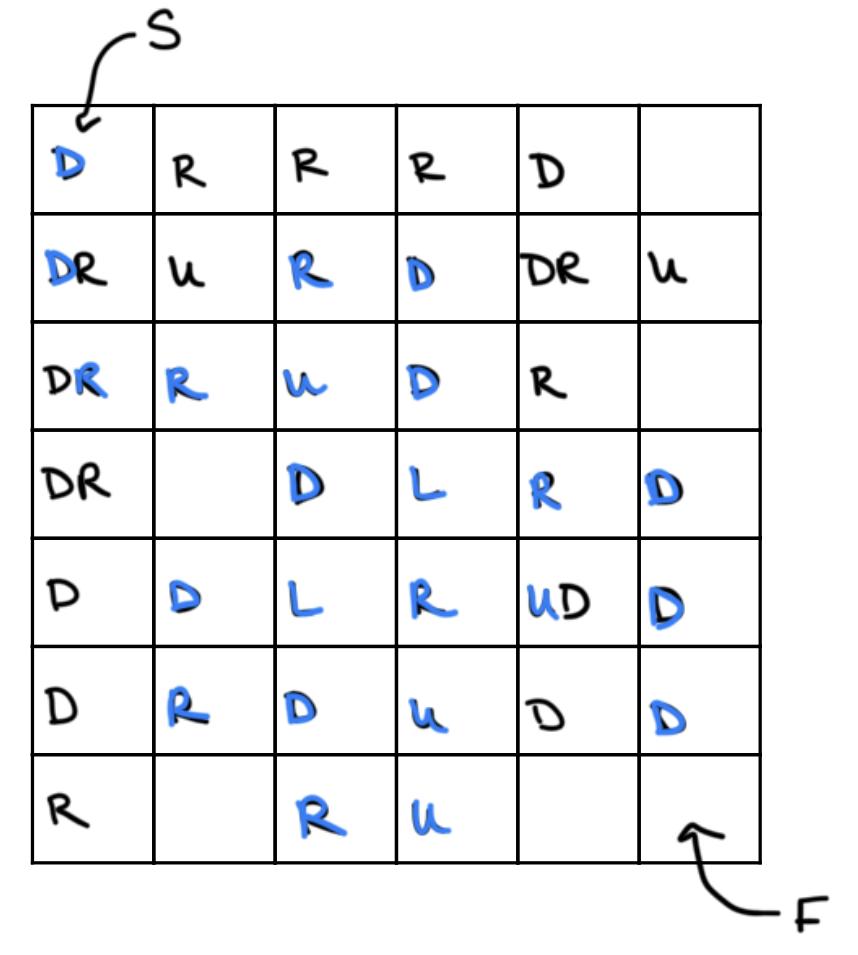
Consider the following maze.



In this part, you will use a 2-dimensional array of queues to represent a maze. The maze above could be represented as a 2-dimensional array of positions, where each position has a list of possible outgoing paths, represented by arrows in the image below. For example, at position [1][4] (noted in red), the possible outgoing paths are Right and Down.



We can represent each outgoing step with a character where D = Down, U = Up, R = Right, and L = Left.



This allows us to represent the maze with a simple text file:

**maze1.txt**

7

6

D

R

R

R

D

DR

U

R

D

DR

U

DR

R

U

D

R

DR

D

L

R

D

D

D

L

R

UD

D

D

R

D

U

D

D

R

R

U

In this text file, the first two lines specify the number of rows and the number of columns respectively (7 X 6 = 42 total positions in the maze). Then follows 42 lines which list the possible outgoing paths in each of the 42 positions, where you loop through each column in a row before moving to the next row (i.e. the order is [0][0], [0][1], [0][2]...[0][5], [1][0], [1][1]...)

In this project, the lists of possible outputs will be represented as Queues of Steps, implementation details discussed below.

To solve a maze, you will traverse the array starting in the upper left-hand corner, using checking each path until you get to the finish (lower right-hand corner). To do this, you will keep track of your current path with a Stack, which allows for easy backtracking when you reach a dead end.

**1.2 Implementation Details**

**(a)** You are provided with the following Java files:

* *EmptyQueueException.java*: use to handle situation when *dequeue* is called on an empty queue
* *EmptyStackException.java*: use to handle situation when *pop* or *peek* are called on an empty stack
* *MazeHasNoSolutionException.java*: use to handle situation when there is no solution to a given maze
* *Step.java*: defined as an enum to guarantee that no values are valid except UP, DOWN, LEFT, or RIGHT--note that each enum value is associated with a String value, letter, making it easy to get a String from the enum value
* *Maze.java*: represents the Maze object as a 2-dimensional array of StepQueue objects
* *Main.java*: use for testing your code; this program reads a maze from a file, builds it using the associated classes, and solves it using MazeSolver. Be sure to keep names of methods used here the same or it won’t run. The output of this program is the solution of the Maze or No Solution. (e.g. the solution to the example above would be the String “DDRRURDDLDLDRDRUURURDDD” printed to System.out.) If a maze has no solution (i.e. throws a MazeHasNoSolutionException, the program catches that exception and prints “No Solution.”)

**(b)** You must implement the following Java files, using the Skeleton code provided for you. You can make changes to the Skeleton code AS LONG AS you are still following the project guidelines AND the code still works with the provided code.

* *StepQueue.java*: This class represents a Queue of Step objects, which will be used to represent all the outgoing paths at a given position in the maze. **You must implement this Queue using a linked list structure.** The inner class *Node* is provided for you, as are a constructor and the *toString* method. It is strongly recommended that you do not change the *toString* method as this is used to test your output and any variation could result in failure on test cases. The methods you must implement are:
  + *isEmpty*: return true if the Queue is empty, false otherwise
  + *size*: return the size of the Queue
  + *enqueue*: add a new item to the back of the Queue (note that the parameter here is a Step, which means you will have to wrap it in a Node in your function). Do not change the parameter.
  + *dequeue*: remove the first item in the Queue and return its value (the Step value, not the Node); if the Queue is empty, throw an exception
* *StepStack.java*: This class represents a Stack of Step objects, which you will use in MazeSolver to keep track of your path. **You must implement an array-based implementation of the Stack.** You are provided with instance variables, a constructor, and a *toString* method. You may make changes to the Skeleton code as long as it still works with the Main program and follows the guidelines of the project. Do not change the *toString* method.You must implement the following methods:
  + *size*: return the current size of the Stack (not the size of the array)
  + *isEmpty*: return true if the Stack is empty, false otherwise
  + *peek*: return the value of the top item on the Stack without removing it; if the Stack is empty, throw an exception
  + *pop*: remove the top item from the Stack and return its value; if the Stack is empty, throw an exception; if the Stack falls below ¼ of the array size, resize the array to be ½ its size
  + *push*: add a new item to the Stack; if the Stack is full, re-size the array to be twice as large
* *MazeSolver.java*: This class contains the algorithm that solves a maze (i.e. finds a path from the Start to the Finish. Note that in all examples, the Start spot will be at position [0][0] (the upper left-hand corner) and the Finish spot will be at the lower right-hand corner. You are provided with instance variables and a constructor. You may change these as long as your code follows the project guidelines and works with the rest of the provided code.

The one function you MUST implement is *findPath*, which searches for a path through the maze from Start to Finish. The basic algorithm is as follows:

**Algorithm** *findPath*

**Input:** *M*[][], a maze represented as a 2-D array of StepQueue objects

**Output:** *path*, a Stack representing the Steps in the path from Start to Finish

*path* := an empty StepStack

*row* := 0 //row index of the current position

*col* := 0 //column index of the current position

**while** the current position is not the Finish position:

*Q* := possible steps from the current position

*step* := *Q.dequeue()*

**if** there is no step (i.e. *Q* is empty)

**then** backtrack (i.e. pop the last step from the stack and move backward along that path accordingly)

**else** make the step, pushing it onto *path*

**end if**

**end while**

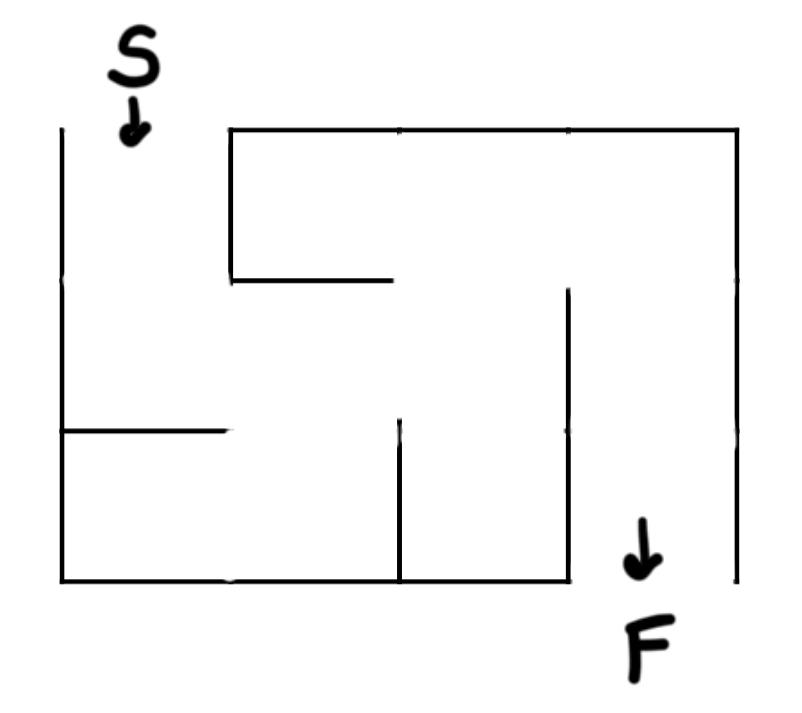
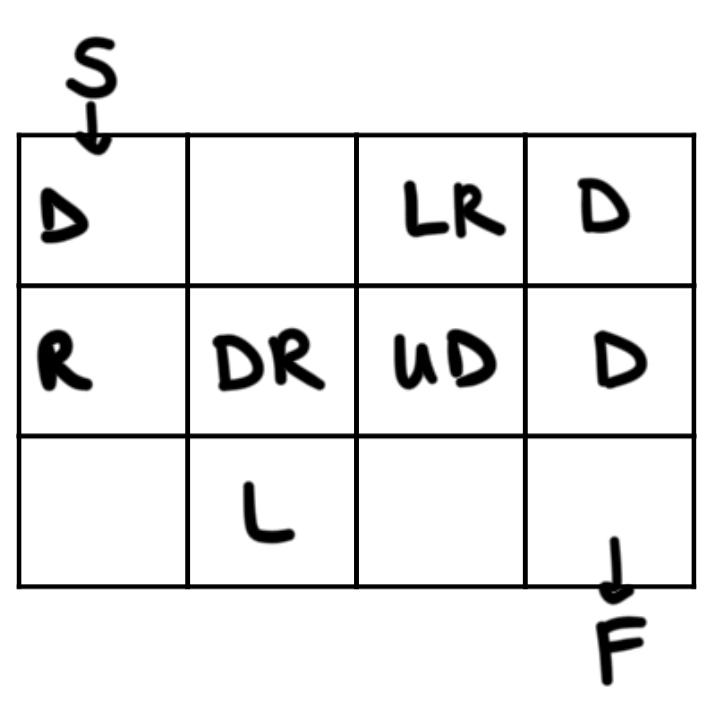
**return** *path*

**NOTE:** This algorithm does not handle the case where there is no solution, but in general no solution means there is no move to make (no step forward--i.e. the current queue is empty--and no step back--i.e. path stack is empty).

You are *allowed* and, in fact, *encouraged* to write helper methods so that the entire algorithm is split into smaller pieces. However, keep in mind that the only method called from Main is *findPath*, which should find the path if there is one (or throw an exception if there is none.)

**(c) Example tracing everything from start to finish:**

On the left is an example maze. On the right, is the 2-D array representation of the same array.

** **

In this project, the maze above would be represented in a text file as follows:

(START OF FILE)

3

4

D

LR

D

R

DR

UD

D

L

(END OF FILE)

*Main*.*java* will read in the file and use *Maze.java* to represent the maze as a 2-D array of StepQueue objects--dimensions here are 3 rows by 4 columns. Then the program will use *MazeSolver.java* to find a path through the maze using the *findPath* method. The following traces *findPath* for this example. Let CP = current position (e.g. (0, 3) means row 0 and column 3). Let P = the stack representing the steps. Let Q = a StepQueue.

CP: (0, 0)

Q = queue at (0, 0): D

P: //currently empty

step = Q.dequeue(): D

step(D)// step down

P.push(D)

P: D

CP: (1, 0)

Q = queue at (1, 0): R

step = Q.dequeue(): R

step(R) //step right

P.push(R)

P: DR

CP: (1, 1)

Q = queue at (1, 1): DR

step = Q.dequeue(): D

step(D)

P.push(D)

P: DRD

CP: (2, 1)

Q = queue at (2, 1): L

step = Q.dequeue(): L

step(L)

P.push(L)

P: DRDL

CP: (2, 0)

Q = queue at (2, 0): <empty>

Q is empty, so backtrack:

P.pop(): L

step(R) //opposite of L

P: DRD

CP: (2, 1)

Q = queue at (2, 1): <empty> //because L was dequeued before

Q is empty, so backtrack:

P.pop(): D

step(U) //opposite of D

P: DR

CP: (1, 1)

Q = queue at (1, 1): R

step = Q.dequeue(): R

step(R)

P.push(R)

P: DRR

CP: (1, 2)

Q = queue at (1, 2): UD

step = Q.dequeue(): U

step(U)

P.push(U)

P: DRRU

CP: (0, 2)

Q = queue at (0, 2): LR

step = Q.dequeue(): L

step(L)

P.push(L)

P: DRRUL

CP: (0, 1)

Q = queue at (0, 1): <empty>

Q is empty, so backtrack:

P.pop(): L

step(R) //opposite of L

P: DRRU

CP: (0, 2)

Q = queue at (0, 2): R

step = Q.dequeue(): R

step(R)

P.push(R)

P: DRRUR

CP: (0, 3)

Q = queue at (0, 3): D

step = Q.dequeue(): D

step(D)

P.push(D)

P: DRRURD

CP: (1, 3)

Q = queue at (1, 3): D

step = Q.dequeue(): D

step(D)

P.push(D)

P: DRRURDD

CP: (2, 3) = FINISH

The correct path is: DRRURDD

**Part 2: Binary Search Trees**

**Description:**

As an extension to Project 1, the programming competition that you started now has several tasks, and as they are accomplished the scores of the participants are updated in real time. In order to accommodate this, you need to implement some way to store that information and efficiently search/update it.

In part 2 you will be implementing a standard binary search tree using a linked list of nodes that each contain a key and data. More specifically, each node will contain a name (the key) and an integer value (their score). In addition to the key and data fields, each node should also have a left and right field, through which their children nodes can be accessed. By definition the key, in this case someone’s name, is used for inserting a new node into the tree as well as searching the tree for an existing node. This means that the nodes should be organized lexicographically with respect to their keys. (Hint: Strings have a compareTo method).

**Explanation of each class**:

* Score: The use of this class is entirely optional. This is essentially just a wrapper class that contains a *UserID (*String) and a *score* (int)
* Node: This class defines the nodes with which make the binary tree. The constructor for this class should take either a String and an int or, if you chose to use it, a Score. In either case, the fields for *key* and *data* should be initialized.
* BinarySearchTree: This is where a majority of your work will be done. It has two fields, *size* (int) and *root* (Node). Size will be equal to the number of nodes in the tree, and root will be the initial point of reference from which you will be able to reach all other nodes.
* Main: Contains the driver code for reading each line of the input file and performing the corresponding instruction.
* MainTest: In order to make testing easier for you and to ensure that no points are missed because of superfluous file differences such as differences in newline characters, we have created JUnit5 tests for you to use. It is imperative that you do NOT make changes to this file except where explicitly stated.

Note: For all classes except MainTest, you are permitted to change any function name, return type, or parameters are you see fit. The skeleton code is only provided to help guide your program structure. However, note that when working with trees, recursion is often the most simple method of implementing a function. If you choose to use recursion, it is often helpful to use wrapper functions which call your recursive functions. Many of the functions provided within the skeleton code are only wrapper functions meaning if you intend to use recursion, you will need to add functions to the classes to complete this project. This is encouraged.

In order to evaluate the correctness of your implementation, your program should provide the following functionality. (The method signature in the skeleton code corresponding to each functionality is given italics):

* A function to insert a Node into the BinarySearchTree; *insertNode(Score newScore)*
* A function to get the height of the tree. The height of the tree is defined as the maximum number of links from the root to a leaf node, therefore a tree containing only a root has a height of 0; *getTreeHeight()*
* A function to get depth of any node in the tree where the depth is defined as the number of links from the root node to the specified node; *getNodeDepth(String searchedKey)*
* A function to get the size of the tree; *getSize()*
* A function to search for a node given its key and return its data; *searchNode(String key)*
* A function to print the in order traversal of the tree; *printInOrderTraversal()*

**Input/Output Format:**

Input format:

* (i <String> <Integer>) Insert a new Node with key = <String> and data = <Integer>
* (size) Prints to System.out “Size of tree: <size\_of\_tree>\n”
* (nodedepth <String>) Prints to System.out “Depth of <String>: <Data\_of\_node\_w/\_key\_String>\n” or “<String> not in tree\n” if there is no node in the tree with key = <String>
* (treeheight) Prints to System.out “Height of tree: <height\_of\_tree>\n”
* (search <String>) Prints to System.out “Score of <String>: <Data\_of\_node\_w/\_key\_String>\n” or “<String> not in tree\n” if there is no node in the tree with key = <String>

|  |  |
| --- | --- |
| Input | Expected Output |
| i Isla 157  i Maple 7  i Thomas 0  treeheight  nodedepth Isla  nodedepth Maple  nodedepth Thomas  search Cooper  i Cooper 50  search Cooper  nodedepth Cooper  size  iot | Height of tree: 2  Depth of Isla: 0  Depth of Maple: 1  Depth of Thomas: 2  Cooper not in tree  Score of Cooper: 50  Depth of Cooper: 1  Size of tree: 4  Begin in order traversal  Cooper  Isla  Maple  Thomas  In order traversal completed |

* (iot) Prints to System.out “Begin in order traversal\n<String1>\n<String2>\n...<StringN>\nIn order traversal complete\n” Where <String1>, <String2>,…<StringN> Are the keys of the nodes visited 1st, 2nd, ...Nth during an in order traversal.