Simply A-Maze-ing: Stack and Queue

Assignment 9

Introduction and setup

The goal of this assignment is to **learn how to implement Stack and Queue ADTs from scratch**, using a Linked List data structure.

After you implement these two ADTs, you will be able to use them for solving mazes. You will see how these two types of queues are used during the maze traversal.

All the starter code, including JUnit tests and some sample mazes can be found in this archive: STARTER. Unzip it and follow the instructions below.

This assignment consists of two parts.

Part I. Mini-List [22 points]

Copy the files in part1 folder into your eclipse project under the rec9 package.

You have the following files in the rec9/part1 folder:
The specifications of Stack and Queue Abstract Data Types:

- StackInterface.java
- QueueInterface.java

The files related to the traversal of the maze:

- Square.java
- Maze.java
- MazeApp.java
- MazeSolver.java

We also provide the JUnit tests for testing your solution:

- MiniListTest.java
- MazeSolverTest.java

You will need to **modify** the following file:

- [ ] MiniList.java

In file MiniList.java we have a basic implementation of the Linked List data structure. This linked list is a singly-linked list. It supports a very minimalistic functionality by providing access only to its ends: we can *add*/*remove*/*read* elements only in the beginning or at the end of this list.

1.1. Read the code

Read the code in file MiniList.java and note the following:

- The class uses *generics* and it stores nodes that contain data of any generic type E
  `public class MiniList<E>`

- There is a private class Node within the *MiniList<E>* class
  `private class Node {  ...  }`
  We do not need this *Node* outside the *MiniList* class: the classes that use our mini-list do not need to know that there are nodes inside the list: they just need to be able to access data at the beginning or the end of the sequence.

- Front methods for MiniList<E>:
  - `public void addFirst(E data)` //add element to front of list
  - `public E getFirst()` // return data of head node
  - `public E removeFirst()` //remove very first item from list

- Back methods for MiniList<E>
  - `public void addLast(E data)` // add item to the very end of list
  - `public E removeLast()` // remove item at the very end of list
    Note a sample traversal loop that is needed to reach the end of the list before we can perform the operations there.

1.2. Test this implementation with MiniListTest.java

Run the JUnit test in MiniListTest.java. You will see that though every method is implemented correctly, the test is stuck at the specific test *testAddLastTime()* which takes too long to complete. This test adds a ton of data, so it takes a lot of time to traverse the list before a new data can be added at the end. We can improve the situation by using a *tail pointer*.

1.3. Adding tail pointer

First, add a Node *tail* reference below the Node *head* instance variable of MiniList<E>.

Now… you must review all the methods and take care of the tail pointer. You should probably update the following operations:
addFirst(E data), removeFirst(), addLast(E data), removeLast() and the constructor of *MiniList()*

Here are some useful notes:

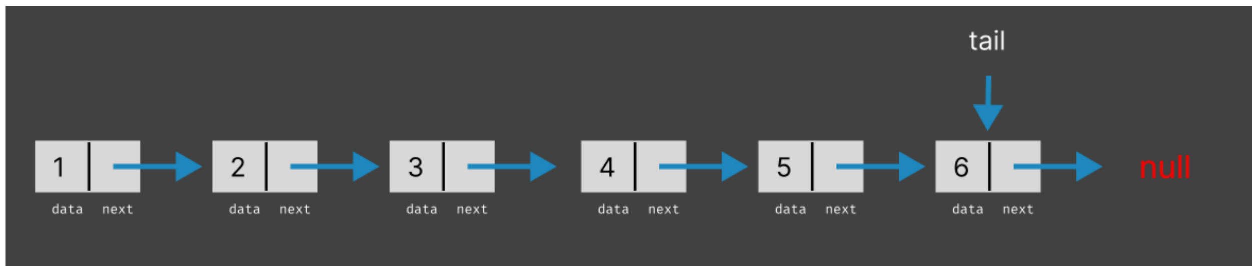
- Every time the *tail* node is removed or changed via an *addLast*() operation, adjust the tail to track the last node.
- If *head* is null, then *tail* has to be null.
- if linked list has one item, then *tail* equals *head*.
- if *head* is removed and the list becomes empty, *tail* must also be null

1.4. Rerun test in MiniListTest.java

All tests should pass now!

If there are new errors, please double check your tail logic.

- have you managed every case where the tail node needs to be updated?
- have you managed every case where the linked list has
    - 0 Nodes ?
    - 1 Node ?
    - N items ?



Your implementation of `MiniList` should be just enough to implement Stack and Queue ADTs.

Part II. Implementing Stack and Queue [18 points]

In this part you will use the MiniList.java to implement Stack and Queue.
Note: Implementing these ADTs by using a different data structure or by using any Java collection will result in zero grade for the Part II.

Add all the java files from the part2 of the starter code into the same rec9 package:

- MyStackTest.java
- MyQueueTest.java
- MazeSolverStack.java
- MazeSolverQueue.java

These files cannot be compiled, because the two files are missing. Add the following **new** files to rec9 package:

- [ ] MyStack.java
- [ ] MyQueue.java

2.1. Implement Stack and Queue

We've been talking about stacks and queues in class, and now it is your time to put that theory to good use. Write two classes `*MyStack<T>*` and `*MyQueue<T>*` that implement the supplied interfaces `*StackInterface*` and `*QueueInterface*`, respectively.

Both `MyStack` and `MyQueue`should make good use of a simple Linked List data structure implemented in Part I.

You can add an instance variable of type *MiniList* inside each of `*MyStack*` and `*MyQueue*`, which could look similar to this:

```
public class MyStack<T> implements StackInterface<T> {
  private MiniList<T> list = new MiniList<T>();

}
```

MyStack

`*MyStack*` implements `*StackInterface*`.
Implement operations declared in this interface using the private variable of type `*MiniList*` to store the elements of the stack. Think which end of the list should be used to make *pop*/*push* operations most efficient.

MyQueue

`*MyQueue*` implements the provided `*QueueADT*` interface.
Use the same `*MiniList*` as the underlying data structure. Also pay attention to which ends of the list you should use for *enqueue*/*dequeue* operations.

2.2. Test your Stack and Queue

After implementing, you should run JUnit tests for `*MyStack*` and `*MyQueue*` that perform testing of your implementations of Stack and Queue. These are provided in MyStackTest.java and MyQueueTest.java respectively.

Run JUnit tests for `MyStack` and `MyQueue`. Pass all the tests.

Part III. Use your work for solving mazes [4 points]

Think about the maze-solving algorithm discussed in Recitation 9.

If you implemented everything correctly, you should be able to run the `MazeApp` program and get a GUI interface that will allow you to visualize the process of traversing the maze using Stack or Queue. You do not need to modify anything in `MazeApp.java` to do that.

The `load` and `quit` buttons operate as you might expect. You can load different files from the *mazes* folder supplied in the starter code, and run different solvers on them. 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. As the animation proceeds, squares that were already explored are painted gray. If the maze is solved, the squares on the solution path are painted yellow. The `getPath()` method is used to get the path and display it in the bottom panel.

Play with the app and observe the difference in maze traversals when the Square objects are added to and removed from the Stack vs. the Queue.

Before the final submission run all test files and make sure all the tests passed:

- MiniListTest.java
- MyStackTest.java
- MyQueueTest.java
- MazeSolverTest.java

Submission

Prepare the submission that contains the content of the rec9 package with **java files only**. **Do not include** the `mazes` folder. Then zip the entire rec9 package and turn it in via Canvas.

In the comments write the following:
- Any known problems or assumptions made in your classes or program.
- Any comments or suggestions you want to share about this assignment.

Now answer the questions in the **Recitation 9 quiz**.