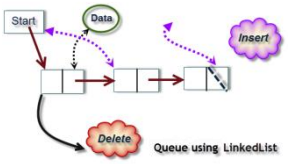
Linked List Queue



A queue-type class *could* be written using a backing array, similar to how you wrote the MyStack class earlier in the year. Also, recall that you did not write a queue class in the queue labs. *The time has come for this – prepare yourself.* However, rather than using a backing array, your queue class will use a *linked list* to store elements.

Understanding linked lists is crucial to understanding the more complex data structures of the future (trees, heaps, graphs, etc.). Future ADTs will rely heavily on a series of references, similar to linked lists.

1. Copy/paste your previous project's MyLinkedList class into the current project. Add the following constructor to your MyLinkedList class:

   ```
   public MyLinkedList(int... vals)
   ```

   The ellipses (`...`) indicates this constructor takes a ***variable number of arguments***. Google "java varargs" for more info! Varargs constructors are really nice for testing, as you can construct objects with initial values (without having to repeatedly `add`). Test that your varargs constructor works.

2. It would be nice if your linked list class could store *any* type of object (rather than just integers), wouldn't it? Thankfully that *is* possible, using a Java language feature called ***generics***.

3. Make the necessary changes to convert your MyLinkedList class to MyLinkedList<T>, such that it can now store any type of object. When using generics, the type parameter is specified after the class name, delimited by angle brackets. Example:

   ```
   public class Nonsense<T>
   {
          private T value; //instance variable of some type T

          public Nonsense(T value) { this.value = value; } //constructor

          public T getValue() { return this.value; } //getter
   }
   ```

   When a new Nonsense object is instantiated, the user of the object should specify a type. The type specified will replace the `T` generic (placeholder) type in the Nonsense class. You've seen this many times, e.g. with ArrayList or Stack objects:

   ```
   ArrayList<String> list = new ArrayList<>();
   ```

   The type parameter in this case is String, and String replaces the generic type (`T` in the example above) everywhere it occurs when the object is instantiated.

   Note: your ListNode inner class does not need to be parameterized – it should use the `T` type supplied to the MyLinkedList class. A ListNode will store an object of type `T` called `val` and a reference to the `next` node (re-declaring the `T` variable in ListNode will hide/shadow MyLinkedList's `T` variable).

Note: When checking nodes' data elements (`val`) for equivalence, you can't use == any longer; this class is now generic, it could store any type! You can't assume it will store primitives.

It's up to the supplied type to tell Java how to determine equivalence between two of its objects (each type should override their own `equals` method). Luckily, all built-in reference types (e.g. String, Integer, etc.) already do this.

A runner class with a `main` is provided; your output should match that in the **"output.txt"** file in the lab folder. Feel free to add more tests as you see fit.

MyQueue<T>

The implementation of a queue class is quite similar to that of a stack class; the functionality merely changes from "LIFO" to "FIFO".

Write a class **MyQueue.java** that implements a queue data structure in a ***generic*** fashion.

Your MyQueue<T> class is essentially a wrapper around a MyLinkedList<T> object (which stores the objects added to the queue), and retrieves elements in a queue-like fashion. MyQueue<T> should implement the QueueADT<T> interface (supplied, in the lab folder) to ensure compatibility with future classes, and should have the following:

Instance variables

`MyLinkedList<T> queue`	the "backbone" of the queue (rather than a regular array or an object of Java's ArrayList or LinkedList class) – what stores the actual objects

Methods

A default and varargs constructor

`boolean isEmpty()`	returns true if the queue is empty
`void offer(T item)`	add an element to the queue
`T poll()`	removes and returns the element at the head of the queue
`int size()`	return the number of elements in the queue
`void clear()`	empty the queue

MazeSolverQueue

Do you recall the mystery surrounding the use of the term "worklist" (and the addition of the abstract MazeSolver class) in the MazeSolver lab? You're about to find out why!

You will again extend the MazeSolver class, this time with a class **MazeSolverQueue.java**. This class should use an instance of your MyQueue<T> class for storing Squares to be explored.

Once you complete the MazeSolverQueue class, import a new copy of the MazeApp class from the source folder (it has a little bit of code added to support switching between a stack and queue solver type), and run its `main` method. Load a maze (text files, you can navigate to the previous project's folder or copy/paste them into this project). Next, click the **"stack"** button on the GUI – this button toggles between using a stack and a queue for the solving algorithm.

Run both and note the differences between stacks and queues for this program. The queue implementation *will* solve mazes (and in some cases, may even take a shorter path), but clearly the LIFO stack is the better choice of data structure for most mazes! (For the curious, try the bigger mazes.)

Finally, given your knowledge of linked lists, it's time to improve the `getPath` method. Previously, the `getPath` method simply returned whether or not the maze was solved (or unsolvable). Wouldn't it be nice if it would return the actual path itself? Make the following changes to support this behavior:

`MazeSolver.step()`

To display the path the solver took from the start to the exit, you need some way of retracing your steps.

Here is the suggestion for adding this functionality: in the `step` method, 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?").

To accomplish this, you could add a new instance variable `Square previous` to the Square class, which will represent the square explored "prior" to the current one; initialize this variable to `null` in the constructor / `reset` method. Then, when a square is being added to the worklist for the first time, you will set the `previous` variable to reference the "current" square (the square that is currently being explored). If the `previous` variable is not `null`, this square has already been placed on the list and shouldn't be added again.

If each Square maintains a reference to the square that added it to the path, which maintains its own reference to the square that added *it*, and so on... you've basically created a linked list!

Make the changes to the necessary classes to support this behavior.

`MazeSolver.getPath()`

Modify the `getPath` method as follows: If the maze isn't solved yet, you should probably return a message indicating such.

If it is solved, this method returns either a String of the solution path as pairs of coordinates `[r,c]` from the start to the exit, or a message indicating no such path exists. You can get the exit path by starting with a reference to the exit square and walking backwards via the `previous` references to the start, like this:

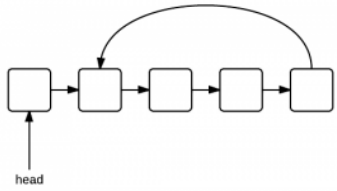
```
Square current = this.maze.getExit();
current = current.getPrevious(); //current now references its predecessor
```

Think about what type of data structure may be helpful to get the path in the correct order... It may be necessary to add other methods / variables to the MazeSolver class to assist this method.

Run MazeApp's `main` method. Solvable mazes should now display the path from the start to the exit as a series of maze locations. Nice!

(Advanced) Cycle detection

In linked lists, a "cycle" is a series of references that continually refer to each other in a cyclical fashion. A picture is worth a thousand words:



Devise an algorithm that will detect a cycle in a singly linked list (use your MyLinkedList class) without using additional data structures (i.e. no additional Lists, Sets, Maps, etc.). To do this without help would be challenging (though definitely possible). Some hints in no particular order:

- Traversing the list and looking for an end makes sense, but it won't work if there *is* a cycle
- Adding a boolean field to the ListNode class that will remember if the node has been "seen" is a good idea, though it requires you to modify the list and therefore won't work in a general sense if you don't control the source (i.e. you're not writing the class)
- A cycle can occur anywhere in the list – the tail (last) node of the list will not necessarily be the node that links back to a previous node

The first hint is given below (change text color to view):

If you have tried your best and are still not having any luck, try the next hint:

(Advanced) Merge `k` sorted lists

Complete the method `ListNode mergeK(ListNode[] heads)` that will merge `k` sorted lists, and return a single list (a ListNode that is the head of the completed list).