Programming Assignment for Module 3, Part 1: Implement a Linked List

You've learned about Linked Lists, now let's apply your knowledge to create a Linked List yourself. In practice, you will almost always use built-in libraries for data structures like Linked Lists, but the notion of a "linked" data structure is so common and critical in computer science that it warrants being sure you can do this yourself. In addition, you'll be making more complex "Linked" data structures in this course and the next, and Linked Lists are the right place to start. Lastly, you'll need your Linked List for the second part of this module's project, which is where you'll auto-generate text which resembles a source text!

Getting Set Up

Before you begin this assignment, make sure you check Part 2 in the setup guide to make sure the starter code has not changed since you downloaded it. If you are an active learner, you will have also received an email about any starter code changes. If there have been any changes, follow the instructions in the setup guide for updating your code base before you begin.

1. Ensure you have the starter code

Be sure you have the starter code you downloaded as part of the first programming assignment on Flesch Score. You should see a package called textgen in that starter code.

To verify everything is setup okay, you can run the "MyLinkedListTester.java" file and you should get JUnit reporting 6/6 tests run with 1 Error and 1 Failure. (This is okay, you'll be fixing it soon!)

2. Open and examine the starter code

In this project, you will write a MyLinkedList class to be your own implementation of an AbstractList. Before you begin coding, you should be comfortable with the principles behind a LinkedList and the notion of extending from AbstractList. For the Java documentation on Abstract Lists, please see here.

Open the starter code for module 2 by expanding the MOOCTextEditor->src folder to see the package **textgen**. You will only be focused on **MyLinkedList.java** and**MyLinkedListTester.java** for this part of the assignment. You can open these two files by double-clicking on them. (Note that there is a MyLinkedListGrader.java file in the folder which holds the tests we run when grading. You won't be using that file yet.)

Notice that MyLinkedList extends AbstractList and that, like AbstractList, it uses Generics so you can hold any type of element. The MyLinkedList class contains a package visible class "LLNode" which represents a single element in a doubly-linked list.

Assignment and Submission Details

Your submission of this assignment is divided into two parts: your code which implements a Linked List (**MyLinkedList.java**) and your code to test a Linked List implementation (**MyLinkedListTester.java**). You will submit a separate file for each part, as described below, but *our guidelines for this project will more closely reflect how you will develop a project in industry*. **We strongly recommend you write your tests as you write your implementation (and vice-versa).**

Step 1: Implement and test creating a Linked List Object, adding elements, and retrieving elements.

To get started, you need to decide if you wish to use sentinel nodes (like we've seen in the videos) or skip sentinel nodes. We encourage you to use sentinels but either choice is okay.

1. A. Review the tester methods provided in MyLinkedListTester.java:

In the file MyLinkedListTester.java, review the setup() and fairly robust testGet() methods in MyLinkedListTester.java. These tests give you an idea of how your Linked List constructor, add method, and get method should perform. You can run them in JUnit by running the MyLinkedListTester.java file. It should automatically run as a JUnit test suite.

1. B. Author the following methods in MyLinkedList.java:

public MyLinkedList()

This constructor should create the Linked List and setup any instance variables as needed.

public E get(int index)

This method returns the node in the list corresponding to the index. For example, if the list has the elements 6 and 9 (in that order) and you call "get" with index 0, you should get back 6. If the method is called with an invalid index (say -1 or 2 in the example above), you will thrown an IndexOutOfBoundsException.

public boolean add(E element)

This method adds an element to the end of the list. Be sure to think about what happens when you add an element to an empty list as well as a list with already existing elements. Drawing a picture of the list will help!

1.C. Run existing JUnit Tests by Running MyLinkedListTester

Run "MyLinkedListTester" and if your methods are working, you should get no errors or failures. If this is the case, the testGet method worked properly and your methods are likely working properly.

1.D. Author "testAddEnd" method in MyLinkedListTester

The tests for "testGet" may not properly stress the add method. To be sure your add method works properly, you may wish to add a couple tests here.

Step 2: Implement and write tests for the remaining methods in MyLinkedList.

As we did in Step 1 above, you'll want to continue implementing and testing your **MyLinkedList**class. To do so, add and test the methods below in the ordering of your choice. You also must add JUnit tests the the **MyLinkedListTester** class to thoroughly test each method. You will be graded not only on the correctness of your implementation, but also on the completeness of your tests.

To facilitate auto-grading, you need to read the method descriptions carefully to ensure they have the correct behavior. Also, make sure you are consistent with our test cases/examples in the main method. But, you have the freedom to implement the doubly-linked list with, or without, a sentinel head and tail node. Our videos detail how to do this with the sentinel nodes, but feel free to omit the sentinels for extra challenge.

public int size()

This method returns the size of the list. Be careful to update your size instance variable when adding and removing elements. To test this method, use the method in MyLinkedListTester called testSize

public void add(int index, E element)

This method adds an element at the specified index. If elements exist at that index, you will move elements at that index (and beyond) up, effectively inserting this element at that location in the list. Although drawings are helpful for implementing any method, you will benefit heavily from drawing out what should happen in this method before trying to code it. You will want to throw an IndexOutOfBoundsException if the index provided is not reasonable for inserting an element. Optional: After authoring this version of the add method, you could consider removing redundant code by having the add method you originally wrote just call this method. To test this method, use the method in MyLinkedListTester called testAddAtIndex.

public E remove(int index)

This method removes an element from that location. Similar to the "add" method, we strongly encourage you to draw this out before trying to code it. Like the "get" method, you will want to thrown an IndexOutOfBoundsException if the index provided is out of bounds. To test this method, use the method in MyLinkedListTester called testRemove.

public E set(int index, E element)

This method allows you to change elements in place by specifying a valid index of an existing element and a new value to change that element to. You should return the old value held at that index. Like the "get" method, you will want to thrown an IndexOutOfBoundsException if the index provided is out of bounds. In addition, if someone tries to set a node with a null element, you should throw a NullPointerException. To test this method, use the method in MyLinkedListTester called testSet.

Hints: Potentially Useful (but not required) Methods

public String toString()

Although not included in the required methods for this class, you may find a "toString" method for the list and/or a "toString" method for each node helpful when testing and debugging.

Additional Constructor for LLNode

You may find yourself doing the same operations multiple time in the MyLinkedList class. If so, you may want to consider a constructor that takes not only the element, but the previous and (possibly) next nodes.

What and how to submit

Submission Part 1: Submitting MyLinkedList Implementation

You will need to upload the **MyLinkedList.java** file for grading. The grading will test the behavior of various methods. If you were diligent in self-testing, this should pass with flying colors. But don't worry if we catch a few cases you missed. Programming a LinkedList can be challenging (and we know the common mistakes our students make). If you are stuck on why you are failing some of our tests, you can refer to the MyLinkedListGrader.java for the tests we run.

Submission Part 2: Submitting MyLinkedListTester

You will upload the file **MyLinkedListTester.java** for grading. Here we'll examine your JUnit test methods to ensure you are catching incorrect behavior. We'll run your JUnit Test cases against both working and flawed versions of MyLinkedList implementations to check that your tests pass when the implementation is correct and catch errors when they are incorrect.