CSE12 - Spring 2018 HW #2

Linked Lists

(200 points)

(2 week Assignment)
Due 11:59pm 28 April 2018

In this lab you will implement a doubly linked list, and create JUnit tests to verify its proper operation.

- You will implement a doubly linked list,
- Build a ListIterator for the list
- Perform some big-O and Theta analysis

This assignment is an individual assignment. You may ask Professors/TAs/Tutors for some guidance and help, but you can't copy code. You can, of course, discuss the assignment with your classmates, but don't look at or copy each other's code.

You will submit the following files for this assignment:

- **MyLinkedList.java** -- Provided for you as a starter file. You will add to it in both parts of the assignment.
- **MyLinkedListTester.java** -- Tester files that you will create by copying and augmenting the starter file LinkedListTester.java that we provide
- **README.txt** -- A plain text file that you will create and add information to as prompted below in the assignment.
- **Part2.txt –** Answers to some analysis questions

Part 1 - Doubly Linked Lists

Your first task is to implement a doubly linked list called **MyLinkedList** and a corresponding **ListIterator** for your class.

To ease the time burden, we are providing starter files for both testing and the MyLinkedList class itself. It is highly recommended that you generate the javadocs for the supplied MyLinkedList.java file. You should be able to complete the assignment by properly programming the "Stubbed-out" methods. Make sure to look in the source code for the string "XXX-CHANGE-XXX", these indicate method returns that need to be changed to reflect proper function. the return statements are there so that the supplied MyLinkedList.java will compile.

You should also Look up the -private flag to javadoc. You will find it useful. We recommend printing out the generated javadoc and use it as a checklist

MyLinkedList

Your MyLinkedList class is just a subset of the Java Collection's Framework LinkedList, and therefore should match its behavior on the described subset.

Implementing the LinkedList interface directly is a bit painful, so Java makes it easier by providing a skeletal implementation of the List interface in the AbstractList abstract class. Your MyLinkedList class must extend AbstractList (You will receive no credit if your class does not extend AbstractList directly). The starter file already does this for you.

[Side note, can be skipped: If you look carefully at the AbstractList documentation, you will see that Java recommends that MyLinkedList should actually implement AbstractSequentialList. The reasons for this are somewhat subtle and should make more sense in a week or two.]

In doubly linked lists, the removal of items can be especially tricky as you need to be sure to properly update the neighbor's next and previous pointers, as well as handle the special cases for removal from the head (front) or tail of the list.

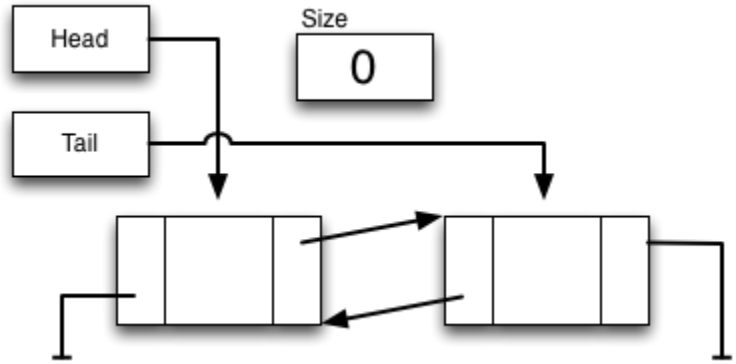
You should not allow "null" objects to be inserted into the list; if the user of your class attempts to do so, your code must throw a NullPointerException. Please note that this specification is different from LinkedList from the Java Collection's Framework, which does allow you to store null objects.

What follows are the methods that you must implement in your MyLinkedList class.
Before you start, read the online documentation of AbstractList and make sure you can answer the following questions (you do not have to submit the answers):

- Which method(s) is (are) defined as abstract in MyLinkedList's superclass?
- What happens if you do not override the add() method of the superclass and a user of your ADT invokes the add() method?
- According to the AbstractList documentation, what are all the different types of exceptions an ArrayList can throw?

Constructors

You should only need to have a single public 0-argument constructor that creates an empty list and initializes all the necessary variables to track your list, as in the following diagram.



Private Methods

Node getNth(int index)

a method that returns the **Node** at a specified index, not the content.

You might also want a removeNode(Node n) method that allowed to keep from duplicating code in your Iterator. But you aren't required to implement one unless you want to.

Public Methods

boolean add(T data)
void add(int index, T data)

add an element into this list (either at end or by index)

throw a NullPointerException if the user tries to add a null pointer

throw IndexOutOfBoundsException as needed (same rules as MyArrayList)

Note: the boolean add method will presumably always return true; it is a boolean function due to the method definition in AbstractList

T get(int i)

get contents at position i

throw IndexOutOfBoundsException as needed

T set(int i,T data)

set the value at index i to data

throw NullPointerException if data is null

throw IndexOutOfBoundsException as needed

T remove(int i)

remove the element from position i in this list

throw IndexOutOfBoundsException as needed

void clear()

remove all elements from the list

boolean isEmpty()

determine if the list is empty

int size()

return the number of elements being stored

Programming hints

- You'll probably want to create a nested class (that is, a class inside a class) to represent a node in your linked list. A nested class is sometimes called an inner class. To accomplish this, you cannot declare a Node class as public, but you can include it in the same file (and even in the same class) as MyLinkedList. If it is contained within **MyLinkedList<T>**, then it can just use the generic label **T** because it is within the class.
-
```
public class MyLinkedList<T> extends AbstractList<T> {
    class Node {    // this is called a Nested Class. Only usable within MyLinkedList
        T data;
        Node next;
        Node prev;

        // more code here
    }

    /* Lots more code will go here */
}
```

Testing your List

Download the supplied file LinkedListTester.java. This is a starter file the defines a small number of tests against the Java Collection's Framework LinkedList class.

- Compile and run LinkedListTester.java as-is.
- **Modify this file to create *at least* 10 (ten) additional meaningful tests on the public methods described above**. Recall lecture, where you often want to test the a) common case (all inputs are valid), b) error case (where bad is given and your code reacts accordingly) and c) edge cases (Null elements is a good edge case for this assignment!). In your README.txt file, briefly describe what each of these tests are attempting to validate. This will test against LinkedList's.
- **Copy** Your LinkedListTester.java and rename it to be MyLinkedListTester.java. (Don't forget to redefine the class from LinkedListTester to MyLinkListTester), and modify the tests to create MyLinkedList objects. You should be able to just change LinkedList to MyLinkedList throughout so you get lines like **MyLinkedList<String> x = new MyLinkedList<String>();**

Part 2 - ListIterator

(Note that a ListIterator isA Iterator, but has additional methods defined). ListIterators define the notion of a "cursor" so that a user of the ListIterator can ask for the list element just before the cursor OR the one following the cursor.

Once your MyLinkedList class is working, you will need to create a ListIterator which is returned by the **listIterator()** method. You can do this one of two ways:

1. Through the use of an anonymous class:

```
public ListIterator<T> listIterator(){
   return  new ListIterator<T>(){
      // TODO - your code  here
   };
}
```

2. Or through the use of an inner helper class (contained inside the definition of the MyLinkedList class). **This is the approach taken in the example file and is highly recommended**. ListIterators are relatively complex classes.

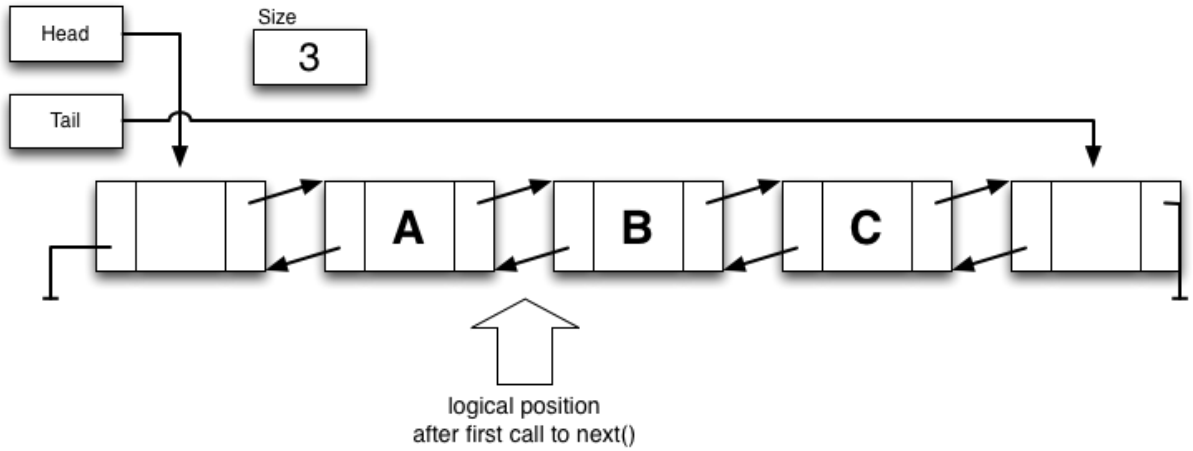
```
class MyListIterator implements ListIterator<T> {

   // class variables here

   public boolean hasNext()  {
         // your code here
   }

   // more methods, etc.
}
```

The ListIterator is able to traverse the list by moving a space at a time in either direction (**make certain to write multiple tests to verify**). It might be helpful to consider that the iterator has size+1 positions in the list: just before the head, between the 0 and 1 index, ..., just after the tail. After one call to next(), the iterator is logically in the state shown below



In either case, you will need to implement all of the methods of a listIterator -- **hasNext()**/**next()**,**hasPrevious()**/**previous()**, **nextIndex()**/**previousIndex()**, **remove()**, **set(x)**, and **add(x)**. See the ListIterator JavaDoc for details, but there are notes below.

boolean hasNext()

Return true if there are more elements when going in the forward direction.

T next() Return the next element in the list when going forward. Throw NoSuchElementException if there is no such element

boolean hasPrevious()

Return true if there are more elements when going in the reverse direction.

T previous() Return the next element in the list when going backwards. Throw NoSuchElementException if there is no such element

int nextIndex()

Return the index of the element that **would** be returned by a call to next()

Return the list size if at the end of the list

int previousIndex()

Return the index of the element that **would** be returned by a call to previous()

Return -1 if at the start of the list

void set(T x) Change the value in the node returned by the most recent next/previous with the new value. Throw an IllegalStateException if neither next nor previous were called

Throw an IllegalStateException if add or remove have been called since the most recent next/previous

void remove() Remove the last element returned by the most recent call to either next/previous Throw an IllegalStateException if neither next nor previous were called Throw an IllegalStateException if add has been called since the most recent next/previous

void add(T x)

Insert the given item into the list immediately before whatever would have been returned by a call to next()

The new item is inserted before the current cursor, so it would be returned by a call to previous() immediately following.

The value of nextIndex or previousIndex both are increased by one

Throw a NullPointerException if x is null.

Which Exceptions to throw for ListIterator?

Your methods should properly throw InvalidStateException, NoSuchElement, and NullPointer exceptions. This means you DO NOT HAVE TO IMPLEMENT throwing ALL exceptions as the javadoc would indicate. In addition to the documentation above, the starter file already indicates through the method header which exceptions you should be throwing in which methods.

Your class MUST support multiple ListIterator instances

When testing your code, we create multiple ListIterator instances on the same Junit test fixture to verify that each instance is independent of all others. When testing this particular feature of your code, we will NOT invoke the remove() method or add() methods (in other words when testing multiple instance of the ListIterator, the list structure will not be modified, elements might be modified with set()). We will test the modification routines (add, remove) when just a single ListIterator instance has been created on a Junit test fixture.

Programming Hints

It is useful to have a number of state fields to simplify the writing of the various methods above. Since the cursor of the ListIterator exists logically between 2 nodes, it is useful to just keep references to the next Node and the previous Node. It may also be helpful to keep an int value of the index of the next node.

If you construct your MyLinkedList to use sentinel nodes as discussed in the book and lecture, and you properly throw exceptions for going out of range, you shouldn't have to worry about checking for null values at the ends of the list since the sentinel nodes are there.

Since set() and remove() both change based on what direction was last being traversed, it makes sense to keep a boolean flag to indicate a forward or reverse direction.

Public Methods to add to MyLinkedList.java

Once you are **sure** your iterator is working, you should override the following methods in MyLinkedList. Each of these should just create a new MyLinkedListIterator and return it.

ListIterator<T> listIterator()

Iterator<T> iterator()

have these factory methods return your ListIterator class for the current list.
Note: You inherit a working ListIterator from AbstractList, but the one you create will be more efficient. We suggest that while you are building and initially testing your ListIterator, you create a differently named factory method to use. You could use names like QQQiterator() and QQQlistIterator() until you are sure it is working correctly. If you jump right into overriding iterator()/listIterator() then things like toString() may stop working for you.
Be sure to have it called just iterator() and listIterator() in your submitted MyLinkedList and JUnit code. SEE THE COMMENTS AT THE END SUPPLIED MyLinkedList.java file.

Testing your Iterator

You'll want to test your **ListIterator** and be sure that it works properly. As a partial test, look at the supplied JUnit test in MyLinkedListTester.java that creates a fibonacci sequence. You should understand how this particular test works as a sample.

You should also look at the supplied code called Sieve.java which is a LinkedList implementation of the Sieve of Eratosthenes to compute prime numbers in a range. It works properly with the Java Collection Framework LinkedList. You should be able to change one boolean variable (See comments), recompile and have it use your MyLinkedList implementation. The program should run properly with either implementation. This is more than a "unit" test, but it is a good validation of your program.

Apportioning your Time

This is a much more comprehensive programming assignment than HW1. If you approach it incorrectly, it will take too long. Suggested amount of time and order of doing things**. Don't try to do the assignment all at once. Spread it out over multiple days and make backups of your in-progress code.**

1. Write tests against LinkedList that will help you develop your Linked list implementation without testing the ListIterator class (beyond what is already in the supplied starting tests). This will help you understand the LinkedList interface. You (obviously) only need to write tests for methods that are common to both LinkedList and MyLinkedList. . Make sure your test suite runs properly against LinkedList instances. (1 - 2 hours)
2. Copy/rename your testing class to MyLinkedListTester, compile it, run it against the supplied outline of MyLinkedList. You should have *many* test failures.
3. Develop the linked list methods (Node inner class, methods). Develop until all the tests you developed passed. Change Sieve.java to use MyLinkedList and verify it works. (2 - 4 hours).
4. Develop some test methods for testing the iterator (see the code and notes above). The ListIterator is more involved because of tracking states. Try some tests that move the iterator forwards and backwards. (1 - 2 hours)
5. Develop the ListIterator implementation using your tests to help you (2 - 4 hours).
6. Uncomment the lines near the end of the starter MyLinkedList.java file, to make your ListIterator active.

README.txt file

Include in your submission a file named README.txt. The contents of the README.txt file should include the following:

1. Your name
2. Any known problems or assumptions made in your classes or program

Look through your programs and make sure you've included your name at the top of all of them. We are expecting you to hand in the following files:

- MyLinkedList.java
- MyLinkedListTester.java
- README

How your assignment will be evaluated.

- Coding Style
    - Does your class and tester properly generate javadoc documentation
    - Is your code readable by others
        - consistent indentation (TABs or Spaces is OK)
        - are variables sensibly named
        - are access modifiers (public, private, protected) used appropriately
        - are helper methods used when needed to reduce code duplication
- Correctness
    - Does your code compile
    - Does it pass all of your unit tests that you defined
    - Does it pass all of our unit tests (we are users of your ADT implementation, you don't get access to our unit tests)
        - We'll check basic functionality, including Exceptions
    - Does your code have any errors (e.g. generates exceptions when it isn't supposed to) (That would be bad).
- Unit test coverage
    - Have you created at least 10 more meaningful unit tests? (that's a bare minimum, you are very likely to have significantly more tests)
    - Does your unit testing approach for listIterator() appear to be sufficient? We supply you with one unit test for listIterator(), that is not sufficient. Good tests will include testing what happens before the head and after the tail

- We're not telling you exactly what to test, that's part of learning to create approaches to developing and debugging more complicated code. **We are also NOT expecting full test coverage**. Your tests should help you develop your code and insure its proper operation.

PART 2 - Analyzing Runtime

In this part of the assignment, you will turn in a traditional "homework" (non-programming assignment)

You will submit the following file for this assignment:

- **Part2.txt**

This will be a plain text file (NOT .docx or .pdf or anything else). You should use the ^ symbol to indicate exponentiation (e.g. n^2 should be written n^2) and write out the words Theta (Θ) and Omega (Ω) where appropriate. At the top of your Part2.txt file, please include the following header:

```
CSE 12 Homework 2
Your Name
Your PID
The date
```

Warm up with Big-O, Big-Theta and Big-Omega (20 points)

True/False. In a section labeled TRUE/FALSE in your Part2.txt file, state whether each of the following equalities are true or false. You should put the number of the question followed by either the word True or False. One answer/line. eg.

```
TRUE/FALSE
 1. True
 2. False
```

and so on.

1. $n^2 + 100 = O(n^4)$
2. $n^2 + 100 = O(n^2)$
3. $n^2 + 100 = O(n)$
4. $n^2 - 1,000,000,000 = O(n)$
5. $n^2 + n = O(n)$
6. $n^2 + n = O(n^2)$
7. $n^2 * n = O(n^2)$
8. $n^2 * n = O(n^3)$
9. $n^2 + \log(n) * n^2 = O(n^2)$

10. $n^2 + \log(n) * n^2 = O(n^2)$
11. $n^2 + 100 = \Omega(n^4)$
12. $n^2 + 100 = \Omega(n^2)$
13. $n^2 + 100 = \Omega(n)$
14. $n^2 + n = \Omega(n)$
15. $n + 100,000 = \Omega(n)$
16. $n^2 + n + 100 = \theta(n)$
17. $n^2 + n + 100 = \theta(n^2)$
18. $n^2 + n + 100 = \theta(n^3)$
19. $n^2 * n + 100 = \theta(n^3)$
20. $100 * n \log(n) = \theta(n \log n)$

Analyzing running time of code (20 points)

In this part you will practice your skills of estimating running time of code snippets. For each piece of code below, state the running time of the snippet in terms of the loop limit variable, *n*. You should assume that the variables n and sum are already declared and have a value. You should express your answer using Big-O or Big-Θ (Theta) notation, though your Big-O bound will only receive full credit if it is a tight bound. We allow you to use Big-O because it is often theconvention to express only upper bounds on algorithms or code, but these upper bounds are generally understood to be as tight as possible. In all cases, your expressed bounds should be simplified as much as possible, with no extra constant factors or additional terms (e.g. O(n) instead of O(2n+5))

For each piece of code, state the running time and then give a short explanation of why that running time is correct. We are not asking for a formal proof--you'll learn how to do that in CSE 101. For now your explanation should simply include an (approximate but reasonable) equation for how many instructions are executed, and then a relationship between your equation and your stated bound. Place your answers in in your **Part2.txt** file in a section labeled RUNTIME.
Here is an example:

Ex.

```
for ( int i = 5; i < n; i++ )
  sum++;
```

Most precise answer:
Running time O(n)
Explanation: There is a single loop that runs n-5 times. Each time the loop runs it executes 1 instruction in the loop header and 1 instruction in the loop header and 1 instruction in the body of the loop. The total number of instructions is 2*(n-5) + 1 (for the last loop check) = 2n-9 = O(n).

A slightly less precise but still acceptable for full credit answer:
Running time: O(n).
Explanation: There is a single loop that runs n-5 times. Each time the loop runs it executes 1 instruction, so the total number of instructions executed is 1* (n-5) = O(n) (also OK: Θ(n)).

```
1. for ( int i = 0; i < n; i+=2 )
       sum++;

2. for ( int i = 1; i < n; i*=2 )
       sum++

3. for ( int i = 0; i < n; i++ )
       for ( int j = 0; j < n; j++  )
           sum++;


4. for ( int i = 0; i < n; i++ )
       sum++
   for ( int j = 0; j < n; j++ )
        sum++
   // The above are two loops one after the other, NOT  nested

5. for ( int i = 0; i < 2*n; i++ )
       sum++

6. for ( int i = 0; i < n*n; i++ )
       sum++;

7. for ( int i = 0; i < n; i++ )
       for ( int j = 0; j < n*n; j++ )
           sum++;

8. for ( int i = 0; i < n; i++ )
       for ( int j = 0; j < 10000; j++ )
           sum++;
```

TURNING IN YOUR ASSIGNMENT

Similar to PR1, we will supply you a bundlePR2 and checkCompilePR2 script that are to be run on the UCSD lab machines under your cs12s<zz> account. The resulting tar file upon successful creation will then be uploaded to gradescope.

The following will be collected

- **MyLinkedList.java**
- **MyLinkedListTester.java**
- **README.txt**.
- **Part2.txt**

PLEASE START ON THIS ASSIGNMENT IN WEEK 3 SO THAT YOU CAN COMPLETE IT!