Project 2: Doubly-Linked Lists, 200 points

(based on Prof. Alvarado's writeup).

Due: Wednesday, January 25th, at 23:59:59

Overview

1. In Homework 2, you will implement a Doubly-Linked List, including an Iterator over that list. In particular, you will implement the List interface and several additional methods. Note that you **MUST** implement a Doubly-Linked List -- you cannot, for instance, instead implement an array-based list.
2. You will also implement a ListIterator class in order to traverse your Doubly-Linked List.
3. You will then create a lot of JUnit tests to check your Doubly-Linked List and Iterator methods.
4. If you want more of a challenge, there are a few Extra Credit problems at the end for you to try out.
5. This Project has an Extra Credit milestone submission. Please refer to the Submission section below for more details.
6. **I strongly recommend that you read the entire write-up before getting started, as you will get a general idea of prioritizing different parts and apportioning your time well**

Warning: This homework onwards you will not be allowed to change any public class/method or variable names in the starter code.

What to submit

Part 1 - Understanding and Testing First

NOTE: **For this part only (the tester), you are allowed to discuss and share ideas about test cases with your classmates.** This is to make the process of developing tests a little more fun.

The Doubly-Linked List you implement will be very similar to Java's own implementation of the LinkedList (link below). Hence in this part, you will write JUnit tests against the Java's LinkedList. Once you have sufficient tests, you will then implement your own Doubly-Linked List in Part 2 and run this tester against your Doubly-Linked List.

1. **First: Understand what LinkedList will do**

In order to write a good tester, you need a deep understanding of how the classes and methods you are testing are supposed to work. So before you start writing your tester, read the Java Documentation of LinkedList to know what these classes and methods are supposed to do.

2. **LinkedListTester**

- Download the supplied file LinkedListTester.java. This is a starter file and it defines a small number of tests against the Java Collection's Framework LinkedList class. This is to allow you to have something to test against before you have your DoublyLinkedList class implemented. Observe how each test is written and what it checks for.

- Compile and run LinkedListTester.java as-is.

- Create a set of meaningful tests on **all** the public methods described in Part 2, which you are responsible for implementing later. Make sure your tests pass against Java's LinkedList class. Provide descriptive javadoc headers for all tests to indicate what each test does.

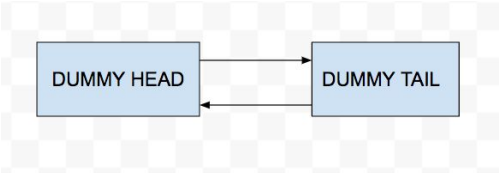
- **After you have completed at least some of part 2,** and are ready to test your DoublyLinkedList class, then rename your LinkedListTester.java to be DoublyLinkedListTester.java. (Don't forget to redefine the class from LinkedListTester to DoublyLinkedListTester), and modify the tests to create DoublyLinkedListTester objects.

- Note that your final tester submitted must contain a test for **every public method.**
- In the end, you will also be graded on how well your tester find errors across a buggy implementation of the linked lists. Please note that we will test
    - your Doubly Linked List against our own Master Tests and
    - your tester file against our implementation of a Doubly Linked List.

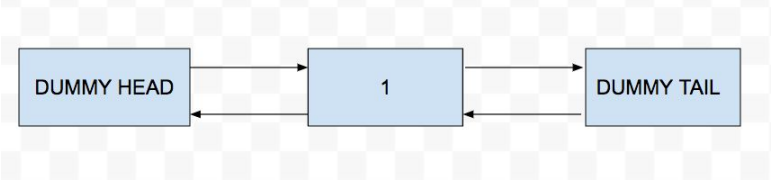
- **Hence to maintain uniformity, you must not change any of the method names/ method signature provided in the starter code.**

Part 2 – Doubly Linked Lists

Empty List (size = 0):



List with size = 1:



- Download the starter file DoublyLinkedList.java. This DoublyLinkedList class will only be a subset of the Java Collection's Framework LinkedList and you only need to implement the methods listed in the table below. **However, the behavior of each method must exactly match that given in the LinkedList javadoc, unless specified otherwise.**

- To make your life easier make sure that your class extends AbstractList, If you are not sure where *@Override* tag came from, then you need to read the online documentation on AbstractList. What happens if you do not override the add() method?

- There are various ways to implement a LinkedList - with only head reference, with both head and tail nodes, with or without dummy nodes. **You must use "dummy" head and tail nodes in this homework.** Refer to the two diagrams above to get a better idea of how the list would look like with dummy head and tail nodes. Think about how having dummy nodes simplifies add/remove operations in the list.

- The Java's LinkedList allows "null" objects to be inserted to the list. However, your DoublyLinkedList will differ from this behavior. You should not allow "null" objects to be inserted into the list and instead throw a NullPointerException if the user attempts to do so.

- Your program will be graded using an automatic script. It will also be manually inspected (to make sure you actually implemented a linked list and not an array-based list). Try to anticipate the kinds of tests we will run on your code -- how might we "stress" your code to its limit?

- **Constructors:** You should only need to have a single public 0-argument constructor that creates an empty list and initializes all the necessary variables.

- **Javadoc style method comments:** You must provide javadoc style comments for every public method. Some methods have already been provided with these comments for your reference.

The following are the methods of the **inner class Node.**

Method name	Description
Node (E element)	Initializes the node with element
Node(E element, Node prevNode, Node nextNode)	Initializes the node with element, prevNode as its predecessor node and nextNode as its successor node.
public void remove()	Removes this node from the list. Links this node's predecessor and successor nodes.
public void setPrev(Node p)	Set p as the predecessor node.
public void setNext(Node n)	Set n as the next node
public void setElement(E e)	Set e as the node's data
public Node getPrev()	Return the node's predecessor
public Node getNext()	Return the node's successor
public E getElement()	Return the node's data

The following are the methods of the **DoublyLinkedList** class.

Tip 1: Make sure to implement all the helper methods (like getNth(int index) first!
Tip 2: Make sure you refer to the table below, comments in the starter code as well as the javadoc description of the method to understand the correct behavior of the method. The behavior of your method must exactly match its specification in order to pass the Master Tests.

Method Name	**Description and link to Java Documentation**	**Exceptions Thrown**
public MyListIterator()	Constructor that is used to initialize the iterator	None
private Node getNth(int index)	a helper method that returns the node at a specified index. (node, and not the content)	None
public boolean add (T data)	Add an element to the end of the list. Note: the boolean add method will presumably always return true; it is a boolean function due to the method definition in AbstractList. add(data)	throw a NullPointerException if a user tries to add a null data
public void add (T data, int index)	Add an element to the list at the given index add(data, index)	throw a NullPointerException if a user tries to add a null pointer throw IndexOutOfBoundsException if index is out of bounds (same rules as MyArrayList)

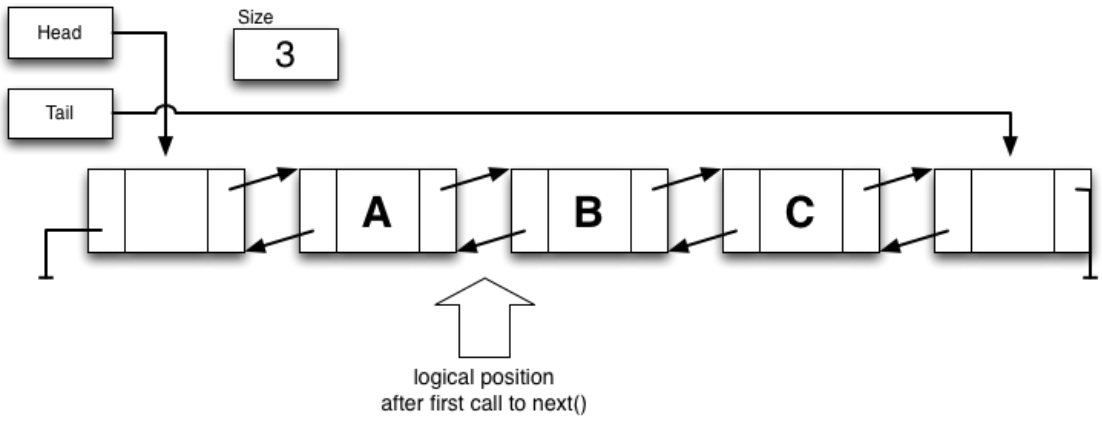
public T get (int i)	get data at position i get(index)	throw IndexOutOfBoundsException as needed
public T set (int i, T data)	set the value at index i to data set(index, data)	throw NullPointerException if data is null throw IndexOutOfBoundsException as needed
public T remove (int i)	remove the element from position i in this list remove(index)	throw IndexOutOfBoundsException as needed
void clear()	remove all elements from the list clear()	None
boolean isEmpty()	determine if the list is empty returns True if empty, false otherwise.	None
int size()	return the number of elements being stored size()	None
public boolean contains(Object o)	Returns true if ‘o’ is found in the list, false otherwise. contains()	Throw NullPointerException if ‘o’ is null

public boolean removeLastOccurrence(Object o)	Removes the last occurrence of ‘o’ in this list (when traversing the list from head to tail). If the list does not contain the element, it is unchanged. removeLastOccurence()	Throw NullPointerException if ‘o’ is null
public int indexOf(Object o)	Returns the index of the first occurrence of ’o’ in this list, or -1 if this list does not contain the element. indexOf( )	Throw NullPointerException if ‘o’ is null

Part 3 – Iterators for DoublyLinkedList

Once your DoublyLinkedList class is working, you will need to create a ListIterator which is returned by the **listIterator()** method. The best approach is to create an inner class (contained inside DoublyLinkedList class).

The ListIterator is able to traverse the list by moving one space at a time in either direction. It might be helpful to consider that the iterator has size+1 positions in the list: just after the sentinel head node (i.e. just before the first node containing data), between the 0 and 1 index, ..., just before the sentinel tail node (i.e., just after the last node containing data). See the diagram below for a better understanding.



Implementation Notes

- Your methods should properly throw IllegalStateException, NoSuchElementException, and NullPointer exceptions. This means you **DO NOT HAVE TO IMPLEMENT throwing ALL exceptions as the javadoc would indicate.** Please refer to the table below.

- You may assume that the methods of the DoublyLinkedList and Iterator will be tested separately and so you need not worry about the effects of one class's methods on the other.

- The starter code does not contain any javadoc style comments for the Iterator methods, unlike some of the methods of DoublyLinkedList. **You must add the javadoc comments yourself, referring to the Javadoc Link and the table below.**

Tip: As in the case of DoublyLinkedList, refer to the table below and the Javadoc description of every method below to understand the correct behavior of the method.

Method name	Method description and Link to Javadoc	Exceptions to throw
public void add(E e)	Adds an element to the list. Refer to the javadoc link to understand where the element is added add(data)	Throw NullPointerException if the data received is null

public boolean hasNext()	Checks if there is another element to be retrieved by calling next hasNext()	None
public boolean hasPrevious()	Checks if there is another element to be retrieved by calling previous hasPrevious()	None
public E next()	Advances through the list by one index, and retrieves the next element next()	Throw NoSuchElementException if there are no more elements remaining in the list for the iterator to retrieve when moving forward
public int nextIndex()	Retrieves the index of the next element (that would be retrieved by next() call) nextIndex()	None
public E previous()	Retreats through the list by one index, and retrieves the previous element previous()	Throws NoSuchElementException if there are no more elements remaining in the list for the iterator to retrieve when moving backwards
public int previousIndex()	Retrieves the index of the previous element (that would be retrieved by previous() call) previousIndex()	None
public void remove()	Removes from the list the last element that was returned by next() or previous() remove()	Throws IllegalStateException if neither next() nor previous() were called, or if add() or remove() were called since the last next()/previous() call.
public void set(E e)	Replaces the last element	Throw NullPointerException if the

	returned by next() or previous() with a given element. set(data)	data given is null Throws IllegalStateException if neither next() nor previous() were called, or if add() or remove() were called since the last next()/previous() call.

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, you must have 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.

- 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 DoublyLinkedList.java

- Once you are **sure** your iterator is working, you should override the following methods in DoublyLinkedList. 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() (as we do in the starter code) 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.**

Apportioning your Time

This is a much more comprehensive 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.

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 DoublyLinkedList. Make sure your test suite runs properly against LinkedList instances. (1-2 hours)

2. Copy/rename your testing class to DoublyLinkedListTester, compile it, run it against the supplied outline of DoublyLinkedList. You should have many test failures.

3. Develop the linked list methods (Node inner class, methods). Develop until all the tests you developed previously pass.

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. (2 - 4 hours)

5. Develop the ListIterator implementation using your tests to help you (2 - 3 hours).

6. Uncomment the lines near the end of the starter DoublyLinkedList.java file, to make your ListIterator active.

7. Note that your final tester must contain a test for **every public method in DoublyLinkedList and Iterator**.

Submission

1. Milestone Submission

This homework has a milestone submission that will get you to work right away and give you extra boost points. This submissions is not mandatory but strongly recommended. You should

submit a "decent quality" code: empty files, a few lines of codes, meaningless code, code that doesn't compile etc will not count. You are allowed to modify your code after the milestone submission but the change should not be drastic.

The first milestone for this homework is on 21st Jan (Saturday). This is an extra credit submission for 10 points, and in order to get full credit, you must complete at least 10 methods of the DoublyLinkedList class. These methods should, at the very least, have their basic functionality completed. However, you need not complete the methods of the Iterator class for the milestone submission. You may change/update the methods of the DoublyLinkedList class after the milestone submission.

Files to submit for Milestone Submission (Saturday, Jan 21):

1. DoublyLinkedList.java
2. DoublyLinkedListTester.java

2. Final Submission

The final submission for this homework is on Wednesday, 25th Jan. You have been provided with a file BasicTester.java as part of the starter files. This file contains a few basic checks that will be run against your code during the final submission. Your code **must** pass these baseline tests to gain **any** points in this homework.

Note: These tests only check the **existence** of all methods in your code and not their proper **functioning**. This is to ensure that your code compiles correctly against our Master Tester.

Files to submit for the Final Submission:

1. DoublyLinkedListTester.java
2. DoublyLinkedList.java (add extra credit methods in this file as well)
3. ExtraCreditTester.java. (optional)

Submit via vocareum and make sure you pass the submission script. It checks the names of your files, compiles it and runs simple JUnit tests.

Grading

How your assignment will be graded:

Coding Style

- Your submission must follow the Coding Style Guidelines on the course website

Correctness

- Does your code compile?
- Does it pass all of your own unit tests?
- Does it pass all of our unit tests?
- Does your code have any errors (e.g. generates exceptions when it isn't supposed to)

Unit test coverage

- Have you tested **every method** of your code?
- Does it detect errors in a reasonably buggy implementation of DoublyLinkedList?
- Does your unit testing approach for listIterator() appear to be sufficient?

Part 4 – Extra credit problems

You have a few problems to choose from

1 (5 points). In your DoublyLinkedList.java class add a method that takes a linked list and concatenates the reversed version of itself. Original list is unchanged. For example: Input list is (3, 5, 7) the output list is (3, 5, 7, 7, 5, 3). Provide a separate file with a JUnit test(s) as well.

Method signature*: public void reverseAndConcat (DoublyLinkedList<E> newList);*
JUnit Test File: ExtraCreditTester.java

2. (5 points): Given two DoublyLinkedLists with integer data, write a method that populates a third list with the sum of the numbers represented by the two lists. Also provide a few JUnit tests in a separate file ExtraCreditTester.java

Example 1:
Input 1: 1 -> 2 -> 3 //represents 123
Input 2: 4 -> 5 -> 6 //represents 456
Result: 5 -> 7 -> 9 //represents 123+456 = 579

Example 2:
Input 1: 1 -> 1 -> 1 -> 1 //represents 1111
Input 2: 9 -> 9 // represents 99

Result: 1 -> 2 -> 1 -> 0 //represents 1111+99 = 1210

Method signature: *public void sumOfTwoLists(DoublyLinkedList<Integer> list2, DoublyLinkedList<Integer> result)*
JUnit Test File: ExtraCreditTester.java

NOTE: You can safely assume that only Integer Lists will be passed as arguments. Also you might have to cast the contents of the input lists to int to perform arithmetic operations on them.

3. (5 points) Given a DoublyLinkedList and a number k, rotate the list counter-clockwise (to the left) k times. Also provide a few JUnit tests in a separate file.

HINT: You shouldn't need to move all the contents around.

Example
Input List: 1 -> 2 -> 3 -> 4 -> 5 -> 6
k : 2
Result: 3 -> 4 -> 5 -> 6 -> 1 -> 2

Method signature: *public void rotateList (int k)*
JUnit Test File: ExtraCreditTester.java

Appendix

1. Exception Handling:

```
//How to throw an exception in methods
int somefunc() throws ThisIsAnException {
 if (SOME_CONDITION)
     throw new ThisIsAnException("MESSAGE");
      //code...
}


//How to catch an exception in tests
try {
      int res = somefunc()
```

```
      //No exception thrown. Normal behavior or not?
     //code..
} catch(ThisIsAnException e) {
      //exception caught. Normal behavior or not?
}
```

Exception_Handling