Lab 7 – Implementing a linked indexed list

In this lab you will be constructing a linked indexed list. You will then use your constructed list to build and test a new linked queue implementation.

Objectives:

- to use existing software components to support building new software
- to gain insight into using linked structures to build a list data structure
- to gain experience applying structural recursion (i.e., recursive class definitions)
- to use your own list ADT to build a new queue ADT

*Problem 1 (of 2): Building & testing your own Linked, Indexed List ADT, using *structural* recursion*

Part 1.1: Creating a shell for class <FOO>LinkedIndexedList

Do not implement anything for now; just read and understand this whole part first.

Move file ***FOOIndexedList.java*** into your structures folder inside ***JavaPackages*** but keep the two test classes (***IndexedListTest.java*** and ***LinkedQueueTest.java***) plus class ***TestClassStats.java*** in your lab folder. As before, rename ***FOOIndexedList.java*** with the (capitalized) initials you are using for your structure classes, then open the three provided files. Change all instances of ***<FOO>*** to the (capitalized) initials you are using for your structure classes; change the package name from ***<foo>structures*** to the name of your structures package.

Now, create an appropriately named (<***YOUR INITIALS***>***LinkedIndexedList***) class in your structures folder inside ***JavaPackages***. This class should eventually look very similar to your linked stack and queue classes but will implement your <***FOO>IndexedList*** interface instead; as a result, the next logical step would be to copy all methods from this interface to your newly created class.

Like your linked stack and queue classes from before, your linked indexed list should include an instance variable of an inner class (we will call it ***ListNode*** this time), called ***head*** to store the first node of the linked indexed list. In addition, you will need to maintain the usual ***size*** integer instance variable. As a result, your class should contain the following two **`private`** instance variables (and only these two): ***head*** and ***size***. *Do not add* methods or instance variables to this class beyond what is asked. Also, don't change names of fields and methods; otherwise, you're likely to encounter failed JUnit tests.

Add a default constructor to your class to initialize ***size*** to zero and ***head*** to a new empty ***ListNode***. The very last node in your list should always be an empty, terminal node, which ***head*** refers to whenever the list is empty. Do not forget to complete the documentation for your class as you build it; use **`{@inheritDoc}`** when possible.

Next, copy the ***StackNode*** inner class from your ***<FOO>LinkedStack*** class into your linked indexed list class, and modify it to be:

```
private class ListNode extends ZHOneWayListNode<ElementType, ListNode>
    implements Serializable
```

with the same two constructors, renamed accordingly. Remember that class ***ListNode*** inherits all **`public`** and **`protected`** methods from the ***ZHOneWayListNode*** class. Please remind yourself of how the class ***ZHOneWayListNode*** works by referring to *https://faculty.csbsju.edu/irahal/SW/ZHStructures/*.

For now, methods that return integers, boolean or objects should simply return -1, false and null, respectively. This should allow your linked list ADT class to compile without errors, even though you have not actually implemented any of the methods yet.

Before we go into implementation details in the coming parts, take time to understand the four categories of outer class methods, based on how they interact with the internal chain of nodes. Do not implement anything for now; just read and understand. Here are the categories:

(1) Methods that operate directly on the ***size*** instance variable (for efficiency) and do not reference the nodes at all; these methods include **`isEmpty`** and **`size`**.

(2) Methods that simply call inherited methods of the ***ListNode*** inner-class on the ***head*** instance variable. For example, the ***ListNode*** class inherits **`contains`** and **`iterator`** methods from the ***ZHOneWayListNode*** class, so the corresponding outer-class methods can just call the inner-class methods on the ***head*** instance variable.

(3) Methods that access the list nodes, including **`get`**, **`set`**, **`addElementAt`**, **`removeElementAt`**, **`indexOf`**, and **`lastIndexOf`**, lend themselves to a two-pronged structural recursive approach: public user-friendly outer-class methods call similarly named private *recursive* methods in inner class ***ListNode*** on the ***head*** instance variable. The outer class methods may only do very little work, before the recursive call, such as checking for out-of-bounds indices. You must use recursion rather than iteration to implement this category of methods.

(4) Finally, methods that access the list nodes but may be easier to implement iteratively (i.e., without recursion); in this case, you should implement the methods completely in the outer class and use loops rather than recursion. Methods **`subList`** and **`containsDuplicates`** fall in this category. In general, if you can implement the method both iteratively and recursively, you should choose the more efficient implementation; if the implementations are about equally efficient, choose the one that has simpler code (usually the recursive version).

Do not add instance variables or other methods to either the outer class or the inner class unless the write-up clearly calls for them.

Part 1.2: Studying the animation

We have created a visualization to help you understand/remember the logic needed to complete your ADT. Please visit Indexed Linked List Implementation under ***Lab07_LinkedIndexedList*** on *https://faculty.csbsju.edu/irahal/SW/DataStructuresVisualizations/Algorithms.html*.

The visualization is super cool in that it contains step-by-step process descriptions (IN BLUE) to help you fully grasp the logic, at least for the major functionalities; so, make sure you play it at slow speeds and use the *Step Back*/*Forward* buttons (after you pause) frequently. Take plenty of time to study the visualization including the following important points which focus on creating, adding to, and removing from a list.

- initial set up when list is still empty; notice how ***head*** points to an empty node (i.e., it is NOT ***null***); as the list grows, the very last node should always be the empty node which ***head*** initially points to
- method **`addElementAt`** when adding to an empty list and at the end, middle, and start of a non-empty list; what parameter index values are not valid? Where is it best to check for valid index values?
- method **`removeElementAt`** when removing from an empty list and from the end, middle, start of a non-empty list; what parameter index values are not valid? Where is it best to check for valid index values?

Make sure to keep referring to the visualization. Note that, in addition to the points above, the visualization illustrates the processes for most other methods in your ADT as well.

Part 1.3: Completing and testing the first batch of methods in class <FOO>LinkedIndexedList

You can implement several of the methods in the outer class by reusing corresponding methods in your ***<FOO>LinkedStack*** class. Keep a copy of the code for that class open so you can refer to it and even copy code from it. However, be careful not to make accidental changes in that file while you work.

Consider test class ***IndexedListTest.java***; its ***init*** method calls the constructor and method **`addElementAt`** but to implement and test method **`addElementAt`**, we first need to implement methods **`isEmpty`**, **`get`** and **`size`**. As described earlier, implementing methods in the first category (**`isEmpty`** and **`size`**) is straightforward since they operate directly on the ***size*** instance variable so start with these.

Next, move to **`get`** and **`addElementAt`**. These belong to the third category, so we must implement them **recursively** which makes them more complex. These public outer-class methods simply use the ***head*** instance variable to make corresponding calls to private inner-class recursive versions with the same names; add these methods to the inner-class ***ListNode***. For example, the **`get`** method in the outer class should return the result of

calling **`this.head.get(…)`**, which requires us to define a recursive **`get`** method in the ***ListNode*** class, since one doesn't currently exist.

Once you complete the methods so far, you should be able to test your implementations. Make sure your tests are successful up to this point before you proceed.

Part 1.4: Completing and testing the second batch of methods in class <FOO>LinkedIndexedList

To implement the remaining methods in your list ADT while simultaneously testing them, please follow the same approach from the previous part. Next, complete methods in the second category (i.e., **`contains`** and **`iterator`**) and test them. Recall that these simply call inherited methods of the ***ListNode*** inner-class on the ***head*** instance variable (parent class ***ZHOneWayListNode*** already provides us with a **`contains`** and **`iterator`** methods).

Once corresponding tests for methods **`contains`** and **`iterator`** are successful, turn your attention to the remaining methods in the third category starting with **`set`**, **`indexOf`** and **`lastIndexOf`**. Recall that you must implement ALL methods in this category using structural recursion rather than iteration (or even functional recursion). Also note that index-based methods (i.e., based on location of elements in the list) will require a parameter representing the index we are currently at in the list. Most outer-class methods in this category already include an ***index*** parameter except for methods **`indexOf`** and **`lastIndexOf`**. These two methods are a little tricky, so we will give you a little more guidance with them in this part.

The inner-class auxiliary **`indexOf`** method needs to keep track of the node number (starting at zero) until it finds the desired item or gets to the end of the list. Here is the heading for this method:

The outer-class method returns the value of the call **`this.head.indexOf(0, element)`**. As the auxiliary method makes recursive calls, it adds one to the value of the ***index*** parameter each time, ensuring that ***index*** is always the position of the current node in the list. Recursion terminates either when parameter ***element*** is equal to the current node's ***element*** or when it reaches the terminal node.

```
/*
 * Searches for element starting at a given index in the list; returns index of first match or -1 if  no match is found.
 *
 * @param  index to identify where to search, which is the current location in the list
 * @param element what we are searching for
 * @return index of first occurrence of the element searched for or -1 if the element is not in list
 */
private int indexOf(int index, ElementType element)
```

You will need to complete method **`lastIndexOf`** before you are able to run all tests for method **`indexOf`**. **`lastIndexOf`** returns the position of the last occurrence of an element, so the auxiliary method must always search to the end of the list since the final occurrence may be yet to come. In addition, it needs to keep track of the last index (if any) where it found the element. We use the additional parameter **`indexOfLastMatch`** to pass this value. Here is the heading for this method:

```
/**
 * Searches for element starting at a given index in the list; returns index of last match or -1 if no match is found.
 *
 * @param  index to identify where to search, which is the current location in the list
 * @param  indexOfLastMatch the last position where we found a match or -1 if we have not found a match yet
 * @param  element what we are searching for
 * @return  index of last occurrence of the element searched for or -1 if the element is not in list
 */
private int lastIndexOf(int index, int indexOfLastMatch, ElementType
```

Here, the outer-class method returns the value of the call **`this.head.lastIndexOf(0,−1,element)`**, where the second parameter −1 indicates that when we start searching, we have not found anything yet. As

with the previous method, the auxiliary **`lastIndexOf`** method increments the value of the ***index*** parameter with each recursive call; it also updates the value of **`indexOfLastMatch`** whenever it finds a match. Recursion terminates at the end of the list, where it returns **`indexOfLastMatch`**.

Once tests for methods **`set`**, **`indexOf`**, and **`lastIndexOf`** are successful, move on to method **`removeElementAt`**; you are mostly on your own to figure this one out (PS: the visualization should be super useful).

Test all methods up to this point are successful before you proceed.

Part 1.5: Completing and testing category 4 methods in class <FOO>LinkedIndexedList

The fourth and final category of methods (**`subList`** and **`containsDuplicates`**) should be implemented ****non-recursively**** in the outer class. They are independent of each other so you can implement them in any order by following their *javadoc* carefully. Please note that with the help of method **`get(int index)`** in the outer class, you are able to use *for* loops to iterate over the elements in the list by location. In addition, since our class implements **`Iterable`**, you may use an enhanced *for* loop instead if location is irrelevant to the task at hand. Make sure to test as you implement.

When you have finished testing and believe your implementation is correct, show it to the lab instructor or the TA.

Problem 2 (of 2): Building and testing a new Queue ADT based on your Linked, Indexed List

Say what now? Yes, you read that right! You can use an ADT you developed to develop another ADT! Isn't life just plain cool!

The linked queue implementation you built in an earlier lab uses a linked structure called ***QueueNode*** to store the nodes of the queue. Recall that we had to manually maintain the linked structure ourselves using ***size*** and the ***front*** and ***rear QueueNodes***. In this lab problem you are to design and implement a new linked queue class that replaces all instance variable with a single instance of your <***FOO***>***LinkedIndexedList*** class. This will prove to be a much simpler task since your linked list instance variable already takes care of maintaining its own underlying linked structure (so we do not have to!)

The provided ***IndexedListQueueTest*** class allows you to test your implementation as you build it. If you have not done so already, open the file and change the **`import <foo>structures.*;`** statement to your structures package, and again change all instances of "<***FOO***>" to the (capitalized) initials you are using for your structures.

Note that this is the same test class you used to test your previous queue implementation. Use this class as a guide to figure out the proper order to implement your methods while simultaneously testing them.

Create an appropriately named (<***YOUR INITIALS***>***IndexedListQueue***) class in your ***JavaPackages/<your initials>structures*** package. This class should implement interface ***ZHQueue<ElementType>*** available in the ***zhstructures*** package. It should contain the following (*and only the following*) instance variable (with your capitalized initials instead of <***FOO***>):

```
<FOO>IndexedList<ElementType> innerList
```

Since instance variable **`innerList`** is of type ***<FOO>IndexedList***, it provides you with many functionalities including ones you can use to build a queue. Your new linked queue class constructor should initialize **`innerList`** to be an empty <***FOO>LinkedIndexedList***. All methods in this class should make appropriate calls on this instance variable so I expect you to be able to complete the remainder of this queue class on your own. First, you need to decide on which end to ***enqueue*** to and ***dequeue*** from, then decide how to use the indexed list methods to implement the queue methods.

It should be possible to implement *every* queue method by a single call to an indexed list method.

When you have finished testing and believe your implementation is correct, show it to the lab instructor or the TA.