**Lab 7 – Implementing a linked indexed list**

In this lab you will be constructing a linked indexed list. You’ll 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 use your own list ADT in order to build a new queue ADT

***Problem 1: Building and testing your own Linked, Indexed List ADT***

### *Part 1.1: Creating class <FOO>LinkedIndexedList*

### Get the lab files; move file *FOOIndexedList.java* into your *JavaPackages*/<*your initials*>*structures* package but keep the two test classes (*IndexedListTest.java* and *LinkedQueueTest.java*) in your lab folder.

### As before, replace *FOO* in *FOOIndexedList.java* with the (capitalized) initials you are using for your structures classes, then open the file. Change all instances of *<FOO>* to the (capitalized) initials you are using for your structures 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 *JavaPackages/<your initials>structures* package. 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.

Similar to your linked stack and queue classes from before, your linked indexed list should include an instance of an inner class (we’ll call it *ListNode* this time) called *firstNode* to store the first node of the linked indexed list. In addition, you’ll need to maintain the usual *size* integer instance variable. So, in total, your class should contain the following two *private* instances variables *and only these two*: *firstNode* and *size*.

Add a default constructor to your class to initialize *size* to zero and *firstNode* to a new empty *ListNode*. The very last node in your list should always be an empty, terminal node, which *firstNode* refers to whenever the list is empty.

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

**protected** **class** ListNode **extends** ZHOneWayListNode<ElementType, ListNode>

with the same two constructors, renamed accordingly. Please take some time to remind yourself of how the class *ZHOneWayListNode* works by referring to its documentation on <http://www.users.csbsju.edu/~csweb/ZHStructures/index.html>. Remember that the *ListNode* class inherits all public and protected methods from the *ZHOneWayListNode* class.

Make sure your class compiles without errors, even though you haven’t actually implemented any of the methods yet.

### *Part 1.2: Completing methods in class <FOO>LinkedIndexedList*

You can implement several of the methods in the outer class by using 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 if you use your original.

There are four different categories of outer class methods, based on how they interact with the internal chain of nodes. Here are the categories:

* Some methods operate directly on the *size* instance variable (for efficiency) and don’t reference the nodes at all; these methods include isEmpty and size. You should complete these methods first.
* Some methods can simply call inherited methods of the *ListNode* inner-class on the *firstNode* 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 *firstNode* instance variable.
* Several methods that access the nodes, including get, set, addElementAt, removeElementAt, indexOf, and lastIndexOf, lend themselves to a two-pronged recursive approach: outer-class methods call similarly named *recursive* auxiliary methods in the inner class *ListNode* on the *firstNode* instance variable. The outer class methods may to do a little work before the recursive call, such as checking for out-of-bounds indices.
* Finally, some methods may be easier to implement iteratively (i.e., without recursion); in this case, you should implement the method completely in the outer class. Methods subList and containsDuplicates are ones that you are required to do iteratively. 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.

Start by implementing outer-class methods in the first two categories, and then turn to methods in the third category. These methods will also use the *firstNode* instance variable to make corresponding calls to inner-class methods 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 firstNode.get, which requires us to define a get method in the *ListNode* class, since the get method doesn’t already exist, unlike the contains method which is inherited from the parent class *ZHOneWayListNode*.

You are mostly on your own here; however, you must use recursion rather than iteration via loops. In addition, please keep in mind that any index-based method (i.e., based on location of elements in the list) will probably need an additional parameter representing the index we’re currently at in the list. Most methods in this category already include an index parameter except for methods indexOf and lastIndexOf. These last two methods are a little tricky, so we’ll give you a little more guidance with them.

The auxiliary indexOf method in the inner-class 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:

/\*\*

\* Searches for parameter element starting at a given index in the

\* list and returns index of first match or -1 if no match is found.

\*

\* @param index to identify where to start the search,

\* which is the current location in the list

\* @param element what we’re searching for

\* @return index of first occurrence of the element searched for

\* or -1 if the element is not in list

\*/

**protected int** indexOf(**int** index**,** ElementTypeelement)

The outer-class method returns the value of the call firstNode.indexOf(0, element). As the auxiliary method makes recursive calls, it adds one to the value of the *index* parameter each time, insuring that *index* is always the position of the current node in the list. The recursion ends either when *element* is equal to the current node’s *element* or when it reaches the final terminal node.

The lastIndexOf method 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 parameter element starting at a given index in the

\* list and returns index of last match or -1 if no match is found.

\*

\* @param index to identify where to start the search,

\* which is the current location in the list

\* @param indexOfLastMatch the last position where we found a match

\* or -1 if we haven’t found a match yet

\* @param element what we’re searching for

\* @return index of last occurrence of the element searched for

\* or -1 if the element is not in list

\*/

**protected int** lastIndexOf**(int** index**,**

**int** indexOfLastMatch**,**

ElementTypeelement**)**

Here, the outer-class method returns the value of the call firstNode.lastIndexOf(0, −1, element), where the second parameter −1 indicates that when we start searching, we haven’t already found anything. 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. The recursion ends at the end of the list, where it returns the appropriate index value.

The fourth and final category of methods (methods subList and containsDuplicates) should be implemented non-recursively in the outer class. You need to follow their javadoc carefully in order to do that. 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.

### *Part 1.3: Testing the linked indexed list class*

Class *IndexedListTest* allows you to test your class once it compiles. If you haven’t 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. When you have finished testing and believe your implementation is correct, show it to the lab instructor or the TA.

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

### That's right—you actually can use an ADT you developed to develop another ADT! Isn’t life just cool!

### *Part 2.1: Creating a new linked Queue class*

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 the *front* and *rear* *QueueNode*s. In this lab problem you are to design and implement a new linked queue class that uses an instance variable 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 don’t have to!)

Start by creating 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.

### *Part 2.2: Testing the new linked queue class*

The *IndexedListQueueTest* class allows you to test your class once it compiles. Note that this is the same test class you used to test your previous queue implementation.

If you haven’t 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. When you have finished testing and believe your implementation is correct, show it to the lab instructor or the TA.