Lab 4 – Implementing, testing, and using linked stack and queue classes

Like before, you are to continue doing "***Pair Programming***" with your assigned partner. Remember to switch roles halfway through lab (teams of 3 should switch every hour).

Preliminaries

In this lab you will:

- complete and test a linked-stack class that implements the *ZHStack* interface,
- create and test a linked-queue class that implements the *ZHQueue* interface, and
- modify the infix-to-postfix programs from the previous lab to use your stack and queue classes instead of the ones in the Java Collections Framework (JCF)

If you have not already done so, please create a new empty folder in your *JavaPackages* folder and name it <***your initials***>*structures* (all small case without the spaces and the < & > signs; this would be ***imr****structures* for me). We will use this new folder as a package to store all data structures created in this lab. Super exciting!!!!

Problem 1: Completing and testing your own Linked Stack ADT

Part 1.1: Completing class FOOLinkedStack to implement interface ZHStack

Move the *FOOLinkedStack.java* file into your *JavaPackages/****your initials*** *structures* folder but *keep the test classes* in your lab folder. Rename *FOOLinkedStack.java* with the (capitalized) initials you are using for your structures (e.g., ***IMR****LinkedStack* in my case), then open the file. Change all instances of "***<FOO>***" to the (capitalized) initials you are using for your structure classes to match the name of the class and change the package declaration from ***<foo>structures*** to the name of your structures package (e.g. ***imr****structures* in my case).

The provided class *FOOLinkedStack* contains all the instance variables and methods needed to implement interface *ZHStack* which extends interface *ZHCollection* (which, in turn, extends interface *Iterable*). Please study the complete documentation for interfaces *ZHStack* and *ZHCollection* by visiting the following URL and selecting the desired class/interface from the left menu: https://faculty.csbsju.edu/irahal/ZHStructures/index.html.

You are asked to complete class *FOOLinkedStack* by writing code needed for all methods flagged with **"COMPLETE ME"**. For the most part, you should use the given ***peek*** method already completed for you as a template and follow class discussions on linked stacks in order to do this. *Do not add* additional methods or instance variables to this class.

Keep in mind the following important notes (which may differ from what was discussed in class):

- We are including a *size* instance variable to keep track of the number of elements in the stack; your implementations of methods `isEmpty` and `size` should utilize the *size* instance variable to be efficient
- Instead of creating an inner node class from scratch, we are using a different inner class called *StackNode* which extends *abstract* class *ZHOneWayListNode*, to store the elements of the stack

Notice how the constructor of your linked stack class sets the *top* instance variable to be a new empty node; the very last node in your stack should always be an empty node which *top* initially points to. You should use method `isEmpty` of class *ZHOneWayListNode* to test for the empty node.

Class *ZHOneWayListNode* (which is extended by inner class *StackNode*) defines a generic node that can be used to store elements of different linked ADTs such as stacks, queues, lists, etc. by creating inner classes that extend *ZHOneWayListNode* inside those ADTs thus allowing the developer to add additional functionalities useful for the ADT at hand (beyond the ones provided in the generic *ZHOneWayListNode* class). In the case of a linked stack, the inner class *StackNode* need not add any functionalities to class *ZHOneWayListNode* aside from the provided constructors; in other words, the inner class is complete. The inherited methods we are interested in are listed in the inner class as comments.

Please take some time to familiarize yourself with class *ZHOneWayListNode* by referring to its documentation on https://faculty.csbsju.edu/irahal/ZHStructures/index.html (select the desired class/interface from left menu).

Notice that *ZHOneWayListNode* implements the *ZHCollection* interface; thus, the *ZHCollection* methods in your linked stack class (i.e., `contains`, `iterator`, `isEmpty` and `size`) can be implemented directly by calling the same methods on the *top* node instance variable. However, because we're keeping a *size* instance variable in the stack, you should make the implementation of the `isEmpty` and `size` methods more efficient by using the *size* variable which keeps track of the number of elements in the stack.

Your linked stack class resides inside your own package, and not in the *zhstructures* package, so you will not be able to access the *protected* instance variables *element* and *next* of the *StackNode* class since they are inherited from the *ZHOneWayListNode* class and only available to classes in the *zhstructures* package. Instead, you will need to use the `getElement`, `getNext`, `setElement(ElementType element)`, and `setNext(StackNode next)` public methods of the *ZHOneWayListNode* class, which the *Stacknode* class inherits.

Complete class *FOOLinkedStack* until it compiles without errors. Make sure you properly increment and decrement the *size* instance variable as you *pop* and *push* from the stack in order to keep track of the correct number of elements in the stack.

Part 1.2: Testing your LinkedStack class

Class *FOOLinkedStackTest* allows you to test your class once it compiles. KEEP THIS CLASS IN YOUR LAB FOLDER. If you haven't done so already, rename it with the (capitalized) initials you are using for your structures, change the `import <foo>structures.*;` statement to your structures package, and 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: Implementing your own Linked Queue ADT

Part 2.1: Creating your class LinkedQueue to implement interface ZHQueue

Create an appropriately named (<*YOUR INITIALS*>*LinkedQueue*) class with the (capitalized) initials you are using for your structures (e.g., *IMRLinkedQueue* in my case) inside your *JavaPackages/*<*your initials*>*structures* package. You will use your complete linked stack class from Problem 1 as a model.

This class should implement the *ZHQueue* interface and therefore include all methods from this interface as well as the *ZHCollection* interface (refer to *ZHQueue*'s documentation by visiting https://faculty.csbsju.edu/irahal/ZHStructures/index.html and selecting the desired class/interface from the left menu).

Recall how the queue differs from the stack. In a stack, elements are added and removed at the same end—at the *top*. In a queue, elements are added at one end and removed from the other. The linked stack, then, only needs a reference to one end of the linked structure—the `protected` *StackNode* instance variable *top*. The queue will need references to both ends of the structure, so it needs two `protected` instance variables of type *QueueNode*; let's call them *front* and *rear*. We'll `enqueue` at the *rear* and `dequeue` and *peek* at the *front*. In addition, we will again use a *size* instance variable to keep track of the number of elements in the queue. Thus, your class should contain the following three instance variables (*and only these*): *front*, *rear* and *size*.

Now think about which queue end you designate as *front* and which as *rear*; this is an important question. You can do it either way, but one way is a lot easier than the other. Recall your class discussions on linked queues. Either way, one of the instance variables should contain a reference to the first node in the queue and one to the last (empty) node. Note that when queue is initially empty, it should contain only a single empty node which both ends—*front* and *rear*—point to. In addition, the last node in the structure should always be empty. You should draw some pictures at this point and consider what you need to do to perform `enqueue` and `dequeue` operations.

Complete and document your linked queue class using your linked stack class as a model. When your class is complete and compiles correctly, go on to Part 2.2.

Part 2.2: Testing your LinkedQueue class

The *FOOLinkedQueueTest* class allows you to test your class once it compiles. *Keep this class in your lab folder*. If you haven't done so already, rename it with the (capitalized) initials you are using for your structures, change the `import <foo>structures.*;` statement to your structures package, and 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 3: Using your Stack and Queue ADT implementations in the infix-to-postfix application

Create copies of your complete *Tokenizer.java*, *PostfixEvaluator.java*, and *InfixToPostfix.java* programs (from lab 3) inside package *expressions* and rename them as *TokenizerV2.java*, *PostfixEvaluatorV2.java*, and *InfixToPostfixV2.java*. Make all the necessary changes so that these three programs use the linked stack and queue implementations developed in this lab (available in YOUR *JavaPackages/<your initials>structures* package) instead of JCF's (Java Collections Framework) versions. This will likely result in additional *import* statements and changes to constructor and some method calls.

Use the provided *TokenizerV2Test.java* and *InfixToPostfixV2Driver.java* classes in your lab folder (*keep them in your lab folder*) to test the changes you made above actually work. (You will need to make the same kind of modifications to the names of classes and to import statements in these as well.)