# Lab 5 – Implementing, testing, and using linked stacks and queues

Today you'll start working with a ***new partner***. Take the time to introduce yourselves and share a few interesting facts. Don't be mean.

***Objectives***:

- to complete and test a linked-stack class that implements the ***ZHStack*** interface in the ***zhstructures*** package
- to create and test a linked-queue class that implements the ***ZHQueue*** interface in the ***zhstructures*** package
- to modify the infix-to-postfix programs from the previous lab to use your stack and queue ADTs instead of Java's

## *Problem 1 (of 3): Implementing your own Linked Stack ADT*

Read all problem description *before* starting.

***Preliminaries:***

Move the provided ***FOOLinkedStack.java*** file into your structures folder inside ***JavaPackages*** but *keep the 5 test classes plus class **TestClassStats.java*** inside your lab folder. Rename ***FOOLinkedStack.java*** with the (capitalized) initials you are using for your structures (such as ***IMR****LinkedStack* for me), then open the file. As before, change all instances of ***<FOO>*** or ***FOO*** to the (capitalized) initials you are using for your structure classes; similarly, replace occurrences of ***<foo>*** or ***foo***, with your lower-case initials.

Class ***FOOLinkedStack*** already contains all the instance variables and methods needed to implement interface ***ZHStack,*** which extends interface ***ZHCollection*** (which, in turn, extends Java's ***Iterable***). Please study both interfaces by visiting the complete documentation of the ***zhstructures*** package linked below, and selecting the desired target from the left menu: *https://faculty.csbsju.edu/irahal/SW/ZHStructures/*

In this first problem, you are asked to complete your ***FOOLinkedStack*** by implementing and testing all methods flagged with "`//COMPLETE ME***`…". *Do not add* methods or instance variables to this class. Also, don't change names of fields and methods; otherwise, you're likely to encounter failed JUnit tests.

Class ***FOOLinkedStackTest*** allows you to test your stack ADT as you build it. KEEP THIS CLASS IN YOUR LAB FOLDER. If you have not 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.

The logic needed to complete class ***FOOLinkedStack*** is summarized nicely for you in the *Stack: Linked List Implementation* visualization under ***Lab05_LinkedStacks_Queues*** on *https://faculty.csbsju.edu/irahal/SW/DataStructuresVisualizations/Algorithms.html*. Take plenty of time to study the visualization. As before, the observer should be in charge of the visualization on one machine, while the driver writes code on the other; switch roles often.

Note the following important points

- initial setup when stack is still empty; notice how ***top*** points to an empty node (i.e., it is NOT ***null***); as the stack grows, the very last node should always be the empty node which ***top*** initially points to
- ***push*** method works by creating a new node (for the pushed element) pointing to the current top of the stack; the ***top*** and ***size*** instance variables are then updated. Do you notice any difference in the logic when we ***push*** to an empty vs. non-empty stack? Are there any restrictions on ***push*** (should it ALWAYS work)?
- ***pop*** method works by updating the ***top*** and ***size*** instance variables after saving the popped element to return it. Again, do you notice any difference in logic when ***pop*** results in an empty vs. non-empty stack? Are there any restrictions on ***pop***?
- How about ***peek***?

Once you are comfortable with the logic, you may begin the journey of testing WHILE implementing your linked stack ADT starting with (1) the default (and only) constructor, and (2) ***push*** (since both are used in the ***init*** method in the test class), followed by (3) ***peek*** (used to test ***push*** and ***pop***) and then by running the provided test class to ensure corresponding tests are successful. Next, move on to ***pop***, followed by testing. Finally complete and test each of the remaining methods, one at a time.

Keep the following in mind:

- The ***size*** instance variable keeps track of the number of elements on the stack so make sure you properly increment and decrement ***size*** as you ***push*** and ***pop*** to/from 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 an inner class called ***StackNode*** which extends ***abstract*** class ***ZHOneWayListNode***, from ***zhstructures***, to store the elements of the stack; class ***ZHOneWayListNode*** defines a generic node class 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 allowing the developer to add additional functionalities useful for the ADT at hand (beyond the ones provided in the generic ***ZHOneWayListNode*** class). Familiarize yourself with class ***ZHOneWayListNode*** by visiting the following page (select target from left menu) *https://faculty.csbsju.edu/irahal/SW/ZHStructures/*.

  In the case of a linked stack, the inner class ***StackNode*** SHOULD not add any functionalities to class ***ZHOneWayListNode*** aside from the provided constructors; in other words, the provided inner class is complete. For your convenience, inherited methods of interest are shown in the comments right above the inner class.
- You should use method ***isEmpty*** of class ***ZHOneWayListNode*** to test for the empty node.

To implement the ***ZHCollection*** methods in your linked stack class (i.e., ***contains***, ***iterator***, ***isEmpty*** and ***size***), you can simply call similar methods on the ***top*** node instance variable; for an example, please see the ***iterator*** method already completed for you. However, because we are maintaining 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 on 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***, and ***setNext*** public methods of the ***ZHOneWayListNode*** class, inherited by inner class ***Stacknode***.

When you have finished implementing, documenting, and testing your stack ADT, show it to the lab instructor or the TA.

## *Problem 2 (of 3): Implementing your own Linked Queue ADT*

Now it is time to create a linked queue class using your linked stack from ***Problem 1*** as a model. Create an appropriately named class (***<YOUR INITIALS>LinkedQueue***) in your structures folder inside ***JavaPackages***. This class should implement the ***ZHQueue*** interface in the ***zhstructures*** package and therefore include all methods from this interface as well as interface ***ZHCollection***. Apply the same setup steps to class ***FOOLinkedQueueTest*** as you did for class ***FOOLinkedStackTest*** in ***Problem 1***.

***ZHQueue***'s documentation can be found here *https://faculty.csbsju.edu/irahal/SW/ZHStructures/* and the logic needed to complete your queue implementation can be visualized using the *Queue: Linked List Implementation* visualization under ***Lab05_LinkedStacks_Queues***.

Take plenty of time to study the visualization including the following important points:

- How to do the initial setup when queue is empty?

- How ***enqueue*** works? Any difference in logic when we ***enqueue*** to empty vs. non-empty queue? Are there any restrictions on ***enqueue***?
- How ***dequeue*** works? Any difference in logic when ***dequeue*** results in an empty vs. non-empty queue? Are there any restrictions on ***dequeue***?
- How about ***peek***?

Recall how the queue differs from the stack. In a stack, elements are added and removed at the same end—the ***top***. In a queue, elements are added at one end and removed from the other. So, while the linked stack only needs a reference to one end of the linked structure—the ***private StackNode*** instance variable ***top***, the queue will need references to both ends: two ***private*** instance variables of type ***QueueNode*** commonly referred to as ***front*** and ***rear***. We will ***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. As a result, your class should contain the following three ***private*** instance variables (*and only these, named exactly as shown*): ***front***, ***rear***, and ***size***.

***front*** will contain a reference to the first node in the queue and ***rear*** to the last (always empty) node. Note that when the queue is initially empty, it should contain only a single empty node which both ends—***front*** and ***rear***—point to. As with our stack implementation, the last node in the linked structure (i.e., the ***rear***) should always be empty.

Use your linked stack class as a model, to complete, document and test your linked queue class. Like your linked stack class, your queue implementation class should include only a default constructor along with a ***private*** inner class ***QueueNode*** that extends ***ZHOneWayListNode*** and includes two constructors (and no additional fields or methods just like inner class ***StackNode***).

Remember to use the provided queue ADT test class to test your implementation ***as you build it***; this test class is very similar to the stack ADT test class from ***Problem 1*** so implement and test your queue ADT methods in similar order. When you have finished implementing and testing your queue ADT, show it to the lab instructor or the TA.

## *Problem 3 (of 3): Using your Stack and Queue ADTs in the Infix-to-Postfix Application*

Create copies of your complete ***Tokenizer.java***, ***PostfixEvaluator.java***, and ***InfixToPostfix.java*** programs from the previous lab inside package ***expressions*** and rename them as *Tokenizer**V2**.java*, *PostfixEvaluator**V2**.java*, and *InfixToPostfix**V2**.java*. Make the necessary changes so that these three programs use the linked stack and queue implementations developed in this lab instead of Java's. (Consider additional ***import*** statements and changes to constructor calls as well as method names).

Use the provided ***TokenizerTestV2.java*** , ***InfixToPostfixTestV2.java*** and ***PostfixEvaluatorV2.java*** test classes in your lab folder to test that the changes you made above actually work. (You'll need to figure out if similar modifications are needed here as well).