CSIS 10B Assignment 3 Stacks & Queues

Goal

In this assignment you will create two Stack and two Queue classes each, using different implementation strategies. At the end, you will copy in your solution to the Palindrome problem in assignment2 and use your own Stack and Queue classes to solve the complete problem.

Resources

- Chapter 5 and 6: Stacks and Stack Implementations
- Chapters 10 and 11: Queue and Queue Implementations
- You may also wish to check out more applications of Stacks and Queues in Sedgewick's lecture on the topic.

Java Files

• *StackArray.java* • *StackLink.java* • *StackTester.java*	• *AQueue.java* • *LQueue.java* • *TestAQueue.java*

Part A – Stacks, Introduction

The ADT stack is used in many areas of computer science, from parsing code in compilers to managing the runtime environment during method invokations. It embodies many of the features of the bag class we used last week, especially in its simplicity of implementation. A stack literally represents a pile of items, and provides access only to the top item.

For simplicity, the stacks we build in this lab will hold items of class Object, meaning they will handle generic types but we are responsible for ensuring the proper types are added, and we'll have to cast items back to their original type as we pop them off the stack. Here are the methods we will implement:

`boolean`	**`isEmpty`**`()` Checks for empty stack
`Object`	**`peek`**`()` Returns the top item from stack without removing
`Object`	**`pop`**`()` Pops an item off the top of stack
`void`	**`push`**`(Object item)` Pushes a new item onto top of stack
`int`	**`size`**`()` Gets the number of items in the stack
`String`	**`toString`**`()` Creates and returns a string version of stack

The first task is to create a stack based on an array. Then you'll do the same thing with a linked chain of nodes.

Pre-Lab Visualization

StackArray vs StackLink

The array based stack (**StackArray**) will use an internal array to maintain the items on the stack. As we saw in lecture, the bottom of the stack will be located in cell 0 of the array, and items will be added to successive array cells.

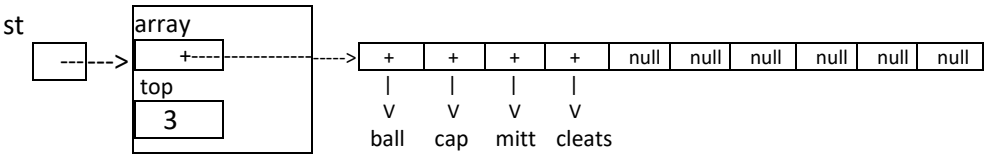
We'll use an integer member to keep track of where the top of the stack is located. We can also use this to tell how large our stack is.

Here is some code that creates one of our simple array based stacks:

```
StackArray st = new StackArray();
st.push("ball");
st.push("cap");
st.push("mitt");
st.push("cleats");
```

Notice we are not using interfaces or type specifiers in angle brackets as we would in a full up version of a Stack. Our stack is storing items in an array of objects.

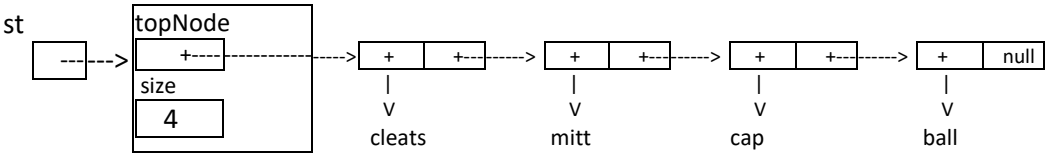
The memory map of the above stack will look something like this



(and in simpler notation would be something like st → [top: 3, array:]→ball,cap,mitt,cleats)

Note that the top member indicates that the top of the stack is in cell 3.

If we had instead create a stack from linked nodes (**StackLink** class), we would draw our stack like this:



(or in simpler notation: st→ [size: 4, topNode]→cleats→mitt→cap→ball)

Notice that the order of the items is reversed from the array stack and we keep track of the top node with the Node reference topNode. Since it's not easy to tell how many nodes we have in our stack, we'll also use a member variable size to keep track of the number of nodes.

Using the above as a guide, draw pictures of the two stacks that are formed by the following statements:

```
StackArray sa = new StackArray();
StackLink  sl = new StackLink();
sa.push("a"); sa.push("b"); sa.push("c"); sa.push("d");
sl.push(sa.pop());
sl.push(sa.peek());
sl.push(sl.peek());
sa.push(sl.pop());
```

Directed Lab Work

StackArray class

Step 1. Add the constructor, define isEmpty, and toString methods to the StackArray class

This constructor method simply initializes member array to a new array of objects with the default array size. It also sets the top member to -1, a signal that the stack is empty

There is already a stub for isEmpty. This method should check whether top is -1 and return true if it is, false if not.

There is also a stub for toString. The toString method should create a String result, then use a for loop to add each item in the stack **starting at the bottom** to the end of result. Then return result.

Checkpoint: activate the lines in StackTester for problem 1, by adding an extra '/' character in front of the block comment for problem 1. Run the program and observe the results. You should have an empty stack.

Step 2. Define the push and size methods

Add code to the push method. To push an item on the stack, simply add it to the cell right after top, and then add one to top. What should happen if there is no space left?

One thing you can do is resize the array. use array = Arrays.copyOf(array, 2*array.length); to double the size of the array if needed. This statement copies the old array into an array of twice the size, returning the address to store in the array variable, replacing the reference to the old array.

The size method should just return top+1

Test these methods by executing problem 2 in StackTester.

Step 3. Define peek and pop methods

Peek should return a reference to the top item in the array

Pop should store a reference to the top item in a temp variable, put null in the array cell located at top, and reduce top by 1. Finally it should return the reference to the top item (the temp variable).

Test these methods by executing problem 3 in StackTester. Once you are finished with step 3, it's time to move over to the StackLink class. Comment out problems 1 through 3 and activate problem 4 in StackTester.

StackLink class

The StackLink class uses a private Node class to store the data in the stack. The private Node field topNode refers to the last item added (pushed) onto the stack, and the private int field size maintains a count of the number of nodes in the stack.

Step 4. Define **constructor**, **isEmpty**, and **toString**

The stackLink constructor will just assign null to topNode and 0 to size.
The isEmpty method will return true if topNode is null

The toString method has a challenge: if we traverse the chain left to right, we will build our String "top first" and it will be flipped with respect to the StackArray class toString. For the same stack shown in the pictures above, the StackArray would print **ball cap mitt cleats** (top on the right), while the StackLink object would print **cleats mitt cap ball** (top on the left). It makes more sense to show the top on the right, so we can modify toString for StackLink to print the list out in reverse. We can do this by changing the way we add items to the result string.

The algorithm for toString is:

1) Create a Node current and set to topNode
2) Create a String result and assign it the empty string "";
3) while current is not null
    - add the data in the node referred to by current (current.data) to the **beginning** of the result string. (**result = current.data + " " + result**; adds the new item to the front of the result string, that's the opposite of how we did it for the linked bag).
    - advance current to the next node
4) When finished, return result.

Step 5. Define push and size methods

Algorithm for push: This is very similar to the add method for the LinkedBag class. Check the lab 6 handout for more help if you need it.

Step 6. Define peek and pop methods

Algorithm for pop: This is very similar to the remove method for the LinkedBag class. Check the lab 6 handout for more help if you need it.

Algorithm for peek: just return the data in the topNode.

Test your methods. Then, when finished**, go back and analyze the Big-Oh growth of all the methods in both of your Stack classes, including the constructors**. Add a comment right after the method header with the Big-Oh notation, such as O(1), O(N), O(N^2) etc. (hint, they are probably all either O(1) or O(N). When you are finished, move on to the Queue portion of the assignment

Part B Queues, Introduction

This portion of the assignment focuses on another constrained linear data structure, the queue. The elements in a queue are ordered from least recently added (the front) to most recently added (the rear). Insertions are performed at the rear of the queue, and deletions are performed at the front. You use the enqueue operation to insert elements and the dequeue operation to remove elements. Although we won't be using interfaces in this assignment, the Queue interface which defines the methods our Queues must implement, is defined below for reference.

```
public interface Queue
{
    // Queue manipulation operations
    public void enqueue ( Object newElement );  // Enqueue element at rear
    public Object dequeue ( );                  // Dequeue element from front
    public void clear ( );                      // Remove all elements from queue

    // Queue status operations
    public Object getFront();              // get item at front without removing
    public boolean isEmpty ( );            // Is Queue empty?
    public boolean isFull ( );             // Is Queue full?
    public String toString( );             // Outputs a String representation of Q
 }
```

Pre-Lab Visualization

1) To make sure you understand how a Queue works, show the contents of the Queue at each step while performing the following operations on Queue testQ. Just list the contents from left to right, where left is the front of the queue:

Operation	Resulting Queue items (front on left)
testQ.enqueue('a');	
testQ.enqueue('b');	
testQ.enqueue('c');	
testQ.dequeue();	
testQ.dequeue();	

2) Show the contents of the Queue at each step while performing the following operations on Queue testQ. Just list the contents from left to right, where left is the front of the queue:

Operation	Resulting Queue items (front on left)
testQ.enqueue('a');	
testQ.enqueue('b');	
testQ.enqueue(testQ.dequeue());	
testQ.enqueue(testQ.getFront());	
testQ.dequeue();	

Directed Lab Work

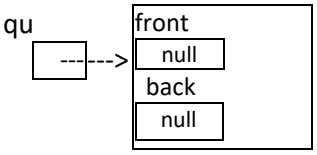
LQueue Class

To begin, we will develop the linked node version of the Queue class in LQueue and test it with the QueueTester program. This class defines a private Node class that will be used internally to store the data in an LQueue object. It also uses two Node reference variables, (front and back) to refer to the first and last Nodes in the queue.

The following diagrams should help you visualize how the different queue methods will work.

Step 1) Define the Constructor, isEmpty, and toString methods

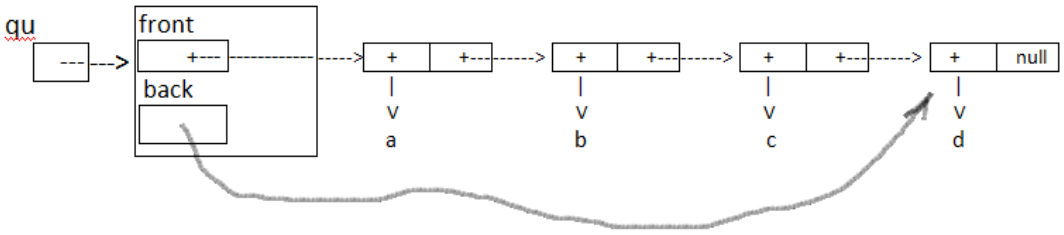
Constructor: set both `front` and `back` Node references to null (this is what an empty queue looks like), so that when we execute the statement **LQueue qu = new LQueue();** we get the following object in memory



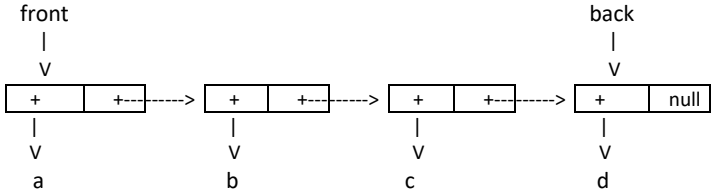
isEmpty: this should return true if both front and back are null

toString: when we eventually add items to our queue, such as with the statements
qu.enqueu("a"); qu.enqueu("b"); qu.enqueue("c"); qu.enqueue("d");

our memory model of the queue will look something like this:



but for convenience, we will simplify our drawing to look like this, showing front and back pointing to their respective nodes:



The toString method will just need to create a **current** Node reference like the Stack, Bag and List classes we've

been working on, and iterate over the Nodes, adding each node's data to the result variable.

Step 2) define the enqueue method

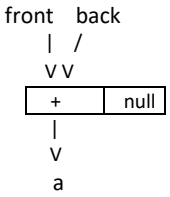
enqueue: this method adds an item to the front of the queue.

Algorithm

a) create `newNode` with the new data
b) if the Queue is empty we'll just add it as the only node in Queue, it will be both front and back like so:

set `front` to refer to `newNode`
set `back` to refer to `newNode`

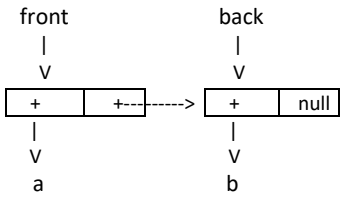
Suppose we execute **qu.enqueu("a");** on an empty queue. We would expect following memory map:



c) otherwise (meaning, the queue is not empty) we'll add to the end of the chain of nodes, like so:

set the next pointer of back to refer to the newNode
set back to refer to the newNode

Suppose we execute **qu.enqueu("b");** on our queue from the last example. We would expect the following memory map:



Step 3) define the dequeue and getFront methods

dequeue: this is very similar to the pop method for the StackLink class. However when removing the last item you have to check to make sure that **back** is also set to null.

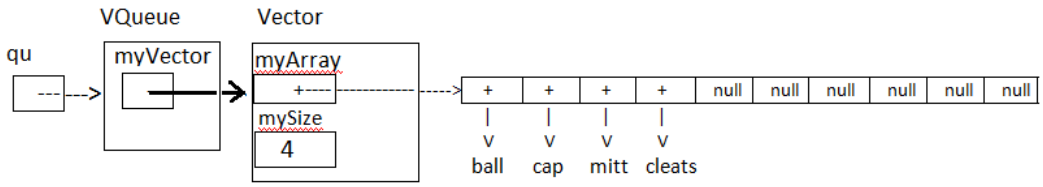
getFront: is very similar to peek for the StackLink class.

VQueue<E> class

Just to show there are many ways of implementing a Queue, we will now create a VQueue class, which will be an adapter class for an internal **Vector** object. What is a Vector? A Vector is like our ArrayList class, except it starts counting from 0 instead of 1. If you would like to explore what a Vector object is, see the **FunWithVectors** program in the assignment 3 download. Feel free to add lines to this demo app. The Vector class implements the Java List Interface, as you can see in the javadoc at http://docs.oracle.com/javase/6/docs/api/java/util/Vector.html

By using a Vector to hold our queue's data, we essentially get to use a "smart" array inside our VQueue object, instead of a normal array. This smart array, (or Vector) object, will manage the data we store in our queue. So we won't be managing an array or a linked chain of nodes of our own, but instead use the Vector object, an object that has its own internal array with methods that allow you to add to and remove data from it.

The Vector object then becomes like a wrapper around your queue's data, like another layer of software between the client (code where our VQueue is being used) and the data in the queue. If we create a VQueue object with the statement, VQueue<String> qu = new VQueue<String>(); and add some items to it, the memory map picture would look something like this:



In other words, our VQueue object now has only one instance field, a reference to a Vector (**myVector**) where all of the data is stored. The internal fields of the Vector are shown, assuming its array is called myArray and its size field is called mySize. We don't need to know what these fields are called, nor will we refer to them directly, they are just shown to help you form a picture of what is going on. When we wish to write code that enqueues or dequeues items to or from our queue we will simply use the Vector add and remove methods to pass the information to or from our internal vector.

The Java Vector class is very similar to our ArrayList class. It manages items by storing them in an array, and allows insertion and deletion anywhere in the list (as shown in FunWithVectors in the assignment3 download). The Vector class also supports generics and automatically resizes the array which means we never have to worry about it filling up. It also has a large number of methods for modifying it. For our purposes, the following Vector methods will be most useful:

VECTOR METHODS

```
public void add(E item);  // adds item to the end of the array
public E remove(int index);   // removes item at cell index and returns. Copies over any gap that may occur in array
public E get(int index);   // returns without removing the item at cell index
public boolean isEmpty();
public void clear();
```

Since Vector supports generics, we'll make our VQueue class also support generics.

Step 4) Declare the Vector<E> instance field in VQueue, as indicated near the top of the class file. This variable will refer to our internal Vector object. For example,

```
Vector<E> myVector;
```

Then Define the Constructor, isEmpty and toString methods

Constructor: this method will initialize myVector as a new Vector of <E>

isEmpty: this method will just return true if myVector is empty, false otherwise

Step 5) Define the enqueue method

This method just adds an item to the back of the vector. We will consider location 0 to be the front, so the back will be at the other end, deeper into the Vector's internal array. Luckily the Vector add method adds an item automatically to the end of the list.

Step 6) Define the dequeue and getFront methods

The dequeue method should remove and return the item located at the front of the queue, or, location 0 in the vector. The Vector remove method takes a number for the cell we want to remove, in this case, location 0. Note that the vector's own remove method will take care of shifting items over to fill in the gap caused by the removal.

The getFront just returns the item at the front of the queue, or location 0 in our queues internal vector. Use one of the Vector methods listed above that is appropriate for this task.

Test your methods. Then, when finished**, go back and analyze the Big-Oh growth of all the methods in both of your Stack classes, including the constructors**. Add a comment right after the method header with the Big-Oh notation, such as O(1), O(N), O(N^2) etc. (hint, they are probably all either O(1) or O(N).