# Homework 6 :: Stacks & Queues

Overview

In this assignment you will be writing one implementation of a `Stack` and two implementations of a `Queue`. Then you will be writing three methods that will use your `Stack` and `Queue` implementations.

As with all assignments, you will be graded in part on your coding style. Your code should be easy to read, organized, and well-documented. Be consistent in your use of indentation and braces. See the style guide for more details..

Background :: Stack

A `Stack` is an abstract data type that allows only a limited number of operations on a collection of data. Elements are stored in order of insertion and the elements are removed in reverse order. Because the last element that is inserted is the first to be removed, it is often referred to as a Last-in First-out (LIFO) collection. For this assignment, we created a `Java` interface that specifies the operations that any implementation of a `Stack` must have:
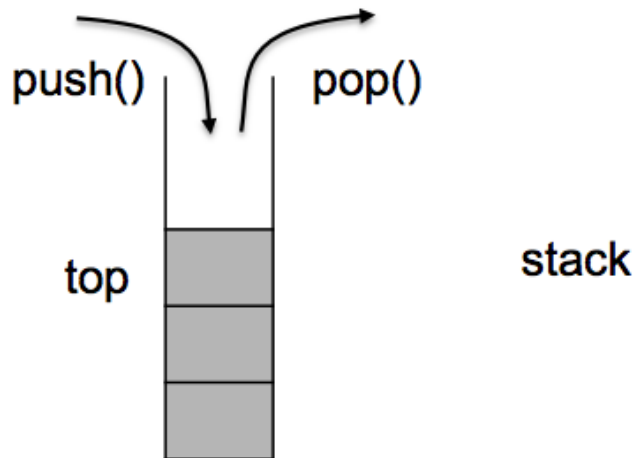
There are several ways that `Stacks` can be implemented. For example, you can use an dynamic array, an `ArrayList`, a linked list of nodes, or a `LinkedList`. For this assignment you will implement a stack using an `array`.

Background :: Queue

A `Queue` is another abstract data type that allows only a limited number of operations on a collection of data. Again, elements are stored in order of insertion. But with queues the elements are removed in the same order. Because the first element that is inserted is the first to be removed, it is often referred to as First-in First-out (FIFO) collection. For this assignment, we created a `Java` interface that specifies the operations that any implementation of a queue must have:
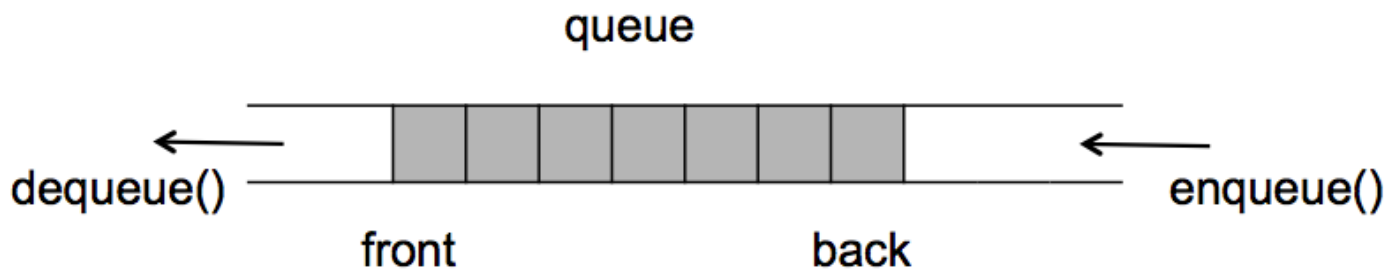
Again, there are several ways that Queues can be implemented. For this assignment you will write two implementations Queue, one using an array and other using two Stacks.

1. MyStack — an interface for the abstract data type stack (given to you)
2. ArrayStack — a class you will complete that uses an array to implement MyStack
3. MyQueue — an interface for the abstract data type Queue (given to you)
4. ArrayQueue — a class you will complete that uses a circular array to implement MyQueue
5. TwoStackQueue — a class you will complete that uses two stacks to implement MyQueue
6. StackQueueSolver — a class you will complete that solves several problems using Stack and/or Queue

Exercise :: Identify

Download the Basecode and fill in the required fields like so:

```
/**
 * @author [First Name] [Last Name] <[Andrew ID]>
 * @section [Section Letter]
 */

/**
 * @author Jess Vird <jvirdo>
 * @section A
 */
```

Exercise :: Examine

Inspect each of the methods that have been provided to you. You are not responsible for understanding each line of code, but you should understand the overall algorithm and concept.

Exercise :: ArrayStack

In the ArrayStack class, implement the methods of the MyStack interface as specified. Be sure to throw exceptions when required. Your stack should be unbounded, like an ArrayList is, where it can grow in size. All the operations must take amortized O(1) time in the worst case.

In addition, implement the toString method so that it returns a String in exactly the following format (three examples shown with integers):

```
top [ ] bottom // (this shows an empty stack)
top [ 6 ] bottom // (this shows the stack with 1 element)
top [ 5 2 8 ] bottom // (this shows the stack with 3 elements)
```

NOTE: The grader checks that your toString makes a string EXACTLY the same as the above examples (number of spaces, capitalization, etc) and comments are NOT part of the string

Be sure to test your class thoroughly, as you will be using it later in the assignment.

Exercise :: ArrayQueue

In order to have an Queue implementation using an array that has O(1) amortized runtime for enqueue and O(1) amortized runtime for dequeue, you will not move the elements of the array. You will

maintain both the index of the front and the back of the queue. Every time you enqueue an element you increment the back index. Similarly, every time you dequeue an element you increment the front index.
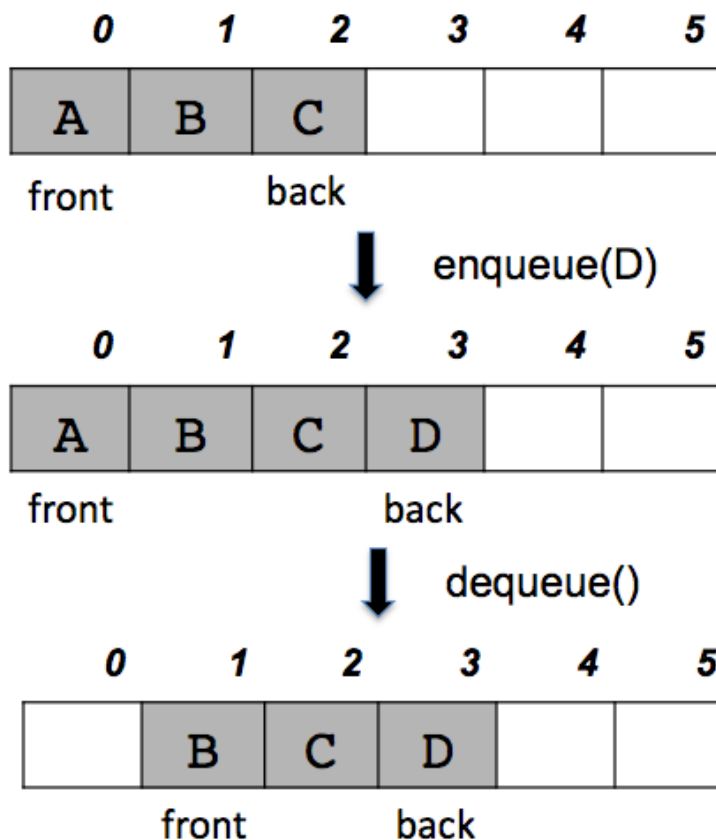
The problem is that, as you enqueue and dequeue elements from the queue, the data in the queue is stored towards the end of the array and space frees up at the beginning of the array. To reuse any unused space at the front of the array, we will consider the array to be a circular array. That is, whenever the we reach the end of the array, we "wrap around" to the beginning of the array. At various stages it is possible that elements could be at the front of the array, the middle of the array, the end of the array, or wrapped around the end of the array. (HINT: You may find that the modulus (%) operator is handy for keeping the front and back indices within the index bounds of the array.)
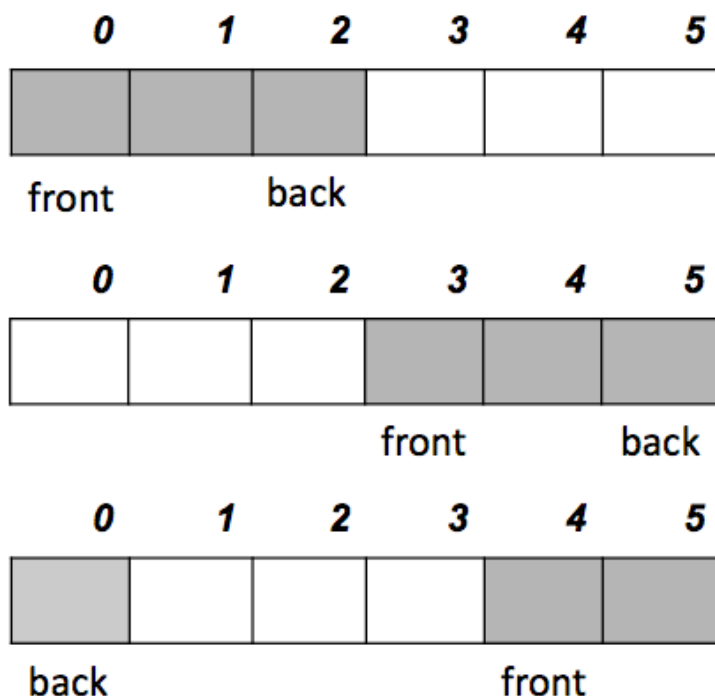
Some configurations of three elements in a queue

Notice that no element in the array moves. Once an element is placed in the array, it stays put and the queue moves forward so that eventually the element is at the front of the queue and is removed with the next dequeue operation.

As with `ArrayLists`, when the array is full, we can expand the array by copying the queue data to a new longer array. The front of the queue should be at the front of the new array (index 0).

Although not strictly necessary, it is convenient to have a field that keeps the count of the number of elements in the queue. (It's also possible to compute the index of the back of the queue from the front index and the number of elements in the queue.)

Implement the methods of the `ArrayQueue` including the `toString` method, which should return a string in exactly the following format (two examples shown with integers):

```
front: 6 back: 0
front [ 5 2 8 ] back // this shows the queue with 3 elements wrapped, where the array has length 8
```

Be sure to test your class thoroughly, as you will be using it later in the assignment.

NOTE: The grader checks that your toString makes a string EXACTLY the same as the above examples (number of spaces, capitalization, etc) and comments are NOT part of the string

Be sure to test your class thoroughly, as you will be using it later in the assignment.

Exercise :: TwoStackQueue

A `Queue` can also be implemented using two stacks. One `Stack` is used to enqueue elements, while the other is used to dequeue elements. That is, when an element is added to the queue it is pushed on the `in` stack. When the `out` stack is empty, both `dequeue` and `peek` operations must first transfer the contents of the `in` stack to the `out` stack.

queue made with two stacks
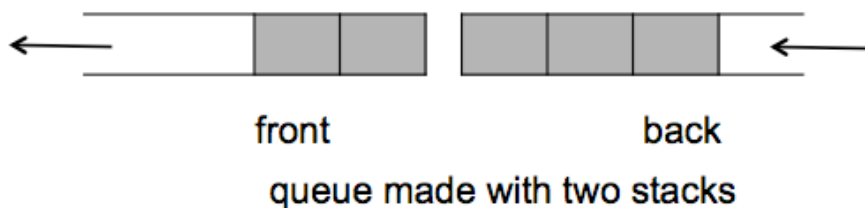
In the figure above, the `in` Stack is represented by the stack on the right (to `enqueue`) and the out stack is represented by the stack on the left (to `dequeue`). As strange as this implementation may seem, it is a common implementation for purely "functional" programming languages.

Again, the runtime of the `enqueue` operation is O(1), as it is simply a push onto the stack. The runtime of the `dequeue` operation is not so simple. It is sometimes O(1), as it is a `pop` operation. Other times it is an O(n) operation, as it has to transfer all the data from the in stack to the out stack. Dequeue is amortized O(1) time: Each element is pushed on one stack and then transferred to the other stack exactly once.

For `TwoStackQueue`, the only collection data type you may use is the `MyStack` collection type, should you require any additional storage. This class fields and additional storage must by of type `MyStack`, which means you may use only `pop`, `peek`, `push`, and `isEmpty`. In particular, you may not add extra methods to your `ArrayStack` class for the convenience of this class.

Implement the methods of the `TwoStackQueue`. When you implement the `toString` method, it must format the returned string in the specified format exactly. Use a vertical bar to indicate which data is in the out stack:

```
front [ 4 | ] back // queue with 1 element on the out stack and the in stack empty
front [ 5 | 2 8 ] back // queue with 1 element on the out stack and 2 elements on the in stack
```

NOTE: The grader checks that your toString makes a string EXACTLY the same as the above examples (number of spaces, capitalization, etc) and comments are NOT part of the string

Be sure to test your class thoroughly, as you will be using it later in the assignment.

Exercise :: StackQueueSolver

Complete the following methods:

1. At the Ann Oy Bank, a bank teller decides to go to lunch and a line forms with m people. Number the customers 1, 2, ..., m. When the teller returns, the teller tells the person at the head of the line to go to the back of the line, doing this n times. Then the teller serves the person at the head of the line. The teller repeats this process until the line is empty (assuming no one else enters the bank). Write a method lastCustomer that has two parameters that specify the number of customers initially in line (m) and the number of customers sent to the back of the line each time (n). The method should return the number of the customer that is served LAST. You may assume that m > 0 and 1 <= n <= m. Use a Queue to compute your solution.

   In a comment above the method write the worst-case runtime (in Big-O) of your method.

2. Write a method areEqual that returns true if the two stacks of strings specified in the parameters have the same elements in the same order, and false otherwise. Two elements are the same if they refer to the same object. The method may remove elements from the stacks, but it must return the elements to the stacks in the same order to restore the stacks to their original state. The only additional data structure that it can use as auxiliary storage is a single stack (i.e., no arrays, no ArrayLists, no LinkedLists no Queues, no Strings...). The method may also only use O(1) additional space.

   In a comment above the method, write the worst-case runtime (in Big-O) of your method when the two stacks have n elements.

3. Write a method duplicateStack that returns a new stack of Integers containing the same elements and in the same order as the stack specified in the parameter. The method should create a new stack and fill it with the same data elements as the given stack. (You do not need to duplicate the contents of the elements.) Before the method finishes, it must restore the contents of the original stack to its original state (same contents in the same order). Besides the new stack that the method returns, the only additional data structure that it can use is a single queue. The method may also only use O(1) additional space.

   In a comment above the method write the worst-case runtime (in Big-O) of your method when the specified stack has n elements.

Submitting your Work

When you have completed the assignment and tested your code thoroughly, create a .zip file on your work. Only include the following file(s):

1. ArrayStack.java
2. ArrayQueue.java
3. StackQueueSolver.java
4. TwoStackQueue.java

Do not include any .jar or .class files when you submit, and zip only the files listed above. Do not zip not an entire folder containing the file(s).

Once you've zipped your files, visit the Autolab site -- there is a link on the top of this page -- and upload your zip file.

Keep a local copy for your records.