Stacks and Queues

CS 106 - Introduction to Data Structures

Lab 4 - one week

You will be given some starter files for this lab (ArrayStack.java, Stack.java, Queue.java, Deque.java, EmptyQueueException, FullStackException, FullQueue-Exception). **You are not allowed to make any changes to these files.**

1 Testing

Implement the main method in Main.java to test all the methods in the implementations you'll create of `TwoStacksQueue` and `ArrayDeque`. You should include enough tests to clearly demonstrate that your implementation works.

2 Queues

Your first task for this lab is to implement the Queue interface. You will do this in a class called TwoStacksQueue (a stub implementation is provided for you in the starter files) that uses two ArrayStacks to store the data for a queue. Since you are implementing the Queue interface, you should be sure that your Queue behaves appropriately, i.e., that items are handled in a First-In-First-Out (FIFO) way. Since you are using two stacks to simulate a Queue, it will certainly not be the most efficient implementation of a Queue and that's ok - just as long as you know that and can analyze the runtime appropriately in the `README` - see below.

Requirements:

1. Implement the Queue interface using two stacks. Your implementation should ensure that the queue is FIFO.

2. You should be sure that you throw the Exceptions that are listed in the interface - i.e., make sure your Queue is not throwing Stack related exceptions that would reveal the implementation.

3. Use two (and no more) `ArrayStack` objects as instance variables.

4. Do *not* use any other array, ArrayList, Linked List, or additional ArrayStacks for any part of your implementation, including as local variables within a method. This means that you should *not* create a `new ArrayStack` more than twice in the class. It *is* ok to store extra pointers to the existing two ArrayStacks.

5. Override the toString method to return a String that contains the contents of the current Queue in the following format:
   `(elment1, element2, ..., elementn)`

6. Your `README` should provide a discussion on the design of your data structure, with particular emphasis on how you implemented the `add` and `remove` operations.

7. Your `README` should provide a worse-case big-Oh analysis of each implemented method, i.e., including all the methods in the interface, the toString method, as well as the constructor.

3 Double-ended Queues (Deques)

A double-ended queue is one where inserting and removing can happen at both ends. You will be implementing the Deque interface using an array in `ArrayDeque.java`. Recall that we discussed in class how to implement the Queue ADT using a circular array. You may also find the discussion in Section 6.3 of your textbook helpful.

Requirements:

1. Implement the included Deque interface using an array in a circular fashion.

2. Override the toString method for ArrayDeque to return a String that contains the contents of the current Deque in the following format:
   `(elment1, element2, ..., elementn)`

4 README

Your README file should at least include the below information:

1. Your name.

2. Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications. Documented bugs will receive less deduction versus uncaught ones.

3. Write-up: Contents as discussed above.

5 Extra Credit Options

All extra credit should be done **after** successful completion of all of the base requirements for this assignment. The number of points awarded for extra credit will be smaller than those for completion of the base requirements and the extra credit is designed to be *harder* than those basic requirements as well. You may choose which of the extra credit options below to pursue and can receive credit for some and not others where that makes sense.

1. Allow the queue and stack implementations from all sections above to grow in size (i.e. remove the need to throw an exception when the queue is full). This should be done *without* modifying the ArrayStacks implementation.

2. Implement a new stack data structure (call it `NewStack`), storing integers, that supports the operations `push`, `pop`, `top`, `size`, `isEmpty`, and an additional operation `minElement`, which returns the smallest element currently in the stack. All operations should run in $O(1)$ worst case time - note that this means no loops of any kind. Explain how your data structure works in your README and justify the $O(1)$. It is acceptable to write a non-generic `NewStack` that only stores integers and doesn't implement the `Stack` interface. Override `toString` for `NewStack` to return a `String` that contains the contents of the current stack in the following format
   `(element1, element2, ..., elementn)`
   where elements were inserted into the stack in order 1 to n. The $O(1)$ requirement doesn't apply to the toString method.