1 Overview

In this lab, we will have some fun with stacks and queues. The lab is split up into 2 parts:

1. MyQueue.java: Implement a queue using two stacks
2. MyStack.java: Implement a stack using two queues

You will then create a set of JUnit tests in MyQueueTest.java and MyStackTest.java to ensure that your new stack and queue implementations works like normal stacks and queues should.

2 Problem 1: MyQueue

2.1 MyQueue implementation

Implement the stubbed out functions in the MyQueue class using two stacks. The function specifications are shown below. The MyQueue class has 2 stacks as instance variables, which represent the inner state of the queue. Therefore, if an object has been dequeued from the queue it should not be in either stack. If an object is enqueued into the queue it should be present in at least one stack.

Note: There is an inefficient and an efficient way to do this. Try to work your way towards a more efficient implementation after first reaching an inefficient way that works. Feel free to ask your TA questions if you are stuck at any step.

Methods to implement (specs also in the .java file):

/**
 * Returns the element at the front of the queue
 * without removing it.
 * @return element at the front of this queue
 */
public K peek() throws NoSuchElementException;
/**
 * Adds the specified element e at the back of this queue.
 * @param k
 */
public void enqueue(final K k);

/**
 * Deletes and returns the element at the front of this queue.
 * @return element at the front of this queue
 */
public K dequeue() throws NoSuchElementException;

/**
 * Tests if this queue is empty.
 * @return True on empty false otherwise.
 */
public boolean isEmpty();

2.2 JUnit: MyQueueTest

Write JUnit tests to add to MyQueueTest to test your implementation of the MyQueue class, to make sure that it operates like a normal queue would. Your JUnit must individually test all four methods described in the interface. You should also add more complex tests as you see fit.

You can take advantage of the setUp() and tearDown() methods which are run before and after every test (You may notice, they are annotated @Before and @After).

3 Problem 2: MyStack

3.1 MyStack implementation

Same idea, this time for a stack. The MyStack class has 2 queues as instance variables, which represent the inner state of the stack. So, like with MyQueue, if an item is pushed onto the stack, it should be in at least one queue, and if an object is popped, it should not be in either queue.

Note: Calling pop() on an empty stack results in a java.util.NoSuchElementException being thrown.

Note: In the java.util.Queue interface enqueue is called offer() and dequeue is called remove(). Use isEmpty() to test if a queue is empty. Use these 3 methods and peek() only.


/**
* Returns the element at the top of the stack
* without popping it.
* @return element at the top of this stack
*/
public K peek() throws NoSuchElementException;

/**
* Push an element onto the top of the stack.
* @param k The element to push.
*/
public void push(K k);

/**
* Pops the top element off the stack.
* @return The top element on the stack.
* @throws NoSuchElementException when popping an empty stack.
*/
public K pop() throws NoSuchElementException;

/**
* Tests if this stack is empty.
* @return True on empty false otherwise.
*/
public boolean isEmpty();

3.2 JUnit: MyStackTest

Write JUnit tests to add to MyStackTest to test your implementation of the MyStack class, to make sure that it operates like a normal stack would.