POSSIBLE MIDTERM 1 QUESTIONS

Computer Science 61BL, Summer 2009

Question 1a. For this question, we use the `IntSequence` class from lab. In particular, we will assume that the following methods have been coded and thoroughly tested for correctness. Pay special attention to the `remove` method, which is slightly different from what you encountered in lab.

```
// Constructor; the argument will be the actual size of the array, or
// equivalently, the (temporary) maximum number of elements it can hold.
public IntSequence (int capacity)

// Returns true when this sequence is empty and returns false otherwise.
public boolean isEmpty ( )

// Returns the number of values in this sequence
// (Note the distinction between the size and the capacity).
public int size ( )

// Returns the value at the given position in the sequence.
public int elementAt (int pos)

// Include the argument among those stored in myValues by placing it in
// the first unused spot in the array and incrementing the count.
public void add (int toBeAdded)

// Removes the specified element at the position provided, and returns it.
public int remove (int position)

// Adds the given element to the specified position in the sequence.
public void insert (int element, int position)
```

We will focus our attention on a special kind of sequence that will only allow us to insert at only one particular location in the sequence. We will call these sequences *restricted sequences*, and we will represent them with the class `RestrictedIntSequence`. In the skeleton for the class defined below, fill in the methods `insertRestricted` and `deleteRestricted`, described as in the comments. In your solution, **take advantage of existing methods in the `IntSequence` class**.

```
public class RestrictedIntSequence extends IntSequence {

    private int myInsertPosition;
    private int myDeletePosition;

    public RestrictedIntSequence (int maxSize, int insertPosition, int deletePosition) {
        super(maxSize);
        myInsertPosition = insertPosition;
        myDeletePosition = deletePosition;
```

```
    }

    // Restricts insertion to the insert position given during the
    // construction of this RestrictedIntSequence. For example, if
    // the insert position was set to 3, elements can only be inserted
    // at the third position. This insertion should only happen if there
    // are enough elements in the RestrictedIntSequence. For example, in
    // the example where the insert position is set to 3, this method will
    // throw an IllegalArgumentException if there are, say, only 2 elements
    // in the IntSequence.
    public void insertRestricted (int element) ___________________________________ {
        if (_____________________) {


        } else {


        }
    }

    // Restricts deletion to the delete position given during the
    // construction of this RestrictedIntSequence. For example, if
    // the delete position was set to 3, elements can only be deleted
    // from the third position. This deletion should only happen if there
    // are enough elements in the RestrictedIntSequence. For example, in
    // the example where the delete position is set to 3, this method will
    // throw an IllegalArgumentException if there are, say, only 2 elements
    // in the IntSequence.
    public int removeRestricted (int element) ___________________________________ {
        if (_____________________) {


        } else {


        }
    }
}
```

Question 1b. Jon claims that since we need to restrict entry to a `RestrictedIntSequence` anyway, we should override the `insert` and `remove` method on the `IntSequence` class, instead of creating entirely new methods called `insertRestricted` and `deleteRestricted`. Explain to Jon why he is wrong in his claim.

Question 1c. Describe **four** test cases, one set each for `insertRestricted` and `deleteRestricted`, that you would use to test each method. Use an example for each test case that you write.

Question 1c. We will now declare two new structures: a **queue** and a **stack**, both of which have their own special importance in computer science. A **queue** is what is known as a **first-in, first-out** structure; just like everyday queues, the first person in the queue is the first person out of the queue. A **stack** is what is known as a **last-in, first-out** structure; just like a stack of plates, the last plate on the stack is the first plate that is removed. We will represent these with the help of our `IntSequence` and `RestrictedIntSequence`

classes.

In the class definition for the `Queue` class below, fill in the constructor, and the methods `addToQueue`, `removeFromQueue` and `isQueueEmpty` as described. In your solution, **take advantage of existing methods in the** `IntSequence` **and** `RestrictedIntSequence` **classes**. The shorter your solution to each question, the higher your score.

```
public class Queue extends RestrictedIntSequence {

    RestrictedIntSequence myQueue;

    public Queue (int maxSize) {
        myQueue = ____________________________________;
    }

    // Adds the given element to the end of a queue.
    public void addToQueue (int newElement) {


    }

    // Removes an element from the front of the queue.
    // If there are no more elements available, this
    // method should throw a NoSuchElementException.
    public int removeFromQueue () _____________________________ {
        try {



        } catch (IllegalArgumentException e) {



        }
    }

    // Returns true if the queue is empty; false otherwise.
    public boolean isQueueEmpty () {
        return ____________________________________________;
    }
}
```

In the class definition for the `Stack` class below, fill in the constructor, and the methods `addToStack`, `removeFromStack` and `isStackEmpty` as described. In your solution, **take advantage of existing methods in the** `IntSequence` **and** `RestrictedIntSequence` **classes**. The shorter your solution to each question, the higher your score.

```
public class Stack extends RestrictedIntSequence {

    RestrictedIntSequence myStack;

    public Stack (int maxSize) {
```

```
        myStack = ____________________________________;
    }

    // Adds the given element to the front of a stack.
    public void addToStack (int newElement) {


    }

    // Removes an element from the front of the stack.
    // If there are no more elements available, this
    // method should throw a NoSuchElementException.
    public int removeFromStack () _____________________________ {
        try {



        } catch (IllegalArgumentException e) {



        }
    }

    // Returns true if the stack is empty; false otherwise.
    public boolean isStackEmpty () {
        return ________________________________________________;
    }
}
```

Question 1d. Kaushik now claims that the `insert` and `remove` methods that the `Queue` and `Stack` classes inherit from their grandparent class `IntSequence` are not required anymore. He would like to ensure that everytime the `insert` and `remove` methods are called, that an error message be printed instead. How would you help him implement such a mechanism?

Question 2.