EECE 310 (Winter Term 2): Midterm Exam

Name: **Student Number:**

Total Time: **1 hour and 15 minutes** (i.e., 75 minutes).

Instructions

1. This exam has four questions printed on 5 pages. Please ensure that you have a complete copy before starting the exam.
2. Write your answers in the space provided. If you need more space, use the back of the paper.
3. **When necessary, make reasonable assumptions and state them clearly in your answers.**

Points Distribution Table

Question	Points	Out of
1A		4
1B		6
2A		5
2B		5
3		5
4		5
Total		30

Question 1: Consider the IntQueue ADT below and answer the questions that follow.

```
public class IntQueue
{
    // IntQueues are mutable, bounded queues of Integers that operate in FIFO order

    private Vector<Integer> queue;  // the rep
    private int capacity;       /         / maximum number of elements that can be stored
    // constructor
    public IntQueue(int n) {
        // REQUIRES: n >= 0
        // EFFECTS: Creates a new empty instance of IntQueue of capacity n
        capacity = n;
        queue = new Vector<Integer>(n);
    }

    // Observers
    public int size() {
        // EFFECTS: Return the number of elements in the queue
        return queue.size();
    }

    // Mutators
```

```
    public void enqueue(int element) throws QueueIsFullException {
        // MODIFIES: this
        // EFFECTS: Appends an element to the end of the queue if queue is not full,
        //          throws the QueueIsFullException otherwise
        if (queue.size() == capacity)
            throw new QueueIsFullException();
        queue.add( new Integer(element) );
    }

    public int dequeue() throws QueueIsEmptyException {
        // MODIFIES: this
        // EFFECTS: Removes an element from the end of the queue if not empty,
        //          throws the QueueIsEmptyException otherwise
        if (queue.size() == 0)
            throw new QueueIsEmptyException();
        return queue.remove(0).intValue();
    }
}
```

You get 1 point for each of the clauses in the above methods.

A. Write the Abstraction Function and RepInvariant for the IntQueue.

AF(C) = c.queue.get(i).intValue, 0 <= i < c.queue.size (1 points)

RI : (1) queue != null && queue.capacity >= 0 && (1 point)

(2) 0 <= queue.size <= queue.capacity (1 point)

(3) queue has only non-null elements (1 point)

B. Using data-type induction, show that the representation invariant is established by the constructor and is preserved by both the *enqueue* and *dequeue* operations. Write the proof clearly showing each step.

HINT: For each clause in the representation invariant, show that it is preserved by the operations. If you like, you may number the clauses for brevity, but show the numbering and use it consistently.

The proof presented below refers to the numbered clauses of the rep invariant in question 1C.

Constructor: Initializes a new empty queue, and sets capacity to a value n >= 0

So it satisfies clause 1 (since queue is allocated and capacity >=0), clause 2 (since size = 0) and clause3 (it has no elements, and hence no non-null elements).

enQueue operation: Assume that the Rep Invariant holds prior to the operation.

if clause 1 holds prior to the operation, it holds after, since neither queue nor capacity is modified.

if clause 2 holds prior to the operation, then either queue.size == capacity or queue.size < capacity. The enqueue operation throws an exception in the former case, and adds an element to the queue in the latter case. In both cases, the value of size at the end of the operation: size' <= capacity. Also, if size >=0 before the call, then clearly size' >=0 after the call. Hence, clause 2 holds after the operation.

If clause 3 holds prior to the operation, it holds after as the enQueue inserts a non-null element, and no other element of the queue is modified.

deQueue operation: Assume that the Rep Invariant holds prior to the operation

if clause 1 holds prior to the operation, it holds after, since neither queue nor capacity is modified.

if clause 2 holds prior to the function, then either queue.size == 0 or queue.size > 0. In the former case, the deQueue operation throws an exception while in the latter case, it removes an element from the queue. Thus, in both cases, the value of size at the end of the operation: size' >= 0. Also, if size <= capacity before the call, then clearly size' <= capacity. Hence, clause 2 holds after the operation.

if clause 3 holds prior to the operation, it holds after because the deQueue operation does not add any new elements to the queue.

NOTE: If your rep-invariant is incorrect, you will not receive any marks for the proof. The proofs for each operation carries 2 points.

Question 2: Consider the following function in Java that compares two IntQueues in Question 1.
A. Write its specification (REQUIRES, MODIFIES and EFFECTS clauses). If a clause is not needed, leave it blank.

```
public static int compare (IntQueue p, IntQueue q)
        throws NullPointerException, UnequalSizes {
  // REQUIRES: p and q are NOT aliased to each other, i.e., p =/= q          (1 point)
  //MODIFIES: p, q                                                          (1 point)
  // EFFECTS: Throws NullPointerException if either p or q is null,          (3 points)
  //          Throws UnequalSizes exception if the queues have different sizes,
  //          Otherwise, return 0 if all the elements of p and q are the same, else
  //          Return -1 if the first element of p that differs from that of q
  //          is greater, and 1 if the element is lesser than the one in q

    if (p == NULL ) throw new NullPointerException();
    if (p.size() != q.size() ) throw new UnequalSizes();
    while (p.size() != 0) {
        int a = p.dequeue();
        int b = q.dequeue();
        if (a > b) return -1;
        if (a < b) return 1;
    }
    return 0;
```

B. Write test-cases to perform **black box** testing of the *compare* procedure. Aim for as few test cases as possible and explain the reason for each test case. **(5 points)**

Reason for test-case	**Input(s) (p, q)**	**Output**
Induce Null Pointer Exception	null, { 1}	NullPointerException
Induce Null Pointer Exception	{1 }, null	NullPointerException
Induce UnequalSizes Exception	{ 1 }, { 1, 2 }	UnequalSizes Exception
Induce UnequalSizes Exception	{ 1, 2}, { 1 }	UnequalSizes Exception
Paths through the spec	{1, 2}, {1, 2}	0
Paths through the spec	{0, 1}, {0, 2}	1
Paths through the spec	{0, 1, 3}, {0, 1, 2}	-1
Boundary	{}, {}	0

NOTE: You will earn 1 point for the first 2 cases, 1 point for the next 2 cases and 2 points for the paths thro' the spec. You will get 1 point for the Boundary test case.

Question 3: Assume that we have added a producer method for the IntQueue ADT (shown below) to initialize the IntQueue from a vector. Explain what is wrong with the code given below (HINT: Consider

whether it is possible to break the representation invariant in any way). Also, rewrite the code to fix the problem(s). If there are multiple things wrong with the code, you need to fix them all. **(5 points)**

```
public IntQueue (Vector<Integer> v, int n ) {
        // REQUIRES: n >= 0
        // EFFECTS: Initializes the queue from the vector v.
        capacity = n;
        queue = v;
}
```

There are two main things wrong with the procedure. First, it exposes the rep by directly initializing the queue with a reference to an external object. Second, it does not check if the vector v in fact satisfies the representation invariant of the ADT (i.e., v may be NULL or its size may exceed the queue's capacity).

To fix the problem, you need to clone the vector within the method and check that the repInvariant holds. The check for this is written as part of the constructor or may be in a separate method (e.g., repOK). Here we show the solution corresponding to the check being a part of the constructor itself.

```
public IntQueue(Vector<Integer> v, int n) throws InvalidRepException {

        capacity = n;

        queue = v.clone();

        if (v==null || v.size() > capacity ) throw new InvalidRepException();

}
```

The first part carries 3 points and the correct version of the constructor carries 2 points.

Question 4. Consider the following sequence of statements in Java. Draw a diagram that represents the state of the heap and the stack after the code is executed. Assume that the statements are within the same scope (i.e., procedure) and are executed consecutively. **(5 points)**

```
int [ ] a = { 1, 2, 3 };
int [ ] b = new int[3];
int [ ] c = b;
int [ ] d = null;
c[1] = 5;
int x = a[2];
String abc = "abc";
String z = abc;
String abcd = z.concat ("d");
```