

## Problem Set 3

Due: 11:59pm Wednesday, March 7

This is the final draft of PS3, containing both problems.

### Exam 1 Notice:

The first exam (a take-home exam) will be posted on the evening of Wed. Mar. 7 and will be due at 6pm on Fri. Mar. 16. **This is a hard deadline. No extensions will be given after this time.** The exam will cover the material in class through Lecture 9 (Mon. Mar. 5), Lab 6 (Tue. Mar. 6), and Problem Sets 1 – 3. Because you should focus on the exam after it is posted, it is **strongly** recommended that you submit PS3 on time (11:59pm Wednesday, March 7).

### Overview:

The purpose of this assignment is to give you practice with implementing abstract data types (ADTs). As usual, it is recommended that you (1) start early and (2) work with a partner (a different one than you worked with on PS1 and PS2).

### Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 1:30pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your final version of `IntVector.java` from Problem 1.
3. a transcript of `java IntVector` for Problem 1.
4. your final version of `QueueCircular.java` from Problem 2.
5. a transcript of `java QueueCircular` for Problem 2.

Each team should also submit a single softcopy (consisting of your entire final `ps3` directory) to the drop directory `~cs230/drop/ps3/username`, where `username` is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following Linux commands in the team member's account where the code is stored:

```
cd /students/username/cs230
cp -R ps3 ~cs230/drop/ps3/username/
```

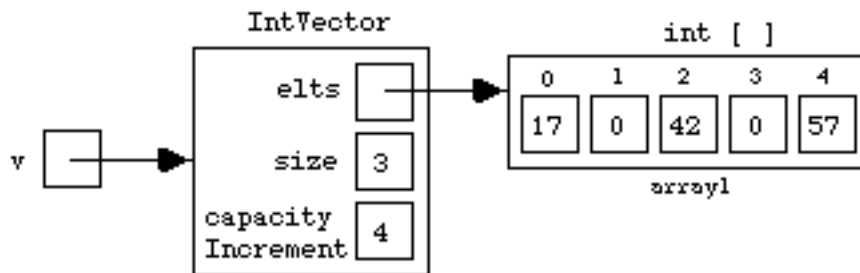
## Problem 1 [50]: An Array Implementation of Integer Vectors

### The IntVector class

We have seen that the Java `Vector` class is a versatile data structure handy for a wide range of uses. However, unlike arrays, the slots of a Java `Vector` instance may not directly hold values of primitive data types like `int`, `double`, and `char`. Instead, primitive data must be packaged in an instance of a wrapper class (e.g., `Integer`, `Double`, `Character`) before they are stored in a vector, and must be unpackaged upon extraction. This is both cumbersome for the programmer and inefficient for execution.<sup>1</sup>

In this problem, we will explore the implementation of an `IntVector` class as a specialized kind of vector that can store only integers. The `IntVector` class has the same kinds of methods as the `Vector<T>` class, except that wherever a `Vector` method uses a value of type `T`, the corresponding `IntVector` method uses an `int`. When using the `IntVector` class to represent a collection of integers, it is not necessary to convert integers to/from instances of the `Integer` class.

We will implement an instance of `IntVector` using an array of integers. Here is a diagram of one instance of `IntVector` that contains, in order, the three integers 17, 0, and 42:

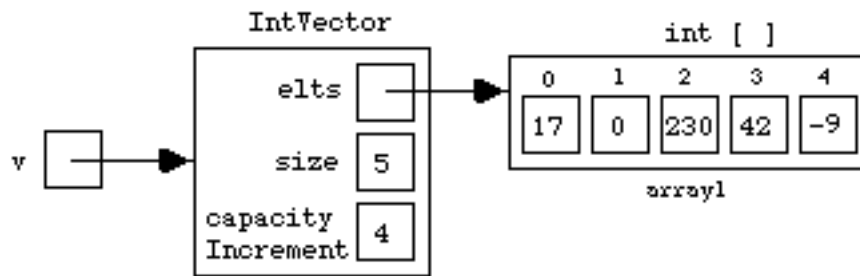


The `elts` instance variable points to an integer array that stores the elements of the vector. The number of slots in the array (known as the *capacity* of the vector) can be any number greater than or equal to the number of elements in the vector. The `size` instance variable indicates the number of slots (from left to right, starting at slot 0) that contain the elements of the vector. The remaining slots are extra space for the vector to “grow into”. These extra slots may contain any integers whatsoever; their values do not matter because they are ignored by the `IntVector` methods. We shall refer to the values of these extra slots as “garbage”. In the above example, the size of 3 indicates that only slots 0 through 2 hold vector elements and slots 3 and 4 hold garbage.

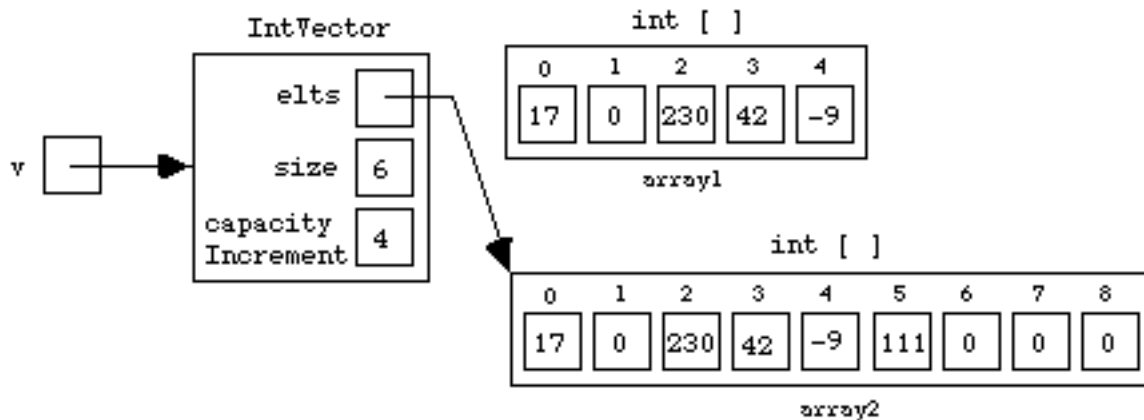
We will assume that as long as the number of elements in the vector remains less than or equal to the capacity, the array held by the `elts` variable does not change. In this case, vector operations that change, add, or remove elements must rearrange the elements within the slots of the existing array. For instance, performing `v.add(-9)` and then `v.add(2,230)` on the above vector representation should change the state of the objects as follows, where the fact that the array label has not changed indicates that it is the same array as in the previous diagram.

---

<sup>1</sup>Java 1.5 has a so-called *autoboxing* feature (see <http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html>) that can automatically perform the packaging and unpackaging for the programmer, thus making wrapper classes less cumbersome. However, the wrapping is still performed, so the inefficiency still remains.



When an element is added or inserted into a vector whose size is equal to its capacity, it is necessary to increase its capacity. This is accomplished by creating a larger integer array, copying the vector elements from the old array to the new array, and making `elts` point to the new array. The number of slots by which the capacity should be increased is indicated by the `capacityIncrement` instance variable. For example, since the `capacityIncrement` is 4 in the example, performing `v.add(111)` installs a new integer array of size  $5 + 4 = 9$  in `elts` before it performs the insertion. The old array will be reclaimed by the Java garbage collector. By default, the garbage slots in the new array will initially contain 0, although after other vector operations the garbage slots might later contain any integer.



We will assume the convention used in Java's `Vector` class that if the `capacityIncrement` is 0, then the capacity of the vector should be doubled when the array needs to grow. For instance, if the `capacityIncrement` had been 0 above, then the new array would have had  $2 * 5 = 10$  slots.

The initial capacity and capacity increment of an `IntVector` instance are specified by arguments to the `IntVector` constructors:

```
// Constructor Methods
public IntVector(int initialCapacity, int capacityIncrement);
public IntVector(int capacityIncrement);
public IntVector();
```

An invocation of the first constructor creates an integer vector with no elements (i.e., `size` is 0) whose initial capacity is `initialCapacity` and whose capacity increment is `capacityIncrement`. In the other constructor methods; the parameters not supplied are assumed to be 0.

The `IntVector` class supports the following subset of the instance methods supported by Java's `Vector` class. If you have questions about the behavior of a particular method, consult the online Java 1.5 API, accessible from the CS230 Documentation page. The `Vector` class is in the `java.util` package. (Of course, the following specifications differ from those in the `Vector` class by replacing every occurrence of `Object` with `int`.)

```

// Instance Methods
public boolean add (int elt); // Note: always returns true.
public void add (int index, int elt);
public int capacity();
public void ensureCapacity (int minCapacity);
public int get (int index);
public int indexOf (int elt);
public int remove (int index);
public void removeAllElements();
public boolean removeElement (int elt);
public int set (int index, int elt);
public int size ();

```

### Implementing the IntVector Class

Your task in this problem is to implement the `IntVector` class using the “extensible array” representation discussed above. You should flesh out the skeletons for the above constructor methods and instance methods in the `IntVector` class in the file `~/cs230/ps3/IntVector.java`. Invoking `java IntVector` tests your implementation on a suite of test cases. The test cases will be run on vectors whose (`initialCapacity`, `capacityIncrement`) are (1,1), (2,8), (3,0), and (0,0).

#### Notes

- The `IntVector(int capacityIncrement)` and `IntVector()` constructor methods can invoke the `IntVector(int initialCapacity, int capacityIncrement)` constructor method using a two-argument `this()` method.
- The case where both the `capacity` and `capacityIncrement` are 0 is problematic, since doubling 0 yields 0 rather than a larger capacity! Your code needs to handle this case. (*Hint*: Using `ensureCapacity` whenever you want to increase the size of the array is helpful for addressing this problem.)
- Your code will need to indicate an error when an index is not within the valid indices of the vector in the `get`, `set`, (two-argument) `add`, and `remove` methods. To do this you should use the follow statement:

```
throw new ArrayIndexOutOfBoundsException(message);
```

where `message` is a string indicating the specific error message.

- The `add(int index, int elt)` method inserts `elt` at index `index` and causes the indices of all other elements in slots `index` and above to be incremented. This requires shifting such elements rightward in the array. Similarly, `remove(int index)` and `removeElement()` can cause elements to be shifted leftward in the array.
- Carefully check the results of `java IntVector` to verify that your operations are working as expected. Turn in a transcript of `java IntVector` as part of your hardcopy submission for this problem.

## Problem 2 [50]: Circular Queues

In Lecture 9, we saw that a queue can be efficiently represented as a mutable list as long as it maintains two pointers to the list: a **front** pointer to the first node of the list (where elements are dequeued) and a **back** pointer to the last node of the list (where elements are enqueued). In this problem, we shall see that it is possible for a queue representation to use just a single **back** pointer if we represent the queue as a **circular list** (a.k.a. a **cyclic list**) – i.e., a list that wraps back on itself.

For instance, in this representation, enqueueing **A**, **B**, and **C** in order onto an initially empty queue would lead to a sequence structures depicted by the box-and-pointer diagrams in Fig. 1.

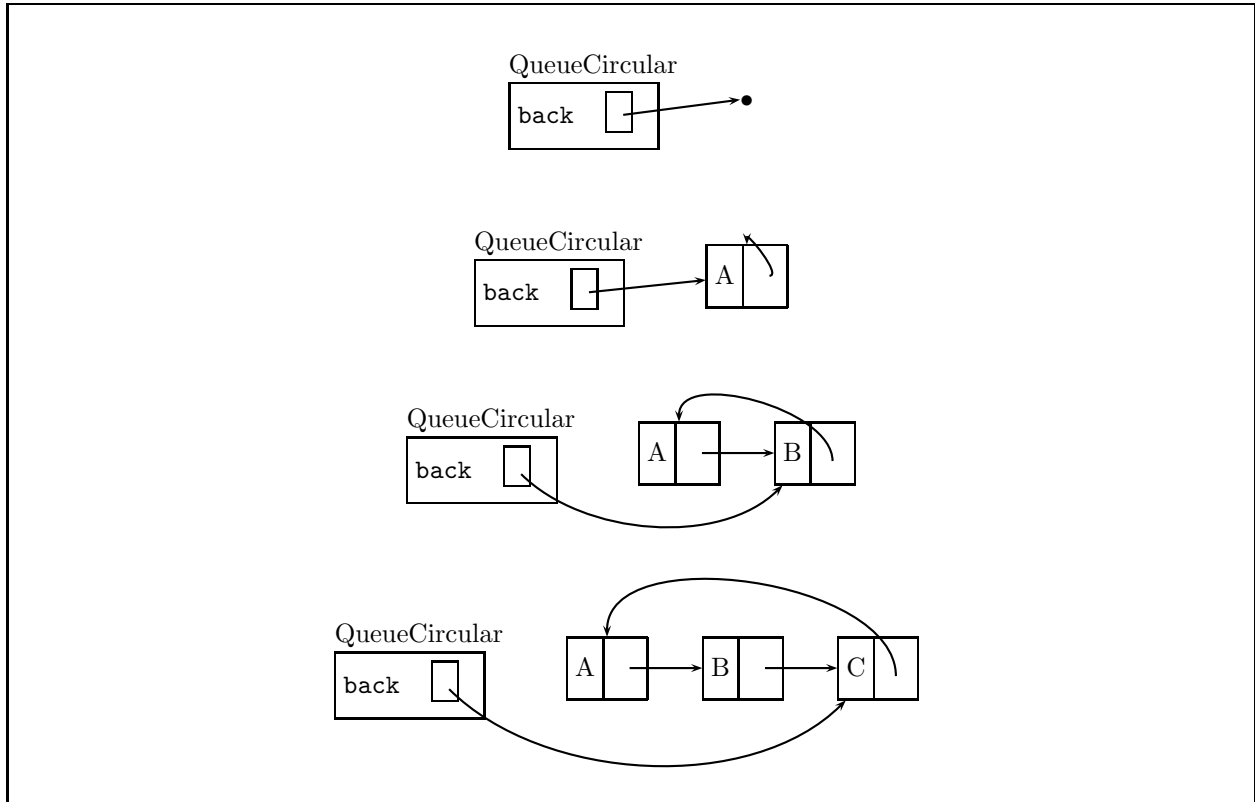


Figure 1: Enqueueing the elements **A**, **B**, and **C** onto an initially empty circular queue.

In the diagrams, the elements stored in the heads of the nodes never change. However, the tails of nodes are rewired to give the depicted structure.

In all but the empty queue case, the **back** instance variable of a `QueueCircular` instance holds a pointer to the back node of the queue (the node holding the most recently enqueued element). This facilitates enqueueing a new back node as well as dequeuing the front (least recently enqueued) node, which is always the tail of the back node.

In this problem, you will implement queues in terms of the circular list representation sketched above. Using the static methods for the mutable lists specified in the `MList<T>` contract in Appendix A, implement the following queue operations in the `Queue<T>` interface:

**Constructor Method:**

```
public QueueCircular<T> ();
```

**Instance Methods:**

```
public boolean isEmpty();
public void enq (T x);
public T deq ();
public T front();
public int size();
public void clear();
public Queue<T> copy();
public toString();
```

The file `QueueCircular.java` contains code skeletons for these methods that you should flesh out. You can test your implementation by invoking `java QueueCircular`. You should submit a transcript of this invocation as part of your hardcopy submission.

*Notes:*

- Many of the stack and queue implementations discussed in lecture and lab can be found in the directories `~/cs230/stack` and `~/cs230/queue`. You are encouraged to study these implementations and experiment with them. In particular, you are *strongly* encouraged to study `~/cs230/queue/QueueTwoEndedMList.java`, which is an `MList`-based version of the two-ended queue implementation presented in Lecture 9. Since `QueueCircular` is *very* closely related to `QueueTwoEndedMList`, it is in your best interest to understand `QueueTwoEndedMList` before starting any coding on this problem.
- The `QueueCircular` skeleton contains a working `iterator()` method that has already been implemented for you.
- The `QueueCircular` class contains the declaration

```
private static MList ML;
```

This declaration allows you to use the abbreviation `ML.` for `MList..`

- As mentioned in lecture, you should always invoke the `empty()` method of the `MList` class with an explicit type argument, as indicated by the following examples:

```
MList.<String>empty(); // Create an empty mutable list of strings
MList.<T>empty(); // Create an empty mutable list with elements of type T.
```

If you neglect to include explicit types for `empty()`, then the Java compiler may give you warnings or errors.

- You should maintain the invariant that the `back` instance variable is either the empty mutable list or a circular mutable list in which `back` points to the node containing the most recently enqueued element. Because `back` is a circular list, it is particularly tricky to define the `size`, `copy`, and `toString`. There are two approaches to writing these methods:

1. In the *surgical* approach, you first temporarily modify `back` so that it is no longer cyclic, perform appropriate operations, and then make `back` cyclic again before returning the result.
2. In the *non-surgical* approach, you directly manipulate `back` as a cyclic list. When traversing `back` using this approach, you must be sure to include an appropriate stopping condition. If you neglect to do this, you will get an infinite recursion, and your program will crash! The best way to debug an infinite recursion is to insert lots of `System.out.println` in your code, and use these to pinpoint which part of your code is causing the infinite recursion.

For the non-surgical approach, it is helpful to know that `==` tests for **pointer equality** of list nodes – it returns true only when two nodes are exactly the same object object in ObjectLand (i.e., they were created by the same invocation of `prepend`).

Because a queue may contain more than one copy of given element, it is *not* correct to compare list nodes by their head elements! Instead, used `==` on the list nodes themselves.

You are welcome to use either of the above two strategies in your `size`, `copy`, and `toString` methods. You can use different strategies for different methods if you like.

- Introduce any auxiliary methods that you find helpful.
- In many of the methods, you will need to treat the case of the empty queue (i.e., where `back` is the empty list) specially.
- As discussed in lecture, we could simplify some operations and/or make them more efficient by adding extra instance variables. For instance, the `size` method would be easier to implement and more efficient if there were a `size` instance variable. However, for this particular problem, you should not add any extra instance variables – `back` should be your *only* instance variable.

## Appendix A: The `MList<T>` class

The `MList<T>` class contains methods for manipulating mutable linked lists. Instances of this class are linked lists whose elements have type `T`. The `MList<T>` class implements the `Iterable<T>` interface.

The `MList<T>` class has no constructor methods. The only way to create `MList<T>` instances is by using the `prepend` and `empty` class methods.

The `MList<T>` class has only a few public instance methods:

**public static String toString ();**

Returns a string representation of this list, in which parentheses delimit the comma-separated string representations of this list's elements.

**public static Iterator<T> iterator ();**

Returns an iterator that generates the elements of the list from front to back.

**public static boolean equals (Object x);**

If `x` is an `MList` instance, returns `true` if it has the same length as this list and each element is equal (via the `equals()` instance method) to the corresponding element of this list. Otherwise returns `false`.

Most of the methods of the `MList<T>` class are class methods. Below are *some* of the class methods of the `MList<T>` class. For the complete set, see `~/cs230/utils/MList.java`.

**public static** <T> MList<T> empty ();  
Returns an empty mutable list. This method should always be invoked with an explicit type argument, as in `MList.<String>empty()`.

**public static** <T> boolean isEmpty (MList<T> L);  
Returns `true` if L is an empty list and `false` otherwise.

**public static** <T> MList<T> prepend (T elt, MList<T> L);  
Returns a new mutable list node whose head is `elt` and whose tail is L.

**public static** <T> T head (MList<T> L);  
Returns the head of the list L.

**public static** <T> MList<T> tail (MList<T> L);  
Returns the tail of the list L.

**public static** <T> void setHead (MList<T> L, T newHead);  
Changes the head of the list L to be `newHead`.

**public static** <T> void setTail (MList<T> L, MList<T> newTail);  
Changes the tail of the list L to be `newTail`.

**public static** <T> void setTail (MList<T> L, MList<T> newTail);  
Changes the tail of the list L to be `newTail`.

**public static** <T> String toString (MList<T> L);  
Returns a string representation of the list L, in which parentheses delimit the comma-separated string representations of the L's elements.

**public static** <T> boolean equals (MList<T> L1, MList<T> L2);  
Returns `true` if L1 and L2 have the same length and each element of L1 is equal (via the `equals()` instance method) to the corresponding element of L2. Otherwise, returns `false`.

**public static** <T> int length (MList<T> L);  
Returns the number of nodes in L.

**public static** <T> MList<T> lastNode (MList<T> L);  
Returns the last node of a non-cyclic list L. Raises an exception if L is empty.

**public static** <T> MList<T> copy (MList<T> L);  
Returns a shallow copy of L. The result consists of new `MList<T>` nodes that share the same elements as L.

*Problem Set Header Page*  
*Please make this the first page of your hardcopy submission.*

**CS230 Problem Set 3**  
**Due 1:30pm Tuesday February 23**

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with*):

*By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.*

Signature(s):

*In the **Time** column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

<b>Part</b>	<b>Time</b>	<b>Score</b>
General Reading		
Problem 1 [50]		
Problem 2 [50]		
<b>Total</b>		