

Problem Set 5

Due: 11:59pm Monday, November 1

Overview:

The purpose of this assignment is to give you practice with implementing collections via linear structures.

Working Together:

If you worked with a partner on a previous problem set and want to work with a partner on this assignment, you are encouraged to choose a different partner. However, you may also work with someone you worked with in the first half of the semester.

Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 6pm 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. For Problem 1, your hardcopy submission should be your final versions of `MaxPQArray.java` and `MinPQArray.java`, your testing transcripts showing that these work as expected.
3. For Problem 2, your hardcopy submission should be your final version of `QueueCircular.java`, and your testing transcripts showing that it works as expected.

Each team should also submit a single softcopy (consisting of your final `ps5` directory) to the drop directory `~cs230/drop/ps5/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 commands in Linux in the account of the team member being used to store the code.

```
cd /students/username/cs230
cp -R ps5 ~cs230/drop/ps5/username/
```

Problem 0: Getting Started

Before starting either programming problem, you should review the collection code that we have been covering in class. In particular, you should carefully study the classes mentioned below after performing a `cvs update -d`.

a. : Flat Collections

Because this assignment involves working with queues and priority queues, you should study the following files:

```

// Queues
~/cs230/flat-collections/Queue.java
~/cs230/flat-collections/QueueTester.java
~/cs230/flat-collections/QueueVectorBF.java
~/cs230/flat-collections/QueueVectorFB.java
~/cs230/flat-collections/QueueTwoEndedMList.java

// Priority Queues
~/cs230/flat-collections/MaxPQ.java
~/cs230/flat-collections/MaxPQTester.java
~/cs230/flat-collections/MaxPQVector.java
~/cs230/flat-collections/MinPQ.java
~/cs230/flat-collections/MinPQTester.java
~/cs230/flat-collections/MinPQVector.java

```

We call these **flat collections** because each implementation file and interface file is self-contained.

b. : Hierarchical Collections

A drawback of the flat collections is that much work is repeated between classes. For instance:

- the `Queue`, `MaxPQ`, and `MinPQ` interfaces are nearly identical;
- the `QueueVectorBF` and `QueueVectorFB` implementations are similar;
- the `MaxPQVector` and `MinPQVector` implementations are similar;
- the `QueueTester`, `MaxPQTester`, and `MinPQTester` testing classes are nearly identical.

Collection interfaces, implementations, and testing classes can be reorganized in a hierarchy that allows common features to be expressed in nodes high in the hierarchy and shared by nodes lower in the hierarchy via inheritance. In this organization, a class need only specify variables and methods that differentiate it from its superclass. This “programming by differences” is a hallmark of object-oriented design.

To understand this organization, carefully study the following files. These specify the same interfaces and implementations as with the flat collections, but do so with a hierarchical organization:

```

// General Collections
~/cs230/collections/Collection.java
~/cs230/collections/CollectionImpl.java

// Queues
~/cs230/collections/Queue.java
~/cs230/collections/QueueImpl.java
~/cs230/collections/QueueVectorBF.java
~/cs230/collections/QueueVectorFB.java
~/cs230/collections/QueueTwoEndedMList.java

// Priority Queues
~/cs230/collections/PQImpl.java
~/cs230/collections/MaxPQ.java
~/cs230/collections/MaxPQVector.java
~/cs230/collections/MinPQ.java
~/cs230/collections/MinPQVector.java

```

Problem 1 [55]: Representing Priority Queues as Arrays

Arrays can be used to efficiently implement arbitrary sized collections as long as a `size` instance variable is used to hold the number of collection elements in the array stored in the `elts` instance variable. Slots with indices 0 through `elts.size - 1` hold the elements of the collection, while slots `size` through `elts.length` hold “garbage elements” that can safely be ignored. When `size = elts.length`, any attempt to add a new element to the collection must increase the size of the array. It’s a good idea to double the size of the array in these cases to make such events rare.

In this problem, you will be implementing priority queues as arrays. This is very similar to the vector implementation of priority queues you studied in Problem 0. Indeed, Java vectors themselves are usually implemented as arrays in the manner described above.

a. [45]: MaxPQArray

In this problem, you are to flesh out an implementation of `MaxPQ` that represents a priority queue using three instance variables:

1. A variable `comp` that holds the `Comparator` used to determine the order of elements. If no comparator is explicitly supplied in the creation of a `MaxPQArray` instance, `comp` should be initialized to an instance of the `ComparableComparator` class.
2. A variable `size` that holds the number of elements in the priority queue.
3. A variable `elts` that holds an array of at least `size` slots, where the slots at indices 0 through `size - 1` store the elements of the priority queue in order from low to high according to the order determined by `comp`. If `elts.length > size`, the slots at indices `size` through `elts.length - 1` are considered to hold “garbage” elements.

Your goal in this part is to flesh out the skeleton of the `MaxPQArray` class in the file `MaxPQArray.java`. This class implements the `MaxPQ` interface. You will need to flesh out the following methods:

Constructor Method:

```
public MaxPQArray (Comparator c);  
public MaxPQArray ();
```

Instance Methods:

```
public void enq (Object x);  
public Object deq ();  
public Object first();  
public int size();  
public boolean isEmpty();  
public void clear();  
public Object clone();  
public Enumeration elements();  
public ObjectList toList();
```

Notes:

- Define whatever auxiliary methods you deem helpful. You should make such methods `private`.
- Be sure to use the `Comparator` instance in `comp` for all element comparisons.
- To access methods from `ObjectList` and `ObjectListOps`, you can use the `OL.` prefix.

- For `elements()`, you should define a new Enumeration class that enumerates the elements in the correct order. You can either define a separate class or declare an anonymous inner subclass of `Enumeration`.
- Test your implementation by invoking the `main` method via `java MinPQArray`. You should turn in a transcript of this invocation.

b. [10]: `MinPQArray`

It is possible to implement a `MinPQ` using the same three instance variables as above. Rather than creating such a class from scratch, it is possible to leverage off the existing `MaxPQArray` class by making `MinPQArray` a subclass of `MaxPQArray` that implements `MinPQ`. The file `MinPQArray.java` contains the skeleton of an implementation of the `MinPQArray` class defined in this way. Your goal is to flesh out the skeleton with the *minimal* number of methods that will make it correct. *Hint:* Study the files `MaxPQVector.java` and `MinPQVector.java`, which contain implementations of queues related in a similar way.

Test your implementation by invoking the `main` method via `java MinPQArray`. You should turn in a transcript of this invocation.

Problem 2 [45]: Circular Queues

In lecture, we saw that a queue can be efficiently represented as a mutable list as long as it maintains pointers to both the front node of the list (where elements are dequeued) and 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 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 following box-and-pointer diagrams in Fig. 1. In the diagrams, the elements stored in the heads of the nodes never change. The tails of nodes are rewired to give the depicted structure.

In all but the empty queue case, the queue variable 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 node (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 mutable and immutable object lists, implement the following queue operations in the destructive interface to queues:

Constructor Method:

```
public QueueCircular ();
```

Instance Methods:

```
public boolean isEmpty();
public void enq (Object x);
public Object deq ();
public Object first();
public int size();
public void clear();
public Object clone();
public ObjectList toList();
```

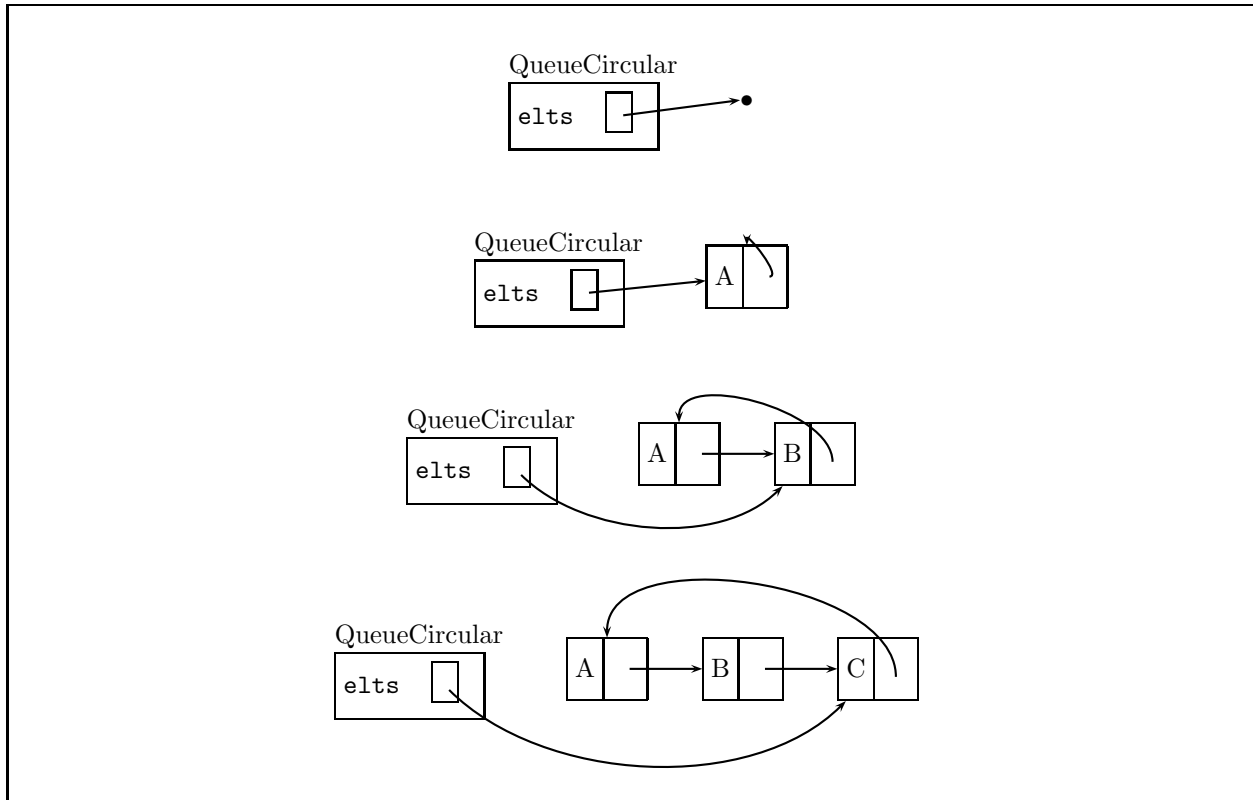


Figure 1: Enqueueing the elements A, B, and C onto an initially empty circular queue.

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.

Notes:

- The `ObjectMList` class exports the standard operations on mutable lists of objects: `empty`, `isEmpty`, `prepend`, `head`, `tail`, `setHead` and `setTail`. The `ObjectMListOps` class exports some additional operations on mutable object lists. Among these are the following (for the complete list of methods, see the file `ObjectMListOps.java`):

public static int length (ObjectMList L);

Returns the number of nodes in L.

public static ObjectMList lastNode (ObjectMList L);

Returns the last node of a non-cyclic list L. Raises an exception if L is empty.

public static ObjectMList copy (ObjectMList L);

Returns a shallow copy of L. The result consists of brand new `ObjectMList` nodes that share the same elements as L.

public static ObjectList toObjectList (ObjectMList L);

Returns an `ObjectList` that has the same elements as L.

public static ObjectMList fromObjectList (ObjectList L);

Returns an `ObjectMList` that has the same elements as L.

- The `ObjectList` and `ObjectListOps` classes export similar operations for immutable object lists (except for `setHead` and `setTail`, which immutable lists do not support).
- To access a methods from the list classes, you can use the prefix `OL.` for both `ObjectList` and `ObjectListOps` and the prefix `OML.` for both `ObjectMList` and `ObjectMListOps`.
- You should maintain the invariant that the `elts` instance variable is either the empty mutable list or a circular mutable list in which `elts` points to the node containing the most recently enqueued element. Because `elts` is a circular list, it is particularly tricky to define the `size`, `clone`, and `toList`. There are two approaches to writing these methods:
 1. In the *surgical* approach, you first temporarily modify `elts` so that it is no longer cyclic, perform appropriate operations, and then make `elts` cyclic again before returning the result.
 2. In the *non-surgical* approach, you directly manipulate `elts` as a cyclic list. When traversing `elts` 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`).

You are welcome to use either of the above two strategies in your `size`, `clone`, and `toList` 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 where `elts` 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 – `elts` should be your only instance variable.

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

CS230 Problem Set 5
Due 11:59pm Monday November 1

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.*

| Part | Time | Score |
|-----------------|-------------|--------------|
| General Reading | | |
| Problem 1 [55] | | |
| Problem 2 [45] | | |
| Total | | |