

Problem Set 6

Due: Wednesday, November 6

Overview: In this problem set, you will get experience with using and implementing stacks, queues, and priority queues.

Download: To begin this assignment, you should download a copy of the directory `~cs230/download/ps6`.

Submission:

- For Problem 1, your hardcopy submission should be your pencil-and-paper drawings from part a, your final versions of `PreOrderElts.java`, `PostOrderElts.java`, and `BreadthFirstElts.java`, and your testing transcripts showing that these work as expected.
- For Problem 2, your hardcopy submission should be your final versions of `MinPQHeadedList.java` and `MaxPQHeadedList.java`, your testing transcripts showing that these work as expected, and your answer to the questions in part c.
- For Problem 3, your hardcopy submission should be your final version of `QueueCircular.java`, and your testing transcripts showing that it works as expected.

Your softcopy submission for this problem should be your entire `ps6` directory.

Remember to include a signed cover sheet (found at the end of this problem set description) at the beginning of your hardcopy submission.

Problem 1 [35]: Tree Enumerations

In class, we have studied various “orders” for traversing the elements of a binary tree: pre-order, in-order, and post-order. In this problem, we shall extend these traversal notions to enumerating the elements of a binary tree. It turns out that stacks and queues are very helpful for implementing tree enumerations.

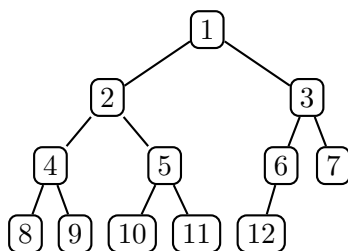
Figs. 1–2 present the implementation of a `InOrderElts` class that enumerates the elements of an `ObjectTree` according to a depth-first, left-to-right in-order traversal. The class has a single instance variable, `stk`, which holds a stack of non-empty `ObjectTrees` that intuitively are “still to be processed”.

The `main` method tests `InOrderElts` in three different ways, according to the nature of `arg` in `java InOrderElts arg`:

1. If `arg` is the string representation of a tree, the elements of the tree are displayed in in-order by `EnumTest.test`. For example:

```
[cs230@koala ps6] java InOrderElts "((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))"
-----
Enumerating elements of ((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))
4
1
5
2
6
3
7
Total number of elements: 7
```

2. If `arg` is a positive integer n , the elements of a breadth-first tree with n nodes labeled with strings (not numbers) 1 through n are displayed in in-order by `EnumTest.test`. Recall from PS5 that a breadth-first tree with n nodes is a binary tree whose n nodes have the binary addresses 1 through n . For example, the breadth-first tree with 12 nodes is:



```

import java.util.Enumeration;

public class InOrderElts implements Enumeration {
    // An enumeration that yields the elements of a tree in an in-order traversal

    private Stack stk; // Invariant: Contains a collection of non-empty ObjectTrees.

    public InOrderElts (ObjectTree t) {
        stk = new StackVector(); // Any stack implementation will do.
        if (! OT.isLeaf(t)) {
            stk.push(t);
        }
    }

    public boolean hasMoreElements() {
        return (! stk.isEmpty()); // There are more elements to enumerate as long
                                   // as stack contains one or more non-empty trees.
    }

    public Object nextElement() {
        if (stk.isEmpty()) {
            // By invariant, there are no more elements
            throw new CollectionException("InOrderElts: no more elements");
        } else {
            ObjectTree t = (ObjectTree) stk.pop();
            // By invariant, t itself is not a leaf, so the following will succeed:
            Object val = OT.value(t);
            ObjectTree lt = OT.left(t);
            ObjectTree rt = OT.right(t);
            ObjectTree lf = OT.leaf();
            if (OT.isLeaf(lt)) {
                if (! OT.isLeaf(rt)) {stk.push(rt);} // Process non-empty right subtree next
                return OT.value(t); // t is "leftless" (has no left subtree) so
                                   // can enumerate its value.
            } else {
                // Push these in the order opposite to that which they will be processed:
                stk.push(OT.node(val, lf, rt)); // 1. Leftless tree with value
                                                // and right subtree of t
                if (! OT.isLeaf(lt)) {stk.push(lt);} // 2. A non-empty left subtree of t
                return nextElement(); // Try again
            }
        }
    }
}

```

Figure 1: Implementation of InOrderElts, Part 1.

```

// ***** TESTING *****

public static void main (String [] args) {
    testString(args[0]);
}

public static void testString (String s) {
    // If s is an integer n , create a breadth first tree with n elements:
    try {
        testTree(breadthTree(Integer.parseInt(s)));
    } catch (NumberFormatException e1) {
        // Otherwise, try to parse s as a string tree representation
        try {
            testTree(OT.fromString(s));
        } catch (Exception e2) {
            // Otherwise treat as the name of a file,
            // in which each line is a number, tree rep, or filename
            Enumeration lines = new FileLines(s);
            while (lines.hasMoreElements()) {
                testString((String) lines.nextElement());
            }
        }
    }
}

public static void testTree (ObjectTree t) {
    System.out.println("-----");
    System.out.println("Enumerating elements of " + t);
    EnumTest.test(new InOrderElts(t));
}

public static ObjectTree breadthTree (int n) {
    // Create a tree with n nodes whose binary addresses are
    // 1 through n and whose values are the string
    // representations of the binary addresses.
    return brTree (1, n);
}

public static ObjectTree brTree (int lo, int hi) {
    if (lo > hi) {
        return OT.leaf();
    } else {
        return OT.node(Integer.toString(lo),
            brTree(lo*2, hi),
            brTree(lo*2 + 1, hi));
    }
}

// Hack for abbreviating ObjectTree and ObjectTreeOps as OT
private static ObjectTreeOps OT;
}

```

Figure 2: Implementation of InOrderElts, Part 2.

Here is a test case based on this tree:

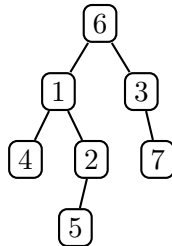
```
[cs230@koala ps6] java InOrderElts 12
-----
Enumerating elements of ((((* 8 *) 4 (* 9 *)) 2 ((* 10 *) 5 (* 11 *))) 1 (((* 12 *) 6 *) 3 (* 7 *)))
8
4
9
2
10
5
11
1
12
6
3
7
Total number of elements: 12
```

3. Otherwise, *arg* is assumed to be the name of a file whose lines are either tree representations, numbers, or other filenames, each of which is processed accordingly.

In this problem, you are asked to understand how `InOrderElts` works and to create similar enumerations for other traversal orders.

a. [7]: Understanding `InOrderElts`

In this part, you will show how `InOrderElts` works in the context of the following tree, which we shall assume is named `A`¹:



Consider the following code:

```
ObjectTree A = ObjectTreeOps.fromString("((( (* 4 *) 1 ((* 5 *) 2 *)) 6 (* 3 (* 7 *)))");
Enumeration e = new InOrderElts(A);
while (e.hasMoreElements) {
    System.out.println(e.nextElement());
}
```

Draw a sequence of “snapshots” of the contents of the stack `stk` at the beginning of *every* call to `nextElement()`. Note that `nextElement()` is called recursively, and the contents of `stk` at the beginning of these recursive calls should also be drawn. In your sequence of snapshots, also indicate where a tree value is returned by the enumeration. In this problem, you should draw the elements of a stack from left to right where the leftmost element is the top of the stack and the rightmost element is the bottom of the stack.

¹Even though the labels look like integers, assume this is an `ObjectTree` with string labels, such as "1", "2", etc.

b. [8]: PreOrderElts

Create a copy of `InOrderElts.java` named `PreOrderElts.java`, and make the following changes:

- Change *all* occurrences of `InOrderElts` to `PreOrderElts`. (Especially don't miss the constructor method name and the constructor method invocation within `testTree`.)
- Change the implementation of the `nextElement` method so that the tree elements are enumerated in *pre-order* rather than in-order.

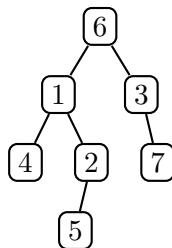
Show that your `PreOrderElts` class works by turning in a transcript of your test cases.

c. [10]: PostOrderElts

Similar to the previous part, create a copy of `InOrderElts.java` named `PostOrderElts.java` that enumerates the elements of a tree in *post-order*. Show that your `PostOrderElts` class works by turning in a transcript of your test cases.

d. [10]: BreadthFirstElts

The *level* of a tree node is the number of edges in the shortest path from the root of the tree to the node. For instance, in the tree



- node 6 is at level 0;
- nodes 1 and 3 are at level 1;
- nodes 4, 2, and 7 are at level 2;
- and node 5 is at level 3.

A *left-to-right breadth-first traversal* visits all the nodes of a tree in order of increasing level, and visits nodes at the same level from left to right. That is, it first visits the root node at level 0, then visits all nodes at level 1 from left to right, then visits all nodes at level 2 from left to right, and so on. In the sample tree shown above, a breadth-first traversal visits the nodes in the following order:

6, 1, 3, 4, 2, 7, 5

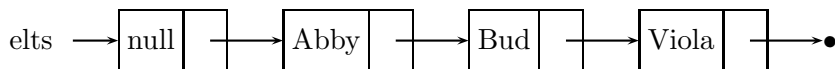
Similar to the previous two parts, create a copy of `InOrderElts.java` named `BreadthFirstElts.java` that enumerates the elements of a tree in *breadth-first order*. In addition to changing the `nextElements` method, you will also have to change the data structure that holds the trees. A stack won't work for breadth-first order – what will? Show that your `BreadFirstElts` class works by turning in a transcript of your test cases.

Problem 2 [35]: Priority Queues

a. [20]: MinPQHeadedList

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

1. A variable `comp` that holds the `Comparator` used to determine the order of elements.
2. A variable `elts` that holds a “headed” mutable object list containing the elements of the priority queue in order from low to high according to the order determined by `comp`. A headed list is one in which the first list node holds a value (let’s say it’s the `null` pointer) that is ignored by the code manipulating the list. For example, the contents of `elts` in a `MinPQHeadedList` that contains the strings `Abby`, `Bud`, and `Viola` would be:



The reason to use a headed list as opposed to a regular (unheaded) list is that it simplifies some manipulations. In part c you will consider in more detail the motivation for headed lists; for now, just assume that you must use them.

3. A variable `size` that holds the number of elements in the priority queue.

Your goal in this part is to flesh out the missing method bodies in the file `MinPQHeadedList.java`, which implements a `MinPQ` using the above instance variables. It’s a good idea first to study the implementation of `MinPQList` in `MinPQList.java` to get a feel for how a list-based implementation of a priority queue should work. However, unlike `MinPQList`, which uses *immutable* object lists, `MinPQHeadedList` uses *mutable* object lists, and you should take full advantage of the mutability. For instance when inserting an element into `elts`, you should do so in a destructive way that creates exactly one new list node.

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

Within the file `MinPQHeadedList.java`, all `ObjectMLList` and `ObjectMLListOps` static methods can be accessed by prefixing the method names with “`OML.`”.

b. [8]: MaxPQHeadedList

It is possible to implement a `MaxPQ` using the same three instance variables as above. Rather than creating such a class from scratch, it is possible to leverage off the existing `MinPQHeadedList` class by making `MaxPQHeadedList` a subclass of `MinPQHeadedList` that implements `MaxPQ`. The file `MaxPQHeadedList.java` contains the skeleton of an implementation of `MaxPQHeadedList` 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 `MaxPQList.java` and `MinPQList.java`, which contain implementations of queues related in a similar way.

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

Within the file `MaxPQHeadedList.java`, all `ObjectMLList` and `ObjectMLListOps` static methods can be accessed by prefixing the method names with “`OML.`”.

c. [7]: Questions

Answer the following questions about priority queue implementations.

1. The number of elements in an instance of `MinPQHeadedList` can be determined directly from `elts`. Why is it a good idea to have a separate `size` variable keeping track of this information?
2. Headed lists are used in data structure implementations because they simplify certain operations. Which particular priority queue operation is simplified by using headed lists vs. regular (unheaded) lists? How would that operation be implemented if `elts` were an *unheaded* list? (Give code!).
3. The `MinPQVector` class includes an instance variable named `invComp`. What is the purpose of this instance variable? Why not just get rid of this instance variable and change the body of `insertIndex` to be:

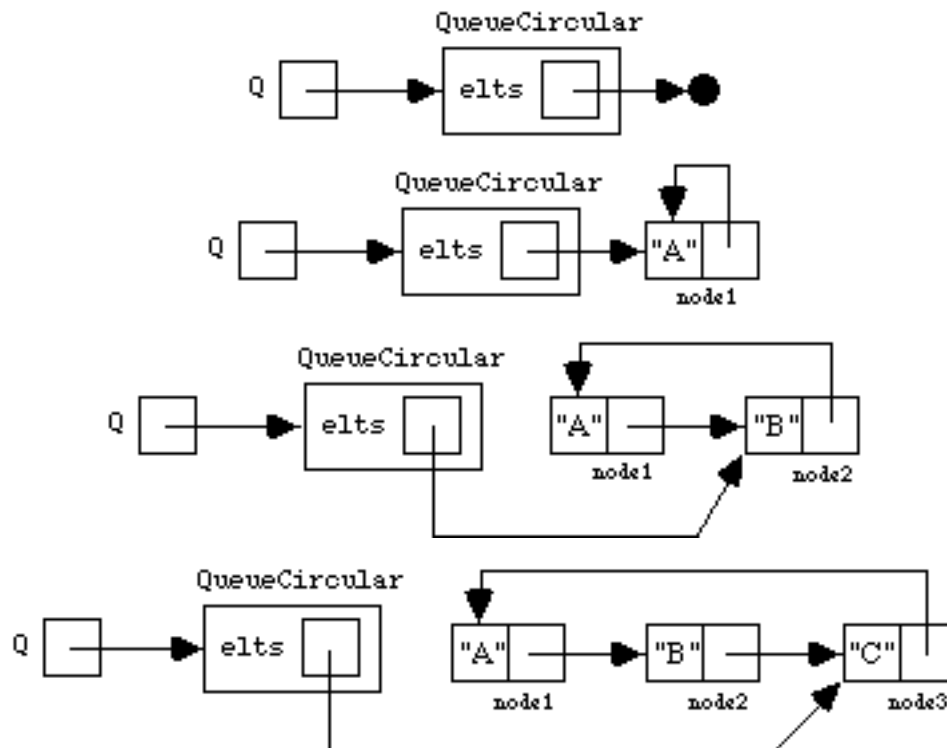
```
return VectorOps.binarySearch(x, elts, new ComparatorInverter(comp));
```

4. The `MinPQHeadedList` class inherits the `comp` instance variable and `comparator` method from the `PQImpl` class. Another implementation of `MinPQ` is `MinPQVector`, whose inheritance chain includes `MaxPQVector`, `QueueVector`, `QueueImpl` but not `PQImpl`. Could the inheritance chain for `MinPQVector` be changed to include `PQImpl`? Would this be a good idea?

Problem 3 [30]: Circular Queues

In lab, you saw that a queue could be efficiently represented as a mutable list as long as it maintained 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 to get by with just a single pointer if we represent the queue as a circular 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 the diagrams, the nodes have been annotated with labels that indicate the identity of the nodes. These labels underscore the fact that, in this representation, the head of a node never changes. However, 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 one most recently enqueued). 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 of `ObjectMList` and `ObjectMListOps`, 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();
```

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 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.

- To access a method from `ObjectMList` or `ObjectMListOps`, you can prefix the method name with “OML.”.

- To access a method from `ObjectList` or `ObjectListOps`, you can prefix the method name with “OL.”.
- 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 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. It is helpful to know that `==` tests for pointer equality of list nodes.

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.

- Introduce any auxiliary methods that you find helpful.
- In many of the methods, you need to treat the case where `elts` is the empty list specially.

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

CS230 Problem Set 6

Due Wednesday, November 6

Name:

Date & Time Submitted:

Collaborators (*anyone you worked with on the problem set*):

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

Signature:

*In the **Time** column, please estimate the time you spend on the parts of this problem set. 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 [35]		
Problem 2 [35]		
Problem 3 [30]		
Total		