

Problem Set 6

Due: 6pm Friday, November 12

Exam 2 Notice:

On Friday, November 12, the second take-home exam will be posted. It will be due at 6pm on Saturday, November 20. **This is a hard deadline. No extensions will be given after this time.**¹ The exam will cover the material in lecture through Lecture 18 (Mon. November 8) and the material in problem sets through PS6, mainly focusing on material introduced since the last exam: binary trees and contracts and implementations of stacks, queues, priority queues, sets, bags, and tables. Because you should focus on the exam, it is **strongly** recommended that you submit PS6 on time, although you may use lateness coupons to turn in PS6 late. But since the exam is worth much more than the problem set, it might be best to cut your losses on PS6 if you are not done with it so that you can focus on the exam.

Overview:

The purpose of this assignment is to give you practice with manipulating trees and implementing collections via trees.

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, submit your final version of `PS6TreeOps.java` and a transcript of running `java PS6TreeOps`.
3. For Problem 2, submit your pencil-and-paper drawings from part a, your final versions of `PostOrderElts.java` (part b) and `BreadthFirstElts.java` (part c), and your testing transcripts showing that these work as expected.
4. For Problem 3, submit your final version of `BagMBSTEntries.java` and a transcript of running `java BagMBSTEntries`.

Each team should also submit a single softcopy (consisting of your final `ps6` directory) to the drop directory `~cs230/drop/ps6/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.

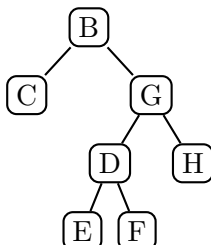
```
cd /students/username/cs230
cp -R ps6 ~cs230/drop/ps6/username/
```

¹Especially since the Red Sox season is over!

Problem 1 [40]: ObjectTree Methods

In this problem, you will define four methods that manipulate immutable binary trees of objects. Skeletons for these methods appear in the file `ps6/PS6TreeOps.java`, which you can access after you perform a `cvcs update -d`.

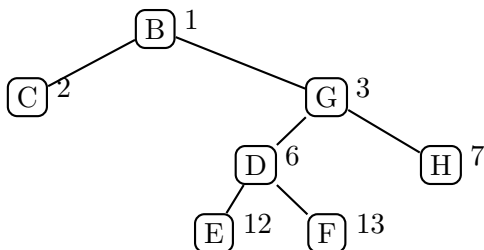
For this problem, it is helpful to know a number of definitions. The sample tree T below will be used as an example in many of the definitions:



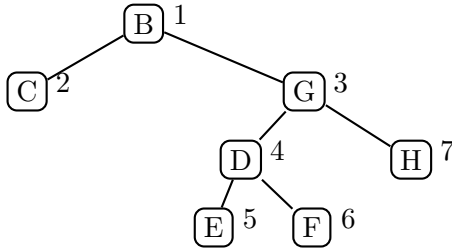
We adopt the convention of not showing leaf nodes in our tree diagrams.

- The *height* of a tree is the longest number of edges from the root to a leaf. The height of T is 4.
- A tree is *balanced* if for every node the height of its two subtrees differ by no more than one. T is not balanced because its left subtree has height 1 and its right subtree has height 3. However, T 's right subtree is balanced.
- An integer tree is a *binary search tree (BST)* if for every node in the tree, the value of the node is \geq all values in its left subtree and \leq all values in its right subtree. T is not a BST because its left subtree contains a value (C) greater than its root. However, the right subtree of T is a BST.
- The *binary address* of a tree node is defined as follows.
 - The binary address of the root of a tree is 1.
 - If the binary address of a node is n , the binary address of its left child is $2n$ and the binary address of its right child is $2n + 1$.

For example, here is a version of T in which each node has been annotated with its binary address:



- The *pre-order address* of a tree node is an integer (starting at 1) that indicates the order in which that node would be visited in a pre-order traversal of the tree. For example, here is a version of T in which each node has been annotated with its pre-order address:



Based on the above definitions, write definitions of the following `ObjectTree` methods within the class `PS6TreeOps`.

a. [10]

public static boolean isBalanced (`ObjectTree t`);

Returns `true` if `t` is balanced and `false` otherwise. You may find it helpful to use the absolute value function `Math.abs` in your solution.

b. [10]

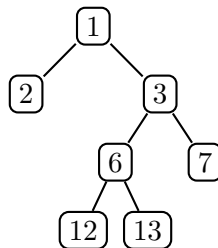
public static boolean isBST (`ObjectTree t`);

Returns `true` if `t` is a binary search tree and `false` otherwise. Be careful – it is *not* sufficient to simply compare the value at the root of the tree to the values at the root of its left and right subtrees. There are many ways to implement `isBST`, some of which use auxiliary methods.

c. [10]:

public static ObjectTree labelBinaryAddress (`ObjectTree t`);

Returns a new tree that has the same structure as `t` where the value at every node is a wrapped integer that is the binary address of the node. For example, here is the result of `labelBinaryAddress(T)`:

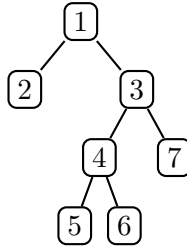


Hint: define `labelBinaryAddress` in terms of an auxiliary recursive method that takes two arguments – a tree and a binary address – and returns a tree.

d. [10]:

public static ObjectTree labelPreOrderAddress (`ObjectTree t`);

Returns a new tree that has the same structure as `t` where the value at every node is a wrapped integer that is the pre-order address of the node. For example, here is the result of `labelPreOrderAddress(T)`:



Hint: define `labelPreOrderAddress` in terms of an auxiliary recursive function that takes two arguments – a tree and a pre-order address – and returns a tree. The `OT.size()` method is particularly helpful here.

Notes:

- Flesh out the skeletons for the methods in the class `PS6TreeOps`.
- You can use the methods of the `ObjectTree` class (Appendix A) and the `ObjectTreeOps` class (Appendix B) using the prefix `OT`.
- The invocation `java PS6TreeOps method-name tree-string` will test the method named `method-name` on the tree whose string representation is `tree-string`. For example:

```
[fturbak@wampeter ps6] java PS6TreeOps labelPreOrderAddress "<<* A *> B <* C *>>"
labelPreOrderAddress( <<* A *> B <* C *>> ) =
<<* 2 *> 1 <* 3 *>>
```

The invocation `java PS6TreeOps method-name` will run some standard test cases for the method named `method-name`. The invocation `java PS6TreeOps` will run standard test cases for all five methods in the problem. You are encouraged to read and understand the testing code and to add any test cases you think are helpful.

Problem 2 [20]: 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`, that 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`:

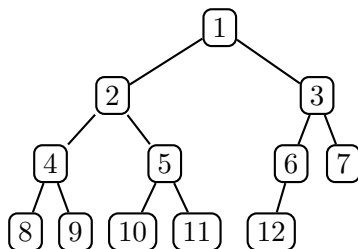
1. If `arg` is the string representation of a tree, the elements of the tree are displayed in in-order by `EnumTest.test`. For example:

```
[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`. A **breadth-first tree** with n nodes is a binary tree whose n nodes have the binary addresses 1 through n and in which each node is labeled with its binary address. The **binary address** of a tree node is defined as follows.

- The binary address of the root of a tree is 1.
- If the binary address of a node is n , the binary address of its left child is $2n$ and the binary address of its right child is $2n + 1$.

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 StackList(); // 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 RuntimeException("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); // Remember to process non-empty right subtree
                }
                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(lf, val, 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(OT.breadthTree(1, 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));
}

// 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. For example, if the file name `Atree.txt` contains the single line

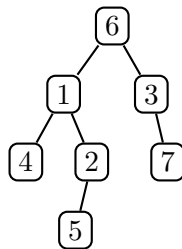
```
<<<<* 4 * > 1 << * 5 * > 2 * >> 6 < * 3 < * 7 * >>>
```

then `java InOrderElts Atree.txt` has the same behavior as the first of the above examples.

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

a. [10]: 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`²:



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.

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

b. [5]: PostOrderElts

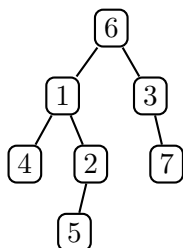
Create a copy of `InOrderElts.java` named `PostOrderElts.java`, and make the following changes:

- Change *all* occurrences of `InOrderElts` to `PostOrderElts`. (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 *post-order* rather than in-order.

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

c. [5]: 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 part (b), 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 `nextElement` 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 3 [40]: Mutable BST of Bag Entries Implementation of Bags

A *bag* is a collection of unordered elements that may contain multiple occurrences of each element. In the CS230 collection hierarchy, the `Bag` interface describes mutable bags. You should study this interface (Appendix D) before proceeding with this problem.

In this problem, you will implement a class `BagMBSTBagEntries` that represents bags as a mutable binary search tree of entries that pair elements in the bag with their number of occurrences. Each entry should be an instance of the following `BagEntry` class:

```
public class BagEntry {

    public Object elt;
    public int num;

    public BagEntry (Object elt, int num) {
        this.elt = elt;
        this.num = num;
    }

    public String toString () {
        return "BagEntry[" + elt + "," + num + "];"
    }

}
```

To improve the running time of the `size()` and `count()` bag operations, the values to be returned by these methods should be cached in instance variables of the `BagMBSTEntries` class.

So instances of `BagMBSTEntries` should have the following instance variables:

- `comp`: a comparator for determining the order of elements.
- `entries`: a mutable binary search tree (i.e., an instance of `ObjectMTree` satisfying the binary search tree property) whose elements are instances of `BagEntry`.
- `size`: the number of element occurrences currently in the bag (includes duplicates).
- `count`: the number of *distinct* elements currently in the bag (does not include duplicates).

For example, Fig. 3 shows one possible representation of an instance of `BagMBSTEntries` that contains two As, three Bs and one C. (The shape of the tree is determined by the order in which the elements were inserted.)

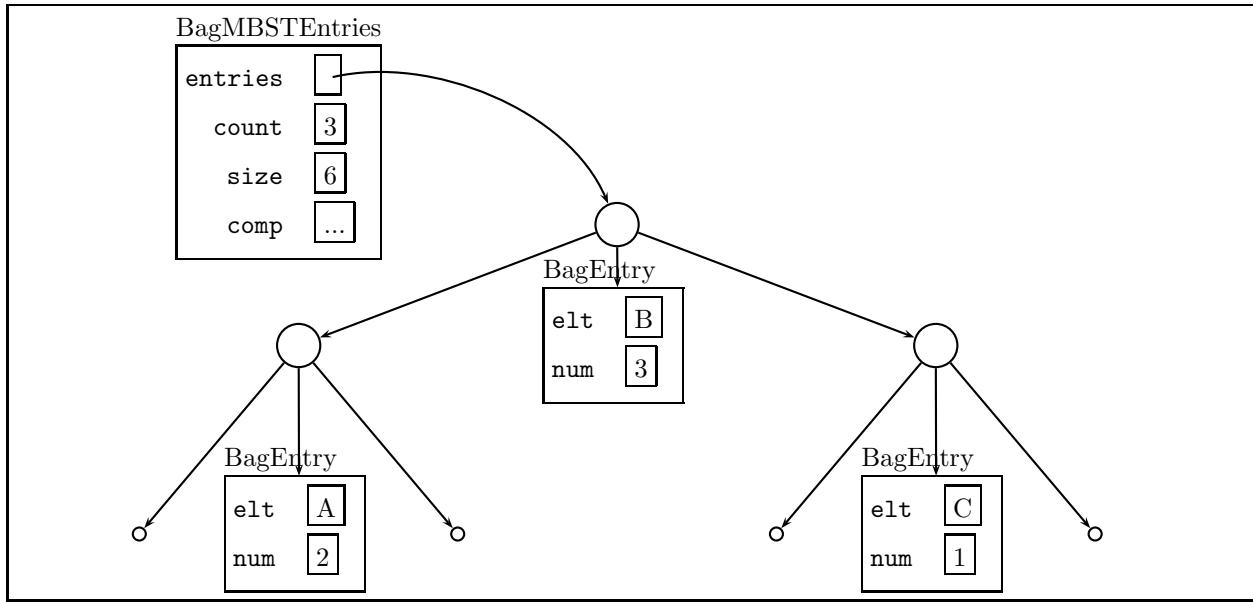


Figure 3: An example of a BagMBSTEntries instance.

To complete this problem, you need to flesh out the following methods of the bag implementation in `BagMBSTEntries.java` using the representation described above:

Constructor Methods

```

public BagMBSTEntries (Comparator c);
public BagMBSTEntries ();

```

Instance Methods

```

public Object choose();
public void clear();
public Object clone();
public int count();
public boolean delete(Object x);
public boolean deleteAll(Object x);
public Object deleteOne();
public boolean isMember(Object x);
public void insert(Object x);
public ObjectList toList();
public int occurrences(Object x);
public int size();

```

Note that many instance methods from the `Bag` interface in Appendix D are missing from the above list. This is because the `BagMBSTEntries` class inherits implementations of these other methods from its superclasses.

Test your implementation by executing `java BagMBSTEntries`, which invokes the bag methods on various simple `BagMBSTEntries` instances. Study the output carefully to make sure that the methods behave as expected. You should turn in the transcript of this test for your final version of the code as part of your hardcopy submission.

Notes:

- Mutable object trees are instances of the class `ObjectMTree`, whose contract is given in Appendix C. The `BagMBSTEntries` class has been configured so that the `ObjectMTree` operations are accessible via the `OMT.` prefix.
- `ObjectList` and `ObjectListOps` operations are accessible via the `OL.` prefix.
- You may define any private auxiliary methods and additional classes that you find helpful for completing this problem.
- Many methods require searching through the `entries` binary search tree to find an existing `BagEntry` or the insertion point for a new `BagEntry`. Additionally, many of these methods not only need to keep track of the current tree node, but also need to keep track of its parent and whether the current node is the left or right child of the parent node. To avoid writing similar code many times, it's a good idea to write a single `findEntry` auxiliary method that abstracts over the process of finding an entry in `entries` and can be invoked many times. The result of `findEntry` is an instance of the `FindEntryInfo` class (see Fig. 4) whose instance variables summarize the information needed by various other methods. (This class is provided for you in the `ps6` directory.) Here is a specification of the `findEntry` method:

```
private FindEntryInfo findEntry (Object x);
```

If a `BagEntry` for `x` exists in the `entries` tree, returns a `FindEntryInfo fei` where:

- `fei.entry` is the entry whose `elt` is `x`,
- `fei.child` is the tree node containing `fei.entry`.
- `fei.parent` is the parent of `fei.child` (or `null` if `fei.child` is the root of `entries`).
- `fei.isChildToLeft` is `true` if `fei.child` is to the left child of `fei.parent` (or there is no parent) and `false` otherwise.

If a `BagEntry` for `x` does not exist in the `entries` tree, returns a `FindEntryInfo fei` where:

- `fei.entry` is `null`.
- `fei.child` is the leaf where `x` would be inserted into the tree.
- `fei.parent` is the node below which `x` would be inserted into the tree.
- `fei.isChildToLeft` is `true` if `x` would be inserted to the left of `fei.parent` and `false` otherwise.

- Depending on how your binary search operations are defined, you might directly use `comp`, or you might need to “lift” `comp` via the `BagEntryComparator` class presented in Fig. 5. This class is provided for you in the `ps6` directory.

```

public class FindEntryInfo {

    public ObjectMTree parent, child;
    public boolean isChildToLeft;
    public BagEntry entry;

    public FindEntryInfo (ObjectMTree parent, ObjectMTree child, boolean isChildToLeft) {
        this.parent = parent;
        this.child = child;
        this.isChildToLeft = isChildToLeft;
        if (ObjectMTree.isLeaf(child)) {
            this.entry = null;
        } else {
            this.entry = (BagEntry) ObjectMTree.value(child);
        }
    }
}

```

Figure 4: The FindEntryInfo class.

```

import java.util.Comparator;

public class BagEntryComparator implements Comparator {

    private Comparator eltComp;

    public BagEntryComparator (Comparator eltComp) {
        this.eltComp = eltComp;
    }

    public int compare (Object x, Object y) {
        return eltComp.compare(((BagEntry) x).elt, ((BagEntry) y).elt);
    }

    public boolean equals (Object c) {
        if (c instanceof BagEntryComparator) {
            return eltComp.equals(((BagEntryComparator) c).eltComp);
        } else {
            return false;
        }
    }
}

```

Figure 5: The BagEntryComparator class.

Appendix A: ObjectTree Contract

The `ObjectTree` class models immutable trees whose nodes hold object values.

Public Class Methods:

public static `ObjectTree leaf ()`;

Returns a leaf – i.e., a distinguished non-value-bearing node that denotes an empty tree.

public static `ObjectTree node (ObjectTree l, Object v, ObjectTree r)`;

Returns a tree node whose left subtree is `l`, whose value is `v`, and whose right subtree is `r`.

public static `boolean isLeaf (ObjectTree t)`;

Returns `true` if `t` is a leaf and `false` otherwise (i.e., if `t` is a tree node).

public static `Object value (ObjectTree t)`;

If `t` is a node, returns the value it holds. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no value.

public static `ObjectTree left (ObjectTree t)`;

If `t` is a node, returns its left subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no left subtree.

public static `ObjectTree right (ObjectTree t)`;

If `t` is a node, returns its right subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no right subtree.

public static `String toString (ObjectTree t)`;

Returns the string representation of `t` (see the description in the `toString()` instance method).

public static `ObjectTree fromString (String s)`;

If `s` is a string representation of an object tree (see the description in the `toString()` instance method), returns an object tree with string elements whose string representation is `s`. Otherwise, throws a `RuntimeException`.

Public Instance Methods:

public `String toString ()`;

Returns a string representation of this tree. In this representation, a leaf is represented by `*` and a node `N` is represented by `<L V R>`, where `L` is the string representation of the left subtree of `N`, `V` is the string representation of the value of `N`, and `R` is the string representation of the right subtree of `N`. For example, the tree created via:

```
node(node(leaf(),"A",leaf()), "B", node(node(leaf(),"C",leaf()), "D", leaf()))
```

has the following string representation:

```
"<<* A *> B <<* C *> D *>>"
```

Appendix B: ObjectTreeOps Contract

The `ObjectTreeOps` class includes the following methods for manipulating object trees:

Public Class Methods:

public static int height (`ObjectTree t`);
Returns the height of `t`.

public static int size (`ObjectTree t`);
Returns the number of nodes in `t`.

public static ObjectList preOrderList (`ObjectTree t`);
Returns the elements of `t` as they would be visited in a pre-order left-to-right depth-first traversal.

public static ObjectList inOrderList (`ObjectTree t`);
Returns the elements of `t` as they would be visited in an in-order left-to-right depth-first traversal.

public static ObjectList postOrderList (`ObjectTree t`);
Returns the elements of `t` as they would be visited in an post-order left-to-right depth-first traversal.

public static Object BSTMin (`ObjectTree t`);
If `t` is a non-empty binary search tree, returns the least element in `t`. If `t` is an empty tree, returns `null`.

public static Object BSTMax (`ObjectTree t`);
If `t` is a non-empty binary search tree, returns the greatest element in `t`. If `t` is an empty tree, returns `null`.

Appendix C: ObjectMTree Contract

The `ObjectMTree` class models mutable trees whose nodes hold object values.

Public Class Methods:

```
public static ObjectMTree leaf ();
```

Returns a leaf – i.e., a distinguished non-value-bearing node that denotes an empty tree.

```
public static ObjectMTree node (ObjectMTree l, Object v, ObjectMTree r);
```

Returns a tree node whose left subtree is `l`, whose value is `v`, and whose right subtree is `r`.

```
public static boolean isLeaf (ObjectMTree t);
```

Returns `true` if `t` is a leaf and `false` otherwise (i.e., if `t` is a tree node).

```
public static Object value (ObjectMTree t);
```

If `t` is a node, returns the value it holds. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no value.

```
public static ObjectMTree left (ObjectMTree t);
```

If `t` is a node, returns its left subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no left subtree.

```
public static ObjectMTree right (ObjectMTree t);
```

If `t` is a node, returns its right subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no right subtree.

```
public static void setValue (ObjectMTree t, Object newValue);
```

If `t` is a node, its value is changed to be `newValue`. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no value.

```
public static void setLeft (ObjectMTree t, ObjectMTree newLeft);
```

If `t` is a node, its left subtree is changed to be `newLeft`. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no left subtree.

```
public static void setRight (ObjectMTree t, ObjectMTree newRight);
```

If `t` is a node, its right subtree is changed to be `newRight`. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no right subtree.

Public Instance Methods:

```
public String toString ();
```

Returns a string representation of this tree. In this representation, a leaf is represented by `*` and a node `N` is represented by `<L V R>`, where `L` is the string representation of the left subtree of `N`, `V` is the string representation of the value of `N`, and `R` is the string representation of the right subtree of `N`. For example, the tree created via:

```
node(node(leaf(),"A",leaf()), "B", node(node(leaf(),"C",leaf()), "D", leaf()))
```

has the following string representation:

```
"<<* A *> B <<* C *> D *>>"
```

Appendix D: Bag Interface

The **Bag** interface describes mutable collections of unordered elements that may contain multiple occurrences of each element. In mathematics, *multiset* is a synonym for *bag*. Each bag instance has a **Comparator** that is used to determine element equality (and may be used in bag implementations to order elements).

Public Instance Methods Inherited from Collection Interface:

public void insert (Object elt);

Modifies this bag by inserting a new occurrence of **elt**.

public Object deleteFirst ();

Deletes and returns an arbitrary element of this bag. Throws a **RuntimeException** if this bag is empty. The **deleteFirst** method is a synonym for **deleteOne**.

public Object first ();

Returns an arbitrary element of this bag. Throws a **RuntimeException** if this bag is empty. The **first** method is a synonym for **choose**.

public int size ();

Returns the number of element occurrences in this bag.

public boolean isEmpty ();

Returns **true** if this bag has no elements, and **false** otherwise.

public void clear ();

Removes all elements from this bag.

public Object clone ();

Returns a "shallow" copy of this bag – i.e. the copied bag structure is new, but elements themselves are shared with the old bag. Operations on the copied bag do *not* affect the original bag and vice versa. Operations on mutable elements of the copied bag *do* affect elements of the original bag, and vice versa.

public Enumeration elements ();

Returns an enumeration of the element occurrences in this bag in an arbitrary order.

public ObjectList toList ();

Returns a list of the element occurrences in this bag in an arbitrary order.

public String name ();

Returns a name indicating the implementation of this bag.

Public Instance Methods Inherited from ComparatorCollection Interface:

public Comparator comparator ();

Returns the comparator used by this bag to compare elements.

public boolean delete (Object elt);

Deletes one occurrence of **elt** from this bag. Returns **true** if **elt** was a member of the bag and **false** otherwise.

public boolean isMember (Object elt);
Returns **true** if **elt** is a member of this bag and **false** otherwise.

public void union (Collection c);
Modify this bag to contain the union of its elements with those of **c**. The union is the result of inserting each element of **c** into this bag. All comparison operations use the comparator of this bag.

public void intersection (Collection c);
Modify this bag to contain the intersection of its elements with those of **c**. The intersection is the result of keeping in this bag only those elements that are also in **c**. The number of occurrences of an element **e** in this bag after the intersection is the minimum of (1) the number of occurrences of **e** in this bag before the intersection and (2) the number of occurrences of **e** in **c**. All comparison operations use the comparator of this bag.

public void difference (Collection c);
Modify this bag to contain the difference of its elements with those of **c**. The difference is the result of deleting each element of **c** from this bag. All comparison operations will use the comparator of this bag.

Public Instance Methods Inherited from UnorderedCollection Interface:

public Object choose ();
Returns an arbitrary element of this bag. Throws a **RuntimeException** if this bag is empty. The **choose** method is a synonym for **first**.

public boolean deleteOne ();
Deletes and returns an arbitrary element of this bag. Throws a **RuntimeException** if this bag is empty. The **deleteOne** method is a synonym for **deleteFirst**.

Public Instance Methods Added by the Bag Interface:

public int count ();
Returns the number of distinct elements in this bag (i.e., ignoring duplicates).

public boolean deleteAll (Object elt);
Deletes all occurrences of **elt** in this bag. Returns **true** if this bag contained at least one occurrence of **elt** before deletion, and **false** otherwise.

public int occurrences (Object elt);
Returns the number of occurrences of **elt** in this bag. Returns 0 if **elt** is not a member of this bag.

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

CS230 Problem Set 6

Due 6pm Friday November 12

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 [40]		
Problem 2 [20]		
Problem 3 [40]		
Total		