

## Problem Set 5

Due: 11:59pm Wednesday, April 18

### Revisions:

- *April 15:* In Problem 1, for the right subtree of  $T$  to be a BST, the labels on the nodes labeled D and E need to be swapped.
- *April 19:*
  - The deadline for this problem set has been extended until **Monday, April 23**.
  - In Problem 2a and 2b, the statement `stk.push(tr)` in the constructor methods `PreOrderIterator()` and `InOrderIterator()` should be `if (!MBT.isLeaf(tr)) stk.push(tr);` (i.e., only non-empty trees should be pushed on the stack).
  - In Problem 3, the `BagEntry` class now includes a `copy()` method. This is helpful for implementing the `copy()` method of `BagMBSTEntries`, which must perform a *deep* copy of the binary search tree rather than a *shallow* copy (why?).
  - In Problem 3, the `FindEntryInfo` class now includes a `toString()` method that is helpful for debugging.
- *April 30:* (1) Renamed `PreOrderElts` and friends to `PreOrderIterator`; (2) changed `Implementating` to `Implementing`.

### Announcements:

Because of the delay in this assignment, the due date of Phase 2 of the Final Project (Program Outline) has been extended from Friday, April 20 to **1:30pm on Tuesday, April 24**. The due date of Problem Set 6 will be **6pm on Friday, April 27**. Note that PS6 is a pencil-and-paper assignment – there is no programming on PS6.

### Overview:

The purpose of this assignment is to give you practice with manipulating binary trees and implementing collections via binary 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 encourage 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 11:59pm 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 `PS5TreeOps.java` and a transcript of running `java PS5TreeOps`.
3. For Problem 2, submit your pencil-and-paper drawings from part a, your final versions of `PostOrderItertor.java` (part b) and `BreadthFirstIterator.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`.

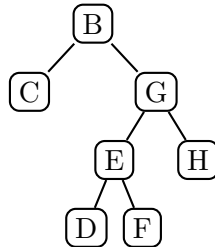
Each team should also submit a single softcopy (consisting of your final `ps5` directory) to the drop directory `~cs230/drop/ps5/username`, where `username` is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following 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 1 [30]: Tree Methods

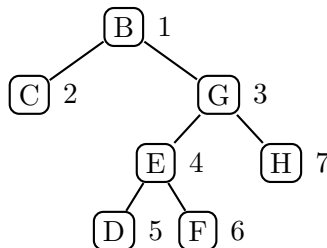
In this problem, you will define three methods that manipulate mutable binary trees. Skeletons for these methods appear in the file `ps5/PS5TreeOps.java`, which you can access after you perform a `cvcs update -d`.

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

- The *height* of a tree is the longest number of edges from its root to a leaf. The height of  $T$  is 4.
- A tree is *height-balanced* (sometimes just called *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 *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 `MBinTree<T>` methods within the class `PS5TreeOps`.

a. [10]

```
public static <T> boolean isBalanced (MBinTree<T> tr);
```

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

b. [10]

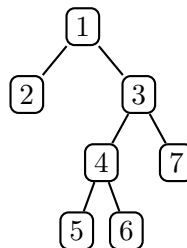
```
public static <T> boolean isBST (MbinTree<T> tr, Comparator<T> comp);
```

Returns `true` if and only if `tr` is a binary search tree relative to the comparator `comp` 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 are closely related to its definition and some of which are not.

c. [10]:

```
public static <T> MbinTree<Integer> labelPreOrderAddress (MbinTree<T> tr);
```

Returns a new tree that has the same structure as `tr` 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 `MBT.size()` method is particularly helpful here.

*Notes:*

- Flesh out the skeletons for the methods in the class `PS5TreeOps`.
- You can use the methods of the `MbinTree<T>` class (Appendix A) using the prefix `MBT.`
- Define any auxiliary methods you find helpful.
- Keep in mind that leaves of different tree types are not interoperable. For example, the leaf `MBT.<Integer>leaf()` cannot be used in a tree of type `MbinTree<String>`.
- The invocation `java PS5TreeOps method-name tree-string` will test the method named `method-name` on the tree whose string representation is `tree-string`. For example:

```
[fturbak@wampeter ps5] java PS5TreeOps labelPreOrderAddress "(( * A * ) B ( * C * ))"
labelPreOrderAddress( (( * A * ) B ( * C * )) ) =
(( * 2 * ) 1 ( * 3 * ))
```

The invocation `java PS5TreeOps method-name` will run some standard test cases for the method named `method-name`. The invocation `java PS5TreeOps` will run standard test cases for all three methods in the problem. You are encouraged to read and understand the testing code and to add any test cases you think are helpful.

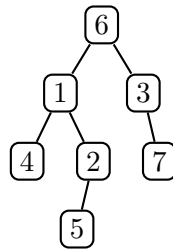
## Problem 2 [25]: Tree Iterators

In Lab #10, you studied various “orders” for traversing the elements of a binary tree: pre-order, in-order, post-order, and breadth-order. In this problem, we shall extend these traversal notions to iterators for the elements of a binary tree. It turns out that stacks and queues are very helpful for implementing tree iterators.

Fig. 1 presents the implementation of a `PreOrderIterator<T>` class that yields the elements of an `MBinTree<T>` instance using a depth-first, left-to-right *pre-order* traversal. The class has a single instance variable, `stk`, that holds a stack of non-empty trees that intuitively are “still to be processed”.

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

1. If `arg` is the string representation of a tree, the elements of the tree are displayed in pre-order. The following example uses the string `"((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))"`, which represents a tree that we shall call *A*:



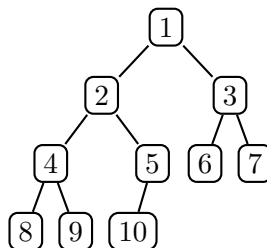
Here is the result of testing `PreOrderIterator` on *A*:

```
[fturbak@puma ps5] java PreOrderIterator "((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))"
-----
Displaying pre-order traversal of ((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))
6 1 4 2 5 3 7
```

2. If `arg` is a positive integer  $n$ , the elements of a breadth-first tree with  $n$  nodes labeled with integers 1 through  $n$  are displayed in pre-order. 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 10 nodes is:



Here is a test case based on this tree, which we shall call *B*:

```
[fturbak@puma ps5] java PreOrderIterator 10
-----
Displaying pre-order traversal of ((( * 8 * ) 4 ( * 9 * )) 2 (( * 10 * ) 5 * )) 1 (( * 6 * ) 3 ( * 7 * )))
1 2 4 8 9 5 10 3 6 7
```

```

import java.util.*; // Import Iterator

public class PreOrderIterator<T> extends SimpleIterator<T> {

    private static MBinTree MBT; // Allow MBinTree to be abbreviate MBT.

    private Stack<MBinTree<T>> stk; // A stack of non-leaf trees

    public PreOrderIterator (MBinTree<T> tr) {
        stk = new StackVector<MBinTree<T>>(); // Any stack implementation will do here.
        if (!MBT.isLeaf(tr)) stk.push(tr);
    }

    public boolean hasNext() { return !stk.isEmpty(); }

    public T next() {
        if (stk.isEmpty()) {
            throw new RuntimeException("PreOrderIterator: no next element");
        } else {
            MBinTree<T> nextTree = stk.pop();
            // Push non empty subtrees, right before left (since want to process left first)
            if (! MBT.isLeaf(MBT.right(nextTree))) stk.push(MBT.right(nextTree));
            if (! MBT.isLeaf(MBT.left(nextTree))) stk.push(MBT.left(nextTree));
            return MBT.value(nextTree);
        }
    }

    // ***** 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(MBT.breadthTree(Integer.parseInt(s))); }
        catch (NumberFormatException e1) {
            // Otherwise, try to parse s as a string tree representation
            try { testTree(MBT.fromString(s)); }
            catch (Exception e2) {
                // Otherwise treat as the name of a file,
                // in which each line is a number, tree rep, or filename
                try {
                    Iterator<String> lines = new FileLines(s, false); // do not include newlines
                    while (lines.hasNext()) {
                        testString(lines.next());
                    }
                } catch (Exception e3) { e3.printStackTrace(); }
            }
        }
    }

    public static <T> void testTree (MBinTree<T> tr) {
        System.out.println("-----");
        System.out.println("Displaying pre-order traversal of " + tr);
        IteratorOps.display(new PreOrderIterator<T>(tr), " ");
        System.out.println();
    }
}

```

Figure 1: Implementation of PreOrderIterator.

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 `trees.txt` contains the two lines

```
(((* 4 *) 1 ((* 5 *) 2 *)) 6 (* 3 (* 7 *)))
10
```

then we get the pre-order traversals of trees *A* and *B*:

```
[fturbak@puma ps5] java PreOrderIterator trees.txt
-----
Displaying pre-order traversal of (((* 4 *) 1 ((* 5 *) 2 *)) 6 (* 3 (* 7 *)))
6 1 4 2 5 3 7
-----
Displaying pre-order traversal of (((* 8 *) 4 (* 9 *)) 2 ((* 10 *) 5 *)) 1 ((* 6 *) 3 (* 7 *)))
1 2 4 8 9 5 10 3 6 7
```

Fig. 2 shows a sequence of snapshots that show the contents of the stack `stk` at the beginning of every call to `next()` when `PreOrderIterator` is tested with tree *A*. In each snapshot, a stack of trees is drawn left-to-right from the top down. The snapshot sequence also indicates where a tree value is yielded by the iterator by `next()`.

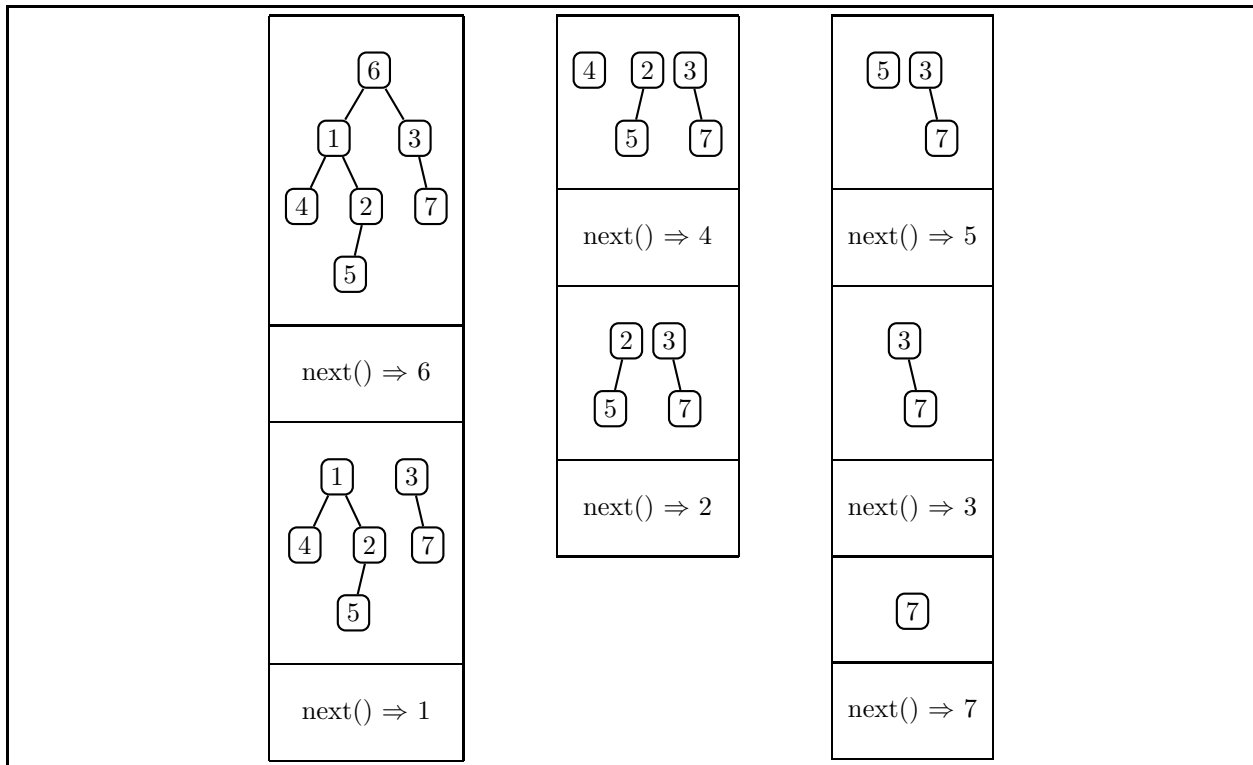


Figure 2: Snapshots of the stack during the pre-order traversal of tree *A*.

In this problem, you are asked to understand and/or implement iterators for other orders of tree traversals.

#### a. [5]: Understanding `InOrderIterator`

Fig. 3 presents the implementation of a `InOrderIterator<T>` class that yields the elements of an `MBinTree<T>` instance using a depth-first, left-to-right *in-order* traversal. Like

```

import java.util.*; // Import Iterator

public class InOrderIterator<T> extends SimpleIterator<T> {

    private static MBinTree MBT; // Allow MBinTree to be abbreviate MBT.
    private Stack<MBinTree<T>> stk; // A stack of non-leaf trees

    public InOrderIterator (MBinTree<T> tr) {
        stk = new StackVector<MBinTree<T>>(); // Any stack implementation will do here.
        if (!MBT.isLeaf(tr)) stk.push(tr);
    }

    public boolean hasNext() { return !stk.isEmpty(); }

    public T next() {
        if (stk.isEmpty()) {
            throw new RuntimeException("inOrderIterator: no next element");
        } else {
            MBinTree<T> nextTree = stk.pop();
            if (MBT.isLeaf(MBT.left(nextTree))) {
                // next is leftless; yield its value after pushing right.
                if (! MBT.isLeaf(MBT.right(nextTree))) stk.push(MBT.right(nextTree));
                return MBT.value(nextTree);
            } else {
                // next has a non-trivial left subtree;
                // push it after pushing leftless root + right subtree
                stk.push(MBT.node(MBT.<T>leaf(), MBT.value(nextTree), MBT.right(nextTree)));
                stk.push(MBT.left(nextTree));
                return next();
            }
        }
    }

    // ***** 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(MBT.breadthTree(Integer.parseInt(s))); }
        catch (NumberFormatException e1) {
            // Otherwise, try to parse s as a string tree representation
            try { testTree(MBT.fromString(s)); }
            catch (Exception e2) {
                // Otherwise treat as the name of a file,
                // in which each line is a number, tree rep, or filename
                try {
                    Iterator<String> lines = new FileLines(s, false); // do not include newlines
                    while (lines.hasNext()) {
                        testString(lines.next());
                    }
                } catch (Exception e3) { e3.printStackTrace(); }
            }
        }
    }

    public static <T> void testTree (MBinTree<T> tr) {
        System.out.println("-----");
        System.out.println("Displaying in-order traversal of " + tr);
        IteratorOps.display(new InOrderIterator<T>(tr), " ");
        System.out.println();
    }
}

```

Figure 3: Implementation of InOrderIterator<T>.

`PreOrderIterator<T>`, it uses a stack of non-empty trees to perform the traversal, but the trees that are pushed onto the stack are different.

Here are some examples of `InOrderIterator` in action on trees *A* and *B*:

```
[fturbak@puma ps5] java InOrderIterator "((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))"
```

```
-----  
Displaying in-order traversal of ((( * 4 * ) 1 (( * 5 * ) 2 * )) 6 ( * 3 ( * 7 * )))  
4 1 5 2 6 3 7
```

```
[fturbak@puma ps5] java InOrderIterator 10
```

```
-----  
Displaying in-order traversal of ((( * 8 * ) 4 ( * 9 * )) 2 (( * 10 * ) 5 * )) 1 (( * 6 * ) 3 ( * 7 * )))  
8 4 9 2 10 5 1 6 3 7
```

In this part, you will show how `InOrderIterator` works on the tree *A*. Draw a sequence of snapshots of the contents of the stack `stk` at the beginning of *every* call to `next()` when `InOrderIterator` traverses the tree *A*. Your snapshots should look similar to those in Fig. 2. As in Fig. 2, you should also indicate where a tree value is yielded by the iterator. Note that in `InOrderIterator`, `next()` is called recursively, so you should also draw the contents of `stk` at the beginning of these recursive calls. Thus, in the snapshots for `InOrderIterator`, there may be several stack snapshots before a value is yielded.

### b. [10]: `PostOrderIterator`

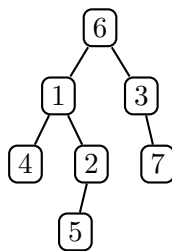
Create a copy of `InOrderIterator.java` named `PostOrderIterator.java`, and make the following changes:

- Change *all* occurrences of `InOrderIterator` to `PostOrderIterator`. (Especially don't miss the constructor method name and the constructor method invocation within `testTree()`.)
- Change the occurrence of `in-order` within the displayed string in `testTree()` to be `post-order`.
- Change the implementation of the `next()` method so that the tree elements are enumerated in *post-order* rather than *in-order*.

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

### c. [10]: `BreadthFirstIterator`

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 *A*



- 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 *A* shown above, a breadth-first traversal visits the nodes in the order 6, 1, 3, 4, 2, 7, 5. Similar to part **b**, create a copy of `InOrderIterator.java` named `BreadthFirstIterator.java` that yields elements of a tree in *breadth-first order*. In addition to changing the `next()` 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 `BreadFirstIterator` class works by turning in a transcript of your test cases.

### Problem 3 [45]: Implementing Bags Using a Mutable BST of Bag Entries

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

In this problem, you will implement a class `BagMBSTBagEntries<T>` that represents a bag with elements of type `T` using 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<T>` class:

```
public class BagEntry<T> {

    public T elt;
    public int num;

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

    public String toString () {
        return elt + ":" + num;
    }

    public BagEntry<T> copy () {
        return new BagEntry<T>(elt,num);
    }

}
```

A `BagMBSTEntries<T>` instance should also include a `Comparator<T>` used to compare elements in the binary search tree. To improve the running time of the `size()` and `countDistinct()` bag operations, the values to be returned by these methods should be cached in instance variables of the `BagMBSTEntries<T>` class.

So instances of `BagMBSTEntries<T>` should have the following four instance variables:

1. **comp**: a comparator of type `Comparator<T>` for determining the order of elements. If **comp** is `null`, this indicates that elements are assumed to implement the `Comparable<T>` interface and their `compareTo()` method should be used for comparisons.
2. **entries**: a mutable binary search tree (i.e., an instance of `MBinTree<T>`) satisfying the binary search tree property at every node, where element comparisons are determined by **comp**) whose elements are instances of `BagEntry<T>`.
3. **size**: the number of element occurrences currently in the bag (includes duplicates).
4. **count**: the number of *distinct* elements currently in the bag (does not include duplicates).

For example, Fig. 4 shows one possible representation of an instance of `BagMBSTEntries` that contains two *A*s, three *B*s and one *C*. (The shape of the tree is determined by the order in which the elements were inserted.)

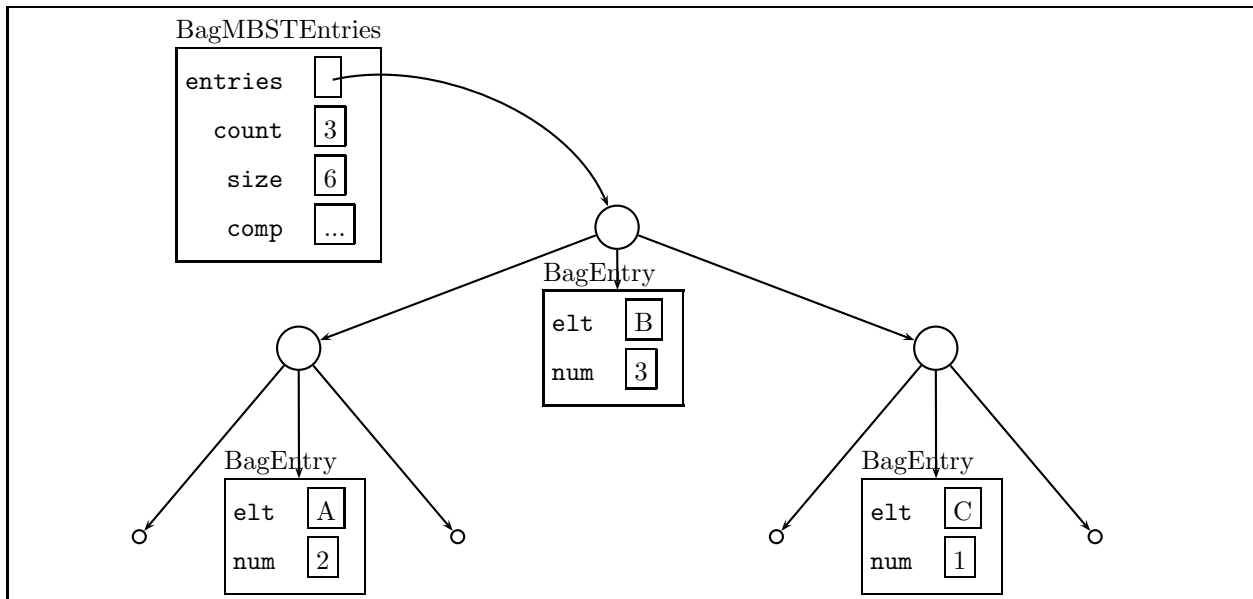


Figure 4: An example of a BagMBSTEntries instance.

To complete this problem, you need to flesh out the following methods of the bag implementation in `~/cs230/ps5/BagMBSTEntries.java` using the representation described above:

#### *Constructor Methods*

```

public BagMBSTEntries (Comparator<T> c);
public BagMBSTEntries (); // uses null for the comparator

```

#### *Instance Methods*

```

public int size ();
public boolean isEmpty ();
public void clear ();
public boolean isMember (T elt);
public boolean insert (T elt);
public boolean delete (T elt);
public T choose ();
public T deleteOne ();
public void union (Bag<T> b)
public void intersection (Bag<T> b)
public void difference (Bag<T> b)
public Bag<T> copy ()
public Iterator<T> iterator ()
public String toString ()
public int multiplicity (T elt)
public int deleteAll (T elt)
public int countDistinct ()
public Iterator<T> distinctElements ()

```

Notes:

- Mutable object trees are instances of the class `MBinTree`, whose contract is given in Appendix A. The `BagMBSTEntries` class has been configured so that the `MBinTree` operations are accessible via the `MBT.` prefix.
- You should *not* use the `MBST<T>` class presented in Lecture #19. However, you are encouraged to study the methods in `MBST.java` (which can be found in the `ps5` directory) because many of the methods you need for `BagMBSTEntries` are similar to those in `MBST.java`.
- You may define any private auxiliary methods and additional classes that you find helpful for completing this problem.
- For your convenience, the variable `leaf` has been defined as follows to simplify the specification of leaves in the `entries` BST:

```
private static MBinTree<BagEntry<T>> leaf = MBT.<BagEntry<T>>leaf();
```

- Think carefully about the `copy` method. It is necessary to create a deep copy of `entries` rather than a shallow copy. Why?
- For the iterators return by `iterator()` and `distinctElements()`, you may use anonymous inner classes or define new classes in a separate file. These iterators should *not* support the `remove()` method. You can specify this by having the iterators extend the `SimpleIterator` abstract class defined in the `ps5` directory, or by explicitly including the following method definition in your iterators:

```
public void remove() {  
    throw new UnsupportedOperationException(  
        "This iterator does not support the remove() operation.");  
}
```

Test your implementation by executing `java BagMBSTEntries`, which invokes the bag methods on various simple `BagMBSTEntries` instances. **Carefully study the output to check 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. To see what the results are supposed to look like, execute `java BagMBSTEntries` in the `ps5/test` subdirectory.

- Many methods require searching through the `entries` binary search tree to find an existing `BagEntry<T>` instance or the insertion point for a new `BagEntry<T>` instance. 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. This is similar to the `find()` method in `MBST.java` that was presented in Lecture #19.

The result of `findEntry()` is an instance of the `FindEntryInfo<T>` class (see Fig. 5) whose instance variables summarize the information needed by various other methods. (This class is provided for you in the `ps5` directory.) Here is a specification of the `findEntry()` method:

```
private FindEntryInfo<T> findEntry (T x);
```

If a `BagEntry<T>` instance for `x` exists in the `entries` tree, returns a `FindEntryInfo<T>` instance `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<T>` instance for `x` does not exist in the `entries` tree, returns a `FindEntryInfo<T>` instance `fei` where:

- `fei.entry` is null.
- `fei.child` is a 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.

```
public class FindEntryInfo<T> {

    public MBinTree<BagEntry<T>> parent, child;
    public boolean isChildToLeft;
    public BagEntry<T> entry;

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

    public String toString () {
        return "FindEntryInfo["
            + "\n parent = " + parent
            + "\n child = " + child
            + "\n isChildToLeft = " + isChildToLeft
            + "\n entry = " + entry
            + "\n]";
    }
}
```

Figure 5: The `FindEntryInfo<T>` class.

- Depending on how your binary search operations are defined, you might directly use `comp`, or you might need to “lift” `comp` via the `BagEntryComparator<T>` class presented in Fig. 6. This class is provided for you in the `ps5` directory.

```

import java.util.Comparator;

public class BagEntryComparator<T> implements Comparator<BagEntry<T>> {

    private Comparator<T> eltComp;

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

    public Comparator<T> eltComp () {
        return eltComp();
    }

    public int compare (BagEntry<T> x, BagEntry<T> y) {
        if (eltComp == null) {
            return ((Comparable<T>) x.elt).compareTo(y.elt);
        } else {
            return eltComp.compare(x.elt, y.elt);
        }
    }

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

```

Figure 6: The `BagEntryComparator<T>` class.

## Appendix A: `MBinTree<T>` Contract

The `MBinTree<T>` class models mutable binary trees whose nodes hold values of type `T`.

*Core Public Class Methods:*

**public static** `<T> MBinTree<T> leaf ()`;

Returns a leaf – i.e., an empty tree (a tree with no nodes).

**public static** `<T> MBinTree<T> node (MBinTree<T> l, Object v, MBinTree<T> r)`;

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

**public static** `<T> boolean isLeaf (MBinTree<T> tr)`;

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

**public static** `<T> MBinTree<T> left (MBinTree<T> tr)`;

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

**public static** `<T> T value (MBinTree<T> tr)`;

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

**public static** <T> MBinTree<T> right (MBinTree<T> tr);

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

**public static** <T> void setLeft (MBinTree<T> tr, MBinTree<T> newLeft);

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

**public static** <T> void setValue (MBinTree<T> tr, T newValue);

If **tr** is a node, changes its value to be **newValue**. If **tr** is a leaf, throws a `RuntimeException` indicating that a leaf has no value.

**public static** <T> void setRight (MBinTree<T> tr, MBinTree<T> newRight);

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

*Derived Public Class Methods:*

All of the following methods can be defined in terms of the core methods, but are provided for convenience.

**public static** <T> int size ();

Returns the number of nodes in this tree. This method “double-counts” nodes in shared subtrees.

**public static** <T> int height ();

Returns the height of this tree – i.e., the length of the longest path from its root to one of its leaves.

**public static** <T> MBinTree<T> copy ();

Returns a shallow copy of this tree — i.e. the copied tree nodes are new, but values themselves are shared with the nodes in the old tree. Operations on the copied tree do *not* affect the original tree and vice versa. Operations on mutable elements of the copied tree *do* affect elements of the original tree, and vice versa.

**public static** MBinTree<Integer> breadthTree (int n);

Returns a complete tree with **n** nodes whose nodes are labelled with their binary addresses. The binary address of the root node is 1, and for any node with binary address *a*, its left subtree (if not a leaf) is rooted at a node with binary address  $2a$  and its right subtree (if not a leaf) is rooted at a node with binary address  $2a + 1$ .

**public static** MBinTree<String> fromString (String s);

If **s** is the string representation of a binary tree (as specified in the `toString()` method), `fromString` returns a binary tree with string values whose string representation is **s**. Throws a `RuntimeException` if **s** is not the string representation of a binary tree.

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

Returns an iterator that yields the elements of this tree in a depth-first left-to-right pre-order traversal of its nodes.

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

Returns an iterator that yields the elements of this tree in a depth-first left-to-right in-order traversal of its nodes.

```
public static <T> Iterator<T> postOrderIterator ();
```

Returns an iterator that yields the elements of this tree in a depth-first left-to-right post-order traversal of its nodes.

```
public static <T> Iterator<T> breadthOrderIterator ();
```

Returns an iterator that yields the elements of this tree in a breadth-first left-to-right traversal of its nodes.

*Public Instance Methods:*

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

Returns a string representation of this binary 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 * ))"
```

```
public boolean equals (Object x);
```

Returns `true` if and only if  $x$  is an `MBinTree<T>` instance with the same shape as this tree and all of the corresponding nodes have values that are equal via (as determined by `equals()`). Returns `false` in all other cases.

## Appendix B: Bag<T> Interface

The `Bag<T>` interface describes mutable collections of unordered elements of type `T` that may contain multiple occurrences of each element. In mathematics, *multiset* is a synonym for *bag*.

*Public Instance Methods Like Those in Set<T>:*

```
public int size ();
```

Returns the number of elements in this bag (counting duplicates). I.e.,  $n$  occurrences of an element contributes  $n$  towards this result.

```
public boolean isEmpty ();
```

Returns `true` if this bag has no elements; `false` otherwise.

```
public void clear ();
```

Removes all elements from this bag.

```
public boolean isMember (T elt);
```

Returns `true` if `elt` is a member of this bag and `false` otherwise.

```
public boolean insert (T elt);
```

Modifies this bag by inserting a new occurrence of `elt`. Always returns `true`, because insertion is always possible with a bag (unlike a set, in which `insert` returns `false` if the set already contains `elt`.)

```
public boolean delete (T elt);
```

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

**public T choose ();**

Returns an arbitrary element of this bag. Throws a `RuntimeException` if this bag is empty.

**public T deleteOne ();**

Deletes and returns an arbitrary element of this bag. Throws a `RuntimeException` if this bag is empty.

**public void union (Bag<T> b);**

Modifies this bag to contain the union of its elements with those of `b`. The union is the result of inserting each element of `b` into this bag. The number of occurrences of an element `e` in this bag after the union is the sum of (1) the number of occurrences of `e` in this bag before the union and (2) the number of occurrences of `e` in `b`. The bag `b` is *not* modified by this method.

**public void intersection (Bag<T> b);**

Modifies this bag to contain the intersection of its elements with those of `b`. The intersection is the result of keeping in this bag only those elements that are also in `b`. 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 `b`. The bag `b` is *not* modified by this method.

**public void difference (Bag<T> b);**

Modifies this bag to contain the intersection of its elements with those of `b`. The difference is the result of deleting each element of `b` from this bag. If there are more occurrences of an element `e` in this bag than in `b`, then the number of occurrences `e` in this bag after the difference is the difference of (1) the number of occurrences of `e` in this bag before the difference and (2) the number of occurrences of `e` in `b`. If the number of occurrences of `e` in this bag is less than or equal to the number in `b`, then the number of occurrences `e` in this bag after the difference is 0. The bag `b` is *not* modified by this method.

**public Bag<T> copy ();**

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. Note: an implementation may specify a more precise return type than `Bag<T>` for this method.

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

Returns an iterator yielding all the elements of this bag (including duplicates) in an arbitrary order. I.e., if there are `n` occurrences of an element, it is yielded `n` times by this iterator.

**public String toString ();**

Returns a string indicating the implementation type and contents of this bag. Typically, the representation `elt:num` is used to indicate that there are `num` occurrences of the value `elt`.

*Public Instance Methods Specific To Bag<T>:*

**public int multiplicity (T elt);**

Returns the number of occurrences of `elt` in this bag. Returns 0 if `elt` is not a element in this bag.

**public int deleteAll (T elt);**

Deletes all occurrences of `elt` in this bag. Returns the number of occurrences deleted.

**public int countDistinct ();**

Returns the number of distinct elements in this bag. I.e.,  $n$  occurrences of an element contributes 1 towards this result.

**public Iterator<T> distinctElements ();**

Returns an iterator that yields the distinct elements in this bag in an arbitrary order. I.e., if there are  $n$  occurrences of an element, it is yielded only once by this iterator.

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

## CS230 Problem Set 5

Due 11:59pm Wednesday April 18

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 1a [10]		
Problem 1b [10]		
Problem 1c [10]		
Problem 2 [25]		
Problem 3 [45]		
<b>Total</b>		