CSE 373, Winter 2013
Homework Assignment #6: The Rotating Dead (25 points)
Due Friday, March 1, 2013, 11:30 PM

This program focuses on implementation of a map using an AVL balanced binary search tree as its internal data structure. Turn in files named `HvZPlayer.java`, `TreeMap.java`, and `TreeSet.java` from the Homework section of the course web page.

You will also need to download the support .java files and input .txt files from the Homework section of the course web site; place them in your Java project and add to it. If you are using Eclipse, place .java files in your project's `src/` folder and .txt files in the project root folder (the parent of `src/`). We do not guarantee that the provided tests are exhaustive; perform additional testing of your own before you submit your work.

Part A (HvZPlayer) Description:

For the first task of this assignment, you'll make a change to the `HvZPlayer` class you implemented in Homework 4. We want to change our `HvZSimulation` program to always show the players in sorted order. To achieve this, we will store `HvZPlayer`s in `TreeSet`s and `TreeMap`s rather than hash-based collections. In later parts of this assignment you will implement those tree-based collections.

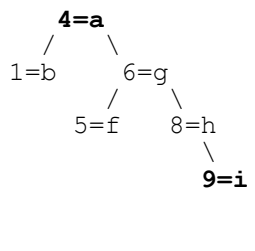
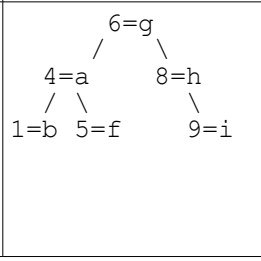
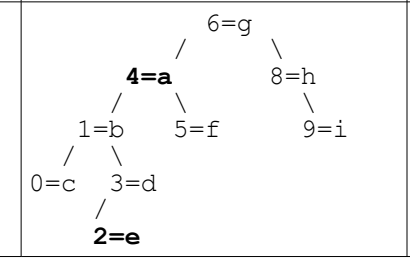
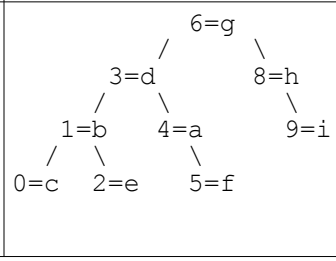
Copy over your HW4 `HvZPlayer.java` into your project for this assignment, then modify it to **implement the `Comparable` interface**. Your `compareTo` method should order players by name in alphabetical order. If two players have the same name, break ties by placing humans earlier in the ordering than zombies. If the players have the same name and human/zombie status, break ties by number of kills, where a player with fewer kills comes earlier in the ordering than one with more kills.

Part B (TreeMap) Description:

In Part B of this assignment, you will write a `TreeMap` class that implements a provided `Map` interface. We provide a class `AbstractTreeMap` that implements an unbalanced normal binary search tree. You should extend the abstract class and override its methods for adding (`put`) and removing (`remove`) nodes so that the tree rebalances itself after each operation.

Your `TreeMap` should extend the `AbstractTreeMap` and turn it into a proper AVL tree that maintains its balance as discussed in class by performing rotations. After any add (put) or removal (`remove`) operation, if the balance factor (the difference between the right and left subtree heights) is greater than 1 or less than -1, rotate the tree appropriately to restore balance.

The diagrams below show a few tree states that require rotations. The first shows the pair 9=i added to a tree, causing the root to become imbalanced. The balance problem corresponds to "case 4" shown in class, so balance is restored by doing a left rotation on the root node 4=a. The third diagram shows a later state of the same tree after the key/value pair 2=e is added, causing the node 4=a to become imbalanced. The balance problem corresponds to "case 2" shown in class, so balance is restored by doing a left-right double rotation, first rotating the 1=b node left, then rotating the 4=a node right.

case 4	after L rotation	case 2	after LR rotation
			

Part B (TreeMap) Implementation Details:

Write a class `TreeMap<K, V>` that implements our `Map<K, V>` interface. The superclass `AbstractTreeMap` already implements almost all of the public methods that the client will call on the map, such as `put`, `get`, `containsKey`, `remove`, `size`, etc. The superclass's code for those methods is correct, but it allows the tree to become unbalanced.

Recall that many tree methods use a public / private pair where the client's public method invokes a private helper that accepts a `TreeNode` parameter. For this assignment the helpers in the abstract class are declared as `protected`, which means that they are mostly private but are visible to your subclass. In your `TreeMap` class, you will replace some of the superclass behavior with your own version by overriding methods. In your overridden method code, you still want to execute the old code in addition to your new code, so you can call the superclass's version using the `super` keyword as shown below:

```
protected TreeNode foo(TreeNode node, int a, String b) {
    TreeNode result = super.foo(node, a, b);   // call superclass version of foo
    <your extra code>

    return result;
}
```

You must write the following methods. You may define additional `private` methods as needed. All methods must run in O(log *N*) average runtime unless otherwise specified. You should not add any fields to the class.

`public` **`TreeMap`**`()`

In this constructor you will initialize a newly constructed empty map. O(1).

`protected TreeNode` **`put`**`(TreeNode node, K key, V value)`

In this method you should add the given key/value pairing to the subtree rooted at the given node of your map, returning the node's new state afterward. If any previous key/value pair exists for that same key, it should be replaced by this new one. The superclass already handles all of this behavior, but your version should ensure that the tree is balanced afterward by performing any necessary rotations.

`protected TreeNode` **`remove`**`(TreeNode node, K key)`

In this method you should remove any existing key/value pair that includes the given key, from the subtree rooted at the given node of your map, returning the node's new state afterward. The superclass already handles all of this behavior, but your version should ensure that the tree is balanced afterward by performing any necessary rotations.

`public String` **`toString`**`()`

In this method you should return a string representation of your map's elements. The string is surrounded by curly braces and contains each key/value pair connected by an equals sign and separated by commas. The pairs should be listed in ascending key order, as they would be returned by an in-order traversal of the tree. For example, for the map in the first picture on the previous page, you would return `"{1=b, 4=a, 5=f, 6=g, 8=h, 9=i}"`. For an empty map you should return `"{}"` and for a one-pair map you should return the one pair in braces such as `"{a=b}"`. This method is O(*N*). Implement this method by traversing the tree a single time and do not call the `keySet` method as a helper.

You may want to look over the code for the `AbstractTreeMap` superclass, because it has some useful methods that you can call in your `TreeMap` code. For example, the method `computeHeight` takes a `TreeNode` parameter and returns an integer representing what its current height should be set to. If you make any change to a node that affects the height of its subtree, you should use this method to update its height field, such as the following:

```
node.height = computeHeight(node);
```

Part C (TreeSet) Implementation Details:

The third and final part of this assignment is to write a class `TreeSet` that implements a provided `Set` interface. It may seem odd for this to be the last part, because a set seems like it would be easier to implement a set than a map. The reason we write the set second is because the code for a tree set can leverage the code you just wrote for the tree map.

A set can be implemented by declaring an internal map from keys to `boolean` values as a field. (Don't extend the `TreeMap` class; store a `TreeMap` as a field inside a `TreeSet`.) When the client wants to add a value to the set, your set can just put a key/value pair into the map, where the key is the element the client wants to add to the set, and the value is `true`. All calls from the client are forwarded to this inner map. In this way you will be leveraging your implementation of one collection to implement another. If you are writing this class the way we intend, all of your methods' bodies should be very short, just 1-2 lines.

You must write the following methods. You may define additional `private` methods as needed. All methods must run in O(log *N*) average runtime unless otherwise specified.

`public` **`TreeSet`**`()`

In this constructor you will initialize a newly constructed empty set. O(1).

`public void` **`add`**`(E element)`

In this method you should add the given element to your set. If the element is already present in the set, no change should occur. You should throw a `NullPointerException` if the element is `null` (the tree map should already handle this for you).

`public void` **`clear`**`()`

In this method you should remove all elements from your set. O(1).

`public boolean` **`contains`**`(E element)`

In this method you should return `true` if the given element is part of your set, and `false` if not. You should throw a `NullPointerException` if the element is `null` (the tree map should already handle this for you).

`public boolean` **`isEmpty`**`()`

In this method you should return `true` if the set does not contain any elements. O(1).

`public Iterator<E>` **`iterator`**`()`

In this method you should return an iterator for traversing the elements of your set. It is fine to use the `keySet` of the internal map as a helper here. You do not need to write an inner iterator class. O(*N*).

`public void` **`remove`**`(E element)`

In this method you should remove the given element from your set, if it was present. You should throw a `NullPointerException` if the element is `null` (the tree map should already handle this for you).

`public int` **`size`**`()`

In this method you should return the number of elements contained in your set. O(1).

`public String` **`toString`**`()`

In this method you should return a string representation of your set's elements. The string is surrounded by square brackets and contains each element separated by commas. The elements should appear in their natural sorted ordering, such as `"[4, 17, 63, 99]"`. For an empty set you should return `"[]"` and for a one-element set you should return the one element in brackets such as `"[hi]"`. This method is O(*N*). It is fine to use the `keySet` of the internal map as a helper here.

When you finish Parts B and C, you can go run the provided `HvZMain` program (similar to the one used in the previous HashMaps vs. Zombies assignment) with your own map/set classes. This provides a good test of your map and set code.

Development Strategy and Hints:

- Look at the AVL tree code written in lecture as an example. You can borrow from the example code, especially the code to help with rotations. The lecture slides outline the various cases for rotation, how to identify each case, and what rotation should be used to fix it. The textbook is also a helpful resource about AVL rotations.
- The superclass includes a method called `print` that you can call to see the state of the tree. This may be helpful for debugging. The jGRASP debugger is also very useful for visually stepping through your tree code and seeing the state of the map after each operation.
- Many bugs come from not properly following the "x = change(x);" pattern described in lecture. When a method accepts a node's state as a parameter, if the method potentially changes that node's state, the method should return the new state. Similarly, any code that calls the method should reassign whatever variable it passes in to the method. Write "x = change(x);", not just "change(x);"
- You may find it helpful to use a `StringBuilder` in your `toString` method. A `StringBuilder` is a mutable buffer for building up a string, and it can be passed as a parameter and modified on each call, even without returning it. This is easier and more efficient than passing and returning various partial strings.

Style Guidelines and Grading:

We will grade your program's behavior and output to give you an **External Correctness** score. You can use the course web site's Output Comparison Tool to check your HvZT simulation and check test case output for your tree map/set.

We will also grade your code quality (**Internal Correctness**). There are many general Java coding styles that you should follow, such as naming, indentation, avoiding redundancy, etc. For a list of these, please see the **Style Guide** document posted on the class web site. You should follow all of its guidelines as appropriate on this and future assignments.

In past assignments we have used collections to solve problems. But in this assignment, you are implementing two collections of your own. **For this reason, you may not use any of Java's built-in data structures (such as the ones from `java.util`) to implement your `TreeMap` or `TreeSet`.** For example, declaring a field of type `ArrayList` or `HashMap` would be unacceptable. A solution that disobeys this restriction will receive a substantial deduction.

Do not declare a value as a private field unless it is necessary to retain it as part of the state of your object. Your tree map should not need any fields, and your tree set should need only one: its inner map.

You must also match the expected **Big-Oh** demands of each method. Relaxing the Big-Oh might make it easier to implement the functionality; but the whole point is implement the map efficiently. So this will be a grading focus. In general, a method should not need to walk down the same part of the tree more than once.

Watch out for **redundant code**. If you are performing identical or very similar commands repeatedly, factor out the common code and logic into a helper method, loop, or other facility to remove the redundancy. In particular, the code for rebalancing a given part of the tree should not be repeated multiple times in your code.

We strongly recommend that you write some **private helper methods** to help you implement the required functionality of `TreeMap`. A helper is necessary to implement behavior that traverses the tree. Our own sample solution has five helpers.

In your method comment headers make sure to comment what exceptions (if any) are thrown by each method and under what conditions they are thrown. In addition to including comment headers on each class and each method, also make sure to include inline comments next to any complex or tricky code, briefly explaining the purpose of that code.