CSE 373, Winter 2011
Programming Project #2: Binary Search Trees and Balanced Trees (60 points)
Step 0: Due Wednesday, February 2, 2011, Beginning of Class
Steps 1 - 3: Due Friday, February 4, 2011, 10:00 PM

This assignment focuses on implementing a Set ADT using binary search trees and AVL trees, one type of balanced binary search trees (BSTs). On Wednesday, February 2, turn in Step 0 (the warm up exercise) at the beginning of class. On Friday, February 4, turn in Steps 1 - 3 electronically. This turnin should consist of four files: `StringTreeSet.java`, `StringAVLTreeSet.java`, `StringSetClient.java`, and `README.txt` (this will contain your answers to the assignment questions as described below). Your Set implementations should implement the `StringSet.java` interface. You will also need the additional supporting files: `StringTreeNode.java`, `StringTreeSet.java`, and `StringAVLTreeSet.java`. All of these files can be found on the course website.

For this assignment you are to add new functionality and modify some existing functionality to the `StringTreeSet` and `StringAVLTreeSet` classes written and discussed in lecture.

Step 0: Warm-up

Draw the AVL tree constructed by successive inserts of 2 1 4 5 9 3 6 7. Say which inserts trigger single (outside) rotations and which inserts trigger double (inside) rotations.

Step 1: Modify `StringTreeSet` to maintain height information in `StringTreeNode`

The goal of this part of the assignment is to augment the `StringTreeSet` introduced in class to maintain height information for each `StringTreeNode` in the `StringTreeSet`. In order to do so, you will be modifying a few existing methods as well as adding a few new methods to the given `StringTreeSet`.

The `StringTreeSet` class already has the following constructors and methods (along with a few others):

`public StringTreeSet()`	constructs an empty tree set
`public boolean add(String value)`	adds a value to the `String` set
`public boolean contains(String value)`	returns `true` if the set contains the specified `String`
`public boolean remove(String value)`	removes the specified `String` from this set if it is present
`public int size()`	returns the number of elements in the set.
`public String toString()`	returns a String representation of `StringTreeSet` with elements in their "natural order"

The version of `StringTreeNode` that you are receiving for this assignment has an additional field, height. This field should be correctly maintained for each `StringTreeNode` in the `StringTreeSet`. The height of a tree is:

- -1 if it is empty,
- 0 if it has only one node, and
- the height of its tallest child plus 1 if there is more than one node.

One reason we want to store the height of a tree node is that we will also want to compute a tree node's balance factor. The balance factor for a tree node, *n*, is the height of *n*'s right subtree minus the height of *n*'s left subtree.

Add the following methods to `StringTreeSet`:

- `protected static int computeHeight(StringTreeNode node)`
  This method returns the height of the given BST according to the definition of height outlined above. This method should be implemented to run in constant time (i.e. use the nodes' height fields rather than actually traversing the tree).

- `protected static int balanceFactor(StringTreeNode node)`
  This method returns the balance factor the given node. This method should be implemented to run in constant time (i.e. use the nodes' height fields rather than actually traversing the tree).
- `public boolean isBalanced()`
  This method returns `true` if the height information in the `StringTreeSet` is correctly maintained and the balance factor of each node is either -1, 0, or 1. In other words, this method returns `true` if the `StringTreeSet` complies with the AVL balance property and `false` otherwise. Your implementation should run in O(n) time and not traverse any unnecessary parts of the tree (i.e. once you know a section of the tree is unbalanced, there is no reason to check branches that have not yet been traversed).

Modify the recursive `add` helper and the recursive `remove` helper to correctly fill in and maintain the `height` field of all `StringTreeNode`s in the Set when a value is added or removed. Your changes should only affect the runtime of the `add` and `remove` operations by a constant (i.e. the runtime of these methods should still be O(log n) after your changes). Additionally, you should modify the `toString` method to print out height information. For example, if "a", "b", and "c" were inserted into a `StringTreeSet` in order, the `toString` method should return the `String "[(a - 2), (b - 1), (c - 0)]"`. In other words, when creating the `String` to represent a node, it should be comprised of an open parenthesis, the value in the node, a space, a dash, another space, the height of the subtree rooted at the node, and a closing parenthesis.

Step 2: Complete `StringAVLTreeSet`

The goal of this part of the assignment is to complete the `StringAVLTreeSet` introduced in class. In order to do so, you will be modifying a few existing methods as well as adding a few new methods to the given `StringAVLTreeSet`.

An AVL tree is a binary search tree with an additional property: it maintains a balance factor of 0, 1, or -1 for each node. When insertions/removals are done on an AVL tree, the AVL property could be violated. For a node, *n*, the AVL property may be violated due to one of four cases:

1. LL Case: An insertion into / removal from the left subtree of the left child of *n*.
2. LR Case: An insertion into / removal from the right subtree of the left child of *n*.
3. RL Case: An insertion into / removal from the left subtree of the right child of *n*.
4. RR Case: An insertion into / removal from the right subtree of the right child of *n*.

The `StringAVLTreeSet` class inherits from `StringTreeSet`. `StringAVLTreeSet` already has the following constructors and methods that start to maintain the AVL property:

`protected StringTreeNode add(StringTreeNode node, String value)`	overrides `StringTreeSet`'s recursive `add` helper to rebalance the BST rooted at the given node when the AVL property is violated
`protected StringTreeNode rebalance( StringTreeNode node)`	if the BST rooted at the given node does not maintain the AVL tree balance condition, restores the AVL tree property for this BST through rotations
`private StringTreeNode rightRotate(StringTreeNode parent)`	right rotates the parent, the parent's left child, and the parent's left child's right subtree in order to fix the LL Case

Add the following methods to `StringAVLTreeSet`:

- `private StringTreeNode leftRotate(StringTreeNode parent)`
  This method should left rotate the parent, the parent's right child, and the parent's right child's left subtree in order to fix the RR Case.
- `private StringTreeNode leftRightRotate(StringTreeNode parent)`
  Since rebalancing the AVL tree has the potential to happen quite frequently, we would like the LR case to be fixed efficiently. Instead of two method calls to the single rotation methods to double rotate, this method contains the necessary statements needed to do a left/right double rotation to fix of the LR Case. There should be no unnecessary variables used, no redundant assignments to reassign left and right subtrees, and no unnecessary calculations performed. In other words, copying and pasting the bodies of the single rotate methods one after another into this method will not earn credit. You have to figure out what the necessary actions for a LR rotate and perform only those actions.
- `private StringTreeNode rightLeftRotate(StringTreeNode parent)`
  Since rebalancing the AVL tree has the potential to happen quite frequently, we would like the RL case to be fixed efficiently. Instead of two method calls to the single rotation methods to double rotate, this method contains the necessary statements needed to do a right/left double rotation to fix of the RL Case. There should be no unnecessary variables used, no redundant assignments to reassign left and right subtrees, and no unnecessary calculations performed. In other words, copying and pasting the bodies of the single rotate methods one after another into this method will not earn credit. You have to figure out what the necessary actions are for a RL rotate and perform only those actions.
- `protected StringTreeNode remove(StringTreeNode node, String value)`
  This method implements a (nonlazy) deletion in our AVL tree. This method first performs a normal BST remove. If you modified the remove method of StringTreeSet correctly, that method will update the heights properly. Next this method checks the balance factors of the nodes from the removed node's parent to the root to ensure they still maintain the AVL balance condition. If a node does not maintain this condition, it should be rebalanced. After rebalancing, this method must continue up the branch to the root checking for other imbalances. Your method should run in O(log N) time.

Modify the `rebalance` method to also account for the symmetric cases (i.e. the RL Case and the RR case).

Step 3: Write-Up

In addition to the code that you turn in, answer the following questions in a file called `README.txt`.

1. Explain why your `isBalanced` method runs in O(n) time.
2. Do you expect the `leftRightRotate` and `rightLeftRotate` methods to save you much time over doing two single rotations? If so, why? If not, why not?
3. Explain why your remove method in the StringAVLTreeSet runs in O(log n) time.

Development and Testing Strategy

We are not providing you with any testing code. You are solely responsible for testing your `StringSets`. All of your testing code should go in the `StringSetClient.java` file that you turn in. You will be graded on how well you have tested your code. For example, your should test the four different cases for inserts and removals.

1. Do the warm-up exercise to make sure you understand the material.
2. Modifying `StringTreeSet` to maintain height information in `StringTreeNode`.
   a. Write the `computeHeight` and `balanceFactor` methods.

   b. Use these methods to modify the recursive `add` and `remove` methods to maintain height information.
   c. Implement the modified `toString` method.
   d. In your client program, test different insert and removal cases to ensure that your heights are maintained properly. Use `toString` to examine the results of the insertions and removals.
   e. Complete the `isBalanced` method.
   f. Add tests to your client program for different BSTs that are and are not balanced.
2. Completing `StringAVLTreeSet`
   a. Write the `leftRotate` method.
   b. Modify the `rebalance` method using `leftRotate` and `rightRotate`.
   c. In your client program, test that your insertions are working for the different cases. Use the `toString` method and the `isBalanced` method to verify your results.
   d. Write the `leftRightRotate` and `rightLeftRotate` methods. Modify the `rebalance` method using `leftRotate`, `rightRotate`, `leftRightRotate` and `rightLeftRotate`.
   e. Does your client program still work?
   f. Write the `remove` method.
   g. Write more tests in your client program to ensure your `remove` works. One idea to test `remove` is to make a list of `String`s. For each of these `String`s in the list, insert them into a `StringAVLTreeSet`. After each insert check to see if your tree is still balanced and the result of the `toString` is what you are expecting. Then, have a similar for loop that goes through your list and removes each element from your `StringAVLTreeSet`. After each insert check to see if your tree is still balanced and the result of the `toString` is what you are expecting.
3. Complete the writeup.

Style Guidelines and Grading

Part of your grade will come from appropriately utilizing a tree data structure to implement your `StringTreeSet.java` and `StringAVLTreeSet.java` classes. There will be significant deductions if you use Java collections to implement these classes. Redundancy is another major grading focus; some methods are similar in behavior or based off of each other's behavior. You should avoid repeated logic as much as possible. Your class may have other methods besides those specified, but any other methods you add should be `private`.

You should follow good general style guidelines such as: making fields `private` and avoiding unnecessary fields; appropriately using control structures like loops and `if`/`else`; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, and exceptions. The files provided to you use "doc comments" format used by Javadoc, but you do not have to do this. For reference, we added approximately 90 lines of code to the `StringTreeSet.java` and `StringAVLTreeSet.java` classes to complete the assignment.