CSE 373, Spring 2012
Homework #4: Good Things Come in Trees (60 points)

Due Wednesday, May 2, 2012, 2:30 PM

This assignment focuses on implementing a Set ADT using binary search trees (BSTs) and AVL trees. Turn in all parts of this assignment electronically. Your turnin should consist of four files: `TrackingStreeSet.java`, `AVLStreeSet.java`, `StringSetClient.java`, and `README.txt` (this will contain your answers to the questions found in Step 0 and 3 as described below). You will also need the additional supporting files: `StringSet.java`, `StreeNode.java`, `StreeSet.java`, `StringSetClient.java`, `AuthorChallenge.java`, the two word lists, and the six book files. All of these files can be found on the course website.

In lecture, we created a Set for `String`s backed by a BST called `StringTreeSet`. In this assignment, we have modified the name of this file and class to be `StreeSet` (String Tree Set) for brevity, but it is the same code we discussed and developed together in lecture. In this assignment you will extend the functionality of our `String` Set to keep track of the height of each node and maintain the AVL balance property for insertions and deletions.

Step 0: Warm-up

For the coding portions of this assignment, we encourage you to draw lots of examples of AVL trees to see how they work for insertions and deletions. For this step, draw the AVL tree constructed by successive inserts of the following values:

```
2  1  4  5  9  3  6  7
```

You will not turn in your drawing, but give your answer to this step by saving the answers to the following questions in your `README.txt`.

1. Give the preorder traversal of your final tree.
2. Which values when inserted caused a single (outside) rotation?
3. Which values when inserted caused a double (inside) rotation?

Step 1: `TrackingStreeSet` to maintain height information

In this part of the assignment you will create a new class called `TrackingStreeSet` in a file named `Tracking-StreeSet.java`. This class is a `String` Set backed by a BST that maintains height information for each node. `TrackingStreeSet` should extend (i.e. be a subclass of) the provided `StreeSet`.

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

`public StreeSet()`	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 `StreeSet` with elements in their "natural order"

The version of `StreeNode` that you are receiving for this assignment has an additional field: height. This field should be correctly maintained for each `StreeNode` in the `TrackingStreeSet`. Recall that 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 to store the height of a tree node is 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.

Implement the following methods in `TrackingStreeSet`:

- `protected static int computeHeight(StreeNode 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 StreeNode add(StreeNode node, String value)`
  This method overrides the recursive helper `add` method found in the provided `StreeSet`. In this method, you should add a value to the BST just as in the regular `StreeSet` and additionally maintain the heights of nodes in the tree. Do *not* rewrite the `add` code found in the `StreeSet` class. `TrackingStreeSet`'s `add` should work exactly like `StreeSet`'s `add` except that after a node is added there is additional functionality to maintain heights of nodes. Therefore, your changes should only affect the runtime of `add` by a constant (i.e. the runtime of this method should be O(log *n*)). The Java keyword `super` is helpful here.
- `protected StreeNode remove(StreeNode node, String value)`
  This method overrides the recursive helper `remove` method found in the provided `StreeSet`. In this method, you should remove a value from the BST just as in the regular `StreeSet` and additionally maintain the heights of nodes in the tree. Do *not* rewrite the `remove` code found in the `StreeSet` class. `TrackingStreeSet`'s `remove` should work exactly like `StreeSet`'s `remove` except that after a node is added there is additional functionality to maintain heights of nodes. Therefore, your changes should only affect the runtime of `remove` by a constant (i.e. the runtime of this method should be O(log *n*)). The Java keyword `super` is helpful here.
- `protected String toString(StreeNode node)`
  This method overrides the recursive helper `toString` method found in the provided `StreeSet`. In this method you should return a `String` representation of the BST that displays height information. For example, if "a", "b", and "c" were inserted into a `TrackingStreeSet` in that order, the `toString` method should return the `String "[(a - 2), (b - 1), (c - 0)]"`. The `String` returned should be an in-order representation of the BST—start with a `[`, end with a `]`, and have a comma and a space separating each node. A portion of the `String` that represents a node should be composed 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.
- `protected static int balanceFactor(StreeNode node)`
  This method returns the balance factor of 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 following two conditions hold true for every node in the BST: (1) the height of the node is correct, and (2) the balance factor of each node is either -1, 0, or 1. In other words, this method returns `true` if the `TrackingStreeSet` 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).

Step 2: `AVLStreeSet` to maintain balance property

In this part of the assignment you will create a new class called `AVLStreeSet` in a file named `AVLStreeSet.java`. This class is a `String` Set backed by an AVL Tree that maintains the AVL balance property. `AVLStreeSet` should extend (i.e. be a subclass of) your `TrackingStreeSet`.

An AVL Tree is a BST 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 six 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*.
5. -20 Case: A removal from the right subtree has occurred causing *n* to have a balance factor of -2 and the left child of *n* has a balance factor of 0.
6. 20 Case: A removal from the left subtree has occurred causing *n* to have a balance factor of 2 and the right child of *n* has a balance factor of 0.

Implement the following methods in `AVLStreeSet`:

- `private StreeNode rightRotate(StreeNode parent)`
  This method should right rotate in order to fix the LL Case. (*Hint*: This code was given and explained in lecture. Just add it to your `AVLStreeSet` file.)
- `private StreeNode leftRotate(StreeNode 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 StreeNode rebalance(StreeNode node)`
  If the AVL Tree rooted at the given node is in violation of the AVL Tree balance condition, this method should restore the AVL tree property for given node through rotations. (*Hint*: An outline of this method was given and explained in lecture. Start with what we gave in lecture, modify it and add code to take care of all of the different cases by which a node can become imbalanced.)
- `protected StreeNode add(StreeNode node, String value)`
  This method overrides the recursive helper `add` method found in your `TrackingStreeSet`. In this method you should add a value to the AVL Tree rooted at the given node just as in the `TrackingStreeSet` and rebalance the given node if the balance property is violated as a result of the addition. Do *not* rewrite the `add` code found in your `TrackingStreeSet`. `AVLStreeSet`'s `add` should work like `TrackingStreeSet`'s `add` except that after a value is added, the node may be rebalanced to maintain the AVL Tree balance property. Your changes should only affect the runtime of `add` by a constant (i.e. the runtime of this method should be O(log *n*)). The Java keyword `super` is helpful here.
- `protected StreeNode remove(StreeNode node, String value)`
  This method overrides the recursive helper `remove` method found in your `TrackingStreeSet`. In this method you should remove the value from the AVL Tree rooted at the given node just as in the `TrackingStreeSet` and rebalance the given node if the balance property is violated as a result of the removal. Do *not* rewrite the `add` code found in your `TrackingStreeSet`. `AVLStreeSet`'s `remove` should work like `TrackingStreeSet`'s `remove` except that after a value is removed, the node may be rebalanced to maintain the AVL Tree balance property. Your changes should only affect the runtime of `add` by a constant (i.e. the runtime of this method should be O(log *n*)). The Java keyword `super` is helpful here.

Step 3: Author Challenge!

There are many literary works that are considered to be of great importance. Amongst these are Cervante's *Don Quixote* (Spain), Shakespeare's *Hamlet* (England), Joyce's *Ulysses* (Ireland), and Tolstoy's *War and Peace* (Russia). In this part of the assignment, you will use our `StreeSet` and your `AVLStreeSet` to determine which of these great works has the most diverse vocabulary using the provided `AuthorChallenge.java` code provided.

`AuthorChallenge` prompts the user if they would like to compare the books against a small list of words or a large list of words and also for a book file (`donquixote.txt`, `hamlet.txt`, `ulysses.txt`, and `warandpeace.txt` have all been provided off of the course homework webpage). `AuthorChallenge` then builds two sets: one using the provided `StreeSet` and another using your `AVLStreeSet`. `AuthorChallenge` determines the percentage of the words used by the book by searching for every word in the chosen word list in both sets. `AuthorChallenge` reports the time it takes to build each set and the time it took to search for all of the words.

Here's an example log of execution:

```
Use (S)mall or (L)arge dictionary? l
Enter book file: ulysses.txt
BST Set build time: 190ms
AVL Set build time: 341ms
BST Set search time: 58ms (19.16% of the words are used)
AVL Set search time: 45ms (19.16% of the words are used)
```

Using `AuthorChallenge`, the provided `StreeSet`, and your `AVLStreeSet`, answer the following questions and save them in your `README.txt`.

1. Run author challenge using the large word list and each of the four book files. Save each log of execution to your `README.txt` as the answer to this question. You might want to run `AuthorChallenge` several times for each set of inputs for a representative log of execution of running times.
2. Which author uses the most diverse vocabulary? In other words, which author has the highest percentage of words used?
3. Explain why your `isBalanced` method runs in O(n) time.
4. Which Set implementation takes longer to build? Why do you think this is the case?
5. Which Set implementation takes longer to search? Why do you think this is the case?

Development and Testing Strategy

We are providing you with a `StringSetClient.java` file that contains one method that tests an AVL Tree removal. We will not be providing you with any other test cases. You should add testing methods like the one provided to this file and turn it in. You will be graded on how well you have tested your code. For example, for `TrackingStreeSet` you should test for trees that are both balanced and imbalanced, and for your `AVLStreeSet` you should test the all the different cases for AVL Tree inserts and removals.

We suggest that you do this assignment in the same order the steps and methods have been provided. After writing each method, we suggest testing that method before moving on to writing other methods. It is easier to find bugs after only a small amount of code has been written rather than waiting until you have entire classes written. In the case of the `rebalance` method, you may want to write a single rebalance case and test that single case before moving onto writing the other rebalance cases.

Before you run `AuthorChallenge` on the large book files, you might want to try to run it on `ulysses_frag.txt` and `warandpeace_frag.txt` to ensure that you are getting the right percentage of words used. It is almost impossible to know if your sets are working if you use the large word file and full book files right away.

Style Guidelines and Grading

A large part of your grade will come from appropriately utilizing a tree data structure to implement your `StreeSet.java` and `AVLStreeSet.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, our solution for `TrackingStreeSet.java` is 51 lines long (32 lines if you ignore blank, closing braces, and commented lines) and our solution for `AVLStreeSet.java` is 95 lines long (53 lines if you ignore blank, closing braces, and commented lines).