Computer Science II — Homework 8 — cs2set and BST's continued

Preliminaries

This assignment is due Friday April 11 at 11:59:59pm. It is worth 100 points toward your homework grade.

Please adhere to the following submission instructions: Place all of your source code files in a folder named hw8 (no spaces, no uppercase, not `hw8 submit`, not `H8`). Make sure none of your file names include spaces. Do not include any extra files. Zip the folder (not just the files in the folder), so that when it unzips a new folder is created, and submit it through the course web page.

Overview

This homework continues our exploration of our `cs2set<T>` class and binary search trees. It is closely linked to material from Lab 10, which is already posted on-line, including all three checkpoints. You are encouraged to complete the lab checkpoints and then continue with this assignment. If you wait until lab to start this assignment, you may struggle.

Your job in this assignment is to implement and test a number of additional member `cs2set<T>` functions, some that mimic functions of `std::set<T>` and others that are specific to the underlying tree implementation. In grading we will attempt to test each function separately (as best we can) so that if you do not successfully complete all parts of the homework you will still earn substantial credit.

The list of member functions to implement is below. Specifics on the interface and are in the version of `cs2set.h` posted at

`http://www.cs.rpi.edu/academics/courses/spring08/cs2/hw8/cs2set.h`

This version includes solutions to all exercises from Lectures 16 and 17, but not the solutions to the lab checkpoints. Please add your solution to the `copy_tree` function from lab; it is the only one necessary. In implementing the new functions required here, please feel free to write additional **private** member functions to help in the solution. Do not change the public interface.

Here is the function list:

- `height`: return the length of the longest path from the root to a leaf. If the tree is empty the height is -1.
- `average_depth`: The depth of a node is the distance from the root of the tree (depth = 0) to the node. The average depth is the average over all nodes in the tree. Unbalanced trees have larger average depth values. If the tree is empty, the average depth is -1.
- `rebalance`: Restructure the tree so that it is balanced. The effect of rebalancing is defined as follows: At each node of the tree, starting from the root position, if there are n nodes to be stored in the tree (the subtree) rooted at the node, including the node itself, then there must be $\lfloor n/2 \rfloor$ in stored in the left subtree and $\lfloor (n-1)/2 \rfloor$ stored in the right subtree. (Note that this definition is recursive!) For example if $n = 13$, then there should be 6 nodes in left subtree, 6 in the right subtree and 1 at the root of the (sub)tree. On the other hand, if $n = 12$ then there must be 6 nodes in

the left subtree, 5 in the right subtree and 1 at the root of the (sub)tree. This must be done efficiently **without creating any new tree nodes**.

- `is_balanced`: Is the tree balanced? The definition here is that at each node of the tree, if there are k nodes in the left subtree then the number of nodes in the right subtree must be $k-1$, k or $k+1$. After rebalancing a tree this property will hold, but other tree structures satisfy the balance property. This definition is stronger than any of the usual definitions of a balanced tree, which you will consider if you take Data Structures and Algorithms.

- `level_order`: Print all the values stored in the set in "level-order", one level per line. In other words, the value at the root should be on the first line, all the values at the children of the root should be on the second line, all the values at the grand-children of the root should be on the third line, etc. All values on each line should be in increasing order. See the on-line example for formatting.

- `operator==`: Are two sets equal? Importantly, this does not mean structurally! Instead, this means the contents of the sets are equal, regardless of the tree structure. This operation must run in time proportional to the number of the values in the set and therefore can not use the `find` member function.

- `lower_bound`: As in `std::set<T>`.

- `upper_bound`: As in `std::set<T>`

- `erase( iterator p )`: Do this efficiently — without starting from the root of the tree. You may be able to use the other `erase` function, but be careful.

- Increment and decrement operators on iterators, as discussed in Lecture 17. You will need to add parent pointers to your `TreeNode<T>` class. Your code to insert, copy and erase must handle this effectively.

I urge you to implement and test the functions one-by-one and to save the iterator functions for last. Also, **for a number of functions you will need to use a temporary, auxiliary container such as a vector** to store pointers to nodes.

Running Your Program

A main program is provided on the course web page. This is the main program we will run from the submission site. We will write a longer program to test your code more extensively. If your code works on the provided main program, but breaks on the second main program, you will still earn significant credit.