CS2124 Data Structures

Assignment 4: Tree Algorithms

For this assignment you'll be implementing several tree-based algorithms that we saw in class. Specifically you're editing only "tree.c". You shouldn't need to change any other files. Here are your algorithms to implement:

Huffman Tree - printHuffmanEncoding

Given the root of a Huffman tree and a character, print the sequence of bits used to encode that character based on the tree.

In each $TNode$ there's a $char*$ called str. The variable str is a list of $chars$ whose Huffman code will be found in the subtree rooted at this node. You should use the str variables of a node's children to decide whether to go left or right.

Reminder: Going left in the Huffman tree prints a '0' and going right prints a '1'.

AVL Tree - rebalanceTree

Here is a brief outline of algorithm for rebalancing the tree using AVL trees:

Reminder: the balance of x is the height of the left subtree of x - height of the right subtree of x.

```
(1) Let $x$ be the node we are starting our rebalance from
(2) While $x$ is not NULL
      1 if the balance of $x$ is $\leq -2$ or $\geq 2$
          (i) Set $z$ equal to the child of $x$ with the greater height
         (ii) if the balance of $x$ and the balance of $z$ have different signs
                (A) if the sign of the balance of $z$ is $+$ right rotate on $z$
                (B) otherwise the balance is $-$ so you left rotate on $z$
        (iii) if the balance of $x$ is $\geq 2$ right rotate on $x$
         (iv) otherwise the balance is $\leq -2$ so you left rotate on $x$
      2 Set $x$ equal to the parent of $x$
```

To test if your tree is balanced you can enable the calls to *checkAVLTree* in driver.c. This function will inform you if there are any balances in your tree outside of 1, 0, and -1. Likewise there is function *printTree* which will print the contents and structure of your tree (elements higher up in your tree are printed with more tabs in front of them).

For my implementation of rebalance I created the following helper functions. You don't have to do this but it may help break this large problem down into several smaller, simpler, problems:

```
TNode* getTallerSubTree(TNode* x);
bool isSameSignBalance(TNode* x, TNode* z);
```

The function *getTallerSubTree* finds and returns the subtree of x with the larger height. The function *isSameSignBalance* returns true if the two given nodes both have balance ≥ 0 or ≤ 0.

Segment Tree - insertSegment and lineStabQuery

Each TNode contains a low and high value. These specify the range of values that this represents on the number line. Each TNode also contains a count, *cnt*, that represents the number of line segments associated with this node.

insertSegment This inserts a given line segment into the tree.

(1) Go down the segment tree starting at the root
(2) If the root is NULL, return.
(3) Else if the given segment is completely to the left or right of the current node's range, return.
(4) Else if the given segment completely covers the range represented by this node: increase *cnt* by 1 and return.
(5) Else: Recursively call insertSegment on the left and right children of this node.

lineStabQuery This checks how many line segments cross a given point, *queryPoint*.

(1) Go down the segment tree starting at the root
(2) If the root is NULL, return 0.
(3) Else if the *queryPoint* is completely to the left or right of the current node's range: return 0.
(4) Else: Recursively call lineStabQuery on the left and right children of this node. Return the sum of their return values and current node's *cnt*.

Deliverables:

Your solution should be submitted as "tree.c". Upload this file to Blackboard under Assignment 4. **Do not zip your file**.

To receive full credit, your code must compile and execute. You should use valgrind to ensure that you do not have any memory leaks.