CS 15 Lab 7: AVL Trees

Introduction

In this lab we will be working with AVL trees. As we saw in lecture, AVL trees are just a kind of binary search tree (BST) but, in addition to the BST invariants, they have the additional invariant that the tree is AVL balanced at every node; that is, the heights of the left and right subtrees of every node differ by at most 1. This property ensures the height of the tree is at most 2 log n (where n is the number of nodes in the tree) and thus both the worst and average time complexity of insert, search, and delete are O(log n).

The key task of an AVL tree implementation is to maintain the balance after each modification of a tree.

Getting Started

To start,

- Create a directory for this lab and move into it
- Copy the lab starter files over from the usual place
- Remember: write a little, compile, link, test, repeat
- Use the provided `Makefile` (do not modify it)

Inserting in AVL Trees

In this lab we will implement AVL trees. To start with compile the code provided (type make in the console) and execute it (with ./lab). You will see that the result is a BST containing the numbers inserted, but it is not balanced. Your task is to make sure that the resulting tree is balanced.

Inserting in a Tree

The function `Node *insert(Node *np, int value)` from the file `AVLTree.cpp` is in charge of insertion. The code is given below and has been implemented for you. You will notice it is quite similar to a simple BST insertion but with some additional lines after the insertion to make sure it remains balanced. You do not need to make any changes in this function.

```
Node *AVLTree::insert(Node *np, int value) {
    //BST insertion start
    if (np == nullptr) {
        return newNode(value);
    } else if (value < np->data) {
        np->left = insert(np->left, value);
    } else if (value >= np->data) {
        np->right = insert(np->right, value);
    }
    //BST insertion end

    //AVL maintenance start
    np->height = 1 + max(height(np->left), height(np->right));
    np = balance (np);
    //AVL maintenance end
    return np;
}
```

Your task for this lab is to implement the following functions:

```
Node *AVLTree::balance    (Node *node)
Node *AVLTree::rightRotate(Node *node)
Node *AVLTree::leftRotate (Node *node)
```

Functions `rightRotate` and `leftRotate` implement the right and left rotations we saw in lecture: given a pointer to a node of the tree they perform a left or right rotation, update the heights of all nodes accordingly, and return a pointer to the new root of the subtree. Function `balance` is similar. You are given a pointer to a subtree that may have just become unbalanced after an insertion. The first task of this function is to check if the tree is indeed unbalanced. If so, make sure that you do whatever rotations are needed to make sure it is balanced (those rotations are handled by invoking rightRotate and leftRotate). Once finished, you return the new root of the tree.

Make sure to update pointers of the nodes in these operations correctly. As usual with handling pointers you must also check for possible null pointers to prevent segmentation faults.

Once you are done, look into `main.cpp`. You will see that the driver only inserts 5 nodes into the tree. Go ahead and change it so that you can insert 17 or even more numbers. Make sure that your tree is balanced no matter what you insert (other than repeated numbers!).

If you implement `balance` and either of the other functions, that's great.

Bonus Questions

How about deletions? Although we have not explained how to handle them, they have 4 cases that are the analogous to those of insertion. Do you feel up to the challenge? If so, implement the private function `bool AVLTree::remove(Node *node, int value)`. You may re-use the existing helper functions or create new ones as you see fit.

Submitting Your Work

To submit your work, run the command `make provide`