CSC 325 Adv Data Structures

Lecture 12

AVL Trees

There are many types of trees that attempt to either maintain being balanced or correct the problem after the tree becomes unbalanced. AVL was one of the first such trees and this data structure was actually developed back in the 60's. It uses simple counting to determine if it is balanced and performs rotations if it is not. Other than that, it follows all the same rules as a BST.

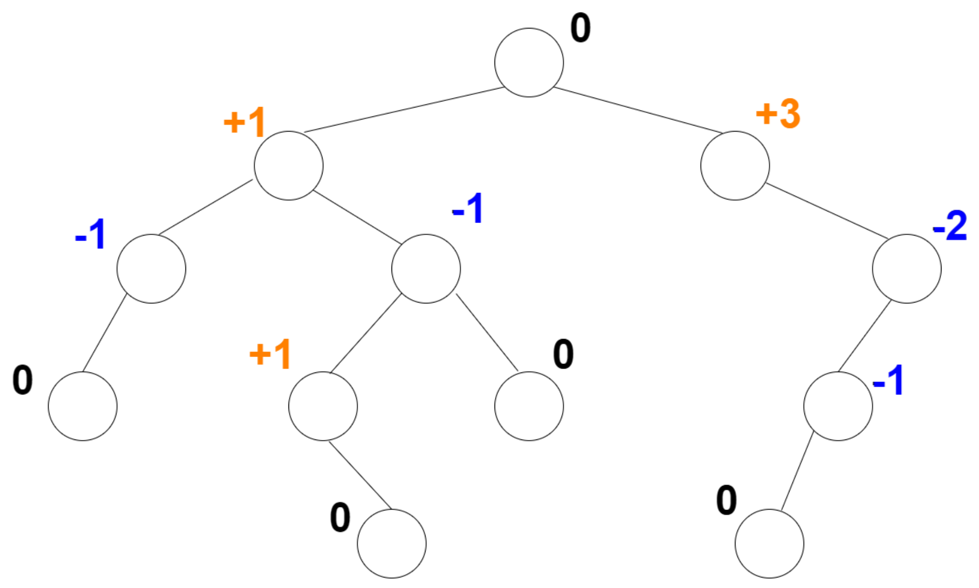
When we covered Red Black trees, I gave you a definition of what is meant by a tree being "balanced". Turns out there are several definitions for this. The Red Black tree definition was that the longest path from root to a null leaf can't be longer than 2x the shortest path from root to a null leaf. For AVL trees, we will be using a different definition of balanced. In fact, this definition finds its way directly into the algorithm. The "balance factor" of a node is the difference in heights between the right subtree and the left subtree. i.e.

$$BalanceFactor(node) = Height(node.rightChild) - Height(node.leftChild)$$

The height of a null node will be considered -1. The possible values for balance factor and their meanings are as follows:

Balance Factor on node	Meaning
-2 or less	Node is "left heavy" and is considered unbalanced. Needs rebalancing
-1	Node is "left heavy" but not considered unbalanced
0	Node is perfectly balanced
+1	Node is "right heavy" but not considered unbalanced
+2 more	Node is "right heavy" and is considered unbalanced. Needs rebalancing

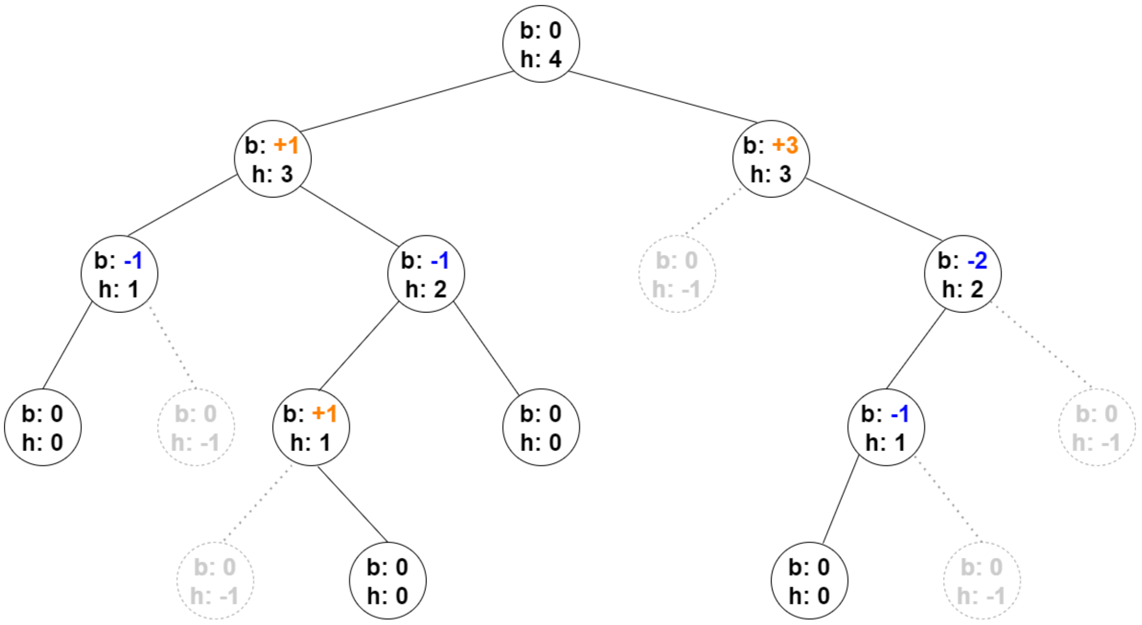
Any node that is considered "unbalanced" (either less than -1 or greater than +1) will need to use predefined rotations to fix the balancing issue. Any node that is considered "balanced" (either -1, 0 or +1) requires no action to be taken. Here is an example of balance factor in action. Given the following tree, the balance factor is written next to each node:



If any node is considered unbalanced, the entire tree is considered unbalanced. To get the balance factor of a node, we need its height. So, we will need to store the height of each node and keep this number up to date. The formula for the height of a node is as follows:

$$Height(node) = Max\big(Height(node.rightChild), Height(node.leftChild)\big) + 1$$

Recall that the height of a null node is considered to be -1. Here is the same tree with the height values filled in ('b' is balance factor and 'h' is height):



Some null nodes are shown to help see how heights and balances are calculated. Remember that height is the max of the heights of the children then add 1 to that number. So, if one child has height of 1 and one child has height of 0, the max is 1. We then add 1 to that number to get height of 2 for our parent. For balance factor we take the height of the right child minus height of the left child. So, for example, if height of the right is -1 (for a null node) and the height of the left is +1, then balance factor would be -1 – 1 which would be -2.

Here are the operations we will go over with AVL trees.

<u>Abstract Data Type: AVL Tree</u>

`insert(value)`	Inserts the given value into the tree using normal BST rules. It then checks the balance of each node along the insert path and rebalances if any are unbalanced.
`search(value)`	Searches for the given value in the tree. Returns true if found, false otherwise.
`min()`	Returns the minimum value in the tree.
`max()`	Returns the maximum value in the tree.
`rebalance(node)`	Rebalances the tree at the given node.

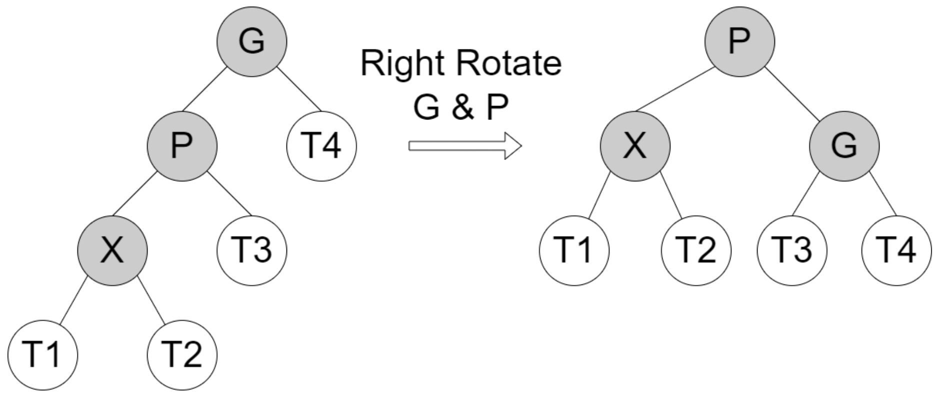
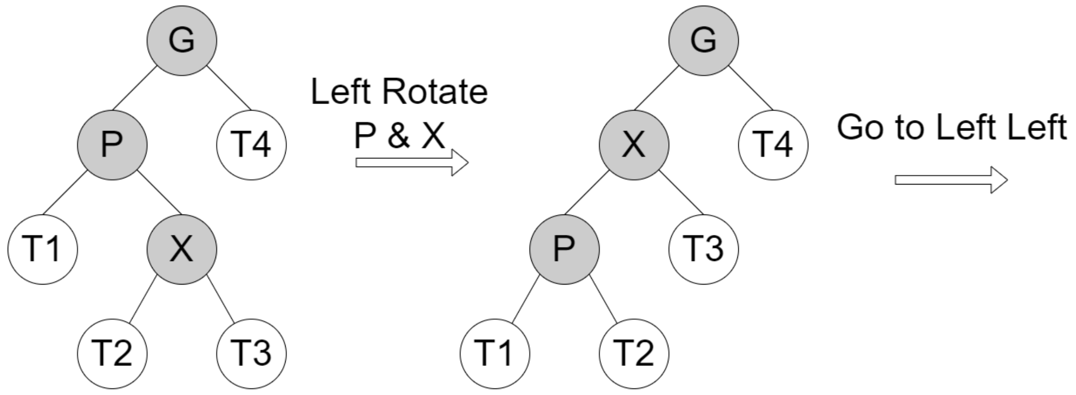
The search, min and max functions are the same as a BST. However, after we insert, we need to check for a rebalance. To do this, we use recursion for the insert function and then when it unwinds, we update the balance factor for each node. This ensures that the balance factor is updated for the parent of the new node, and the parent's parent and so on until reaching the root. If at any point in updating the balance factors of the node, we find a node that is unbalanced, we need to rebalance it. To make this work, we should create our node like so:

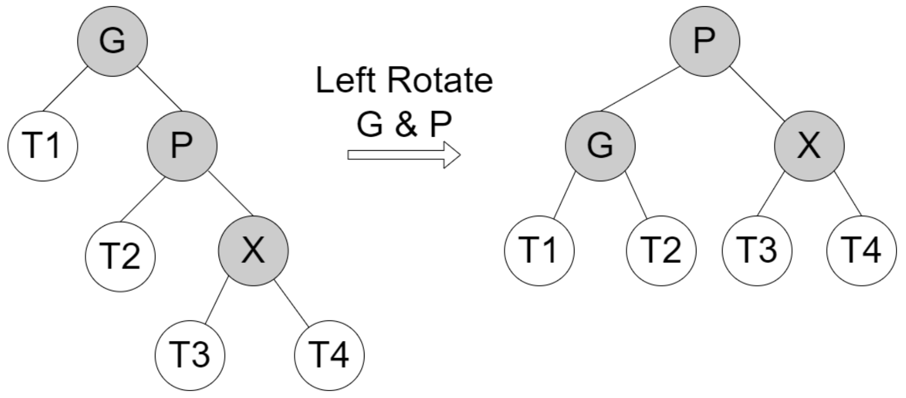
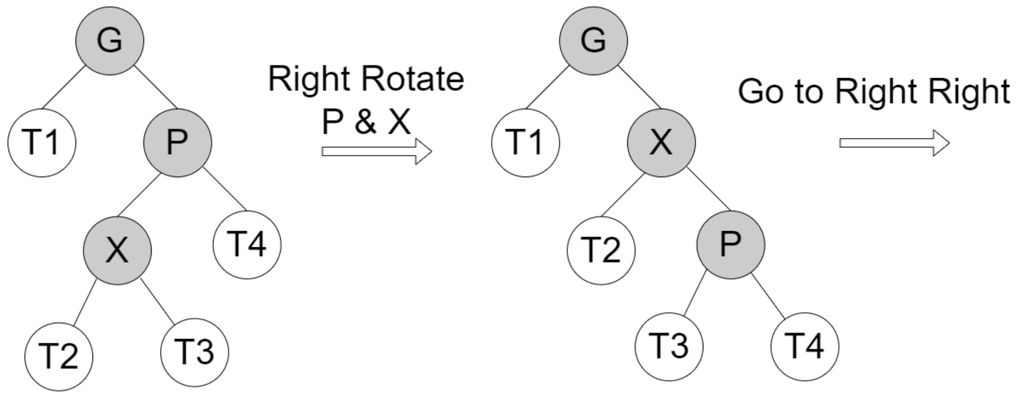
```
class AVLNode {
  private AVLNode left;
  private AVLNode right;
  private int data;
  private int height;

  public void updateHeight() {
    height = Math.max(right.height, left.height) + 1;
  }

  public int balanceFactor() {
    return right.height - left.height;
  }
}
```

We could have a field for balance factor to remember that value, but it's easy enough to calculate this on the fly with a method, plus it may change if the heights of our child nodes change, so a method will suffice to get this value. The rebalance operation works the same as Red Black tree rotations, just without the red and black colors:

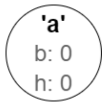
Left Left Case	 G P T4 X T3 T1 T2 Right Rotate G & P P X G T1 T2 T3 T4
Left Right Case	 G P T4 T1 X T2 T3 Left Rotate P & X G X T4 P T3 T1 T2 Go to Left Left

Right Right Case	 G T1 P T2 X T3 T4 Left Rotate G & P P G X T1 T2 T3 T4
Right Left Case	 G T1 P X T4 T2 T3 Right Rotate P & X G T1 X T2 P T3 T4 Go to Right Right

Example

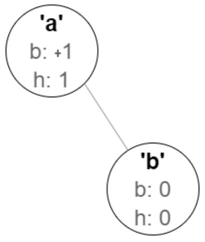
Let's see a simple example of inserting values into an AVL tree.

```
insert('a')
```



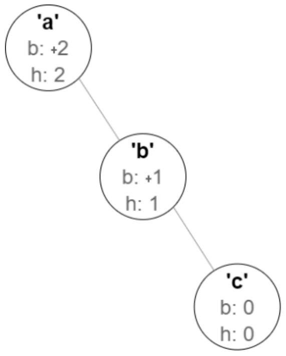
After our initial insertion, the tree now has a non-null root. We calculate the height of the node and its balance factor. For new nodes, this is always 0 for both. Since the new null will have null children, its height will be `max(-1, -1) + 1` which is 0. If we subtract the height of the null nodes, we get 0. So, a newly placed node will have 0 for height and balance factor.

```
insert('b')
```

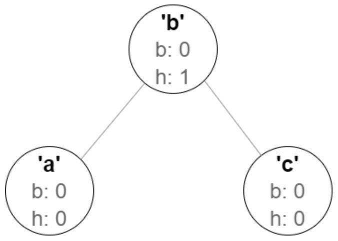


After inserting 'b', we must recalculate the height and balance factor of all nodes go up to the root. The node containing 'b' will have 0 for both height and balance factor, but now the root is at a height of 1 and is slightly right heavy. But not enough to rebalance (refer to the table above on balance factors).

```
insert('c')
```



After inserting 'c' and recalculating the height and checking the balance factors of all nodes up to the root, we have found an imbalance that requires fixing. To be clear, on the path to the root, we are recalculating heights and checking balance factors as we go. AS SOON AS we arrive at a node whose balance factor is +2 (or higher) or -2 (or lower), we immediately fix that before continuing to traverse the tree. So, let's fix this:



According to our cases above, we were in a "right right case" which required a left rotate. After the left rotate, we must recalculate heights and check balance factors. As you can see, all balance factors are fixed now, and the tree is back to being balanced. Also, note that the "search" property (i.e. lesser values to the left and greater values to the right) still holds. It is clear that 'a' is lesser (i.e. comes before in the dictionary) than its parent 'b' and 'c' is greater (i.e. comes after) the parent 'b'.