AVL Trees - LL LR RL and RR rotations

In the last lecture, we saw why AVL trees are important and how rotations in an AVL tree balances it back after an unbalanced insertion. We learned all the different types of rotations we have in AVL trees. We practiced them on the smallest possible AVL tree, but today, we'll push ourselves further with some complex AVL trees and complex situations that arise after insertion.

So, before we proceed with the examples of the rotations we learned, we should know how a binary search tree that is unbalanced can be balanced with a few Rotate Operations.

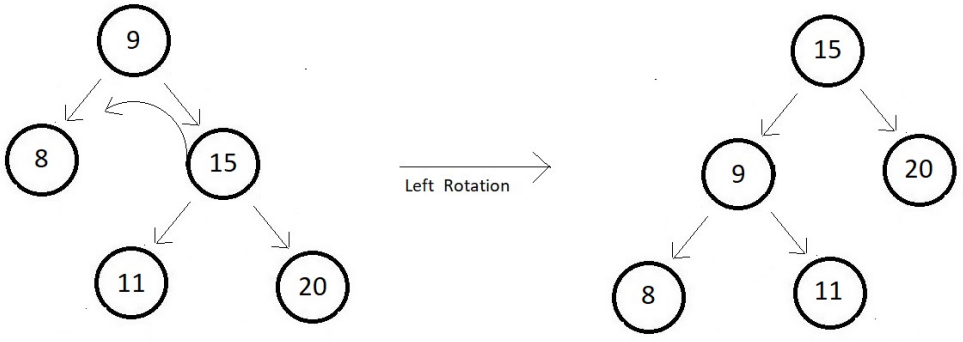
Rotate Operations:

We can perform Rotate operations to balance a binary search tree such that the newly formed tree satisfies all the properties of a binary search tree. Following are the two basic Rotate operations:

1. Left Rotate Operations.
2. Right Rotate Operation

Left Rotate Operations:

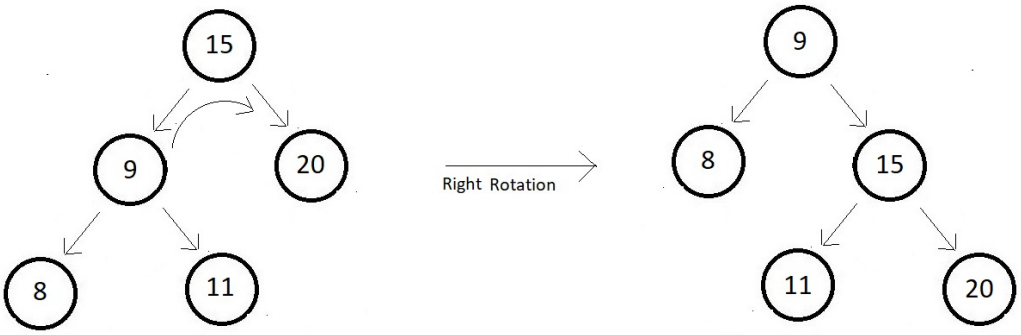
In this Rotate Operation, we move our unbalanced node to the left. Consider a binary search tree given below, and the newly formed tree after its left rotation with respect to the root.



One thing to observe here is that node 11 had to change its parent after the rotation to be able to maintain the balance of the tree.

Right Rotate Operations:

In this Rotate Operation, we move our unbalanced node to the right. Consider a binary search tree given below, and the newly formed tree after its right rotation with respect to the root.

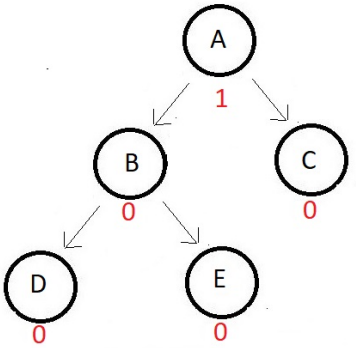


Again, as you can see that node 11 had to change its parent after the rotation to be able to maintain the balance of the tree. And rotating a tree to its left, and then again to the right, yields the same original tree as you can see from the above two examples.

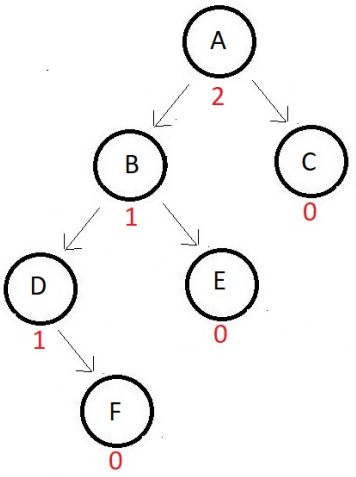
Let's move back to the balancing of the AVL tree after insertion. So, when it comes to complex trees, in order to balance an AVL tree after insertion, we can follow the below-mentioned rules:

1. For Left-Left insertion - Right rotate once with respect to the first imbalance node.
2. For Right-Right insertion - Left rotate once with respect to the first imbalance node.
3. For Left-Right insertion - Left rotate once and then Right rotate once.
4. For Right-Left insertion - Right rotate once and then Left rotate once.

We'll now see how a complex tree gets balanced again after an insertion. Consider the binary search AVL tree below:

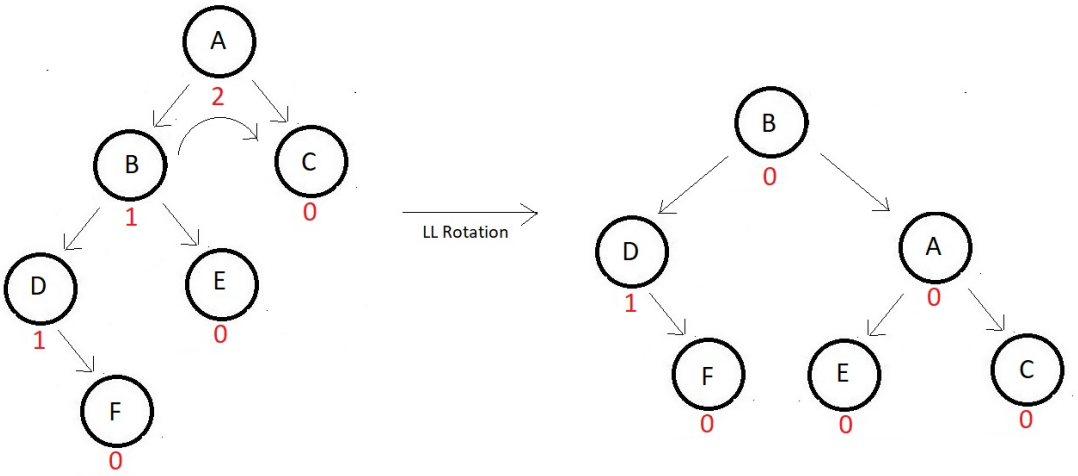


The absolute balance factor of each node is written beside, and you can see how balanced the values are. Now suppose we need to insert an element that gets its position to the right of node D. Now the updated tree looks something like this.



And the tree got imbalanced. Now, you follow these steps.

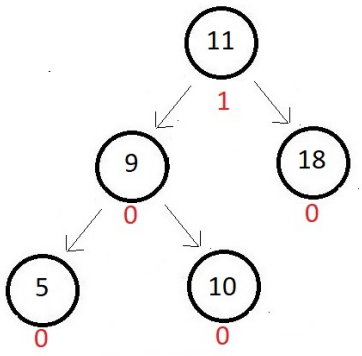
1. The first thing you would do is search for the node which got imbalanced first. We start iterating from the node we inserted at and move upwards looking for that first imbalance node. Here node A is the one we were searching for.
2. Second, you see what type of insertion was this with respect to the node we found. Here, the insertion happened to the left to the left of node A. So, this belongs to the first rule we saw above.
3. Do what the rule says. Here the rule says to right rotate once with respect to the first imbalance node. So, the tree after rotating to the right becomes:



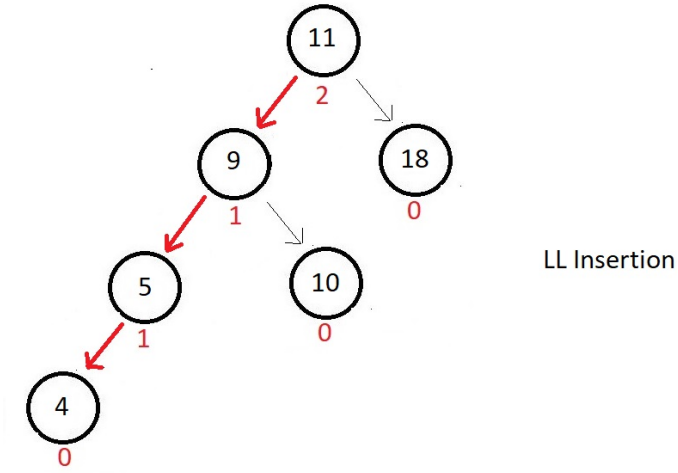
And similarly, had this been a type of right-right insertion, we would have first searched the first imbalanced node, and then would have rotated left with respect to it. But since this was just a demonstration, we would deal with two out of these four in detail now with actual numbers. Due to the similarity between the first and second rotations, we will take the LL rotation first and then one of the LR or RL rotations second.

LL Rotation:

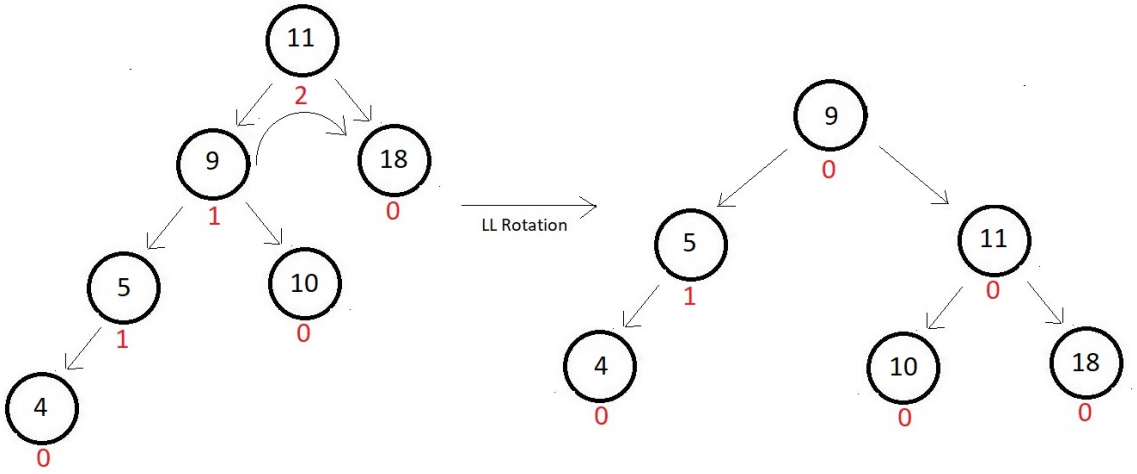
LL rotations were discussed already in the last lecture. We would just use it to balance a relatively complex AVL tree. Consider the one given below.



Absolute balance factors of each of the nodes are written beside. The tree is balanced and good for now. But now, we want to insert a node with data 4. So, that would get inserted to the left to node 5. The updated tree and their balance factors are:



And since this is a case of left-left insertion, we would rotate right once with respect to the root node, since that's the first one to get imbalanced. And in that process, we might lose the position of node 10. So, we give it a new position to the left of node 11 to accommodate it again into the tree. And our tree gets balanced again.

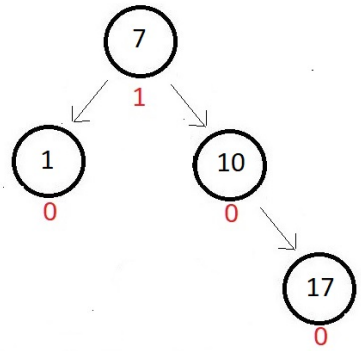


This is just a coincidence that our root node is the one we are rotating with respect to. We could come across examples where the first imbalanced ode is not the root. So, we would rotate with respect to the one we'll find first, not the root.

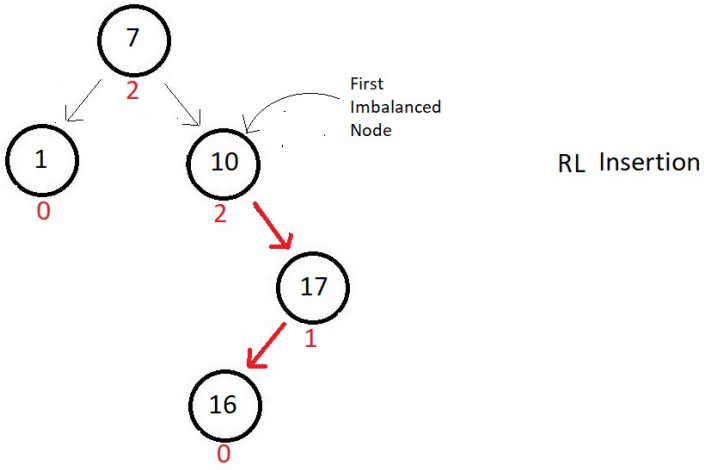
Let's now deal with one of the RL insertion cases. LR would be more or less the same, so, we'll just ignore that.

RL Rotation:

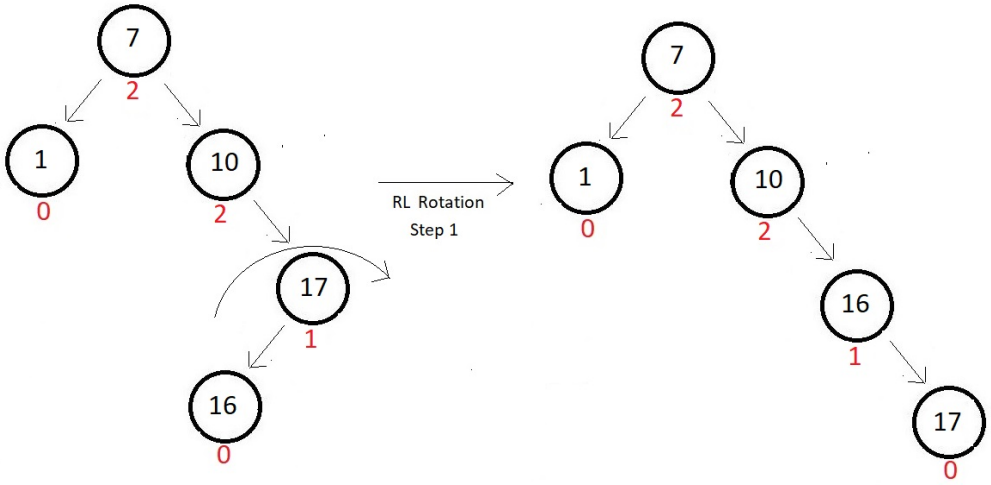
RL rotations were also discussed in the last lecture. We would just use it to balance a relatively complex AVL tree. Consider the one given below.



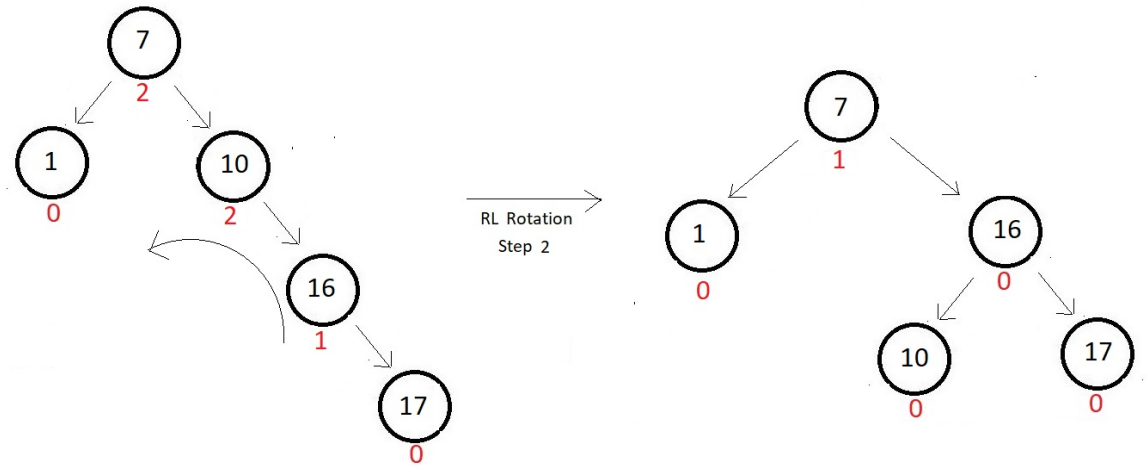
The absolute balance factors of each of the nodes are written beside. The tree is balanced and good for now. But now, we want to insert a node with data 16. So, that would get inserted to the left of node 17. The updated tree and their balance factors are:



And since this is a case of right-left insertion with respect to the first imbalanced node which is node 10, we would first rotate right once with respect to the child of the first imbalanced node which comes into the path of the insertion node. Follow the figure below.



And now, we rotate left with respect to the node we found first imbalanced, here 10. And this would do our job. Our tree gets balanced again.



So, that was what we had for today. One important thing you shouldn't miss while balancing trees. Root nodes have no special significance while you are balancing your trees. It is just about the node you find imbalanced first. It could be the root node sometimes, but it couldn't be either. So, just be careful about that. Practice on your own. Balance your trees yourselves. Hustle on.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial. Till then keep coding.