# AVL Tree

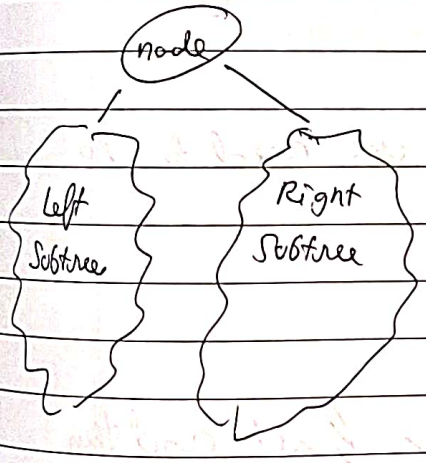
Full understanding of how AVL tree is working we will first start by some rotation techniques.

What is rotation?

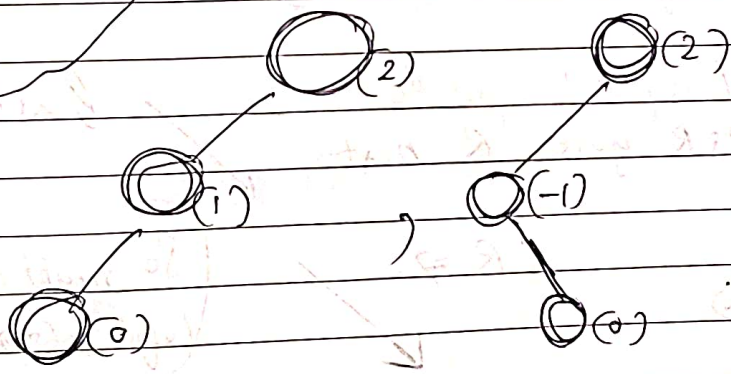
Well AVL trees are self-Balancing trees, to prevent the tree from becoming skew we are applying some checks to see if the nodes are balanced or not and apply rotation accordingly.

Balance Factor → A term which you will hear a lot in AVL trees.

Balance Factor = [Left Subtree Height − Right Subtree Height]



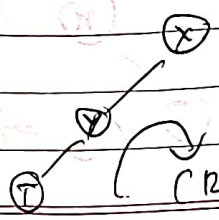

if |Balance Factor| > 1 we mark that node to be unbalance and needs rebalancing




⇒ (Examples of unbalanced trees)

So here comes the rotation part where we introduce lots of rotation to make the balance Factor 1 again.

Case 1 LL → left child of left subtree of the node make it unbalanced.



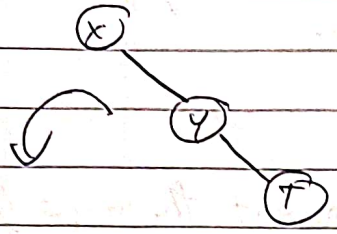

(Right Rotation).

y = x.left
T = y.left

(Rebalance)

x.left = T
y.right = x

Case 2 (RR) Right child of Right subtree of node is making the tree unbalanced

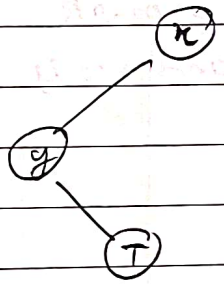



(Left Rotation around node)

```
y = x.right
T = y.right

x.right = T
y.left = x
```

(case 3 LR) right child of left subtree of node is making the tree unbalanced.




In this case of mixture of RL, we need to do two types of Rotation.

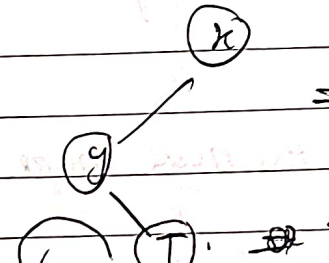
LR → go chronologically

If L is first go to L first and then there is R go to R next.

R

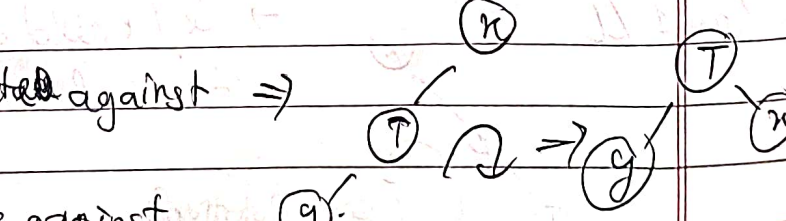
LR ⇒

(So right child of left subtree)




⇒ we have to convert it into LL case.

Step ① Left Rotated against ⇒ y

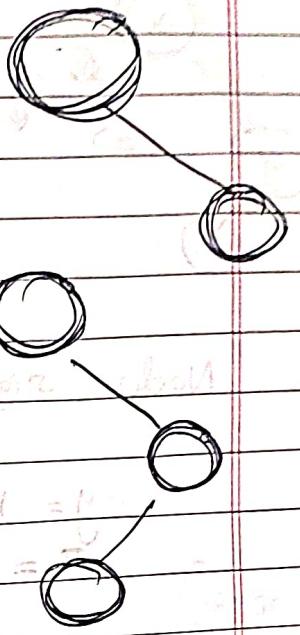



Step ② Right Rotate against x

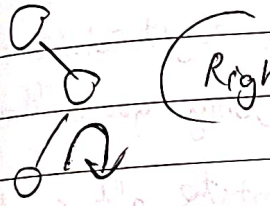
Case 4) RL → Again don't get confused, just go chronologically from left to right

RL → First R Go to right child.

→ then L Go to Left child then

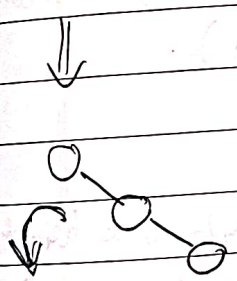


How to rotate ?

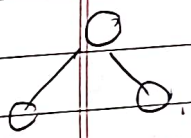


(Right Rotation against this node)

node.right = rightRotate(node.right)

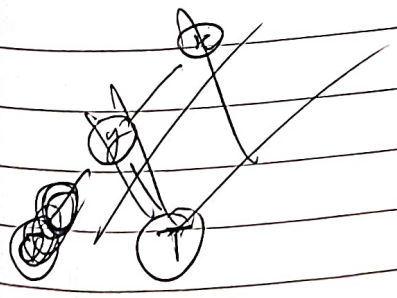


(Left Rotation against root node) ⇒



leftRotate(node).

(Code for Right Rotate) → (PTO)



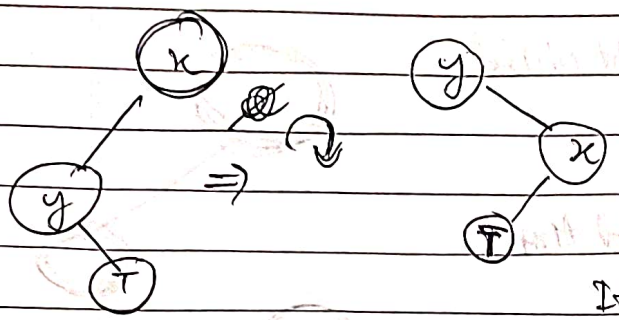
Node rightRotate(Node x) {

y = x.left

T =

x.left = T

(Code For Right Rotation)




Since after rotation abt "x" x becomes the right child of "y". Having a reference to the previous right child of "y" is v. imp. Its "T" in this case

```
Node rightRotate(Node x){
    y = x.left
    T = y.right
```

(Not need to worry abt NPE)

(Since you are doing a right rotate abt x, so you need not worry you will only need to do a right rotate abt x when a left child of x exists).

(Remapping things)

```
    x.left = T
    y.right = x
```

nodes x and y got new children T remains untouched so we calculate height.

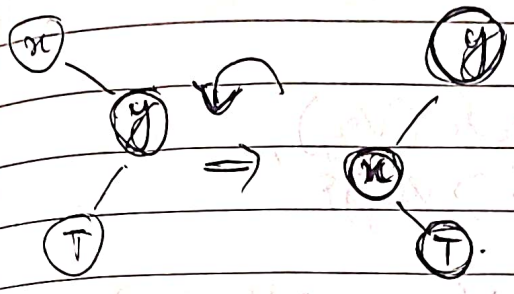
```
    x.height = 1 + Math.max(x.left.height+1, x.right.height+1)
    y.height = 1 + Math.max(y.left.height+1, y.right.height+1)
```

Since y is now the parent of x we have to calculate the height of x first always.

```
    return x;
}
```

P.T.O for calculating (Left Rotation)

Code for Left Rotation.




Since after Left Rotation, abt "x", "y" becomes the root and the left child of "y" now becomes "x", So we must have a previous reference of "y" 's left child.

```
Node leftRotate(Node x){
    y = x.right
    T = y.left
```

(Same explanation, no need to worry abt y being null as you are doing left rotation abt x, x.right will always be present)

remapping things

```
    y.left = x
    x.right = T
```

y is new parent of x, calculate height of x first then of y

and x, both its subtrees have changed hence height needs to be calculated again.

```
    x.height = 1 + Math.max(x.left.height, x.right.height)
    y.height = 1 + Math.max(y.left.height, y.right.height)
    return y;
}
```

(Now Lets write code for deletion & insertion, code for search is pretty simple)

## insertion

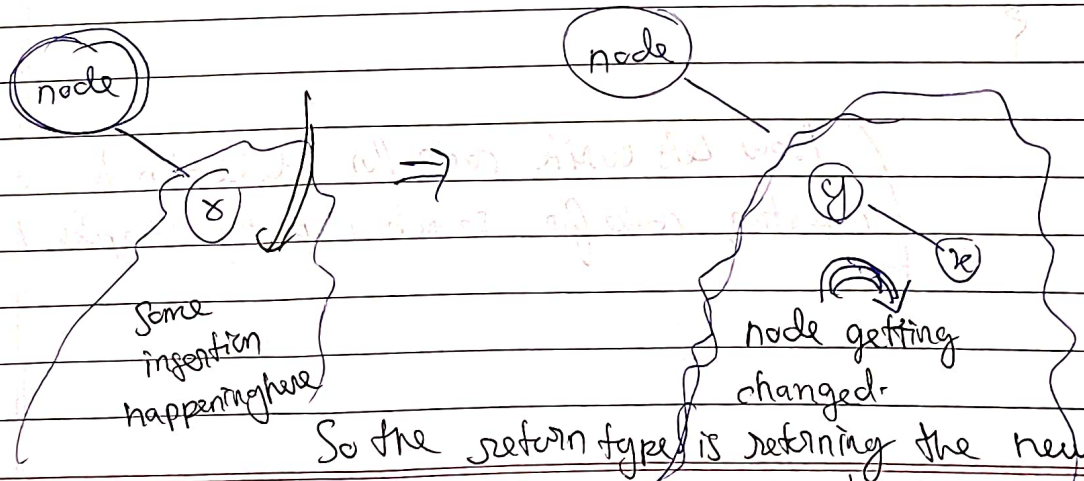
```
Node insert (Node node, int key){
    if (node == null){
        return new Node (key);
    } else if (node.val > key){
        node.left = insert(node.left, key);
    } else if (node.val < key){
        node.right = insert (node.right, key);
    } else{
        return node;
    }
```

(we reached a spot where node is to be inserted, we insert the node)

You might be wondering, that why first of all does the function have a return type Node and why does it return a Node. And then also you might be wondering that why are we doing the mapping

```
node.left = insert (node.left, key).
```

Well, well, well am here to clear all your doubts, as you saw on the previous page we are performing Rotations to keep the tree balanced. Lets say that after insertion in any subtree the root changed due to some rebalancing




So the return type is returning the new node of the subtree where insertion happened. And the left child pointer has to be pointed to this new node by the parent.

Let's come back to the code,

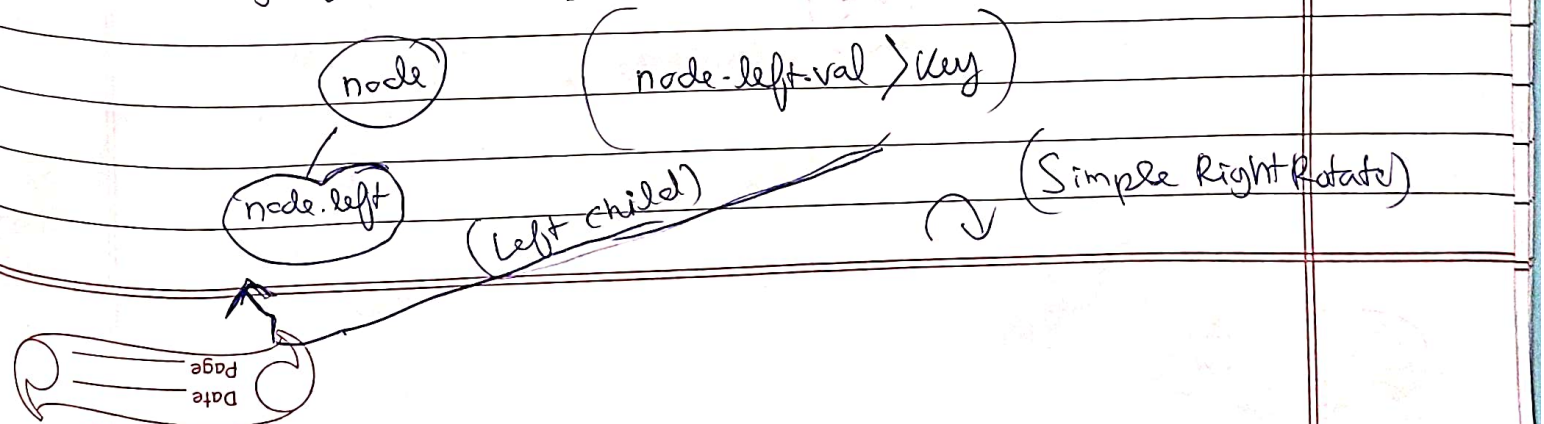
```
Node insert(Node node, int key){
    // the previous page insertion logic,
    Why are we calculating the height at this point?
    node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));
```

↓ Because after this point we will do rebalance checks this node if required, and will loose access to this node once we return the new root of this subtree.
Since its very much possible that the ~~height changed~~ height of either of its left or right subtree height changed, can cause height of this node to be changed.

```
int balanceFactor = getBalanceFactor(node);
if (balanceFactor > 1 && node.left.val > key){
    return rightRotate(node);
}
```

↳ This translates to a LL case balanceFactor > 1 → left subtree bigger.

left Subtree became bigger after key insertion, and which child did the key get inserted?



Case (LR)

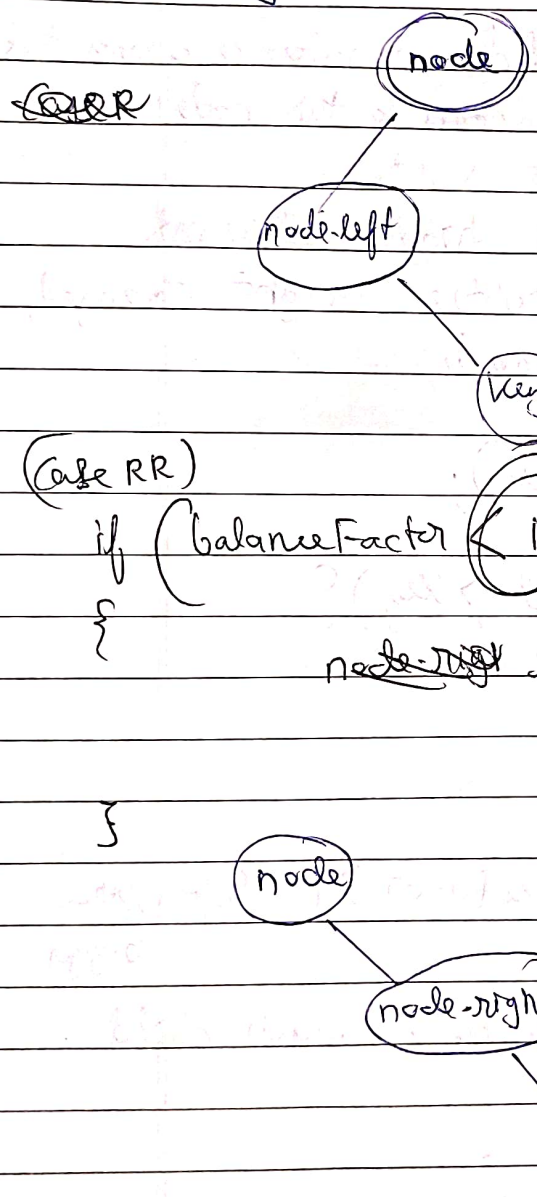
if (balance Factor > 1 && node.left.val < key)

{

node.left = leftRotate (node.left)

return rightRotate (node).

}




mistake, this will be ⩵1

[balanceFactor <= -1]

(case RR)

if (balanceFactor < 1 && node.right.val < key)

{

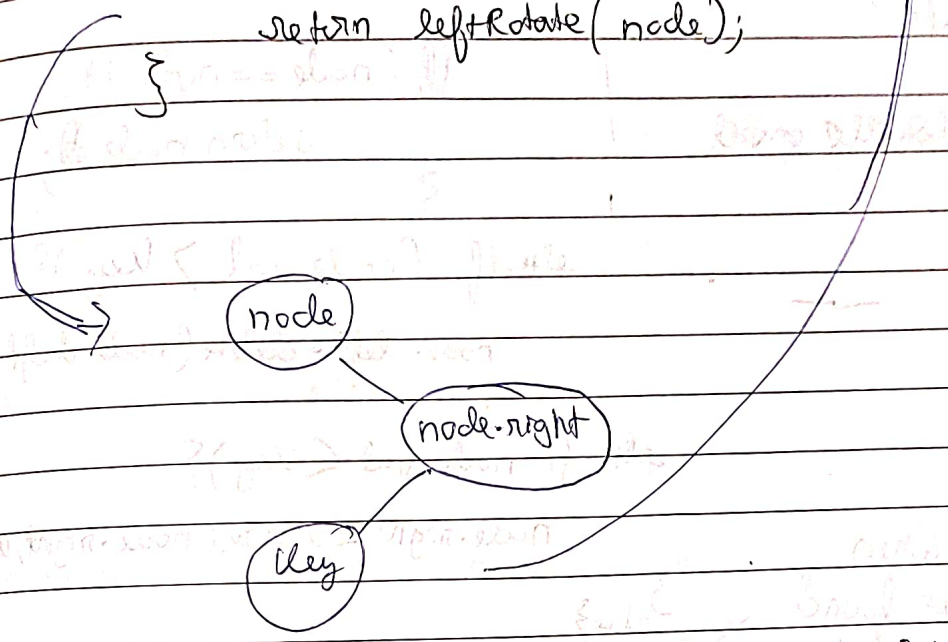
node.right return leftRotate (node);

}

mistake < -1

(Case RL)

```
if ( balanceFactor < -1 && node->right->val < key)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
```



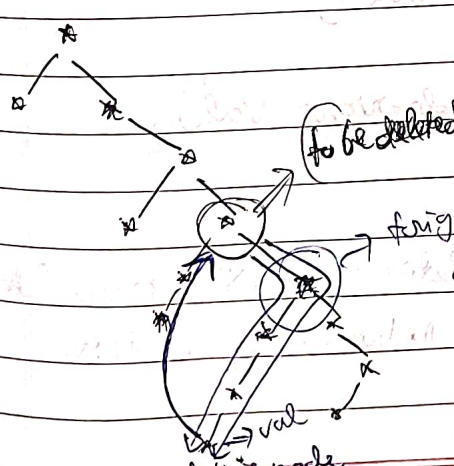

Now lets discuss deletion. → lets revise basic deletion, how did we do basic deletion in BST.

we would first of all find the node.

2 cases arise

1). Node has no child or single child → Just replace the node with that child or null value

2) Node has both left & right children → In that case we find the inorder sucessor of the node and trigger deletion for that value.




(Falls in case 1).

```
private int inorderSuccessor(Node x){
    Node y = x.right;
    int val = INT_MAX
    while(y != null){
        val = y.val;
        y = y.left
    }
    return val;
}
```

```
private Node delete(Node node, int key){
    if(node == null){
        return node;
    }
    else if(node.val > key){
        node.left = delete(node.left, key)
    }
    else if(node.val < key){
        node.right = delete(node.right, key)
    }
```

Case when we found the node exactly.

Case 1: when both or either of the child is null.

```
    if(node.left == null || node.right == null){
        return node.left == null ? node.right : node.left;
    }
```

return either null or whichever child is not null.

Case 2: both children are present

```
    int val = inorderPredecessor(node);
    node.val = val;
    return delete(node.right, val);
}
```

This is customary node deletion in BST, now let's see where does AVL Tree do the rebalancing in this.

After normal BST deletion has been done, we would focus on height re calculation.

node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));

⇓

Again the same logic as subtree due to rebalancing and deletion, would have faced changes in height and we are doing it here since we will loose access to this node later onwards. So alter height now itself

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && getBalanceFactor(node.left) >= 0){

→ LL case do rotation accordingly

}

if (balanceFactor > 1 && getBalanceFactor(node.left) < 0){

→ LR case do rotation accordingly

}

if (balanceFactor < -1 [mistake < -1] && getBalanceFactor(node.right) >= 0){

→ RL case do rotation as we did in insertion.

}

if (balanceFactor < -1 [mistake < -1] && getBalanceFactor(node.right) < 0){

→ RR case.

}

}

While I was studying this conditions of rotations in insertions and deletions of when to rotate in which case I noticed a difference in condition.

Mistake, when < less than comparison do -1

| Insertion | Deletion. |
|---|---|
| LL → if (balanceFactor > 1 && node.left.val > key). | LL → if (balanceFactor > 1 && getBalanceFactor(node.left) ≥ 0) |
| LR → if (balanceFactor > 1 && node.left.val < key) | LR if (balanceFactor > 1 && getBalanceFactor(node.left) < 0) |
| RL → if (balanceFactor < 1 && node.right.val > key) → < -1 | RL if (balanceFactor < 1 && getBalanceFactor(node.right) ≥ 0) → ≤ -1 |
| RR → if (balanceFactor < 1 && node.right.val < key) → < -1 | RR if (balanceFactor < 1 && getBalanceFactor(node.right) < 0) → ≤ -1 |

I was wondering why to keep confusion and why not just have same condition for rotation for both Insertion & Deletion. The "if" conditions of deletions seems to be more generic as they would work for both insertion as well as deletion. And the truth is "yes" they will work for both

But in insertion writting the code in this manner makes it more intuitive and as we have the info about how the key got inserted so we can do it.

But in deletion we have no idea how the deletion might have affected the nodes and where disbalance might have occured. since we don't have a concrete check like [node.left.val < key] condition so we have to check the balanceFactors of the childs.

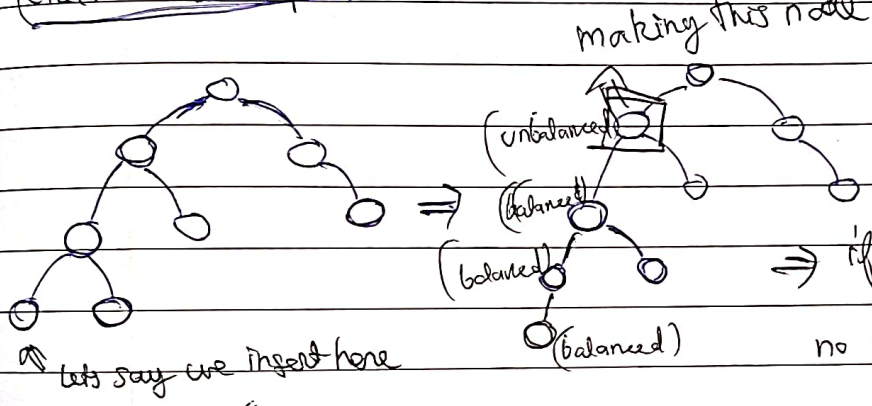
(Coming to the last part)

When I was starting to read about AVL trees I had a very naive doubt. whether it be insertion or deletion, why do we need to do rebalancing of all the ancestors in the path which we came in. Won't balancing the first node itself balance the complete tree.

To debunk this myth of mine, I saw a case in which it might be problem.

Insertion.

(Check this example)




if we just only check after insertion, then we would notice no balancing is required.

But if in the backtracking if we did not apply rebalancing we would leave the marked node as unbalanced causing tree to become skewed.