AVL Tree

1. It is a BST
2. |height of left subtree| - |height of right subtree| = {-1, 0, 1}

Balance factor

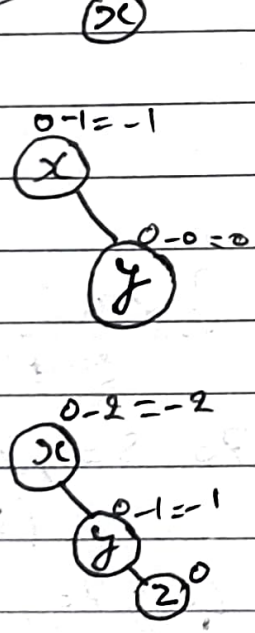
* If the balanced factor is greater than or less than this, then (4) case arises

left Rotation

↓ x, y, z

Case I

left Rotation



- no left and right height
left - height of right = 0 | Balanced

Rtg

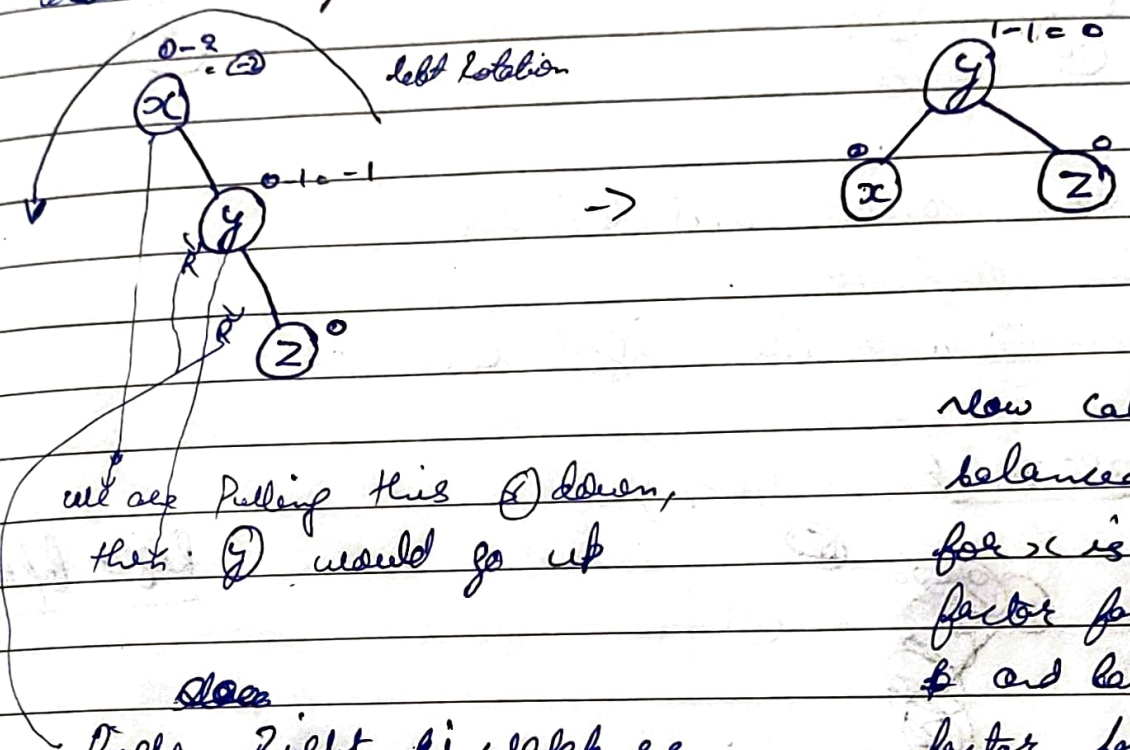
- now left and right subtree is 0 & 1 so |height of left subtree| - height of right subtree| = 0-1 = -1 |Bal

(x & y) both Balanced

- y & z are Balanced factor but x is not Balanced because

height of left subtree of x is 0 & height of right subtree of x is 2 & 0-2 = -2 which not satisfy the Balanced factor condition.

now we do left rotation, note that (left rotation) always the middle element go to root

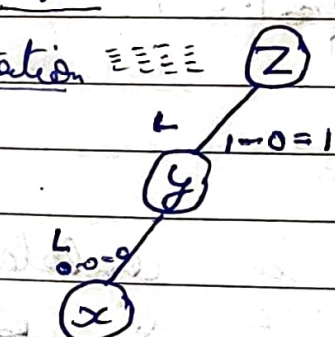


we are pulling this y down, then y would go up

Right Right ki wajah se ha imbalanced, isliea then rotation would be left rotation.

now calculate balanced factor for x is 0, balance factor for z is 0 & and balanced factor for y is 1-1=0

Right-Case II Right Rotation



(2-0 = 2) z is not Balanced factor

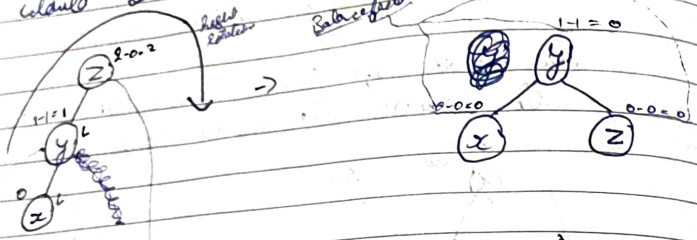
y is balanced factor

x is balanced factor

(height of left subtree - height of right subtree = 1-1, 0, is called Balanced factor)

Thus tree is unbalanced because of

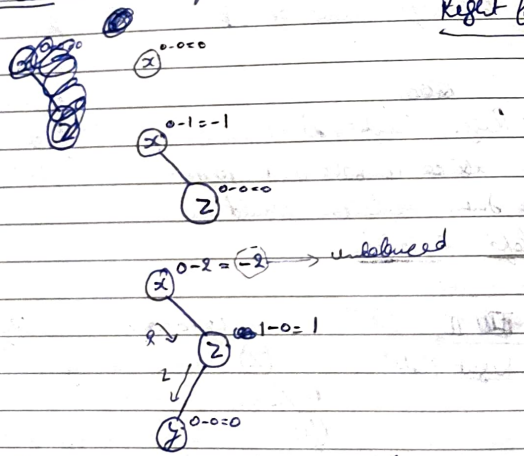
left insertion and again left insertion, so we should balanced it by Right Rotation



medium element will become root
Pull (z) into the right one

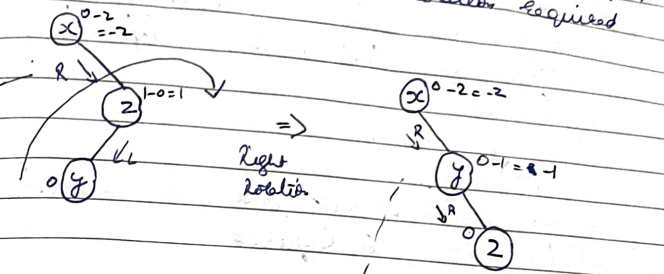
Case III : z, y, x

Right left Rotation



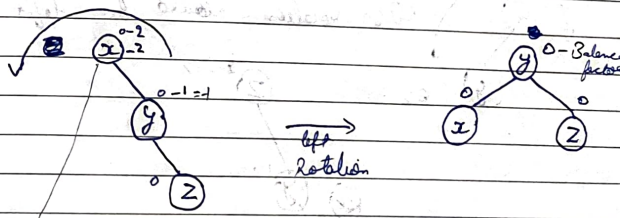
This tree is unbalanced because first Right & the left, so it is called Right left Rotation.

In case I & case II we done only 1 Rotation left & Right but in case III 2 Rotation required



This tree is still unbalanced because (x) do not satisfy balanced factor

This tree is unbalanced because of Right + Right. So we perform left Rotation in this tree.

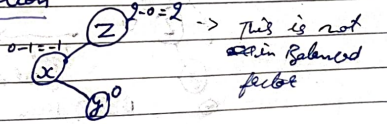


Pull x down, then y would go up, medium element (y) become root

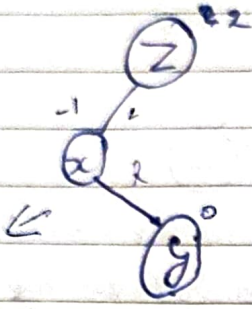
Left Right Rotation — Case IV

Left Right Rotation

z, x, y

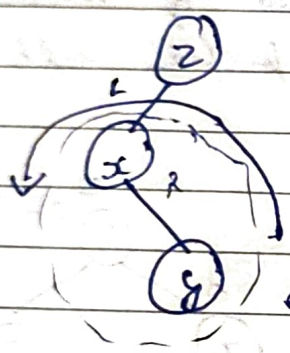


This tree is unbalanced because of first left insertion is there and ~~then~~ then right insertion is there, left and ... then right this is k/a left right rotation

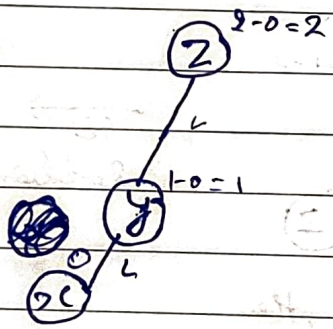


In this case, 2 rotation would be required

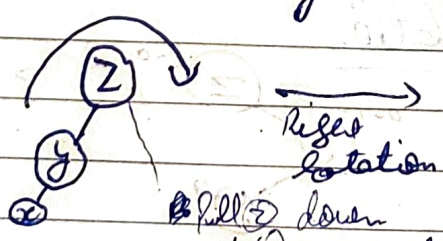
Try to make this tree left + left

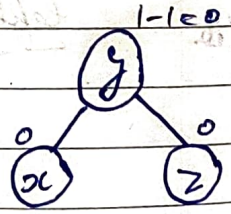


First rotation would be left rotation



Again unbalanced, if tree is unbalanced because of left + left, so we could balanced it by right rotation.





There 4 types of condition could be there to balance out AVL tree

1, left Rotation — ye tumko tb karni ka when tree is unbalanced because of two Right insertion

2, Right Rotation — ye tb karni ka tumko when tree is unbalanced because of two left insertion

3, Right - left Rotation — when tree is unbalanced due to first right and then left insertion

4, left - Right Rotation — when tree is unbalanced due to first left + then right insertion

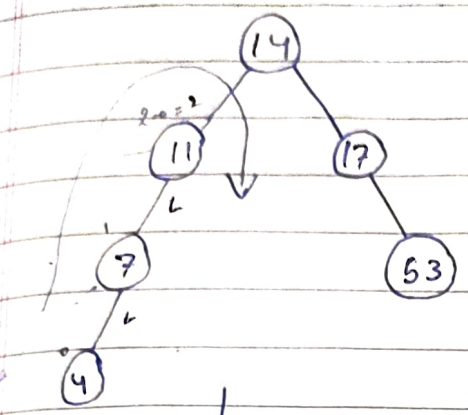
left height - Right height

B.F or balance factor = {-1, 0, 1}

60

1) Construct AVL tree by inserting the following data

14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

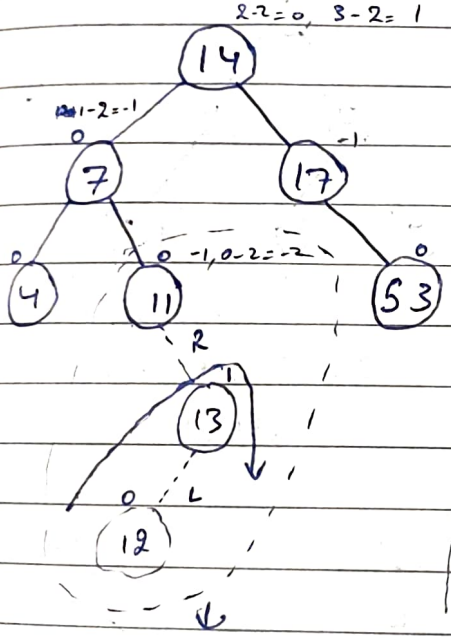


first element is root

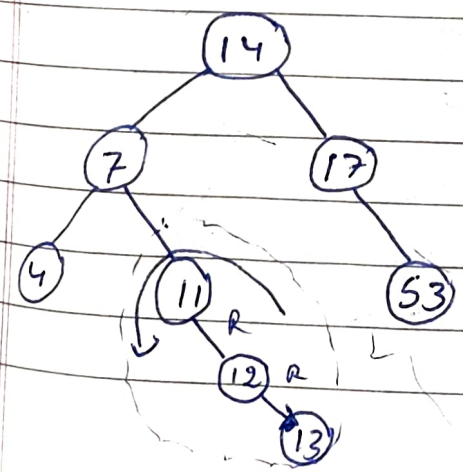
17 greater than 14 balance factor is 0 & for 11 also balance factor is 0

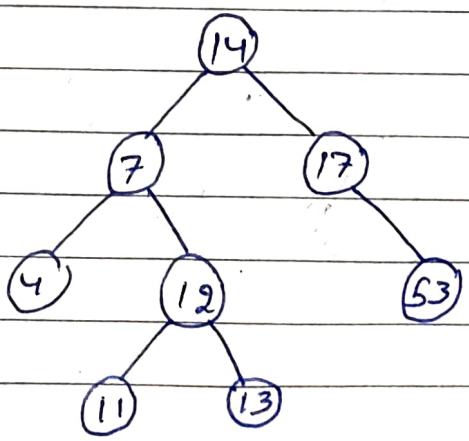
B.F of 7 is 0, now B.F of 11 is 1 & B.F of 14 is 2-1 = 1

B.F of 4 is 0, B.F of 7 is 1, B.F of 11 is 2-0 = 2 so this is not balanced tree. This tree is unbalanced due to left & left rotation or LL rotation, we need to balanced it by right rotation.

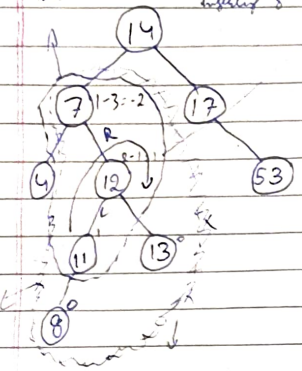


This tree is unbalanced due to right left rotation or RL Rotation, first we make it right right & then we perform left rotation

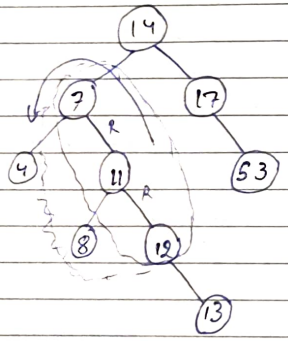




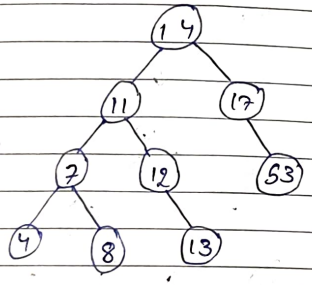
This is our critical node



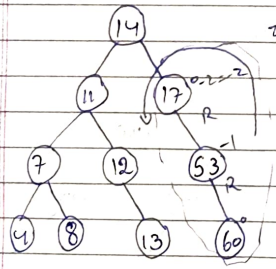
Tree is unbalanced due to Right left Rotation or RL rotation, we can balance it by first making it a Right Right rotation & then perform left Rotation



->

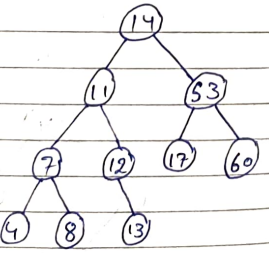


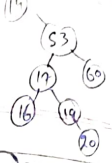
8 is less than 11, so to left of 11 & 8 is greater than 7, so go to right of 7

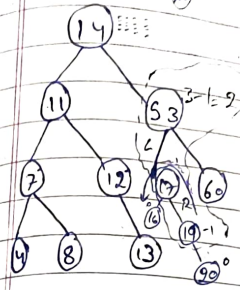


Tree is unbalanced due to Right Right rotation or RR rotation

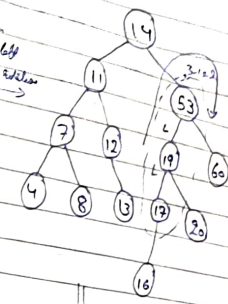
So we need to perform left Rotation to make this right subtree balanced



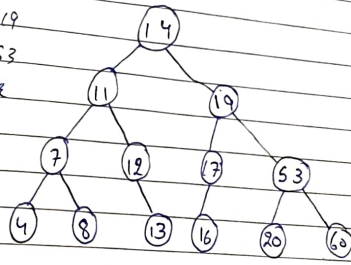




This is unbalanced to left right rotation Try to make left & left & then perform left rotation



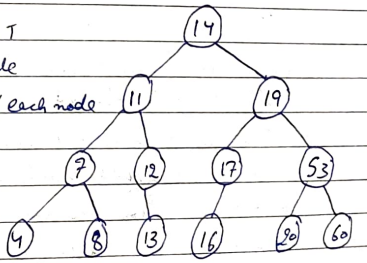
20 is present to the right of 19 but to the right of 19, 53 is present. 20 is lesser than 53 so 20 go to the left of 53



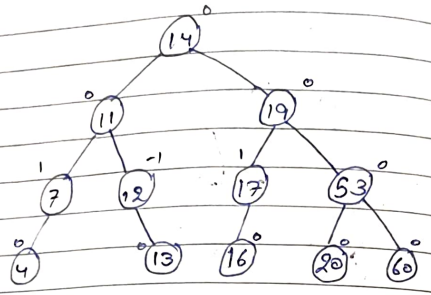
Deletion in AVL Tree

1, Deletion same as in BST
2, after deleting a node check balance factor of each node every time

Delete - 8, 7, 11, 14, 17

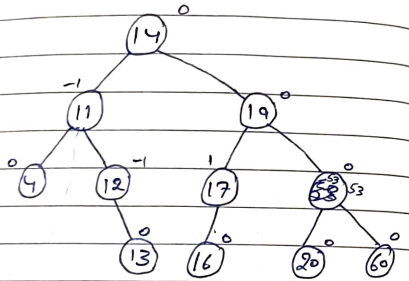


After deleting 8 ->



Tree is balanced ⊂

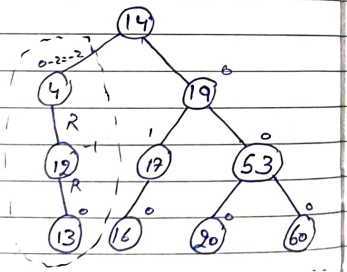
After deleting 7 ->



Tree is balanced ⊂

After deleting 11 ->

Both left & right children of (11) is present, so we can delete (11) by inorder Predecessor or inorder successor. We choose inorder Predecessor.

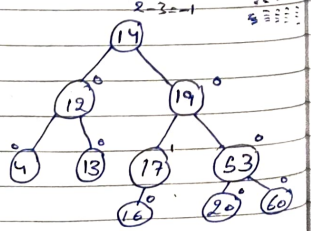


Now check Balanced factor

Tree is not balanced,

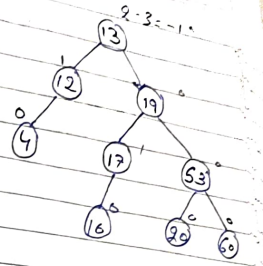
Tree is unbalanced due to Right right rotation or RR rotation, so we can balanced it by using left Rotation

Now again check balanced factor



After deleting 14 ->

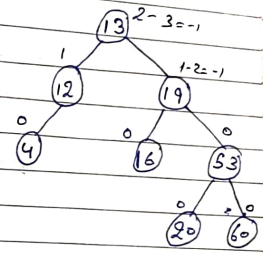
(14) is having both left and right child, so we can delete (14) by inorder Predecessor or inorder successor, so we choose inorder Predecessor



Now check balanced factor

After deleting 17 ->

There is only one child present in 17, so we can easily replace 17 with 16



Now check balanced factor

If tree is balanced we can delete next node, if tree is not balanced first we have to balance it & then we can delete a node

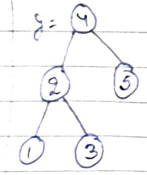
Code for AVL Tree insertion

Binary Search Tree 14. Java

```
int height (Node N) {
    if (N == null)
        return 0;
```

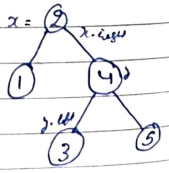
```
    return N.height;
}

int max (int a, int b) {
    return (a > b) ? a : b;
}

Node rightRotate (Node y) {
    Node x = y.left;
    Node temp = x.right;

    x.right = y;
    y.left = temp;

    // update heights
    y.height = max(height(y.left), height(y.right)) + 1;

    x.height = max (height(x.left), height(x.right)) + 1;
    // return new root
    return x;
}
```

Explain: Right rotate



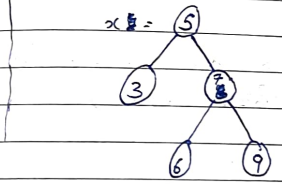
x = 2 (y.left)
temp = 3 (2.right)

x.right = 4 (y)
y.left = 3 (temp)

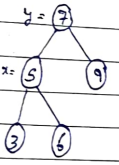



```
Node leftRotate (Node x) {
    Node y = x.right;
    Node temp = y.left;

    y.left = x;
    x.right = temp;
    // update heights
    x.height = max (height(x.left), height(x.right)) + 1;
    y.height = max (height(y.left), height(y.right)) + 1;
    // return new root
    return y;
}
```

Explain: left rotate



y = 7 (x.right)
temp = 6 (7.left)
y.left = 5 (x)
x.right = 6 (temp)




```
int getBalance (Node N) {
    if (N == null)
        return 0;
    return height (N.left) - height (N.right);
}

Node insert (Node node, int key) {
    if (node == null)
        return (new Node (key));
```

```
if (key < node.key)
    node.left = insert(node.left, key);
else if (key > node.key)
    node.right = insert(node.right, key);
else // Duplicate keys not allowed
    return node;

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

int balance = getBalance(node);
// Left Left case
if (balance > 1 && key < node.left.key)
    return rightRotate(node);
// Right Right case
if (balance < -1 && key > node.right.key)
    return leftRotate(node);

// Left Right case
if (balance > 1 && key > node.left.key) {
    node.left = leftRotate(node.left);
    return rightRotate(node);
}

// Right Left case
if (balance < -1 && key < node.right.key) {
    node.right = rightRotate(node.right);
    return leftRotate(node);
}
```



```
// return the (unchanged) node pointer
return node;
}
```

AVL Tree - Delotion (with code)

BinarySearchTree15.java

```
Node deleteNode(Node root, int key)
{
    if (root == null)
        return root;

    if (key < root.key)
        root.left = deleteNode(root.left, key);
    else if (key > root.key)
        root.right = deleteNode(root.right, key);
    else {
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // in order successor (smallest in right subtree)
        root.key = minValue(root.right);

        root.right = deleteNode(root.right, root.key);
    }

    if (root == null)
        return root;
```

```
root.height = max(height(root.left), height(root.right))+1;

int balance = getBalance(root);

                // left left case
if (balance > 1 && getBalance(root.left) >= 0)
        return rightRotate(root);

                // left right case
if (balance > 1 && getBalance(root.left) < 0) {
    root.left = leftRotate(root.left);
    return rightRotate(root);
}

                // Right Right case
if (balance < -1 && getBalance(root.right) <= 0)
        return leftRotate(root);

                // Right left case
if (balance < -1 && getBalance(root.right) > 0) {
    root.right = rightRotate(root.right);
    return leftRotate(root);
}

return root;
}
```