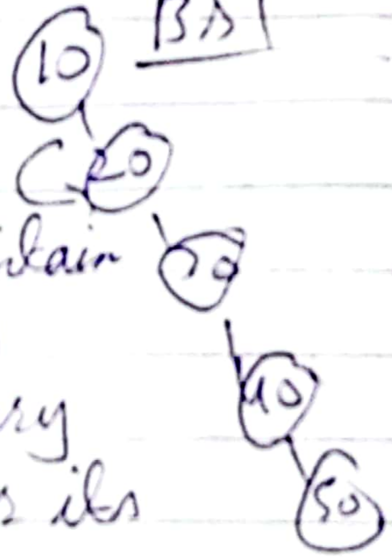
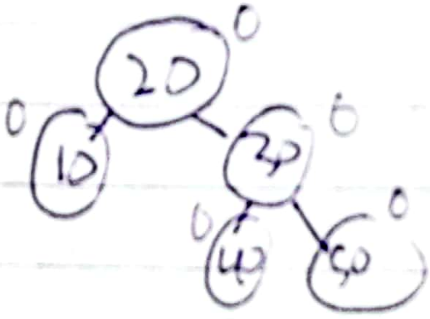
AVL tree is a type of binary search tree (BST) that keeps itself balanced to maintain the basic operations like insert, delete, and search. In an AVL tree, the difference in height called the (balanced factor) b/w the left and right subtrees of any node is at most 1. if the tree become unbalanced after an operation it gets balanced using rotations.

Why it is useful:-> In a single BST, if the tree becomes unbalanced

all nodes are on one side and the time complexity is O(n). but if we use AVL tree it maintain the height perform all operations like insert, delete or search is very fast the time complexity for its is O(Log n).





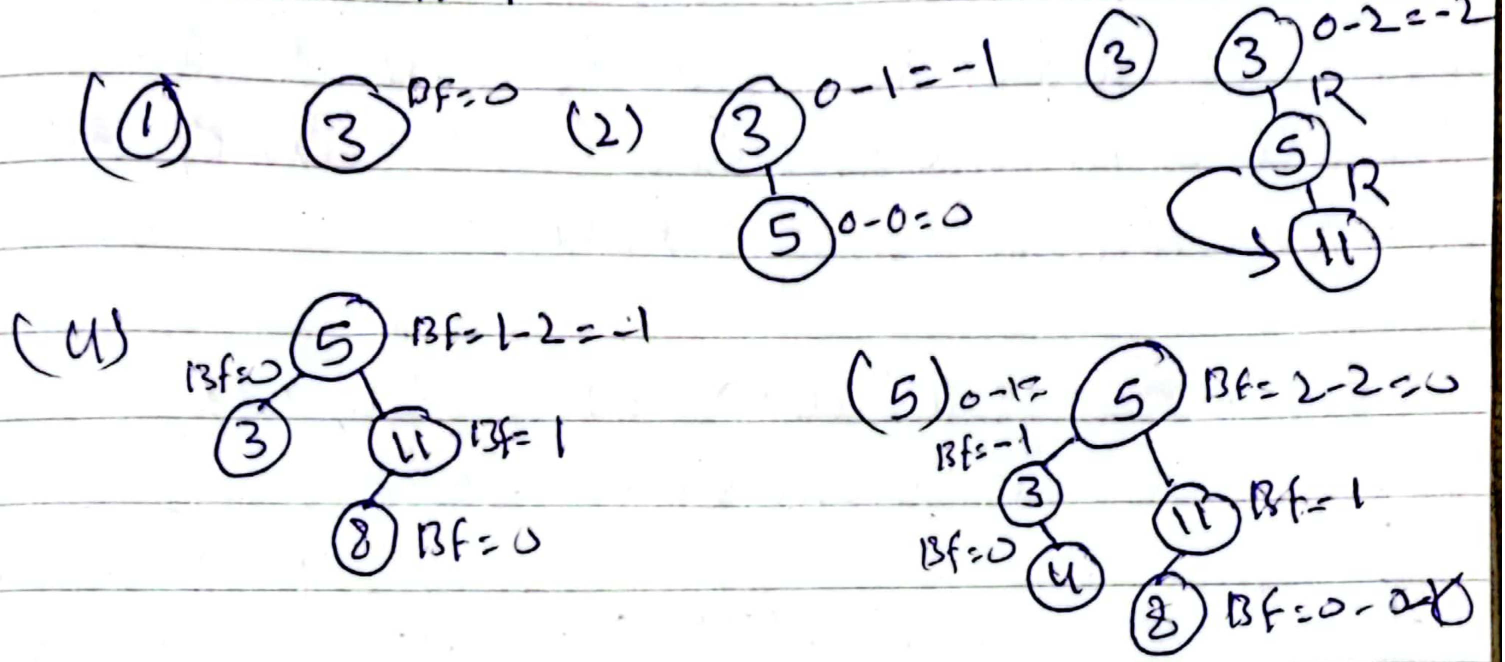
Avl tree Insertion:→

Create Avl tree

Ddlee = 3, 5, 11, 8, 4

BF = height of left subtree - height of right subtree

BF = 0, 1, -1



On this way we insert data in Avl tree when the BST become unbalanced then we rotate the data and become it balanced

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

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

And we also check 4 cases of the rotation method which makes it balance which are R.R, L.L. L.R and R.L.

Helper method to insert a key.

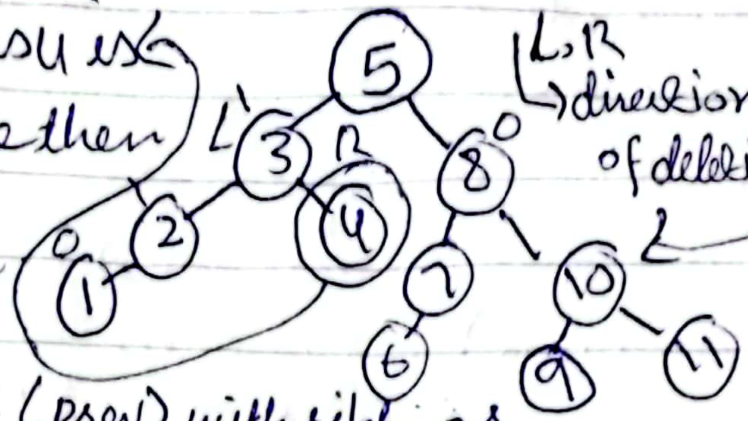
```
public void insert (int k) {
     root = insert(root, k);
  }
```

with time complexity

Delete in AVL tree $\Theta(\log n)$

Delete nodes 4, 8 from the tree given:

These is 4 is leaf node then we direct delete it



L, R direction of deletion

Rules

$R_0 = LL$

$R_1 = LL$

$R_{-1} = LR$

$L_0 = RR$

$L_1 = RL$

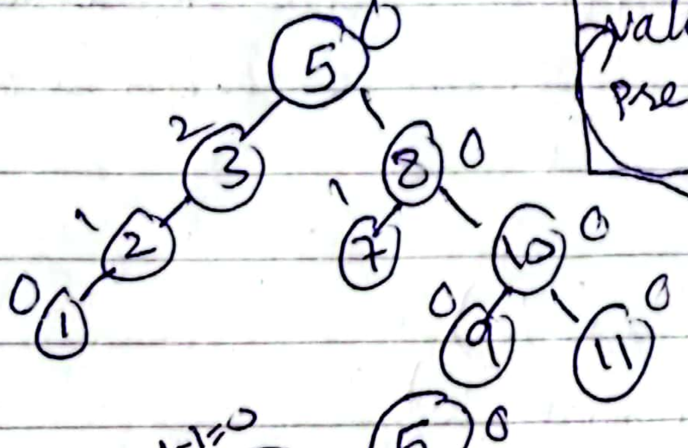
$L_{-1} = RR$

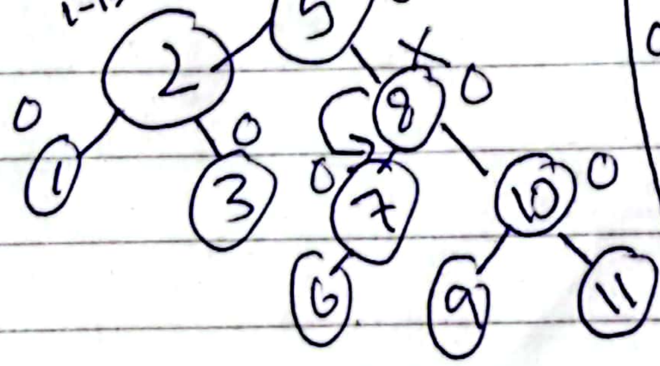
check B.F (prev) with siblings

when we delete leaf node (No child) then we direct delete them.

if the node has one or two child then we use the AVL tree rules.

Delete 4 =→

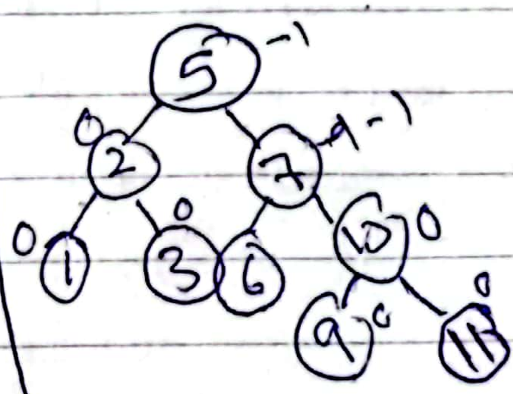




Delete = 8

in this we replace the value of Inorder predecessor.

1 2 3 5 7 8 9 10 11



```
Node delete (Node n, int k){
    if(~~node~~ n==null){ return n }
    if(k < n.ke){
      n.left = delete (n.left, k);
}else if(k > n.k){
    n.right = delete(n.right, k);
}else{
    if((n.left == null) || (n.right==null)){
  Node temp = (n.left!null) ? n.left : n.right;
    if(temp == null){  n = null; }
else{  ~~node~~= temp; }
}else{
    Node temp = getMinValueNode(n.right);
      n.k = temp.k;
      n.right = delete(n.right, temp.k); }}

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

The time complexity of its is $O(\log n)$

and then we check 4 basic operations of balance in this code which are R.R, L.L, R.L, L.R

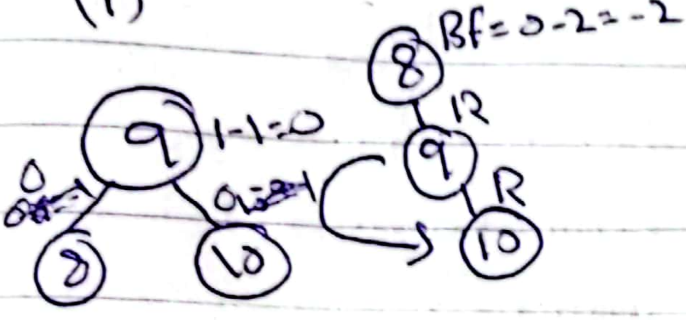
Rotation in a AVL tree

In AVL tree four types of rotation

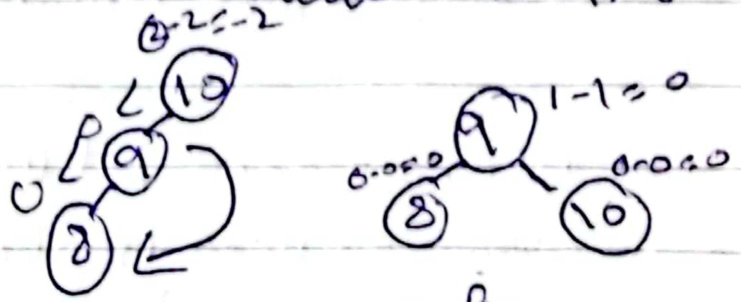
(1) Right to Right :-> 1 rotation
(2) left to left :-> 1 rotation
(3) left to right :-> 2 rotation
(4) Right to left :-> 2 rotation

Data = 8, 9, 10

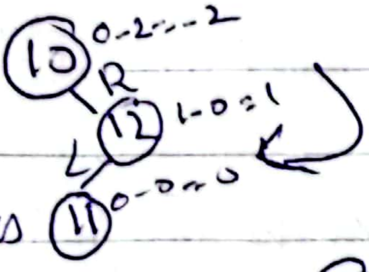
(1)



(2) Data = 10, 9, 8

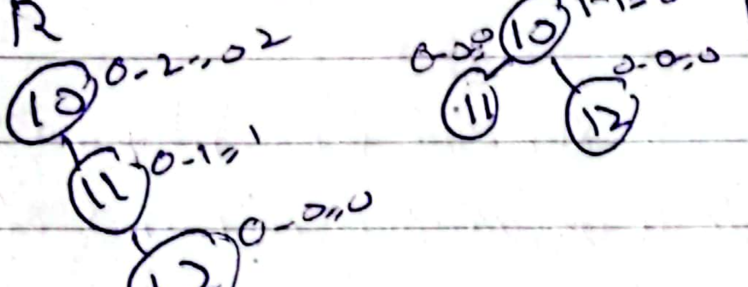


(3) Data = 10, 12, 11



R·L

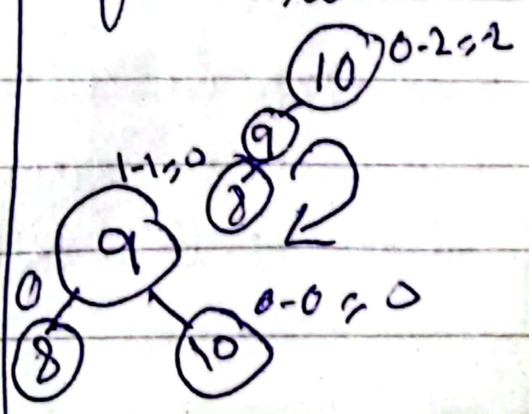
we change this to R·R



Data

(4) 10, 8, 9

L·R first we change this to L·L then we make it self balance.



```
Class AVL tree {
class Node {
  int K; int height;
  Node left, right;
  Node(int K) {
    this.K = k;
    height = 1;

Node rightRotate(Node y) {
    Node x = y.left;
    Node T = x.right;
    x.right = y;
    y.left = T;
    y.height = Math.max(height(y.left), height(y.right))+1;
x.height = Math.max(height(x.left), height(x.right)) +1;
    return x; }
Node leftRotate(Node x) {
  Node y = x.right;
  Node T = y.~~right;~~ left;
  y.left = x;
  x.right = T;
```

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

  int balance = getbalance(n);
   // L.L case
    if(balance>1 && k<n.left.k){
     return rightrotate(n);
     }
   // R.R case
    if(balance<-1 && k>n.right.k){
     return left rotate(n); }
    // L.R case
    if(balance>1 && k>n.left.k){
    n.left= leftrotate(n.left);
      return rightRotate(n);
// R.L case }
    if(balance<-1 && k<n.right.k){
      n.right = rightrotate(n);
     return leftrotate(n);
      } return(n);
```