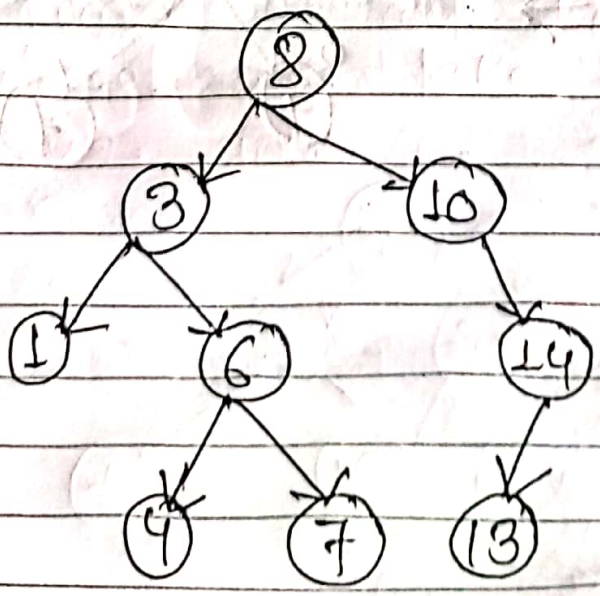
Adelson Velsky & Landis (AVL) Tree

Why do we need an AVL Tree?

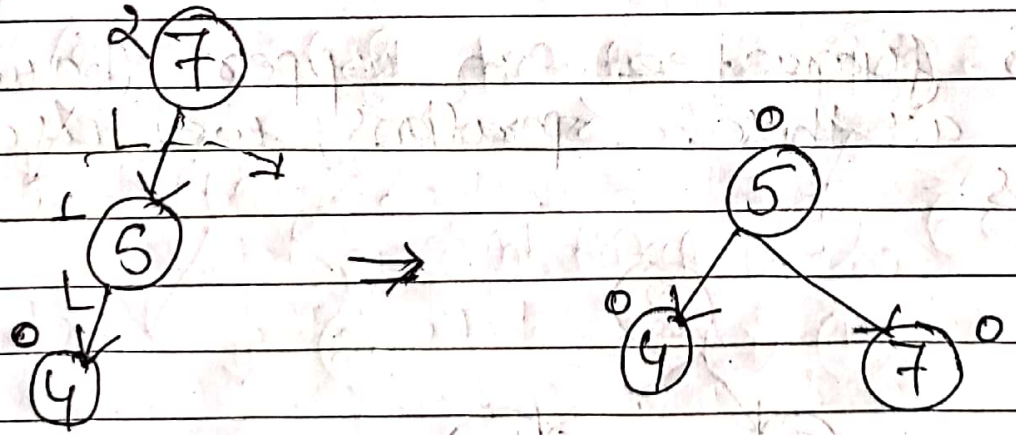
- Almost all the operation in a binary tree of order O(h) where h is the height of the tree.
- If we don't plan our tree properly, this height can get as high as 'n' where n is the number of nodes in a BST.
- To guarantee an upper bound of O(log n) for all these operations, we use balanced trees.



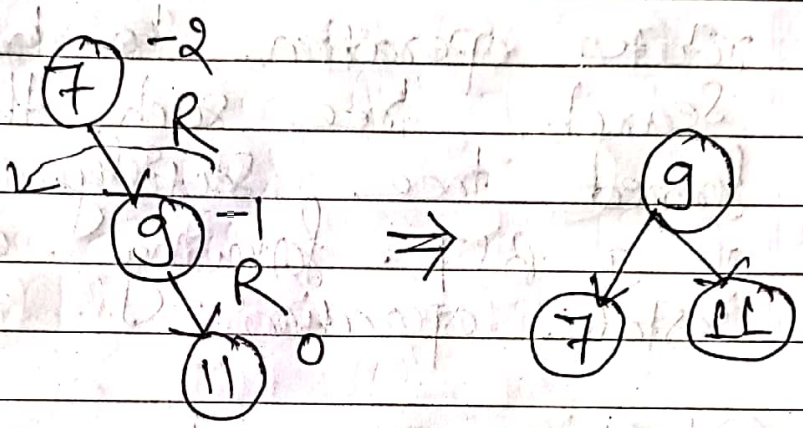
What is an AVL Tree?

- Height Balanced Binary Search trees.
- Balanced factor = Height of left subtree − Height of right subtree
  = H(L) − H(R)
- Balanced factor should be in {−1, 0, 1}.

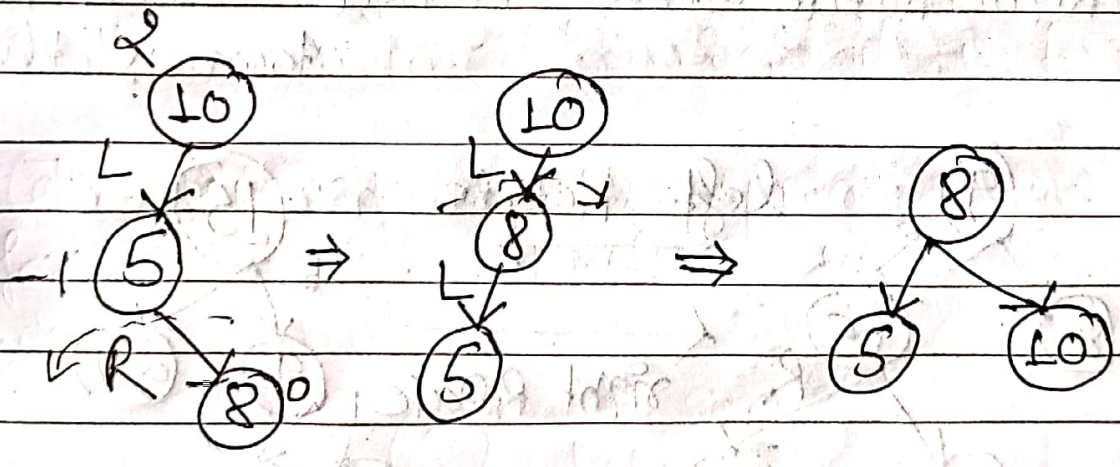
LL Rotation in an AVL



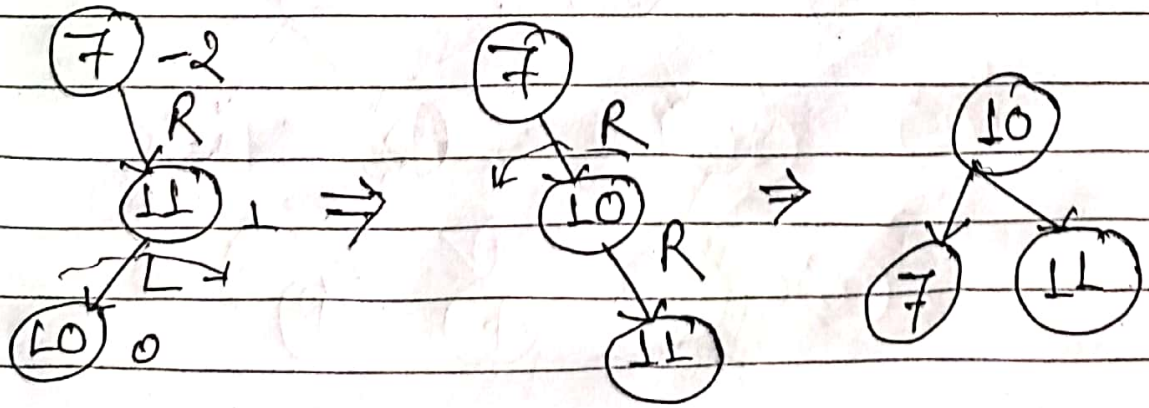
RR Rotation in an AVL



LR Rotation in an AVL



RL Rotation in an AVL

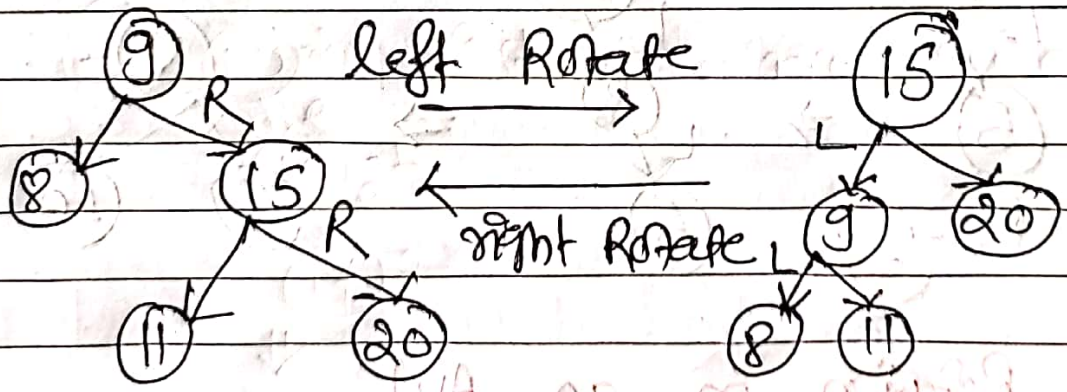


Rotate operations

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

(i) Left Rotate w.r.t a node – node is moved towards the left.

(ii) Right Rotate w.r.t a node – node is moved towards the right.



Balancing a AVL Tree After Insertion

In order to balance an AVL tree after insertion, we can follow the following rules:

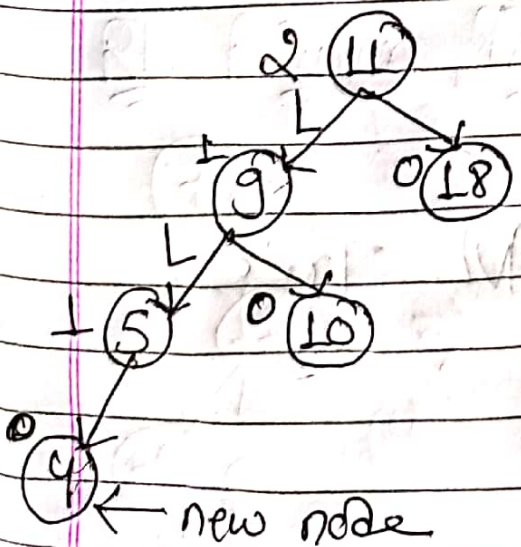
(i). for a left-left insertion → Right rotate once w.r.t the first imbalanced node.

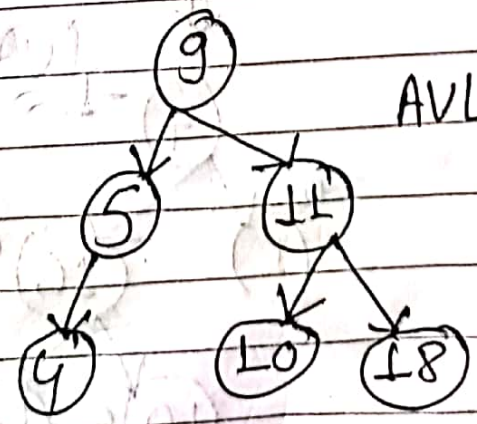
(ii) for a Right-Right insertion → Left rotate once w.r.t first imbalanced node.

(iii) for a Left-Right insertion → left rotate once and then right rotate once.

(iv) for a Right-Left insertion → Right rotate once and then Left rotate once.

Left Left Insertion

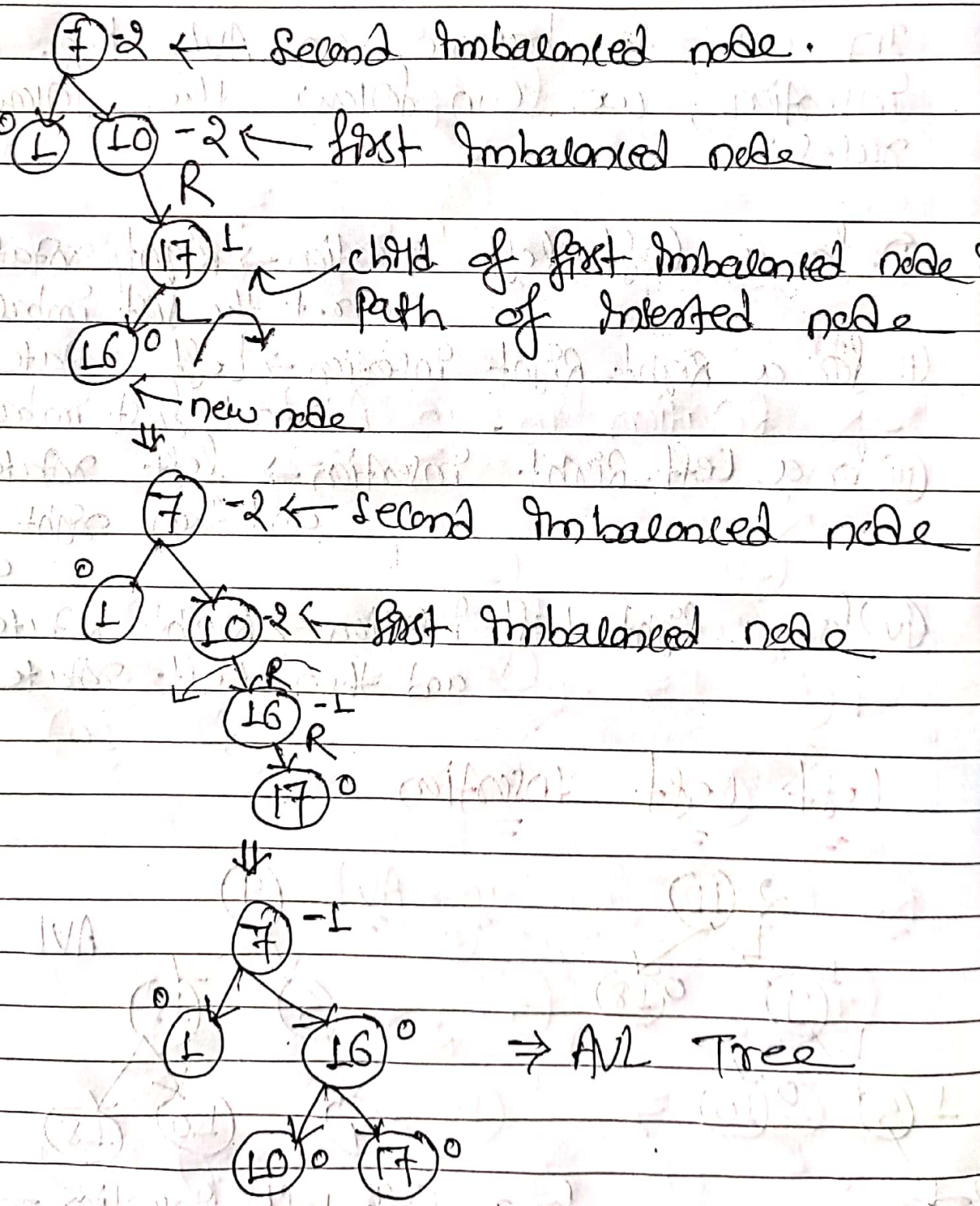




AVL Tree

for a Left-Left insertion → Right rotate once w.r.t the first imbalanced node.

Right_Left Insertion

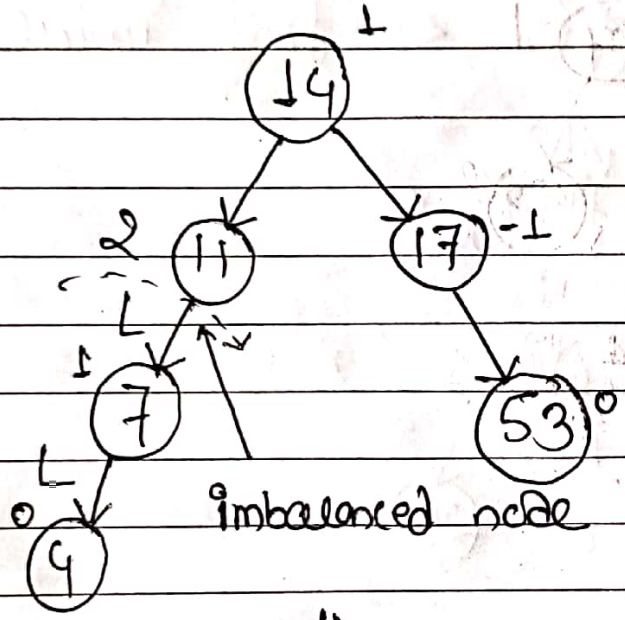


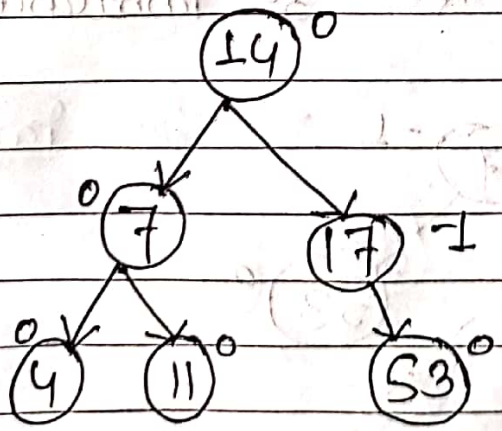
⇒ AVL Tree

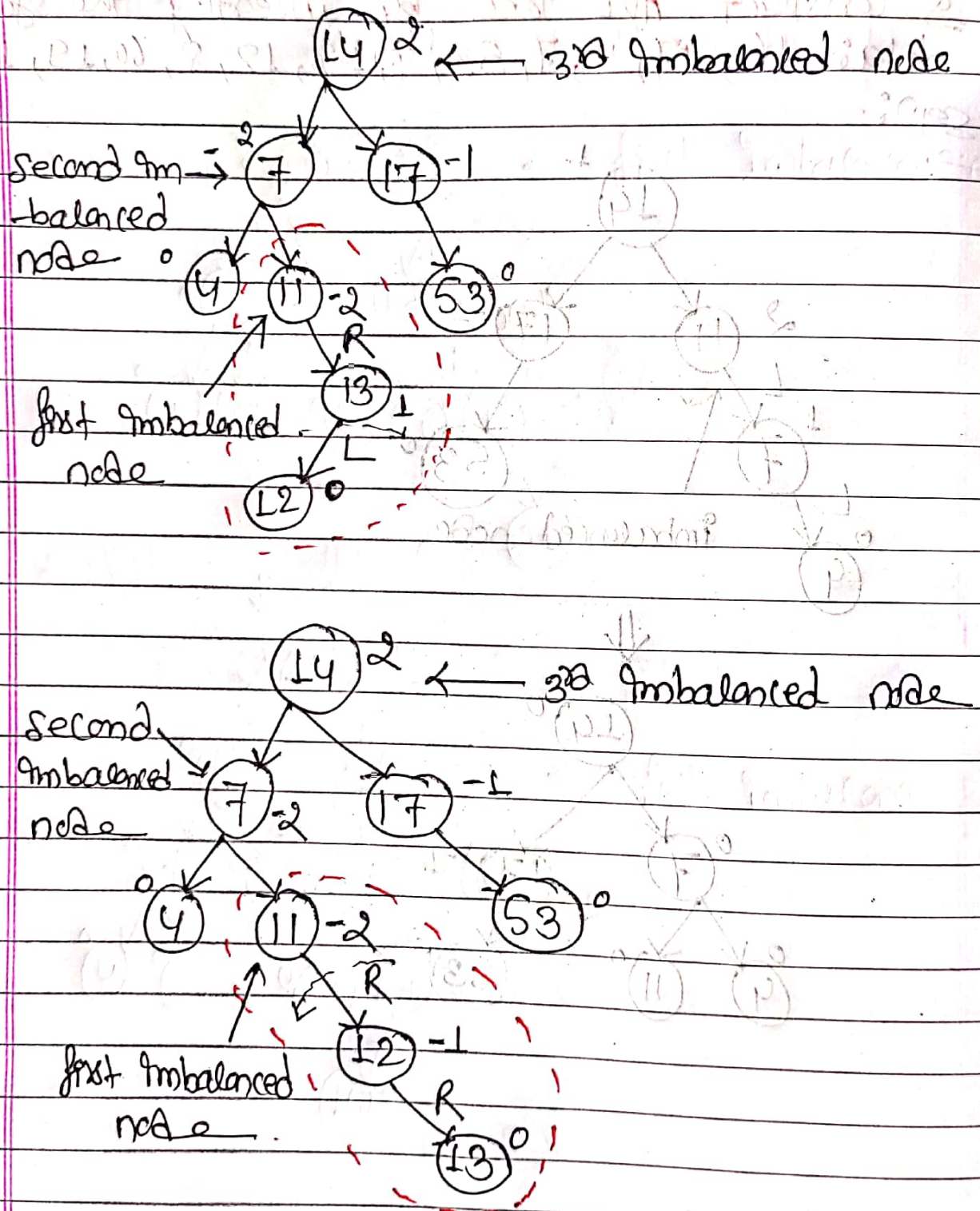
Q. Construct AVL tree by inserting the following data:- 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20.

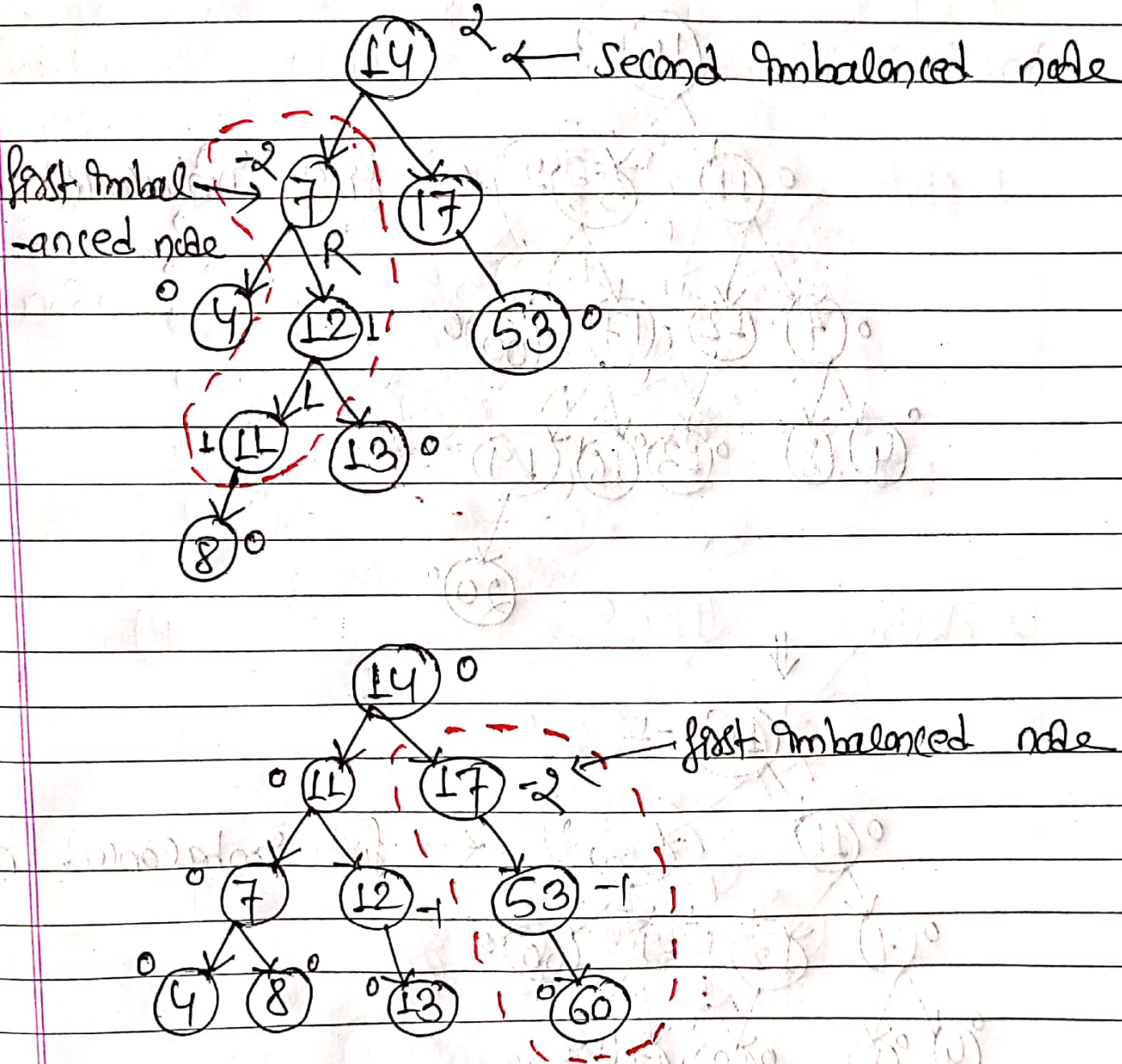
Soln:-

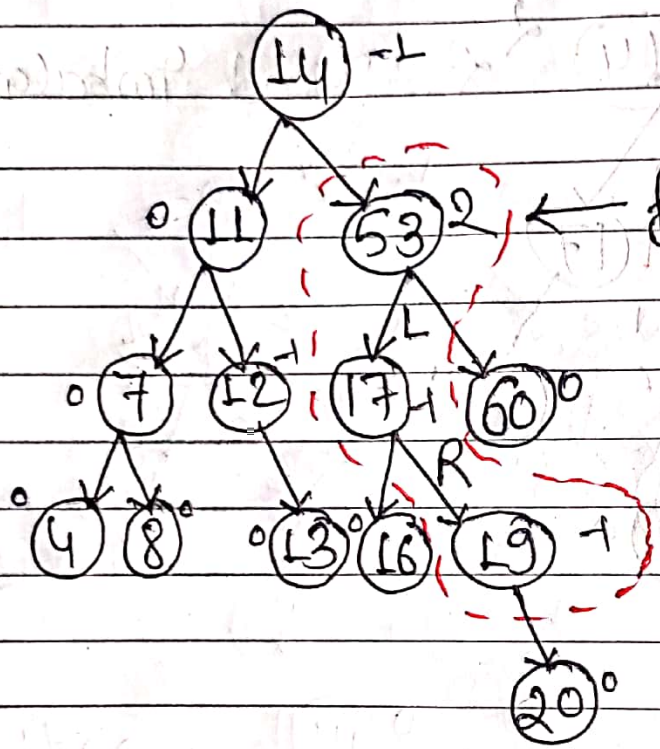
→

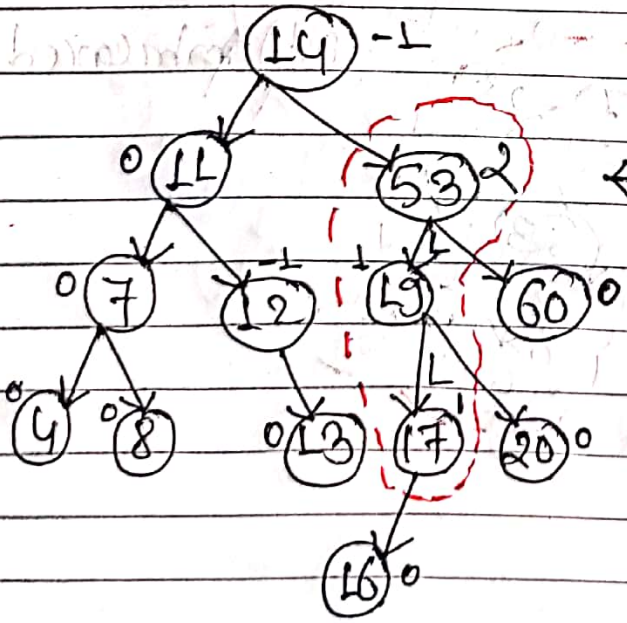


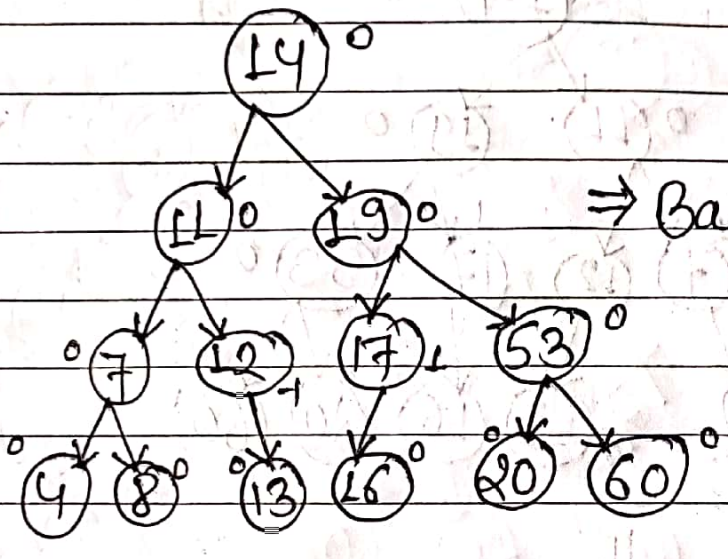










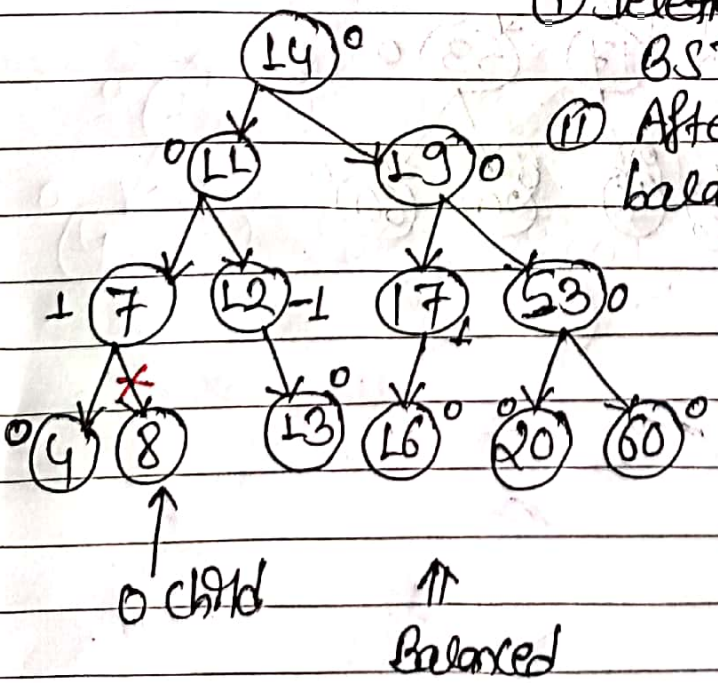


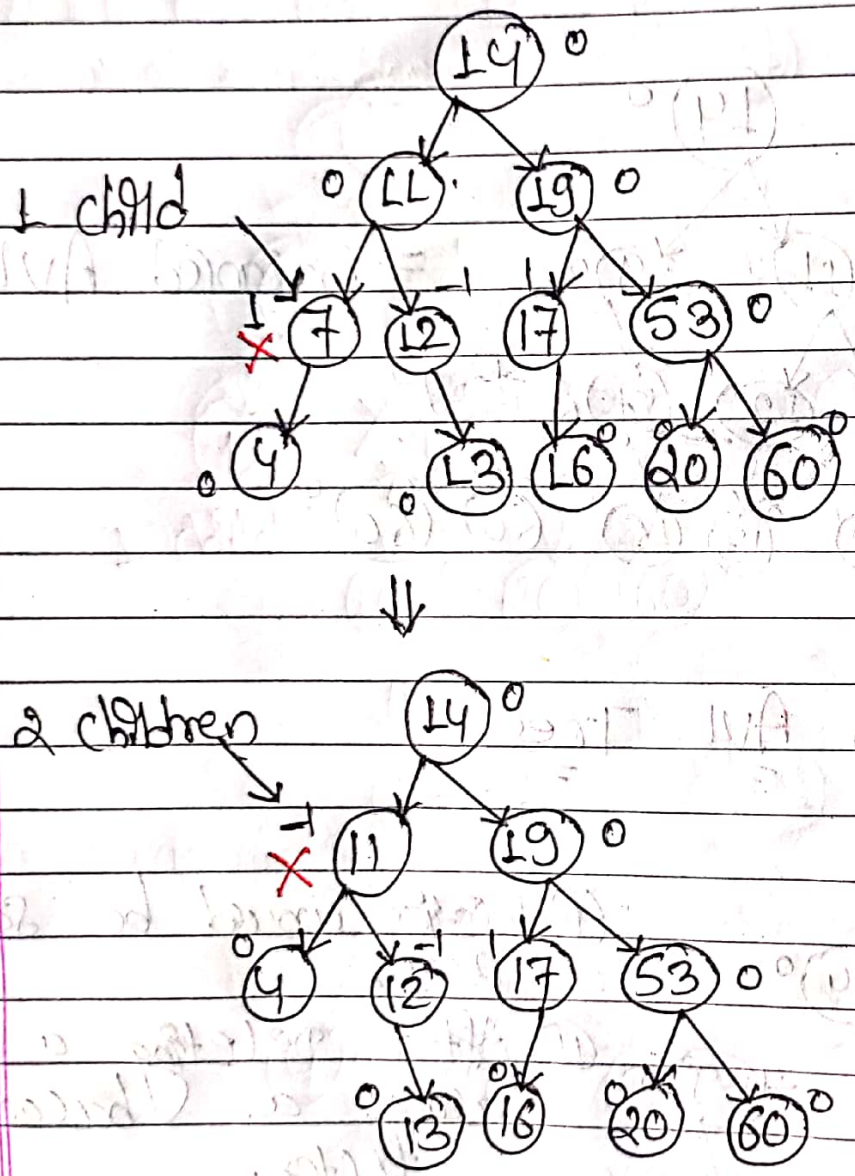
⇒ Balanced AVL Tree

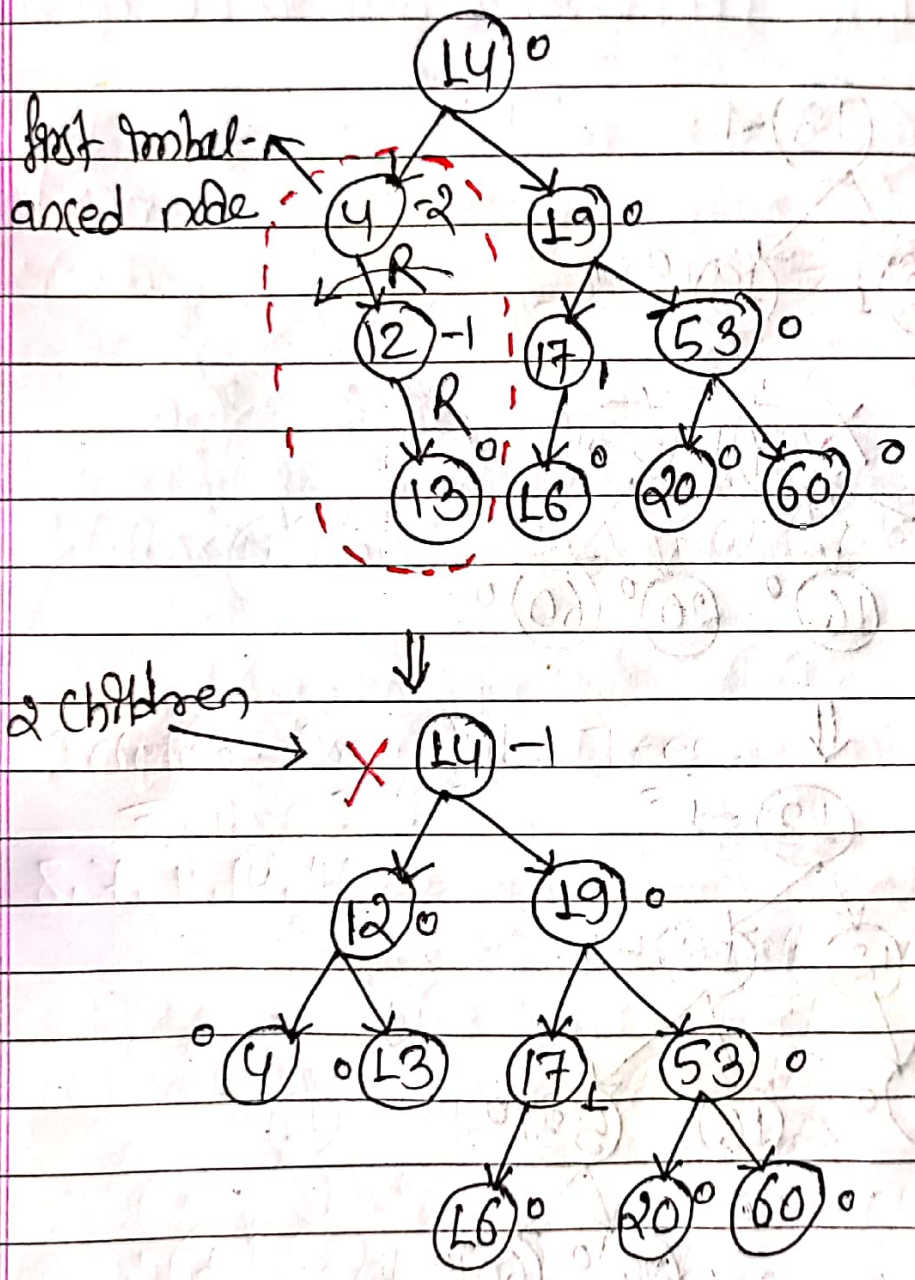
Deletion in AVL Tree

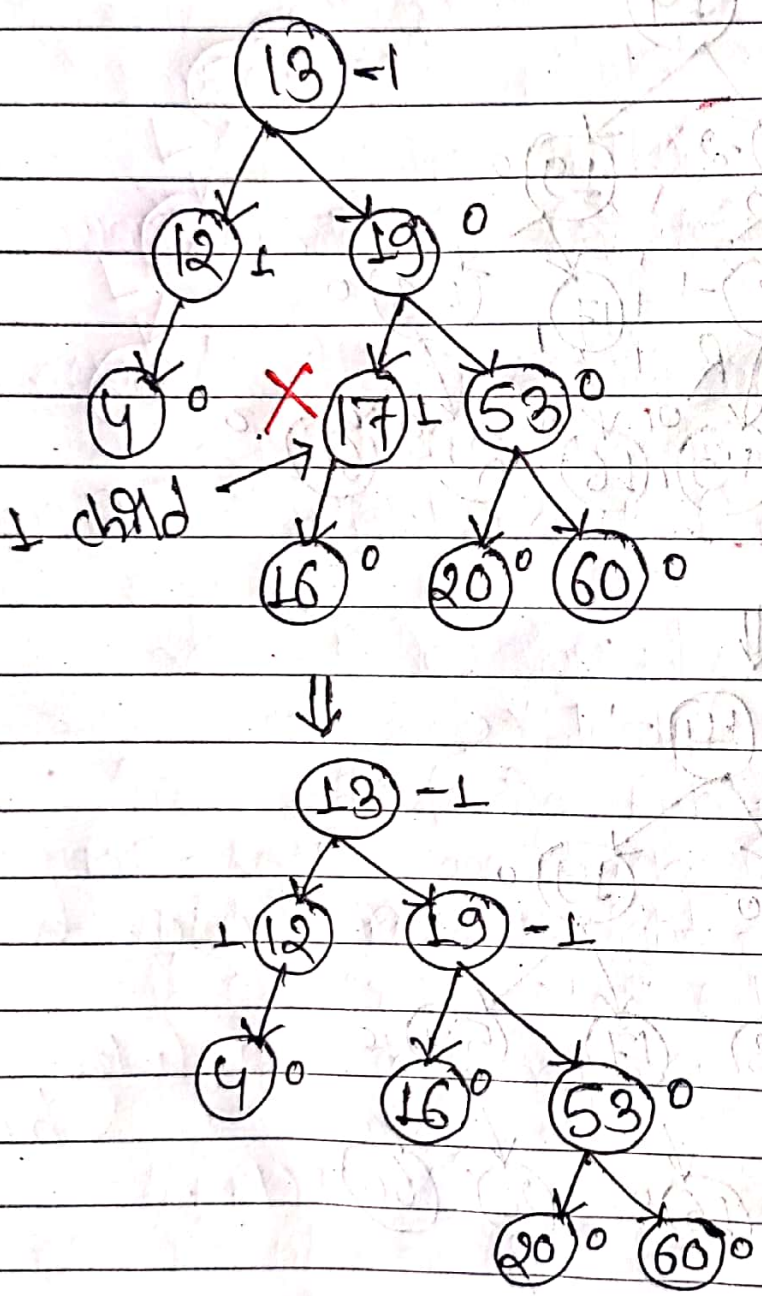
8, 7, 11, 14, 17

(i) Deletion would be same as BST.
(ii) After deleting a node balance a balance factor.









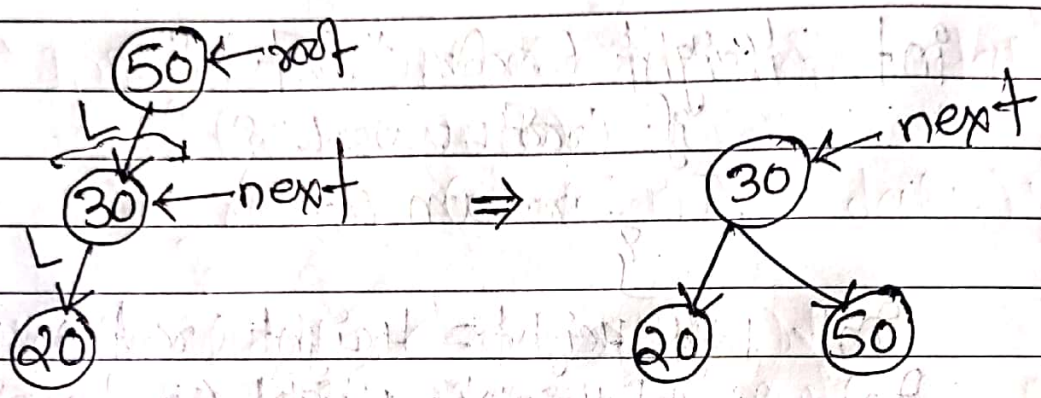
// To Calculate the Height of a node

```
int Height (node *root){
    if (root==NULL){
        return 0;
    }
    int Left_Height = Height (root->left);
    int Right_Height = Height (root->right);
    return max (Left_Height, Right_Height)+1;
}
```

// Balance factor of a node {-1, 0, 1}

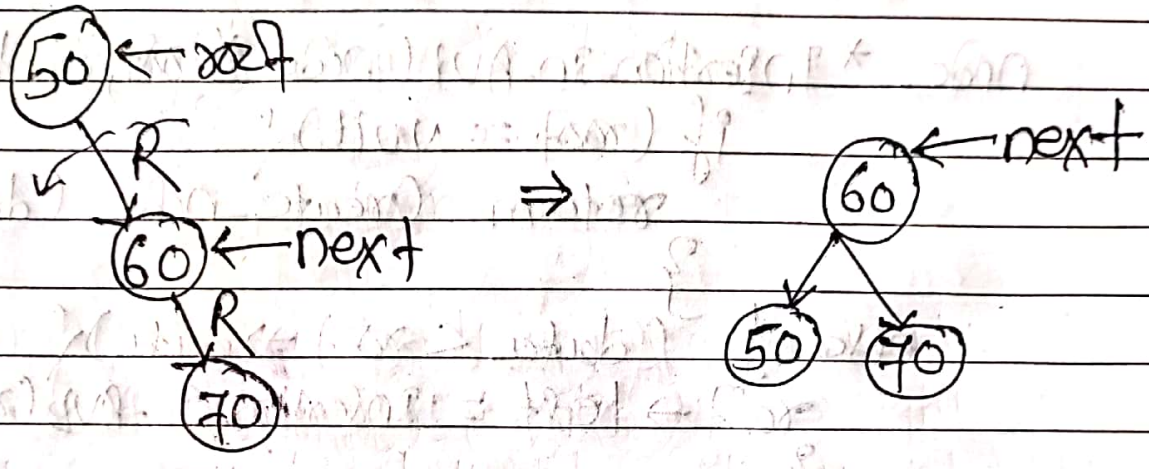
```
int Get_Balance_Factor (node *root){
    return Height (root->left) - Height (root->right);
}
```

// LL Rotation (Unbalance due to LL operation)




```
node * Right_Rotate ( node *root ){
    node *next = root -> left ;
    node * temp = next -> right;
    //performing Right Rotation
    next -> right = root;
    root -> left = temp ;
    return next;
}
```

// RR Rotation (unbalance due to RR operation)




```
node *Left_Rotate( node *root ){
    node *next = root->right;
    node *temp = next->left;
    // performing Left Rotation
    next->left = root;
    root->Right = temp;
    return next;
}
```

// Insertion In AVL

```
node *Insertion_In_AVL(node *root, int data){
    if (root == NULL){
        return (create_node (data));
    }
else if (data < root->data){
    root->left = Insertion_In_AVL (root->left, data);
}
else{
    root->right = Insertion_In_AVL(root->right, data);
}
int BF;
BF = Get_Balance_Factor (root);
// LL Rotation
if (BF > 1 && data < root->left->data){
    return Right_Rotate (root);
}
// RR Rotation
else if (BF < -1 && data > root->right->data){
    return Left_Rotate (root);
}
// RL Rotation
else if (BF < -1 && data < root->right->data){
    root->right = Right_Rotate (root->right);
```

```
        return Left_Rotate(root);
    }
// LR Rotation
    else if(BF>1 && data>root->left->data){
        root->left = Left_Rotate(root->left);
        return Right_Rotate(root);
    }

    return root;
}
```

Deletion In AVL Tree

```
node *Deletion_In_AVL(node *root, int key){
    if (root == NULL){
        return NULL;
    }
    else if (key < root->data){
        root->left = Deletion_In_AVL(root->left, key);
    }
    else if (key > root->data){
        root->right = Deletion_In_AVL(root->right, key);
    }
    else {
        // Case 1 :- 0 child
        if (root->left==NULL && root->right==NULL){
            free(root);
            root = NULL;
            return root;
        }
        // Case 2 :- 1 child
        else if (root->left == NULL){
            node *p = root;
            root = root->right;
            free(p);
            return root;
        }
```

```
else if (root->right == NULL){
        node *p = root;
        root = root->left;
        free(p);
        return root;
    }
    else{
    // Case 3 :- 2 children
     node *I_p = Inorder_predecessor(root);
        root->data = I_p->data;
     root->left = Deletion_In_AVL(root->left, I_p->data);
    }
}

int BF;
BF = Get_Balance_Factor(root);
// LL Rotation
if (BF > 1 && Get_Balance_factor(root->left)>=0){
        return Right_Rotate(root);
    }
// RR Rotation
else if (BF<-1 && Get_Balance_factor(root->right)<=0){
        return Left_Rotate(root);
    }

// LR Rotation
else if (BF>1 && Get_Balance_factor(root->left)==-1){
```

```
root->left = Left_Rotate(root->left);
return Right_Rotate(root);
}
// RL Rotation
else if (BF<-1 && Get_Balance_factor(root->right)
                                    == 1){
root->right = Right_Rotate(root->right);
return Left_Rotate(root);
}
return root;
}
```

Advantages of AVL Trees

- The height of the AVL tree is always balanced the height never grows beyond $\log N$, where N is the total number of nodes in the tree.
- It gives better search time Complexity when Compared to simple Binary search trees.
- AVL trees have self-balancing capabilities.