AVL Trees

Trees with a worst case height of $O(\log_2 n)$ are called balanced trees.
One of the popular balanced trees is AVL tree, which was introduced by Adelson-Velskii and Landis.

If T is a non-empty binary tree with T_L and T_R as its left and right subtree, Then T is an AVL tree if and only if.

1) $|h_L - h_R| \leq 1$, where h_L and h_R are the heights of T_L and T_R respectively and

2) T_L and T_R are AVL trees.

An AVL search tree is a binary search tree that is also an AVL tree. The height of an empty AVL tree is taken as -1.

Complexity of an AVL tree: O(height) = O(log n)

* Representation of an AVL tree

The node in an AVL tree will have additional field bf (for balance factor) in addition to the structure of a node in a binary search tree.

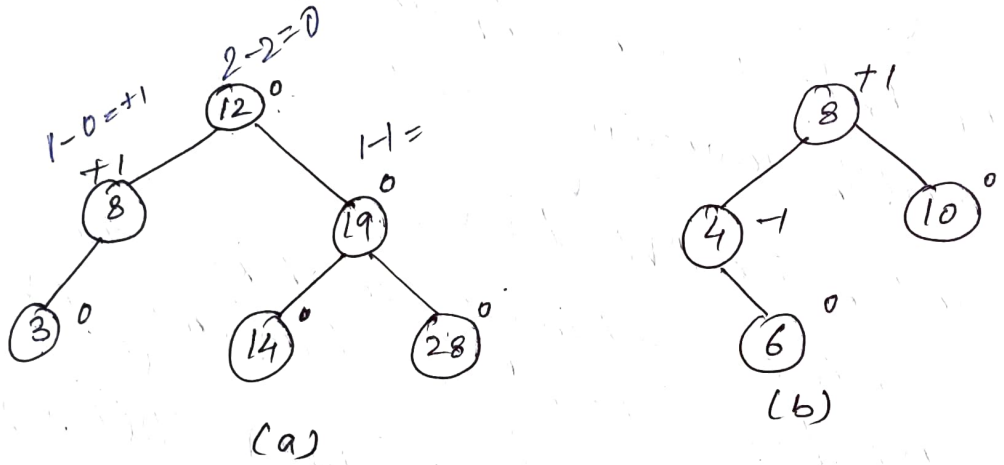
The required memory locations of declarations for representing a node in an AVL tree will look like:

```
typedef struct nodeType {
        struct nodeType *left;
        int info;
        int bf;
```

```
struct nodeType *right;
} avlnode;
avlnode *root;
```

The value for field bf will be chosen as

$$bf = \begin{cases} -1 & \text{if } h_L < h_R \\ 0 & \text{if } h_L = h_R \\ +1 & \text{if } h_L > h_R \end{cases}$$



AVL trees

A binary tree is said to be balanced if the difference between the heights of left and right subtrees of every node in the tree is either −1, 0, +1.

In an AVL tree, every node maintains an extra information known as **balance factor**.

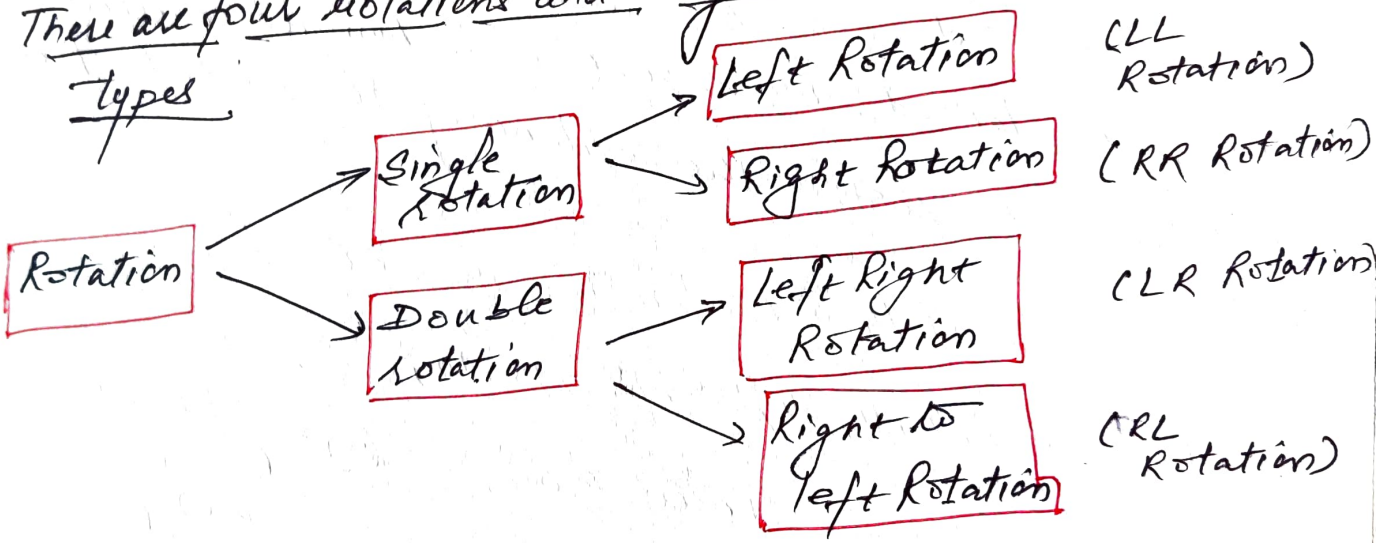
> Balance factor = height of Left subtree − height of right subtree.

AVL Tree Rotation

In AVL tree, after performing operations like insertion & deletion we need to check the balance factor of every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are four rotations and they are classified into two types



Problem	solution
LL	R
RR	L
LR	LR
RL	RL

Searching an AVL Search Tree

Searching an AVL search tree for an element is exactly similar to the method used in a binary search tree.

Insertion in an AVL Search tree

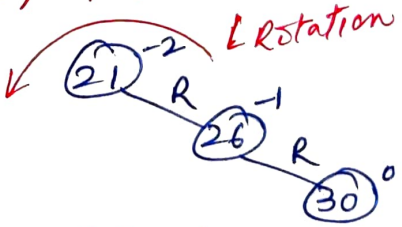
Inserting an element into an AVL search tree in its first phase is similar to that of the one need in a binary search tree. However, if after insertion of an element, the balance factor of any node in the tree is affected so as to render the binary search tree unbalanced, we resort to technique called Rotations to restore the balance of the search tree.

Insert the following elements in AVL tree.

21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7

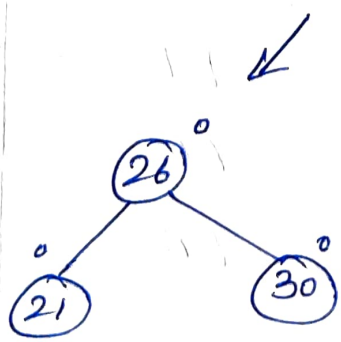
Insert 26

Insert 30

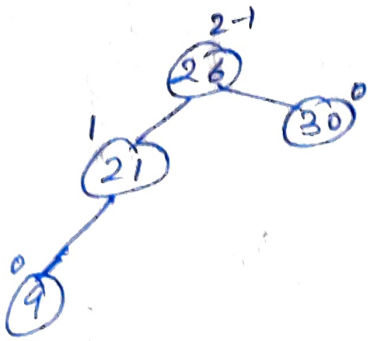


RR → L

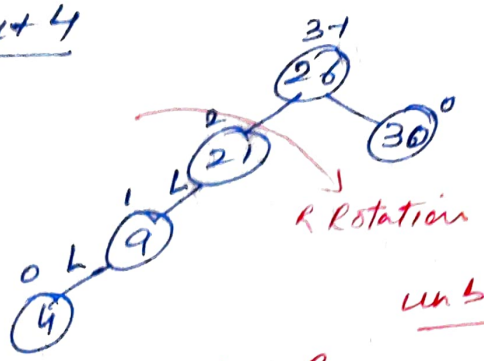
unbalance



Insert 9

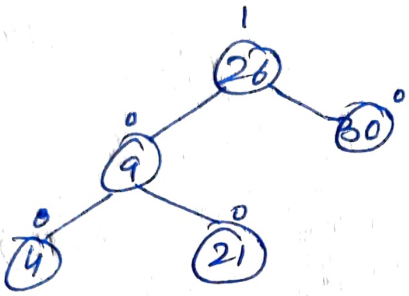


Insert 4

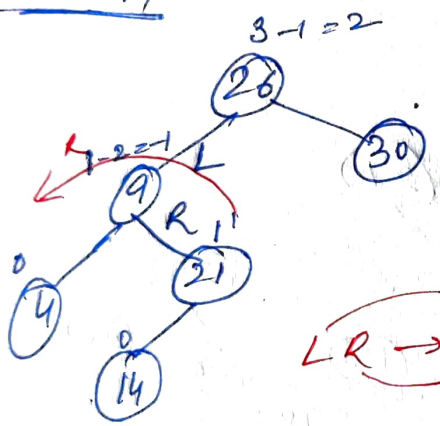


unbalance

LL → R

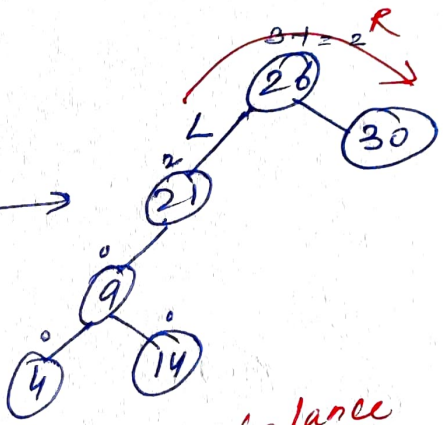


Insert 14

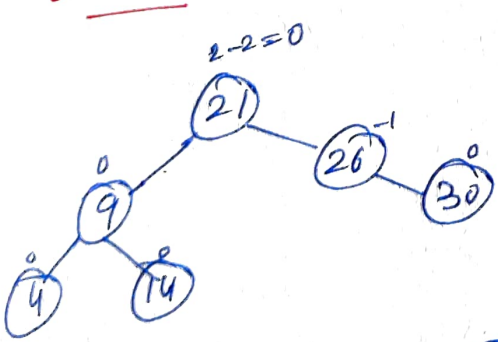


unbalance

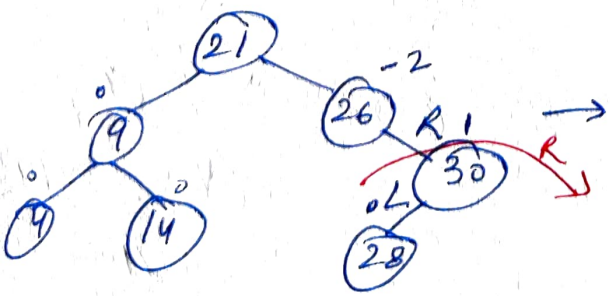
LR → LR

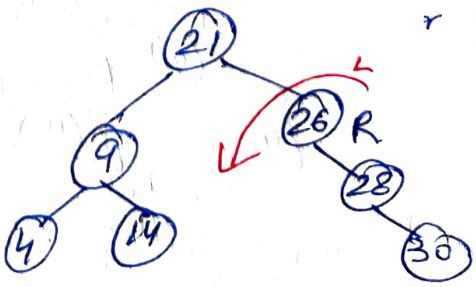


unbalance



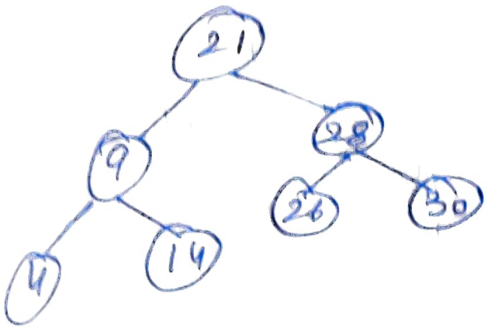
Insert 28



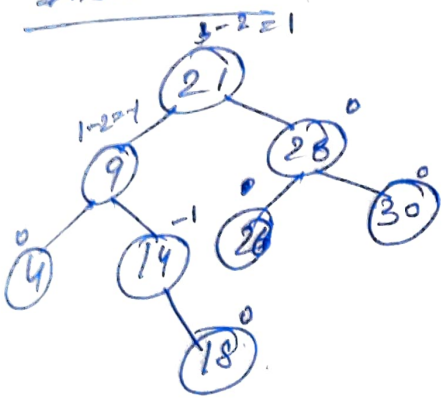


RL → RL

unbalance

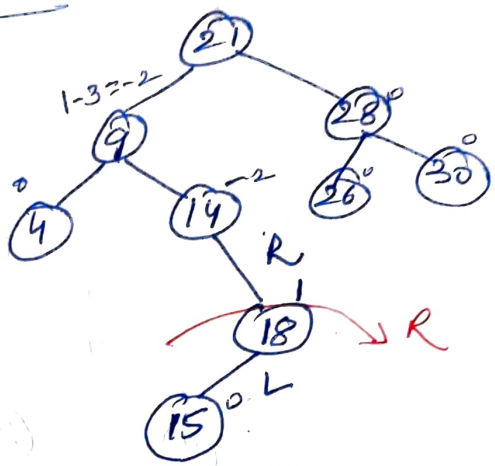


Insert 18



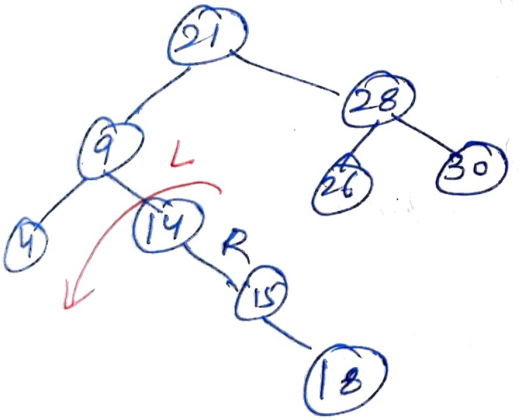
balance

Insert 15

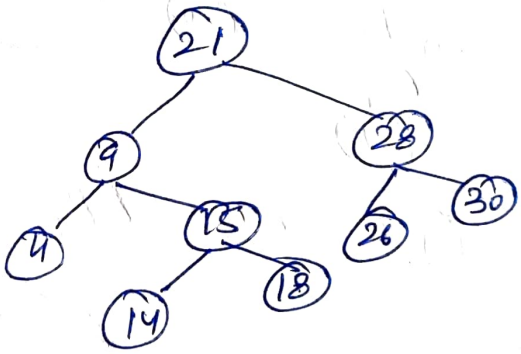


LR → LR

unbalance

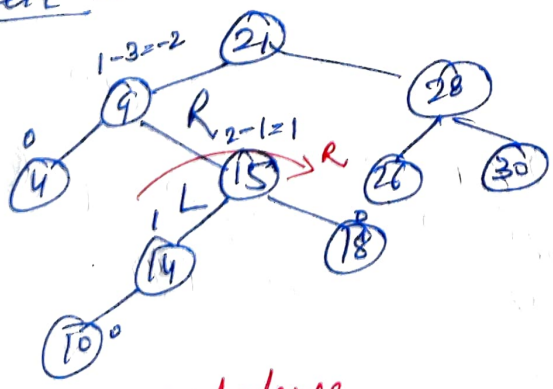


unbalance

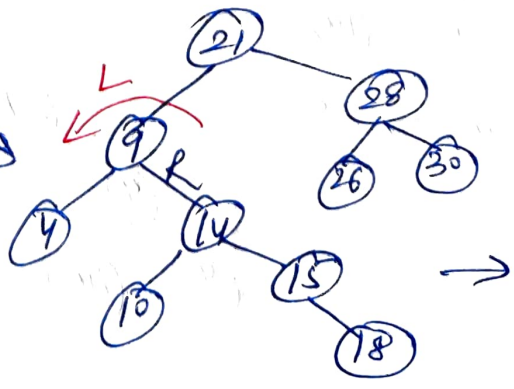


balance

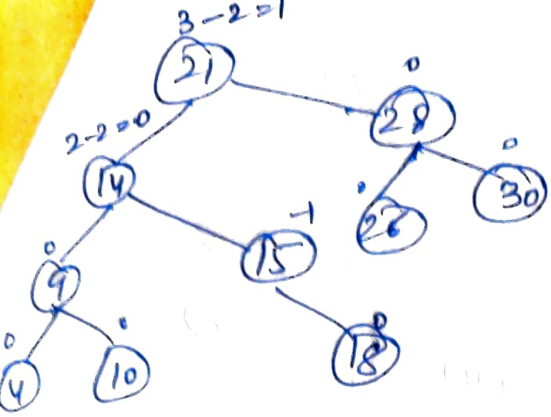
Insert 10



unbalance

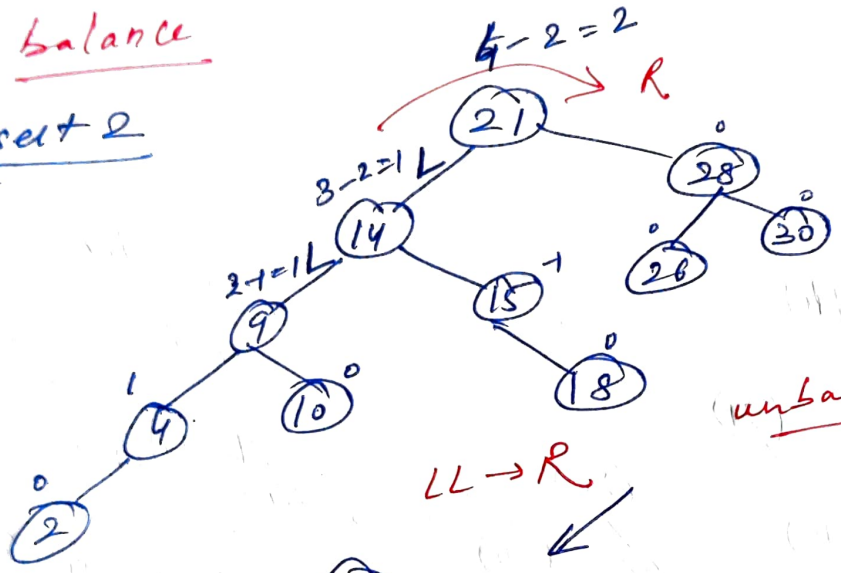


unbalance



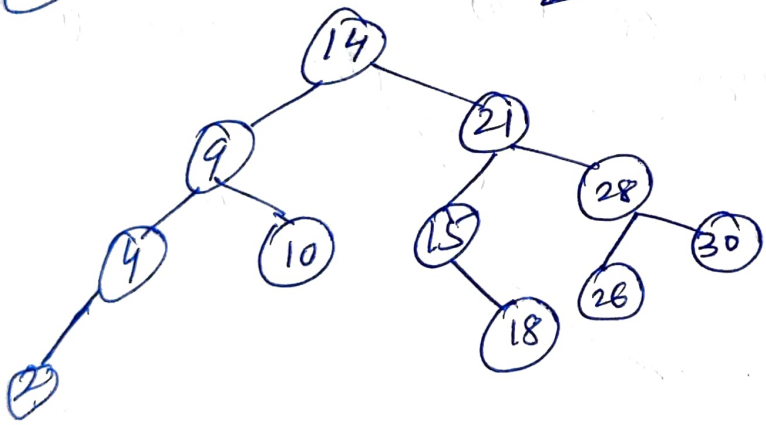
Balance

Insert 2

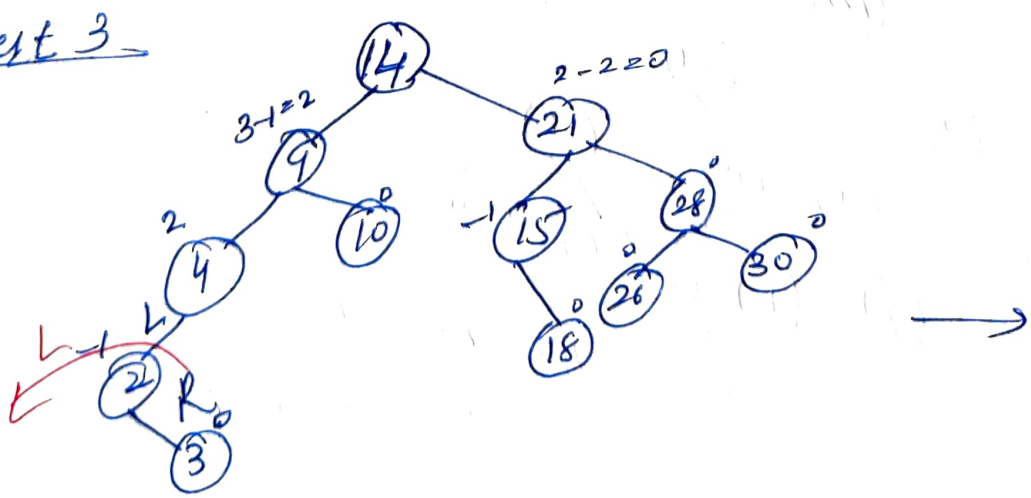


unbalance

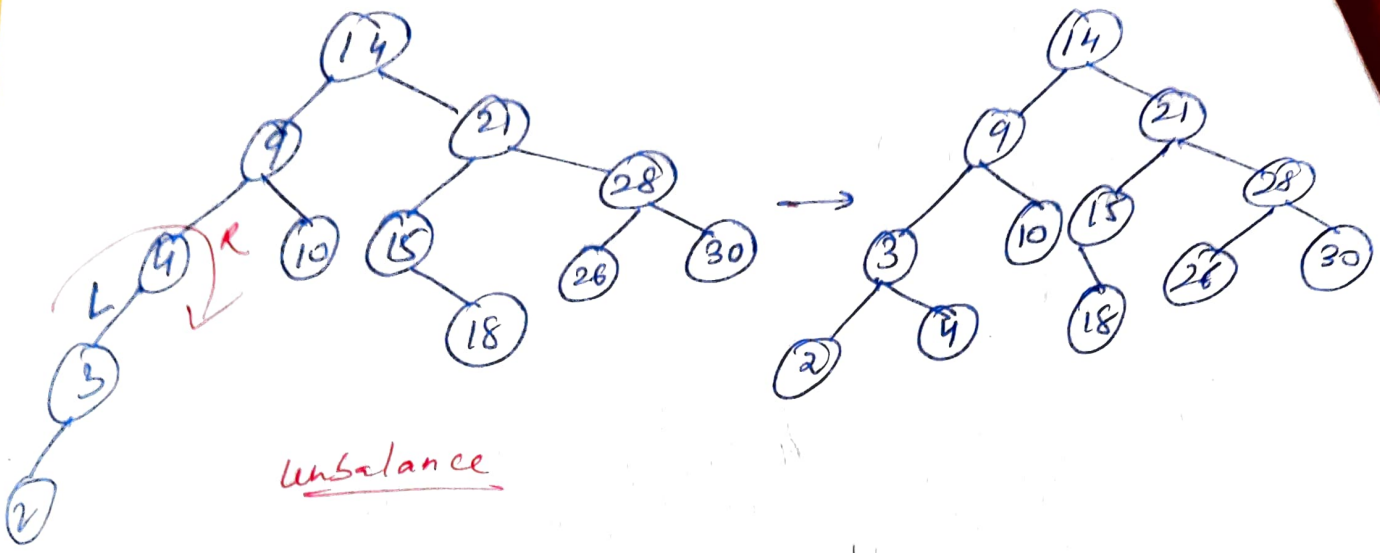
LL → R



Insert 3

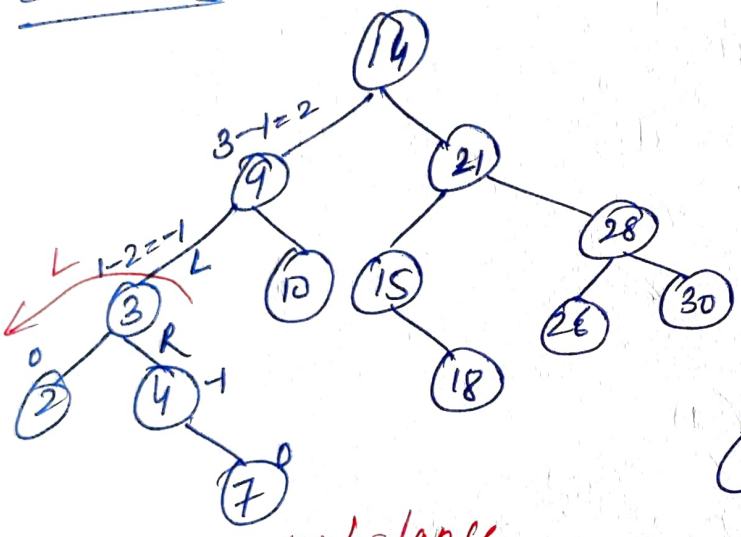


unbalance

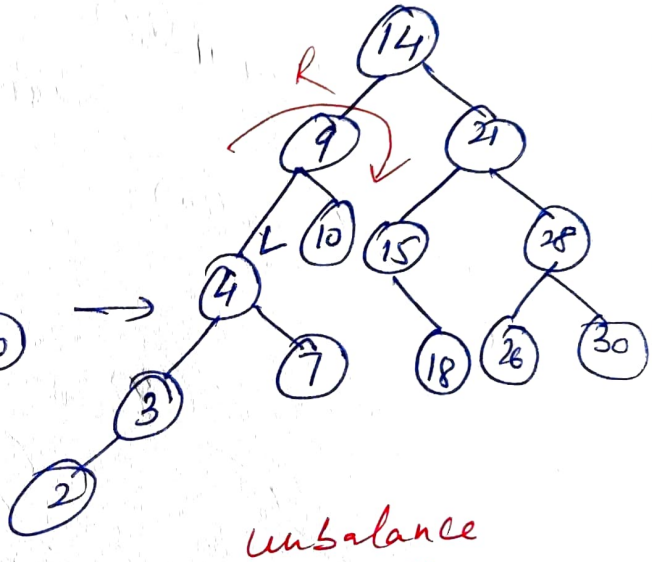


unbalance

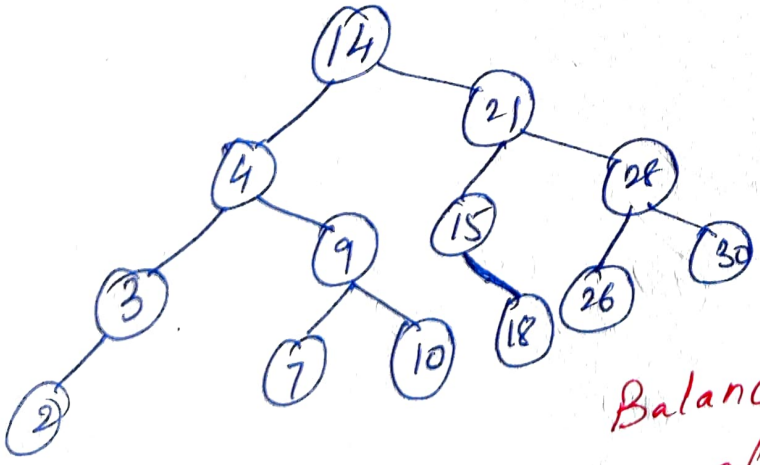
Insert 7



unbalance



unbalance

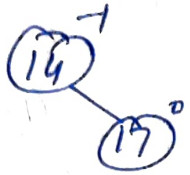


Balanced BST
also AVL tree

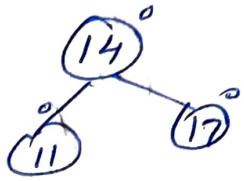
Insert 14, 17, 11, 7, 53, 4, 13, 12 into an empty AVL tree. (5)

(1) Insert 14

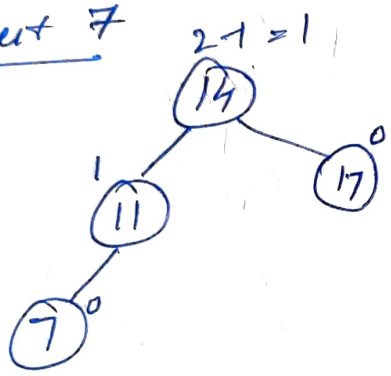
(2) Insert 17



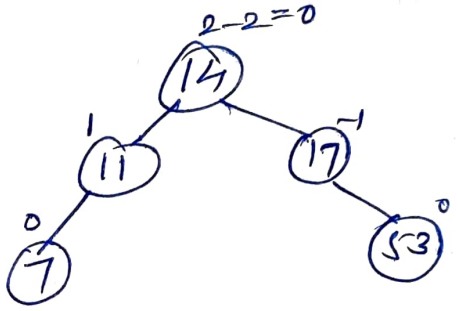
(3) Insert 11



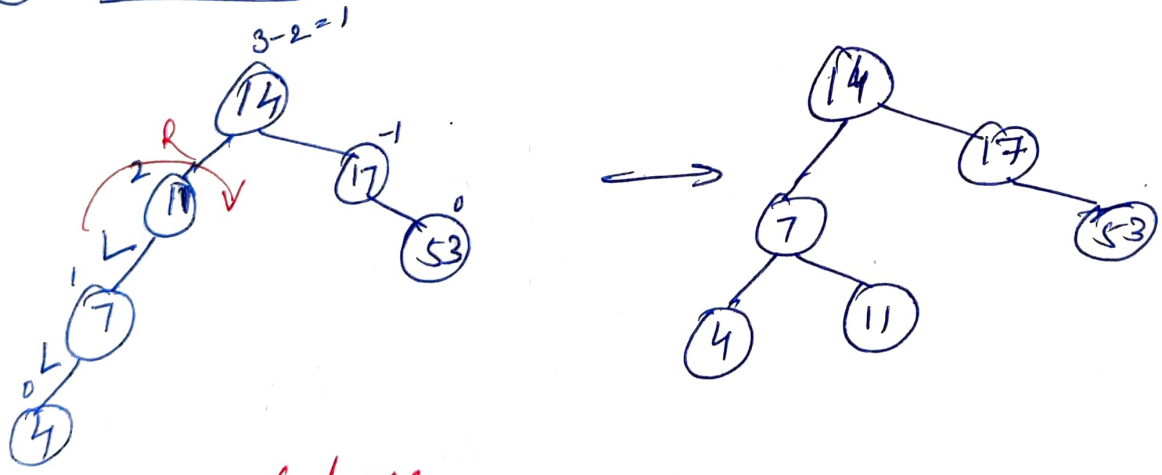
(4) Insert 7



(5) Insert 53



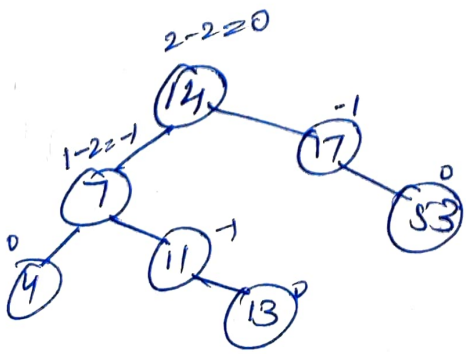
(6) Insert 4



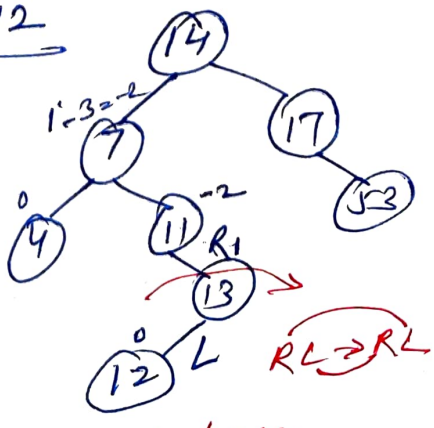
unbalance

LL → R

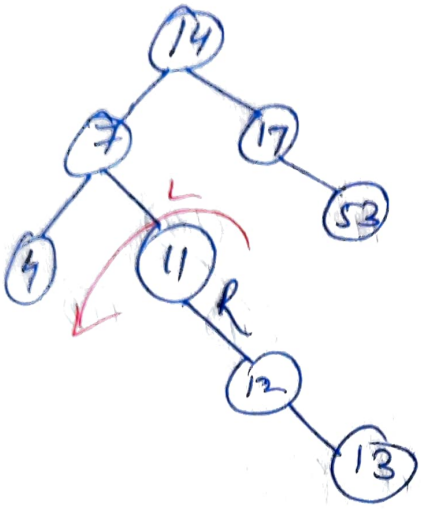
(7) Insert 13



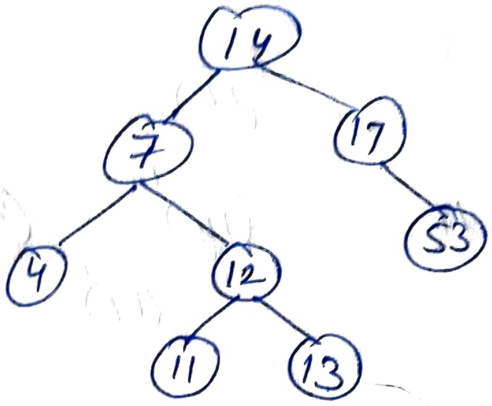
(8) Insert 12



unbalance



unbalance



Balanced
AVL tree