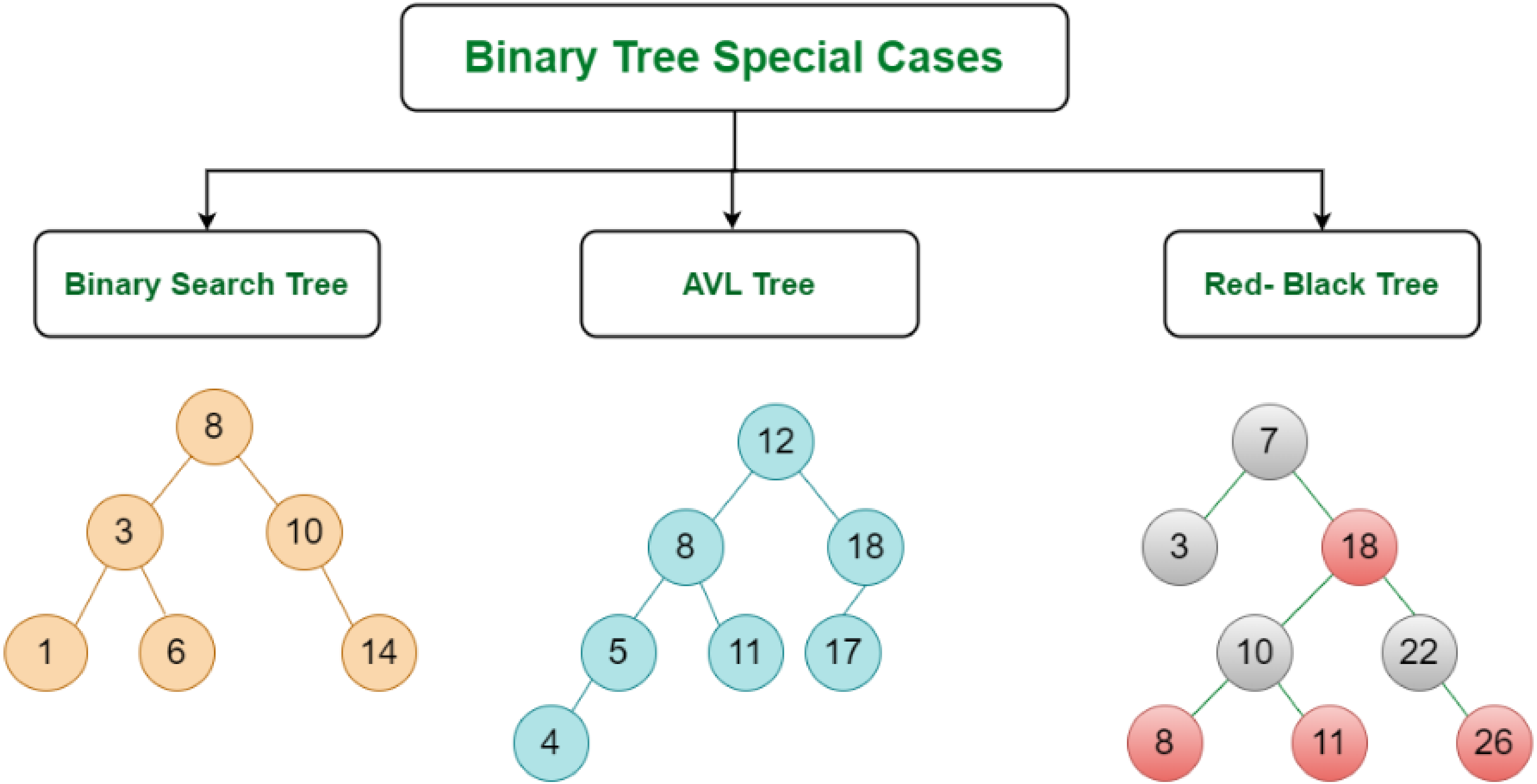
Trees

OPERATIONS ON TREES

Traversing a Binary Tree

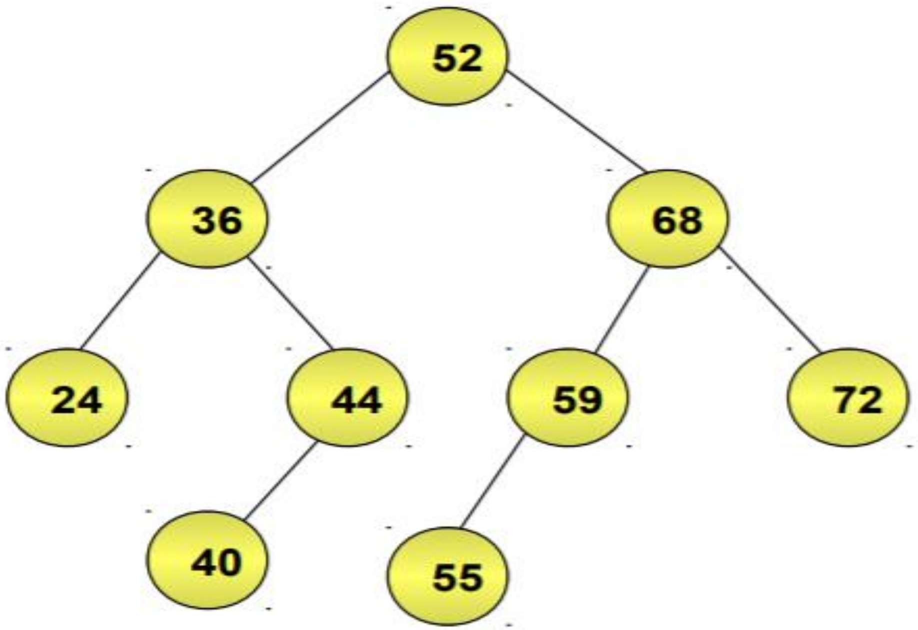
1)TRAVERSING

- You can implement various operations on a binary tree.
- A common operation on a binary tree is traversal.
- Traversal refers to the process of visiting all the nodes of a binary tree once.
- There are three ways for traversing a binary tree:
  - Inorder traversal
  - Preorder traversal
  - Postorder traversal


Binary Tree Special Cases
Binary Search Tree
AVL Tree
Red- Black Tree
8
3
10
1
6
14
12
8
18
5
11
17
4
7
3
18
10
22
8
11
26

Binary Search Tree

- Binary search tree is a binary tree in which every node satisfies the following conditions:
  - All values in the left subtree of a node are less than the value of the node.
  - All values in the right subtree of a node are greater than the value of the node.
- The following is an example of a binary search tree.



Operations on a Binary Search Tree

- The following operations are performed on a binary earch tree...
  - Search
  - Insertion
  - Deletion
  - Traversal

Insertion of a key in a BST

```
Algorithm:- InsertBST (info, left, right, root, key, LOC)
{
    key is the value to be inserted.
    1. call SearchBST ( info, left, right, root, key, LOC , PAR )    // Find the parent of the new node
    2. If ( LOC != NULL)
    2.1 Print " Node alredy exist"
    2.2 Exit
    3. create a node [ new1 =  ( struct node*) malloc ( sizeof( struct node) ) ]
    4. new1 -> info = key
    5. new1 -> left = NULL , new1 -> right = NULL
    6. If ( PAR = NULL ) Then
    6.1 root = new1
    6.2 exit
      elseif ( new1 -> info < PAR -> info)
    6.1 PAR -> left = new1
    6.2 exit
      else
    6.1 PAR -> right = new1
    6.2 exit
 }
```

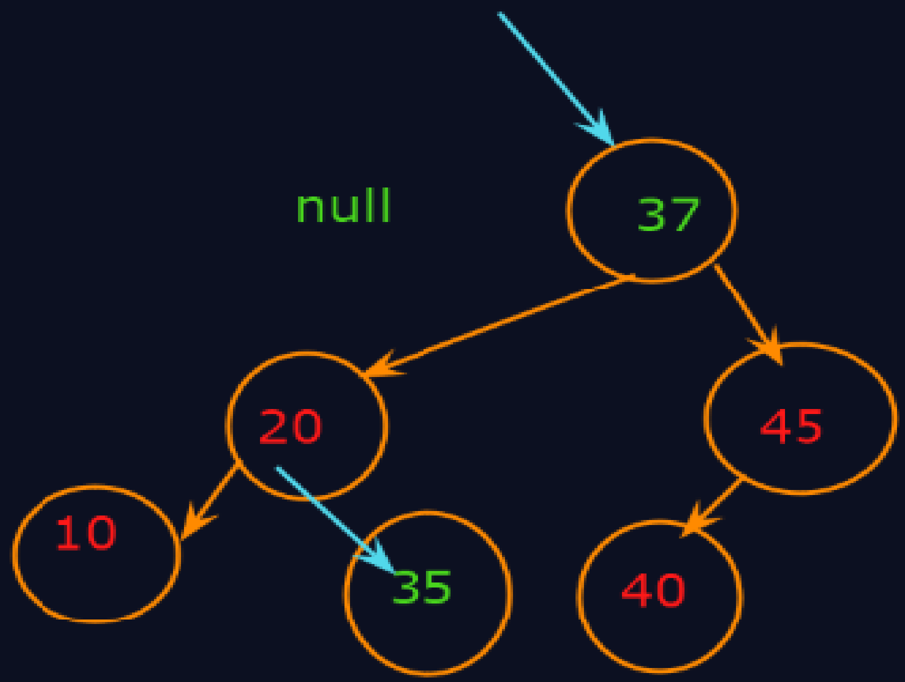
```java
class BST{
    Node root;//starting point of tree

    static class Node{
    int data;
    Node left,right;

    Node(int d)
    {
        data = d;
        left=right=null;
    }
    }

    BST()
    {
        root = null;
    }

    BST(int d)
    {
        root = new Node(d);
    }
```

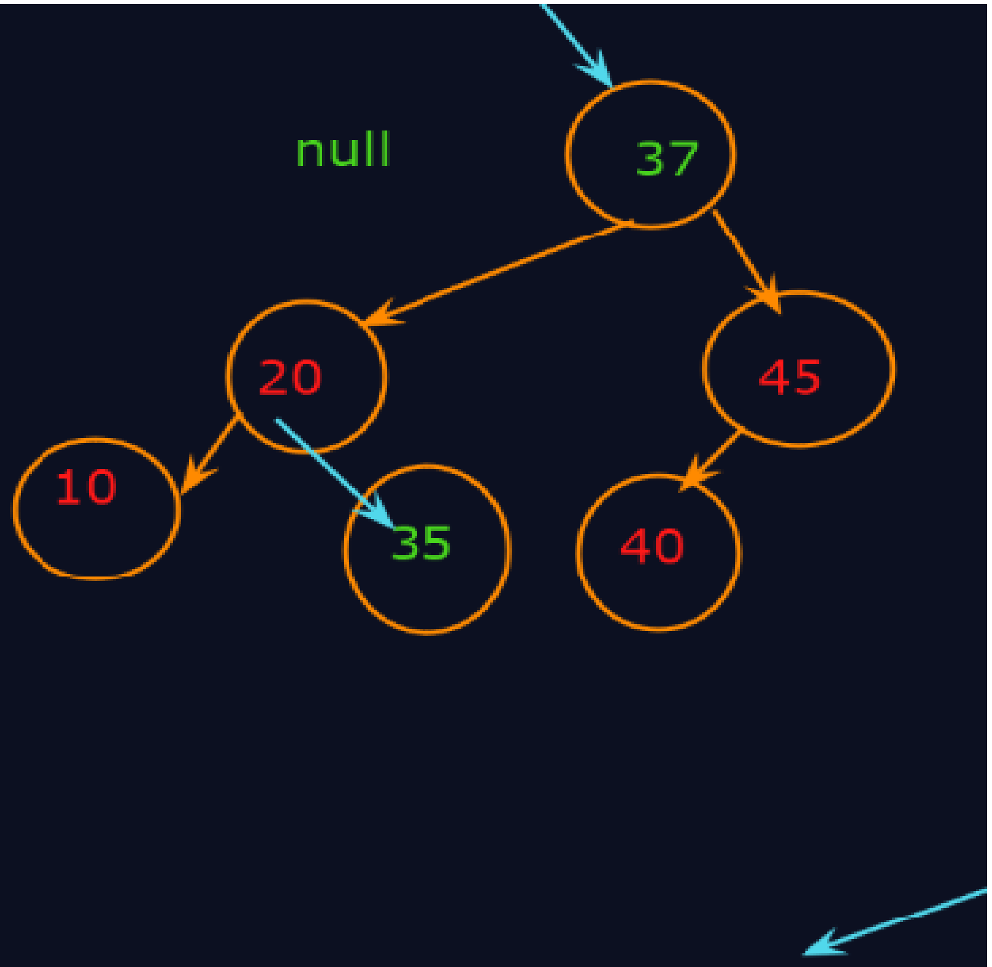


inorder: 10 20 30 37 38 40 45

```
}

Node insert( Node root, int key)
{
    if(root == null)
    {
        root = new Node(key);
        return root;
    }
    if(key <= root.data)
        root.left = insert(root.left, key);
    else
        root.right = insert(root.right, key);
    return root;
}

void printinsert(int key)
{
    root =insert(root, key);
}
```



Inorder: 10 20 30 37 38 40 45

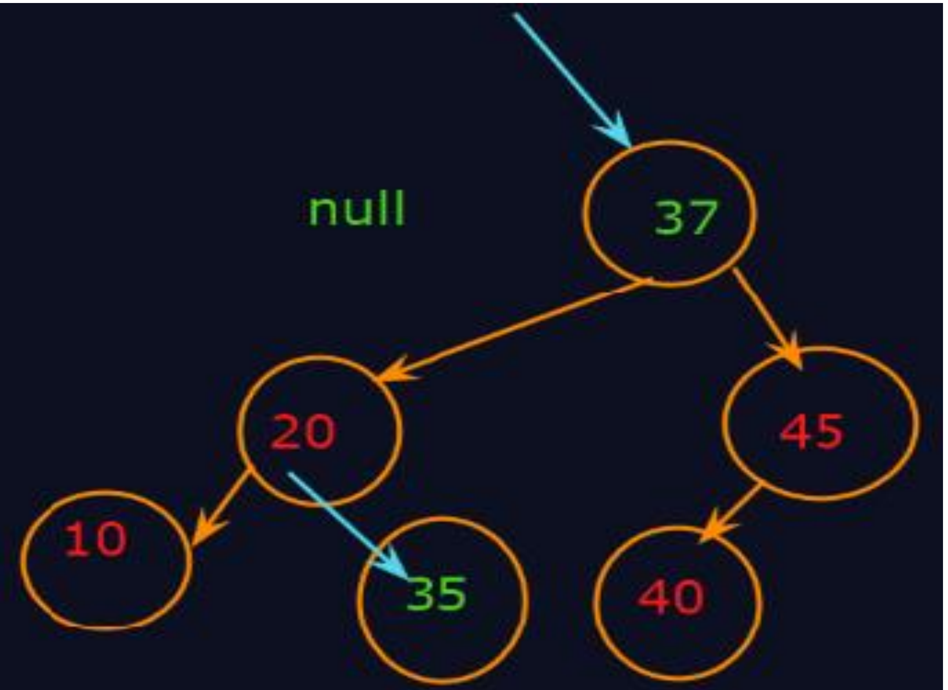
```java
    }

    void printinsert(int key)
    {
        root =insert(root, key);
    }

void printInorder(Node node)
{
    //base condition
    if(node == null)
        return;
    printInorder(node.left);
    System.out.print(node.data+" ");
    printInorder(node.right);

}
 void inorder()
{
    printInorder(root);//call for function
}

Node delete(Node root, int key)
    {
```



Inorder: 10 20 30 37 38 40 45

Deleting Nodes from a Binary Search Tree

- Write an algorithm to locate the position of the node to deleted from a binary search tree.

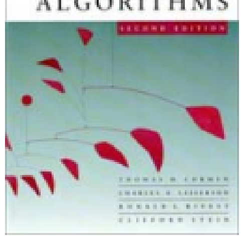
- Delete operation in a binary search tree refers to the process of deleting the specified node from the tree.
- Before implementing a delete operation, you first need to locate the position of the node to be deleted and its parent.
- To locate the position of the node to be deleted and its parent, you need to implement a search operation.

Deleting Nodes from a Binary Search Tree (Contd.)

- Once the nodes are located, there can be three cases:
  - **Case I**: Node to be deleted is the leaf node
  - **Case II**: Node to be deleted has one child (left or right)
  - **Case III**: Node to be deleted has two children

Deletion of a key from a BST

```
Algorithm:- Delete1BST (info, left, right, root, LOC, PAR)
            // When leaf node has no child or only one child
{
    1.  if ( ( LOC -> left = NULL)  and  ( LOC -> right = NULL))
            1.1 Child = NULL
        elseIf ( LOC -> left != NULL)
            1.1 Child = LOC -> left
        else
            1.1 Child = LOC -> right
    2. if ( PAR != NULL)
            2.1   if ( LOC = PAR -> left )
                        2.1.1 PAR -> left = Child
            2.1   else
                        2.1.1 PAR -> right = Child
        else
            2.1 root = Child
 }
```



Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of n items.

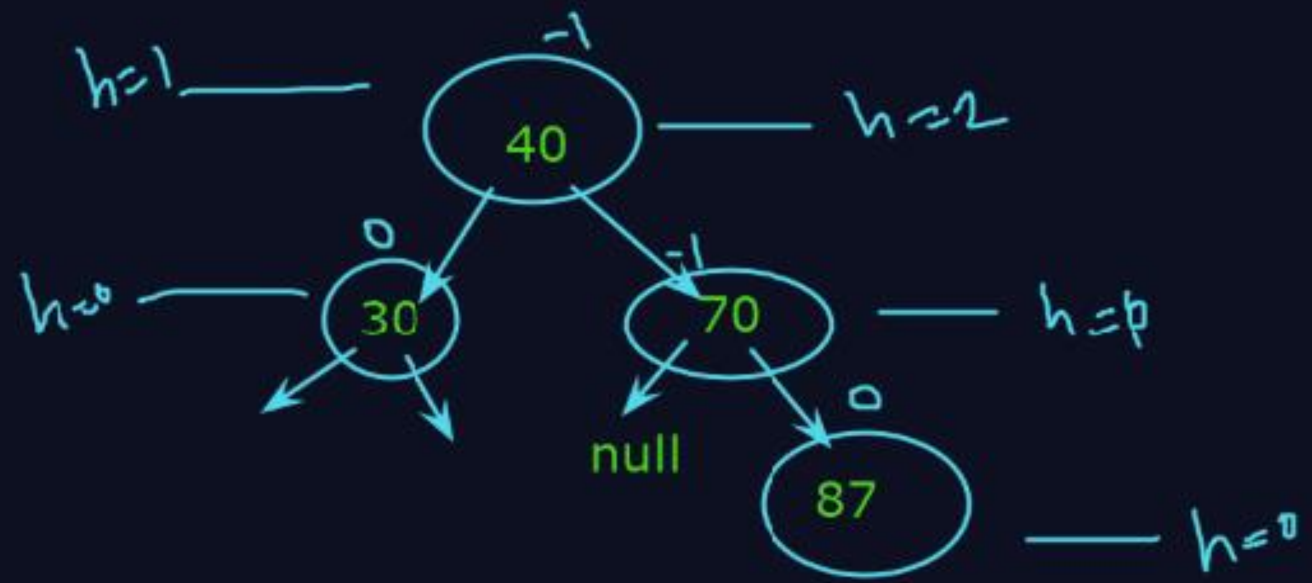
Examples:

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees

AVL Tree

AVL Tree:

-An AVL tree is a self balancing BST where the difference between the left and right subtree (balance factor) of any node is at m

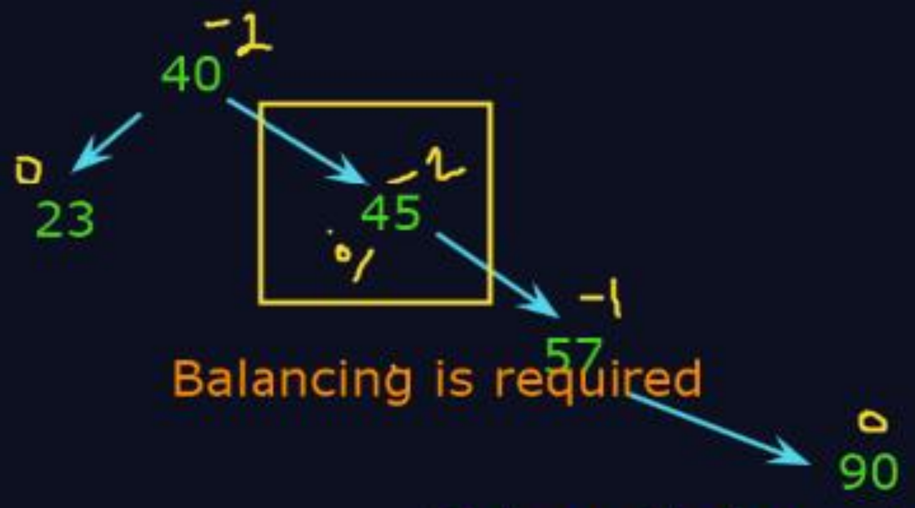


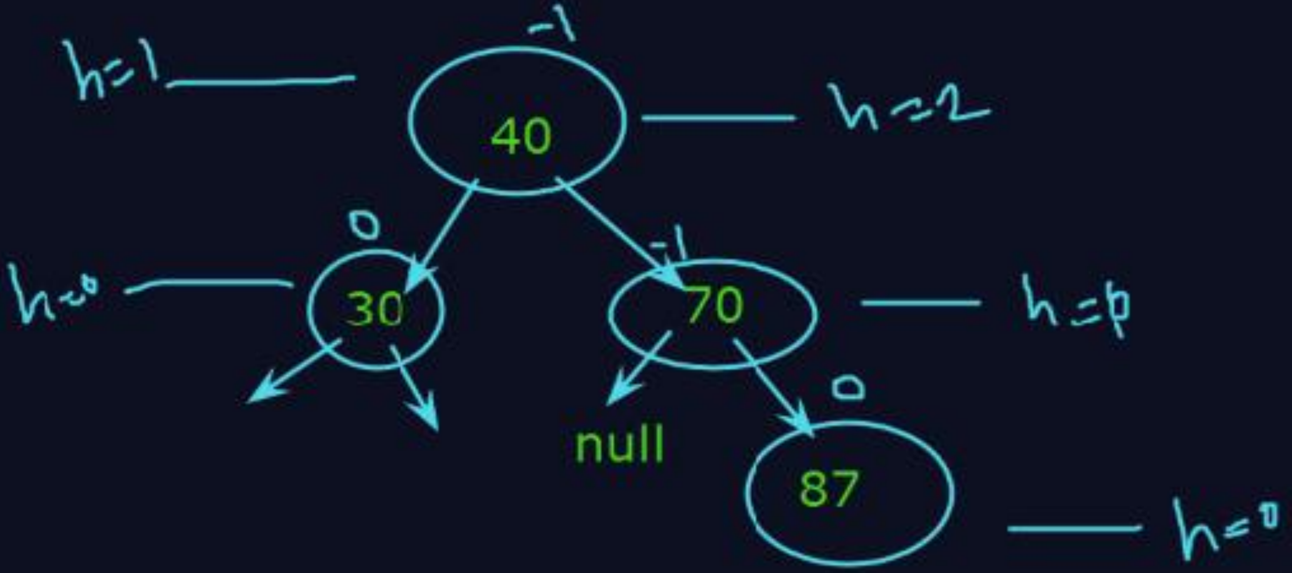
Balance Factor = height of left subtree - height of right subtree

BF = {-1, 0, +1}

AVL Tree:

-An AVL tree is a self balancing BST where the difference betwee
the left and right subtree (balance factor) of any node is at m





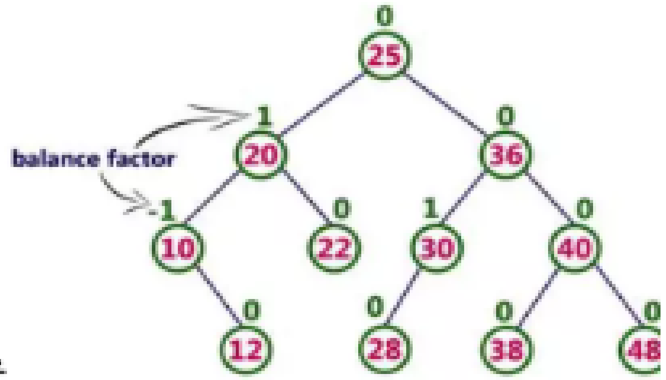
Balance Factor = height of left subtree - height of right subtree

BF = {-1, 0, +1}

Rotations for balancing the tree

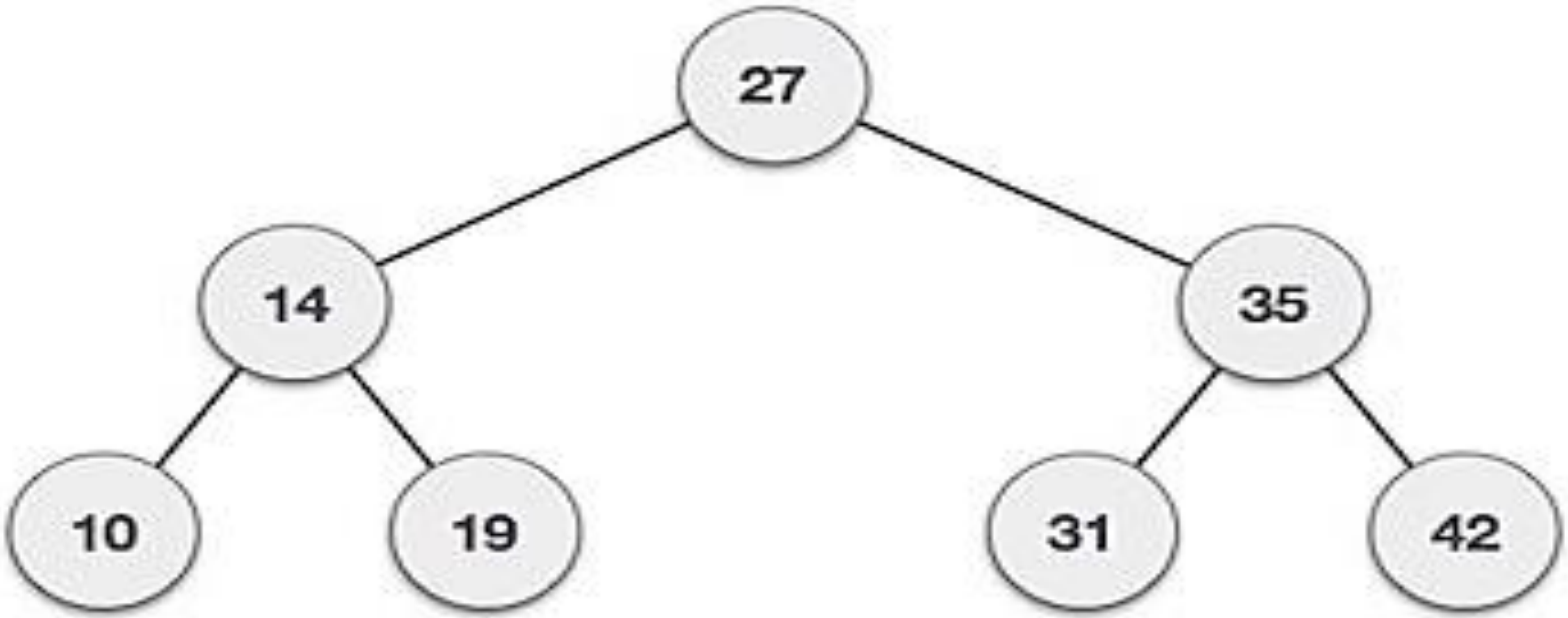
What is AVL Tree?

- AVL invented by G.M. Adelson-Velsky and E.M. Landis. So, name is AVL.
- AVL tree is a height-balanced binary search tree.
- AVL tree, balance factor of every node is either -1, 0 or +1.
- Every node maintains an extra information known as balance factor.
- **Balance factor = heightOfLeftSubtree – heightOfRightSubtree.**
- Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.
- Operation perform Search, Insertion, Deletion with **O(log n)** time complexity.

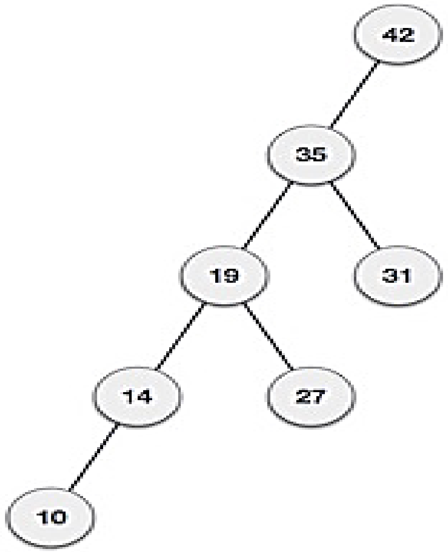


Example:

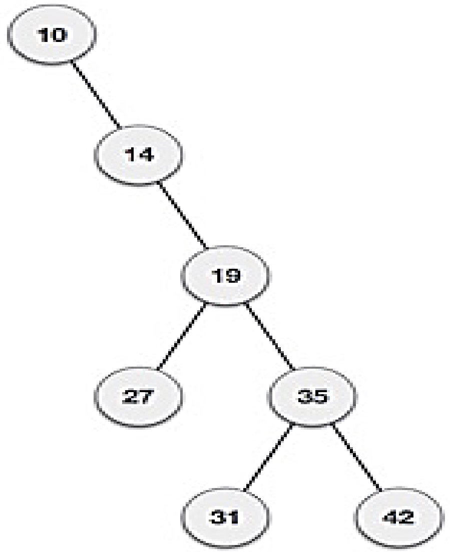
Example



What if the input to binary search tree comes in sorted (ascending or descending) manner?



If input ‘appears’ non-increasing manner



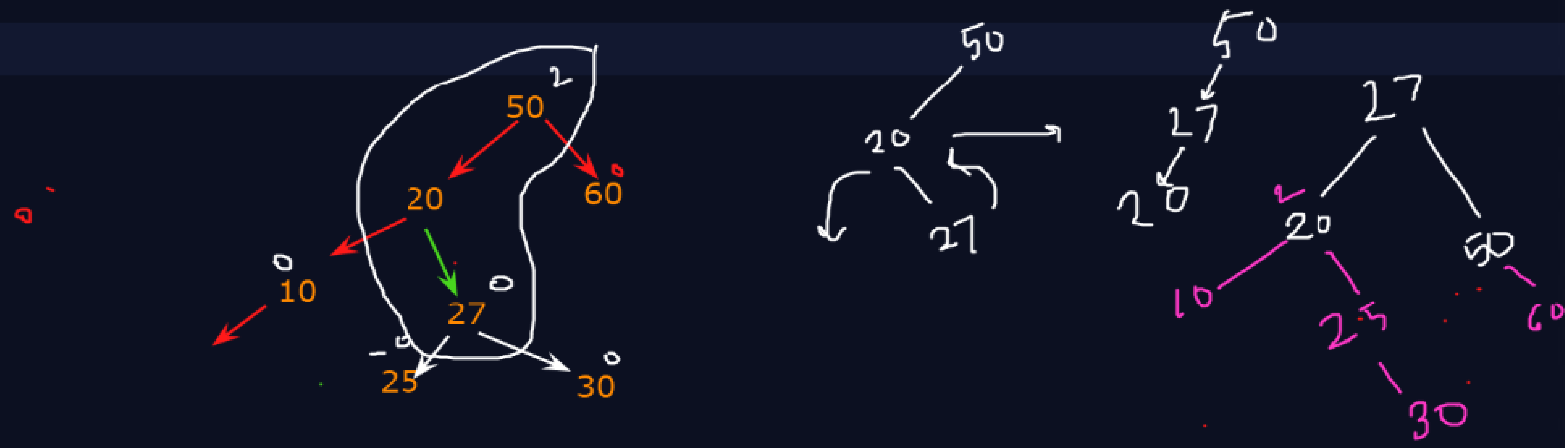
If input ‘appears’ in non-decreasing manner

- Ans: It is observed that BST's worst-case performance closes to linear search algorithms, that is O(n). In real time data we cannot predict data pattern and their frequencies. So a need arises to balance out existing BST.

AVL Tree:

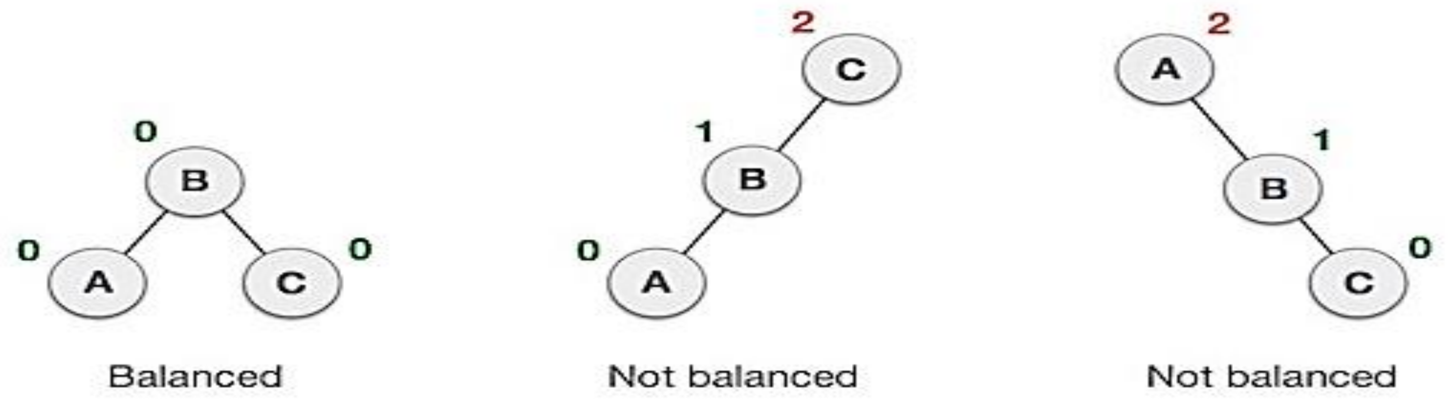
-An AVL tree is a self balancing BST where the difference betwee
the left and right subtree (balance factor) of any node is at m

50, 30 60, 20, 10



AVL Tree

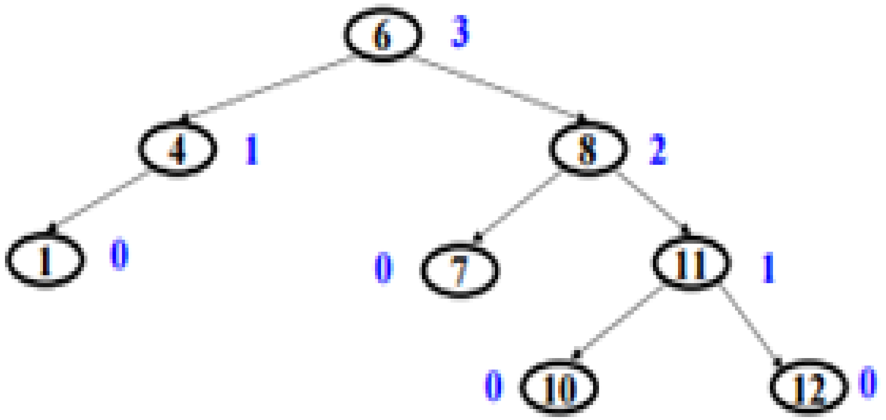
- Named after their inventor *Adelson*, *Velski* & *Landis*,
- AVL trees are height balancing binary search tree.
- AVL tree checks the height of left and right sub-trees and assures that the difference is not more than 1. This difference is called ***Balance Factor***.
- Example:



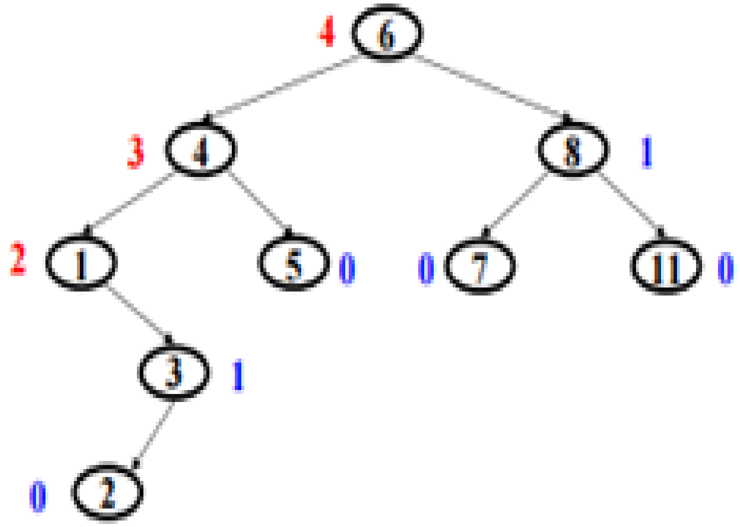
- Problem: In second tree, the left subtree of C has height 2 and right subtree has height 0, so the difference is 2. In third tree, the right subtree of A has height 2 and left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.
-
- *BalanceFactor* = height(left-sutree) − height(right-sutree)
-
- Solution: If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

Q. Check AVL Tree:

An AVL tree?



An AVL tree?



Why AVL Tree?

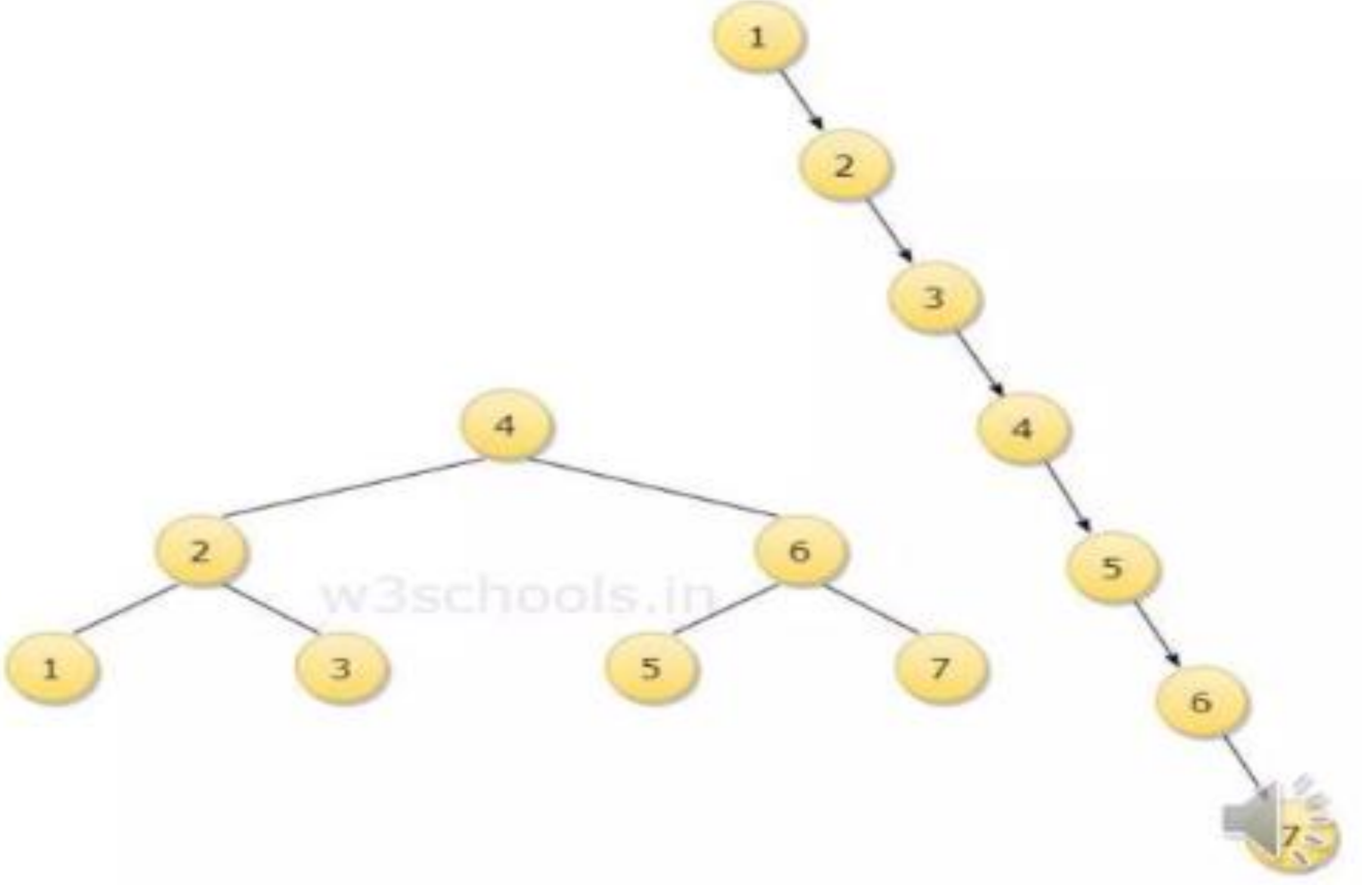
Example: Keys are: 1, 2, 3, 4, 5, 6, 7. Generate Binary Tree.

- **Fig 1:** AVL Tree
- **Fig 2**. Binary Tree

- **Insert 8 in BT**: 7 comparisons.
- **Insert 8 in AVL**: 3 comparisons.

So, AVL Tree:

- **Height balance trees.**
- **Insertion and deletion have low time complexity.**



AVL Rotations

- To make itself balanced, an AVL tree may perform four kinds of rotations –

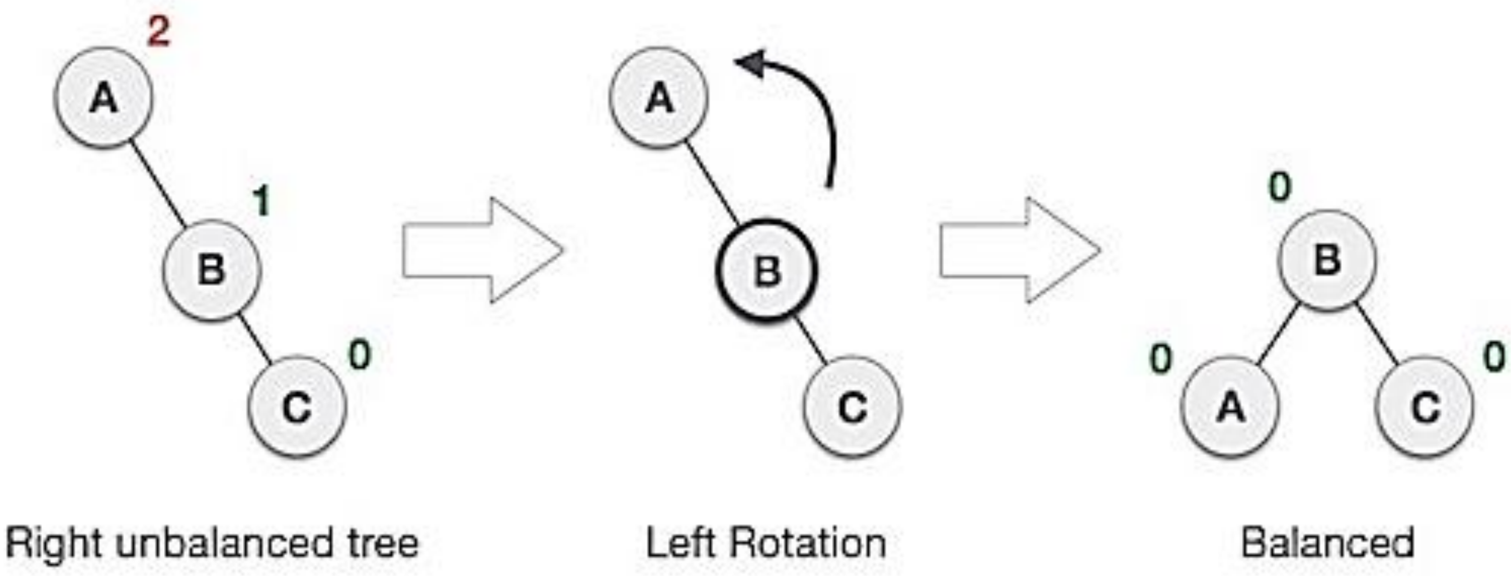
1. Left rotation
2. Right rotation
3. Left-Right rotation
4. Right-Left rotation

-

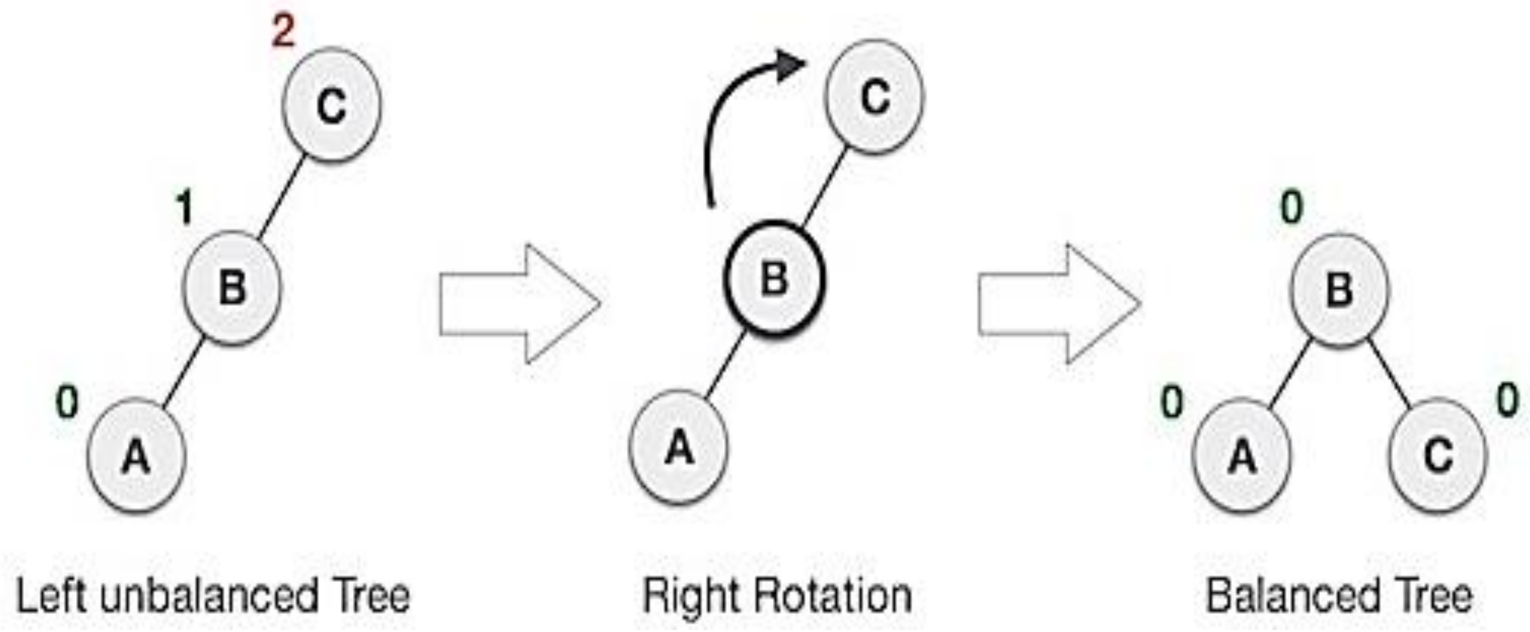
- First two rotations are single rotations and next two rotations are double rotations. Two have an unbalanced tree we at least need a tree of height 2. With this simple tree, let's understand them one by one.

-

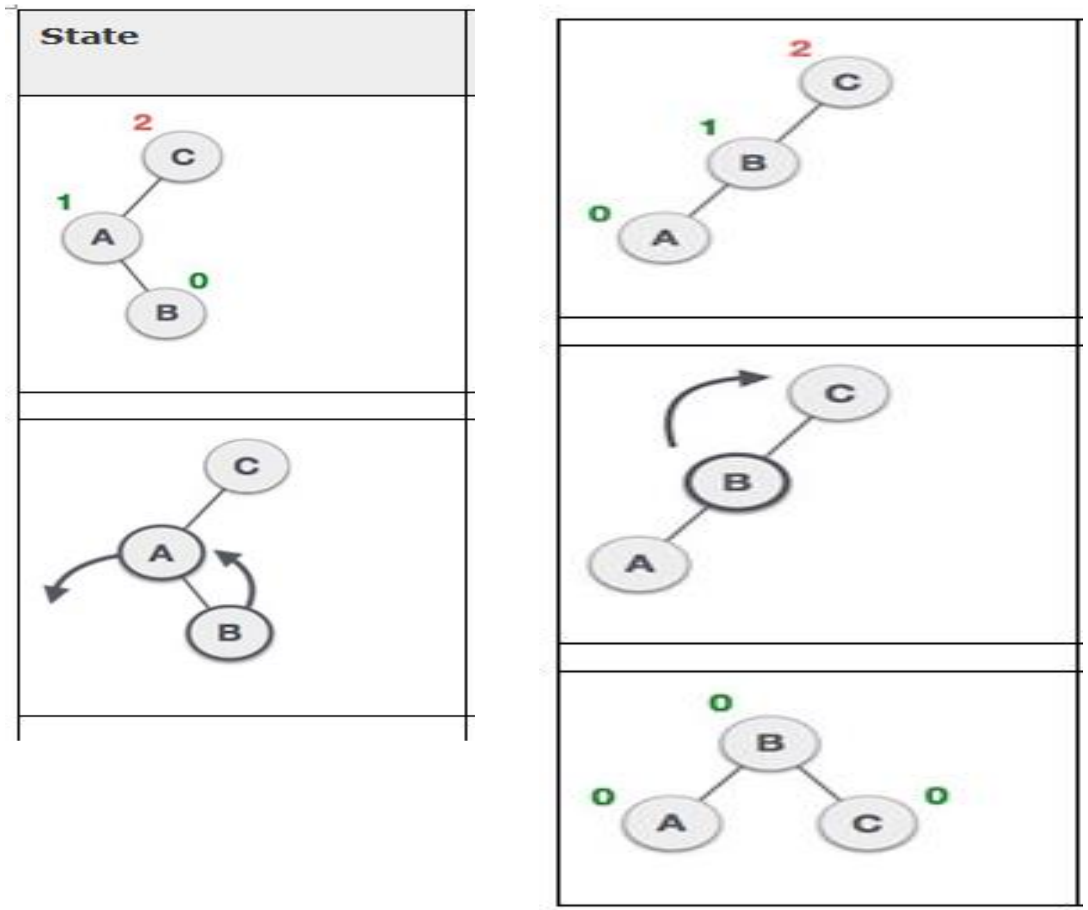
•**Left Rotation : If a tree become unbalanced, when a node is inserted into the right subtree of right subtree, then we perform single left rotation –**



- Right Rotation : AVL tree may become unbalanced if a node is inserted in the left

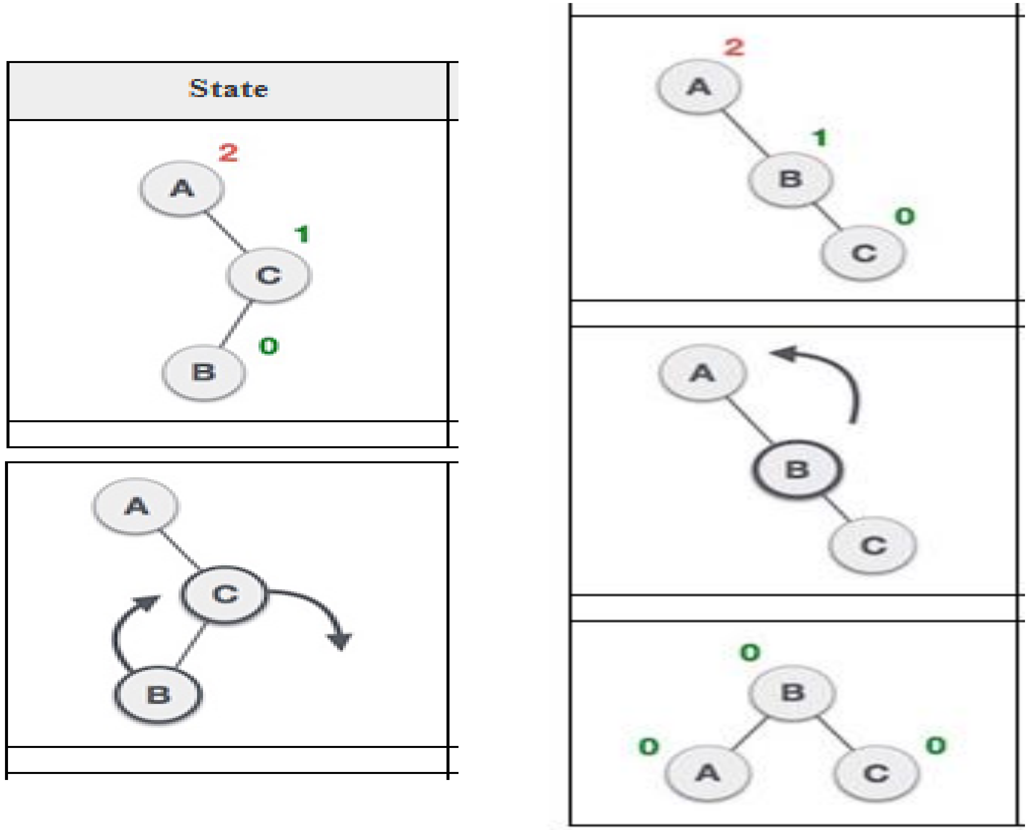


Left-Right Rotation :



- Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is combination of left rotation followed by right rotation.
-

Right-Left Rotation :



•Second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

Insertion Operations In AVL Tree

In AVL Tree, a new node is always inserted as a leaf node.

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

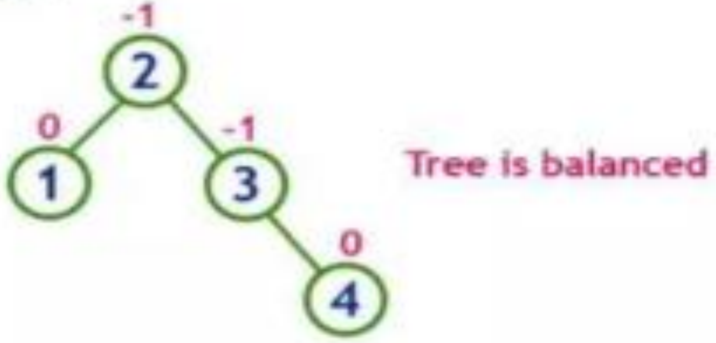
Step 2 - After insertion, check the **Balance Factor** of every node.

Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

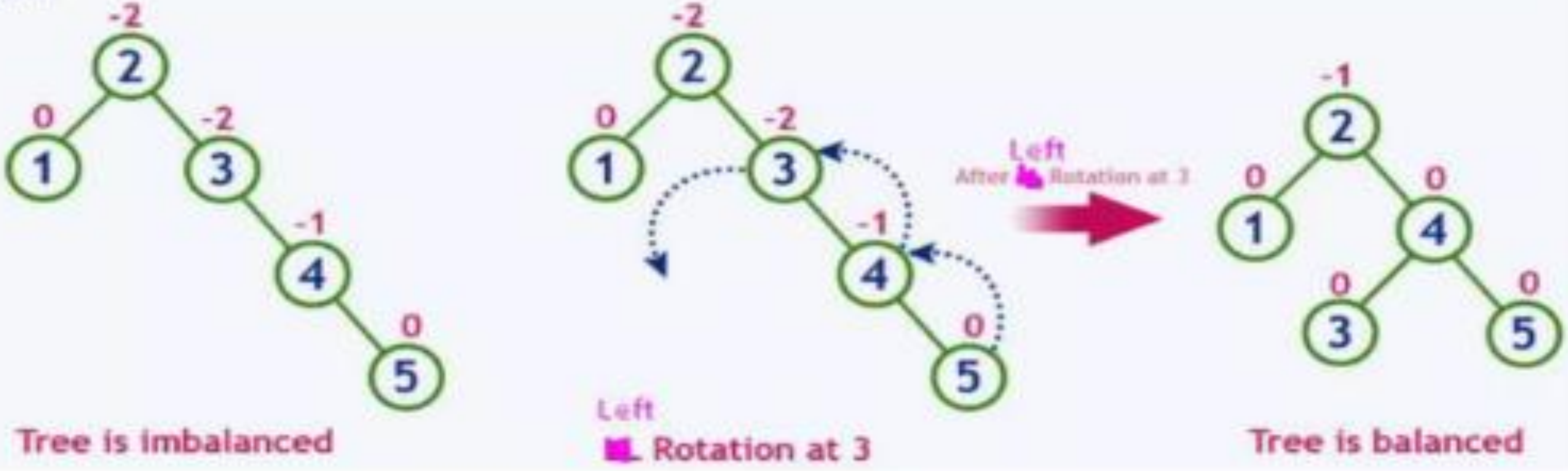
Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Continue..

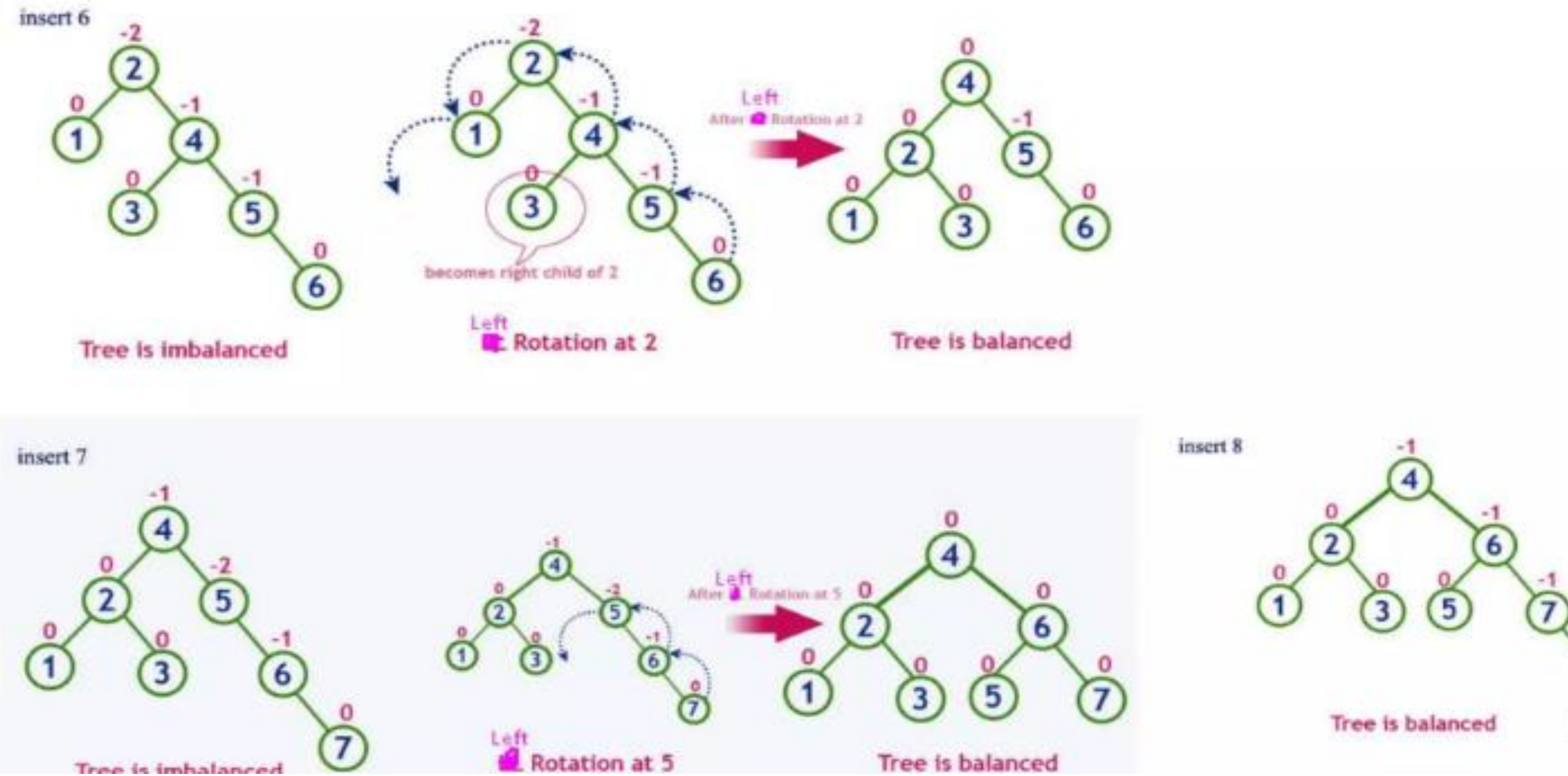
insert 4



insert 5

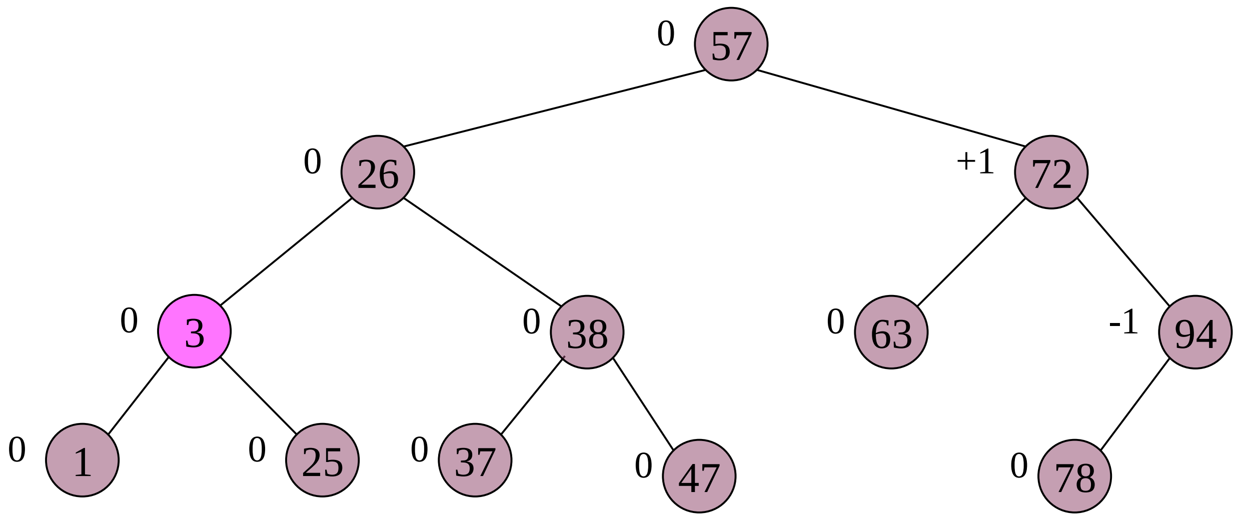


Continue..

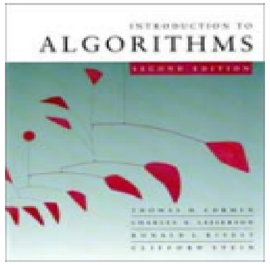

insert 6
Tree is imbalanced
becomes right child of 2
Left Rotation at 2
Left
After Rotation at 2
Tree is balanced
insert 7
Left Rotation at 5
Left
After Rotation at 5
Tree is balanced
insert 8
Tree is balanced

Balance ok

Rebalance and recalculate balance factors



Next step: insert 30 -->

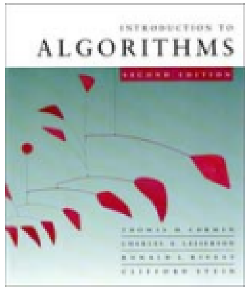


Red-black trees

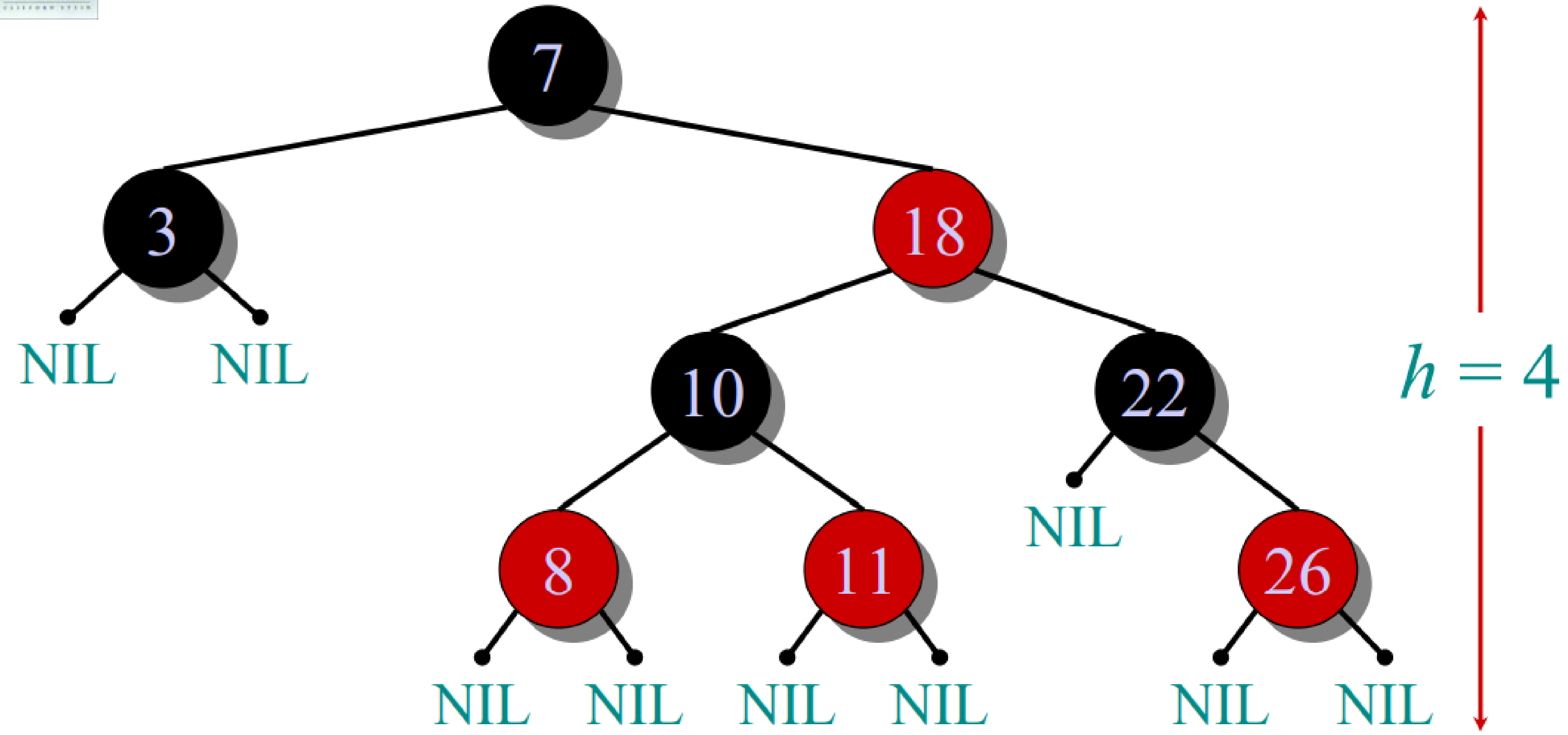
This data structure requires an extra one-bit color field in each node.

Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).



Example of a red-black tree



Thanks