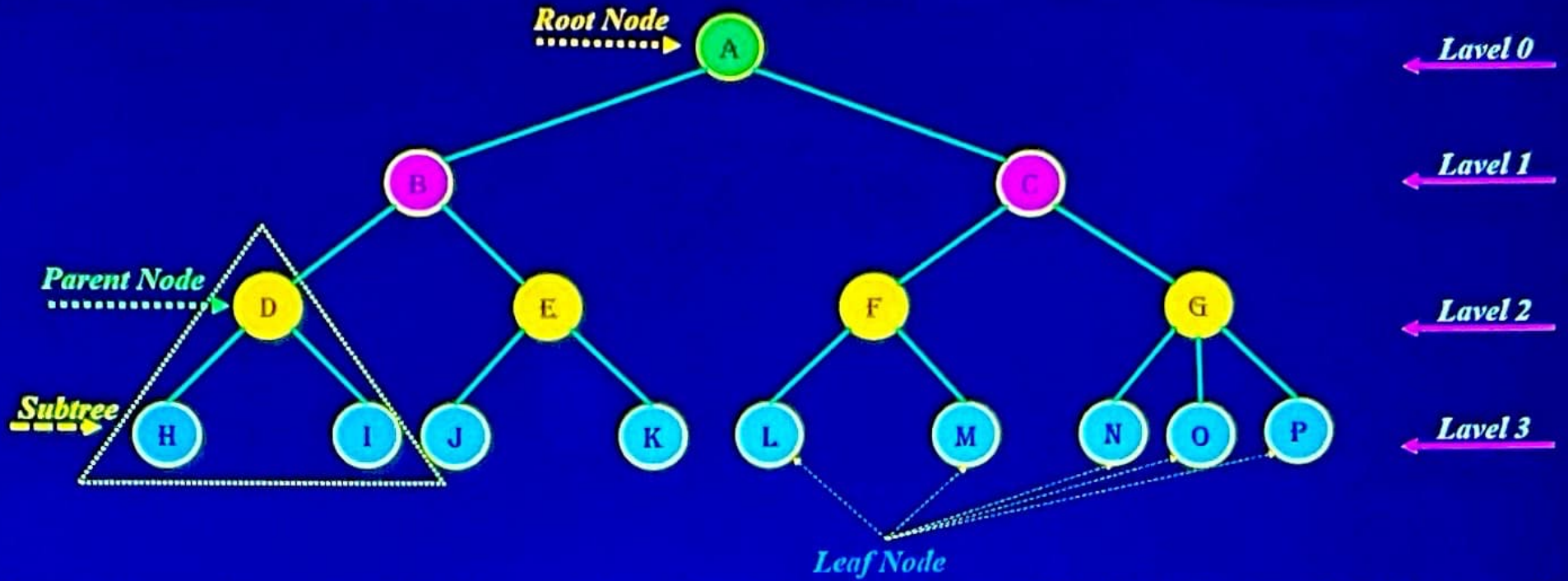
Tree

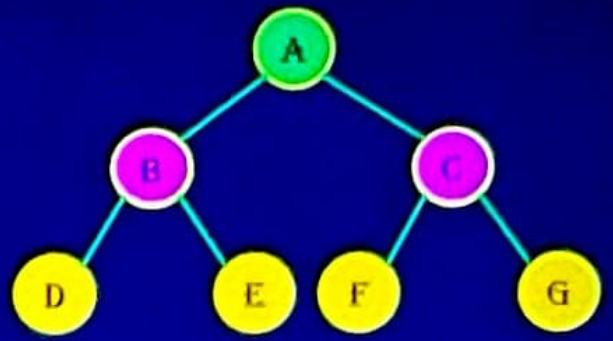
Tree: *A tree is a non-linear data structure that is used to store the data in a hierarchical form. The data will be maintained in the nodes that are linked in hierarchical order.*



The linear data structures such as Array, Linked List, Stack, and Queue store the data sequentially. So the time complexity increases when performing the operation with a large amount of data. The hierarchical order of Trees helps to perform the operations faster than linear data structures.

Tree

Basic Terminology



Node: An element of the tree that stores data. Eg: A, B, C, D, E, F and G
Root Node: The topmost node of the tree (it has no parent). Eg: A
Edge: A connection between two nodes.
Child Node: A node that descends from another node. Eg: B, C, D, E, F and G
Parent Node: A node that has child nodes. Eg: A, B and C
Leaf Node: A node with no children. Eg: D, E, F and G
Siblings: Nodes with the same parent. Eg: D and E, F and G, B and C
Subtree: A tree formed by any node and its descendants. Eg: B, D and E or C, F, and G
Height of Tree: The number of edges in the longest path from the root to a leaf. Eg: 2
Tree Size: The tree size is the number of nodes in the tree. Eg: 7
Height of a Node: The height of a node is the maximum number of edges between the node and a leaf node.

Tree

Types of Trees in Data Structure	
General Tree	Each node can have any number of children.
Binary Tree	A tree with a maximum of two children is allowed for every node.
Binary Search Tree	A Binary Tree with a particular ordering of nodes. i.e., Left child < Parent < Right child
AVL Tree	A self-balanced Binary Search Tree.
Red Black Tree	A self-balanced Binary Search Tree with nodes colored in red or black.
Splay Tree	A self-balanced Binary Search Tree to keep the most recently accessed items on top.
Heap	A self-balanced Binary Tree with a particular ordering of nodes.
B Tree	A self-balanced m-way Binary Search Tree to maintain the sorted stream of data.
B+ Tree	B Tree extended for efficient search, insert and delete operations.
Spanning Tree	A subset of a Graph that has all vertices connected with the minimum number of edges.

Common Operations: Insertion, Deletion, Traversal(Visit all nodes, e.g., Preorder, Inorder, Postorder, Level order), Search(Find a specific value)

B-Tree / B+ Tree – Used in databases and file systems.

Tree

Time & Space Complexity				
Type	Search	Insertion	Deletion	Space Complexity
General Tree	O(n)	O(1) – O(n)	O(1) – O(n)	O(n)
Binary Tree	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(h) → O(log n) best / O(n) worst	O(h) → O(log n) / O(n)	O(h) → O(log n) / O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	O(n)
Red Black Tree	O(log n)	O(log n)	O(log n)	O(n)
Splay Tree	O(log n) (amortized)	O(log n) (amortized)	O(log n) (amortized)	O(n)
Heap (Binary Heap)	O(n)	O(log n)	O(log n)	O(n)
B-Tree	O(log n)	O(log n)	O(log n)	O(n)
B+ Tree	O(log n)	O(log n)	O(log n)	O(n)
Spanning Tree	O(E log V) (for MST)	—	—	O(V + E)

For Binary Search Tree, performance depends on tree balance.
Spanning Tree is not a search or insertion-based structure; its time complexity refers to minimum spanning tree (e.g., Kruskal or Prim algorithm).
AVL Tree (*Adelson-Velsky* and *Landis* Tree)

Tree

Applications

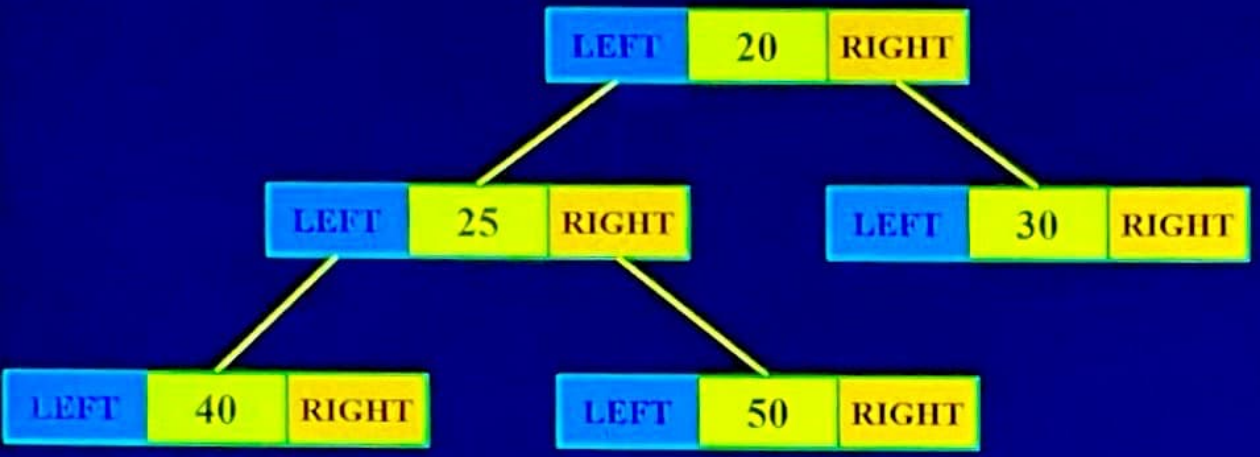
- **File System Organization**– Your computer's folders and files are stored in a tree structure. Where Each folder is a node, and subfolders are child nodes.
- **Website Structure**– A web page's code is stored in a DOM tree, where The browser uses tree to display the web page correctly.
- **Blockchain**– Merkle Trees are used in Bitcoin and Ethereum to store and verify large sets of transactions efficiently.
- **Computer Networks**– panning Trees are used in routers to prevent looping paths in a network.
- **Compression & Encoding**– Huffman Trees are used in ZIP file compression, JPEG images, and MP3 encoding — they reduce file size by storing frequent data more efficiently.
- **Databases and File Storage**– B-Trees / B+ Trees are used in databases like MySQL, Oracle, and file systems (NTFS, ext4) to quickly search, insert, and retrieve data.

BW WW ZG

Binary Tree

A Binary Tree is a form of Tree Data Structure with almost two children for every node.

REPRESENTATION:




```
// Binary Tree node representation
struct Node
{
    int data;    // Actual data
    Node* left;  // Pointer to left child
    Node* right; // Pointer to right child

    // Node constructor
    Node(int d) : data(d), left(NULL), right(NULL) {}
};

// The root node
Node* root;
```

- **Root**: The topmost node.
- **Node**: Node consists of Data, Left and Right pointers.
- **Left**: Pointer to the left child.
- **Right**: Pointer to the right child.

A constructor is a special function in a class that is automatically called when an object is created. It is mainly used to initialize the object's data members.

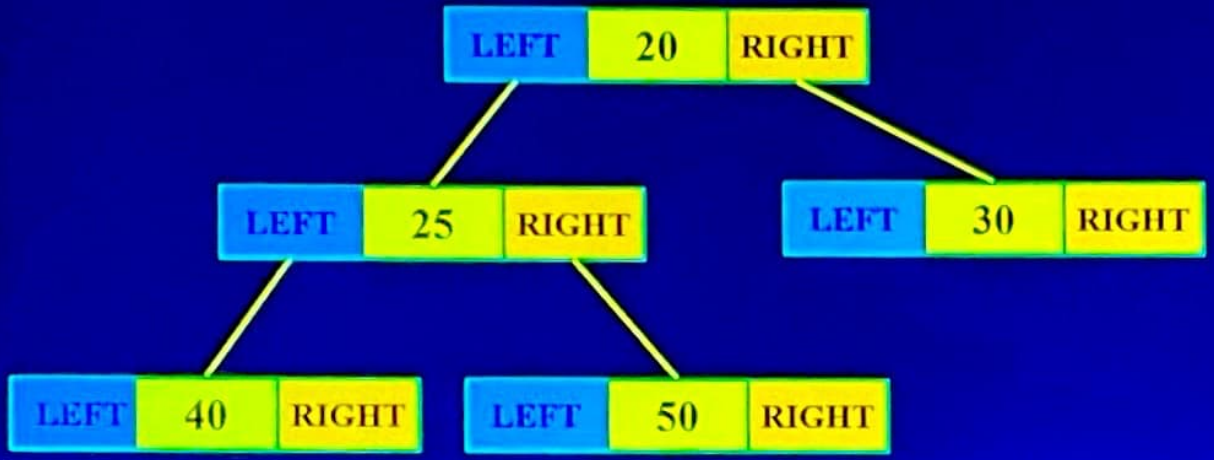
Note : A simple Binary Tree is inefficient for real-time applications due to its $O(n)$ time complexity for search, insertion, and deletion, but it serves as the foundation for efficient structures like Binary Search Trees, AVL Trees, Red-Black Trees, Splay Trees, and Heaps.

Binary Tree Operations

Insertion: In insertion, a new node is added at the first available position found during a level-order traversal, where a node has either its left or right child empty.

Algorithm:

1. Create the new Node with the data and initialize its Left and Right child to NULL.
2. Traverse the tree in Levelorder.
3. Find the first node with either its Left or Right child is free.
4. Link the new node to the free position.



Insert node(70)

Binary Tree Operations

Deletion: A node cannot be deleted directly if it has children; instead, its data is replaced with that of the deepest node, and then the deepest node—which has no children—is deleted.

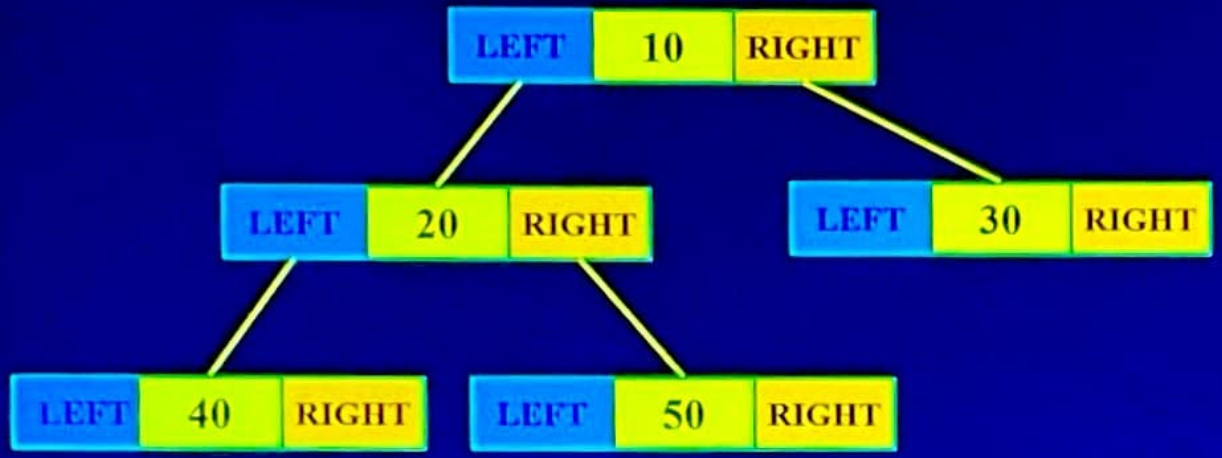
Algorithm:

1. Check the tree is empty and return if true.
2. Check the tree has only a single node and matches the given data. If true, delete the node, set the root to NULL, and return.
3. Otherwise, traverse the tree in Levelorder starting from the root node.
4. Search the target node that contains the given data.
5. Continue the traversal to locate the deepest node.
6. Replace the target node with the deepest node's data.
7. Delete the deepest node.

Binary Tree Operations

Traversals: A Binary Tree can be traversed in any one of the following orders:

- **Preorder** (Root, Left, Right)
- **Postorder** (Left, Right, Root)
- **Inorder** (Left, Root, Right)
- **Levelorder** (Level 0, Level 1, and so on…)

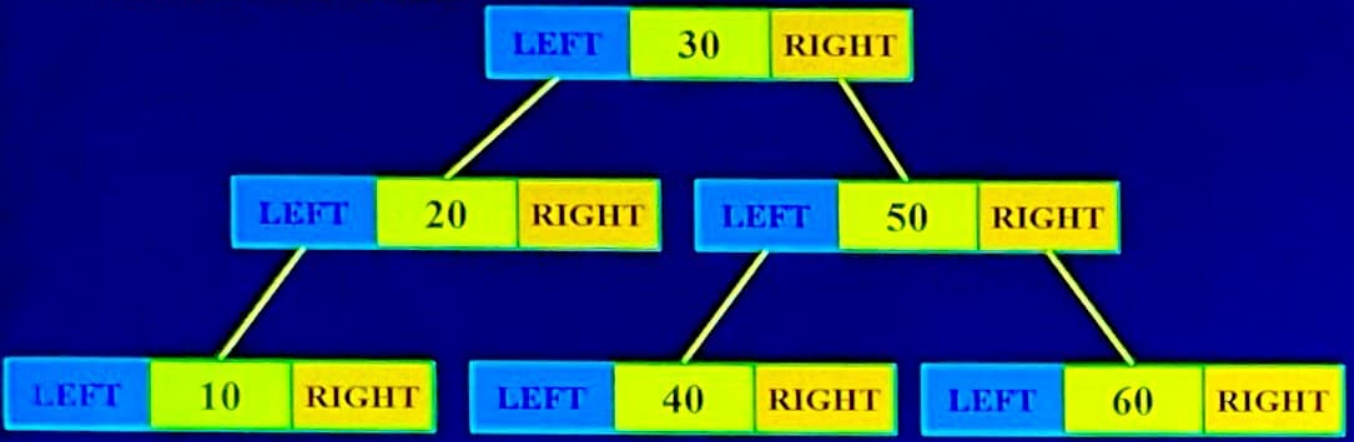


Preorder : 10, 20, 40, 50, 30
Postorder :40, 50, 20, 30, 10
Inorder : 40, 20, 50, 10, 30
Levelorder : 10, 20, 30, 40, 50

Binary Search Tree

A Binary Search Tree is a Binary Tree that contains the nodes sorted in a particular order. The ordering helps to perform the search, insertion, and deletion operations faster than the linear data structures.

REPRESENTATION:




```
// Binary Search Tree node representation
struct Node
{
    int data;    // Actual data
    Node* left;  // Pointer to left child
    Node* right; // Pointer to right child

    // Node constructor
    Node(int d) : data(d), left(NULL), right(NULL) {}
};

// The root node
Node* root;
```

IMPORTANT

- The left node and its subtree are smaller than the current node.
- The right node and its subtree are greater than the current node.
- There must be no duplicate nodes.
- Generally implemented using Linked List

Binary Search Tree Operations

Insertion: A node is inserted in the tree according to its value, requiring traversal based on element comparison to find the correct position for insertion.

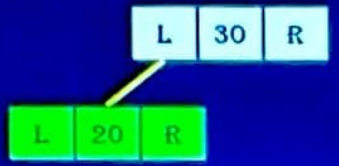
Algorithm:

1. Start the traversal from the root node to find the data location.
2. If the data element is smaller than the current node, traverse on the left subtree.
3. Otherwise, traverse on the right subtree if the data element is greater.
4. Insert the node when the traversal reached its depth and found the location.

EXAMPLE:

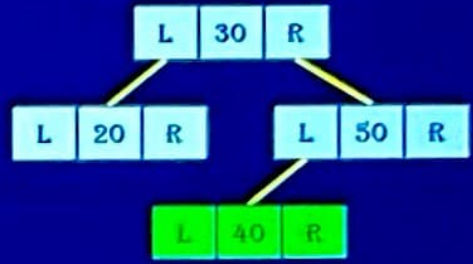
Insert the first node (30)

Insert the node (20)



Insert the node (50)

Insert the node (40)



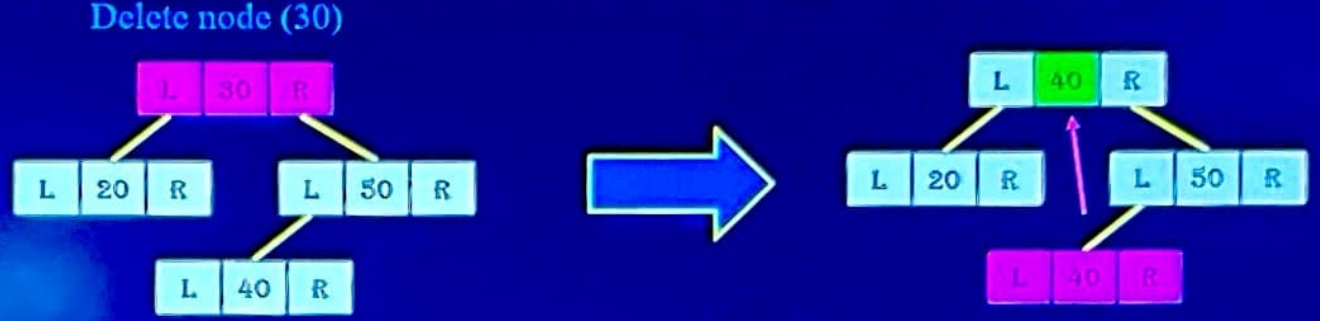
Binary Search Tree Operations

Deletion: Traverse the tree based on the node order to find the data element and delete the node by rearranging the child nodes. In case the target node has two children, this will be replaced by the next smaller element to avoid losing its children.

Algorithm:

1. Start the traversal from the root node to find the target node.
2. If the data element is smaller than the current node, traverse on the left subtree.
3. Otherwise, traverse on the right subtree if the data element is greater.
4. If the target is found, delete the node without losing the reference to its children.
   - No child for the target node: Delete the node and return NULL to unlink from the parent.
   - Only has the right child: Delete the node and link the left child to its parent directly.
   - Only has the left child: Delete the node and link the right child to its parent directly.
   - Both left and right children: Find the next smaller data element (in-order successor) node, and replace the target node with its in-order successor.
5. Otherwise, return NULL if the traversal is completed without finding the element.

Example:



Binary Search Tree Operations

Searching: Search the data element by traversing the tree from the root node. Proceed the search either on its left or right subtree based on the node order.

Algorithm:

1. Start the traversal from the root node to find the target node.
2. If the data matches the current node, return the node.
3. Traverse on the left subtree if the data element is smaller than the current node,
4. Traverse on the right subtree if the data element is greater.
5. Return NULL if the traversal is completed without finding the element.

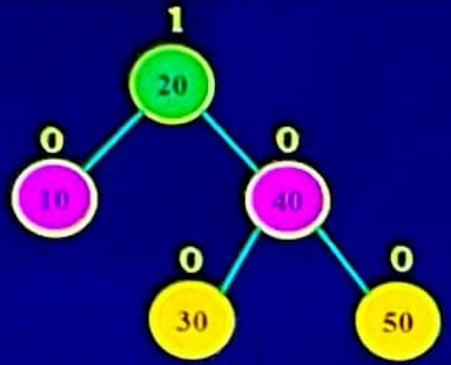
IMPORTANT

- Every BST is a Binary Tree, but not every Binary Tree is a BST.
- Binary Tree can contain duplicates (depends on implementation), while BST usually does not allow duplicates.
- Binary Tree requires traversal of the entire tree for searching → O(n) time, whereas BST performs binary search in O(log n) time (if balanced).
- Inorder traversal of a Binary Tree does not produce sorted output, but inorder traversal of a BST always gives sorted data.
- Binary Tree has no specific order between parent and child nodes, while in a BST, elements follow a specific order — Left < Root < Right.

AVL Tree

AVL Tree is a self-balanced Binary Search Tree (BST) that maintains the optimal height of the tree. This ensures that the operations are faster than the BST in the worst cases as well.

REPRESENTATION:

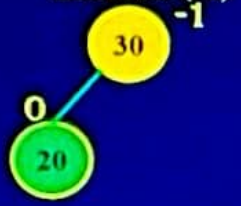


- **Balance Factor**: Node has a balance factor which can be {0, +1, or -1}.
- **Balance factor** = Height (Right subtree) – Height (Left subtree)
- **Height**: The height of a node represents the number of levels

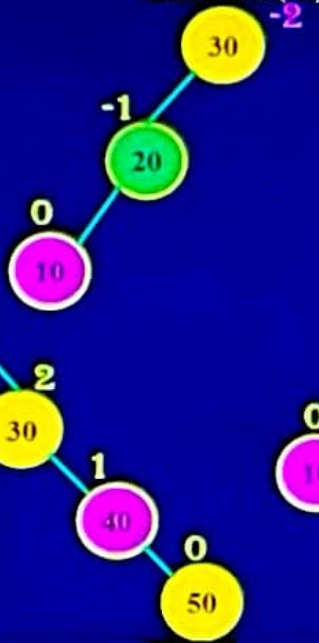
Example:

Insert the first node (30)

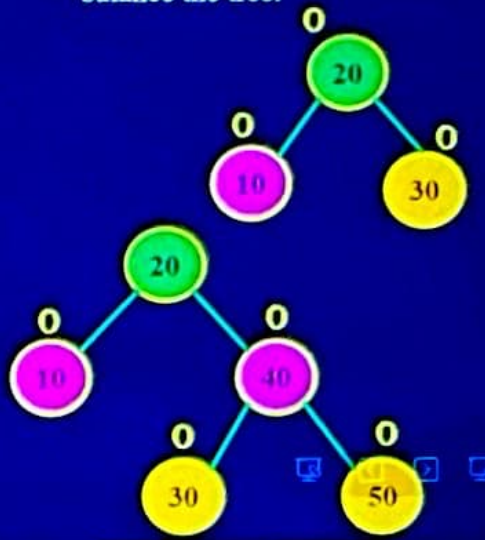
Insert node (20)



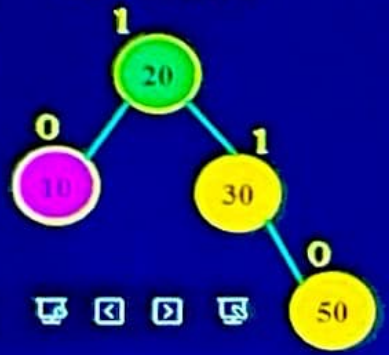
Insert node (10)



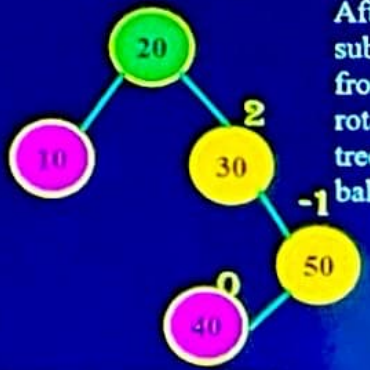
After the insertion, the tree is left-left unbalanced from the node (30). So, perform the right rotation to balance the tree.



Insert node (50)



Insert node (40)



After inserting node (40), the subtree is right-left unbalanced from node (30). Hence double rotations are required to balance the tree. Perform Right-Left rotation to balance the tree.

Tree (Class Questions / Assignments)

Q40. Write a program to implement basic operations (creation, insertion, traversals — preorder, inorder, postorder — and counting nodes) on a Binary Tree.

Q41. Extend the above program (Q40) to include deletion operations in the Binary Tree.

Q42. Write a program to count the total number of nodes, leaf nodes, and internal nodes in a Binary Tree.

Q43. Write a program to implement basic operations on a Binary Search Tree (BST).

Q44. Write a program to find the smallest and largest node in a BST.

Q45. Write a program to implement basic operations on an AVL Tree.