DATA STRUCTURES

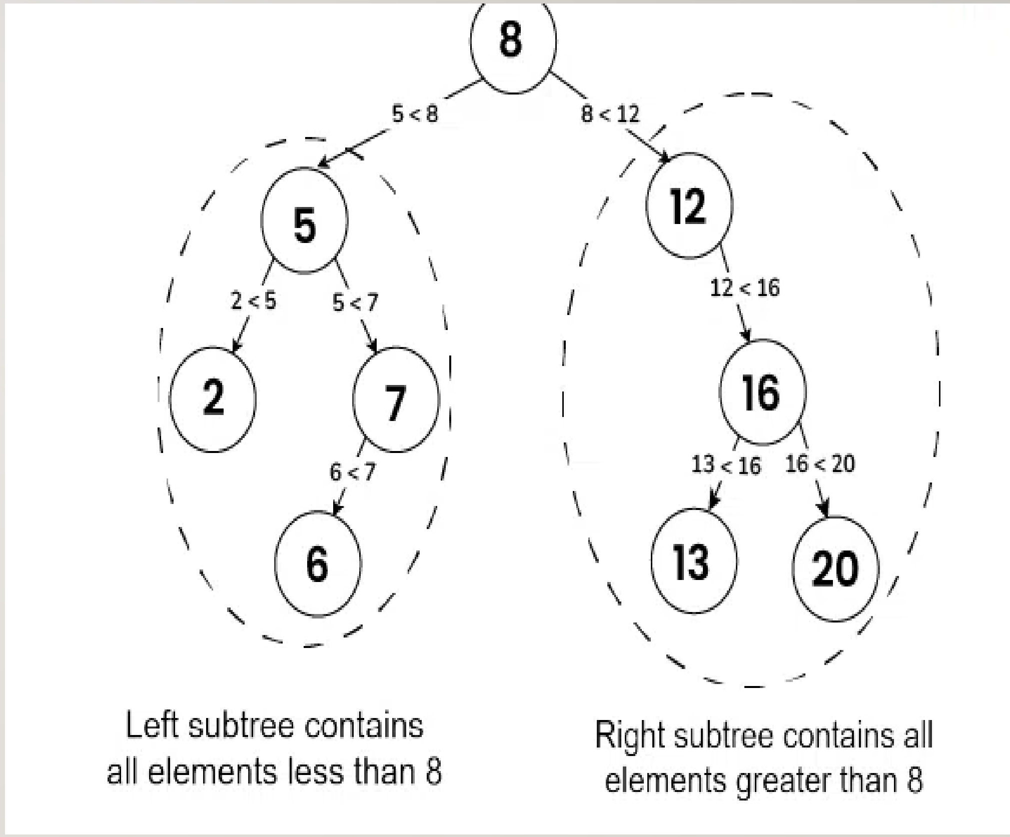
LECTURE-13

TREE

Dr. Sumitra Kisan

Binary Search Tree

- A **Binary Search Tree** is a data structure used in computer science for organizing and storing data in a sorted manner.
- Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child, with the **left** child containing values less than the parent node and the **right** child containing values greater than the parent node.
- This hierarchical structure allows for efficient **searching**, **insertion**, and **deletion** operations on the data stored in the tree.


8
5 < 8
8 < 12
5
12
2 < 5
5 < 7
12 < 16
2
7
16
6 < 7
13 < 16
16 < 20
6
13
20
Left subtree contains all elements less than 8
Right subtree contains all elements greater than 8

Properties of Binary Search Tree

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

Number of Binary Search Trees:

$$\text{Number of distinct Binary Search Trees possible with n distinct keys} = \frac{{}^{2n}C_{n}}{n+1}$$

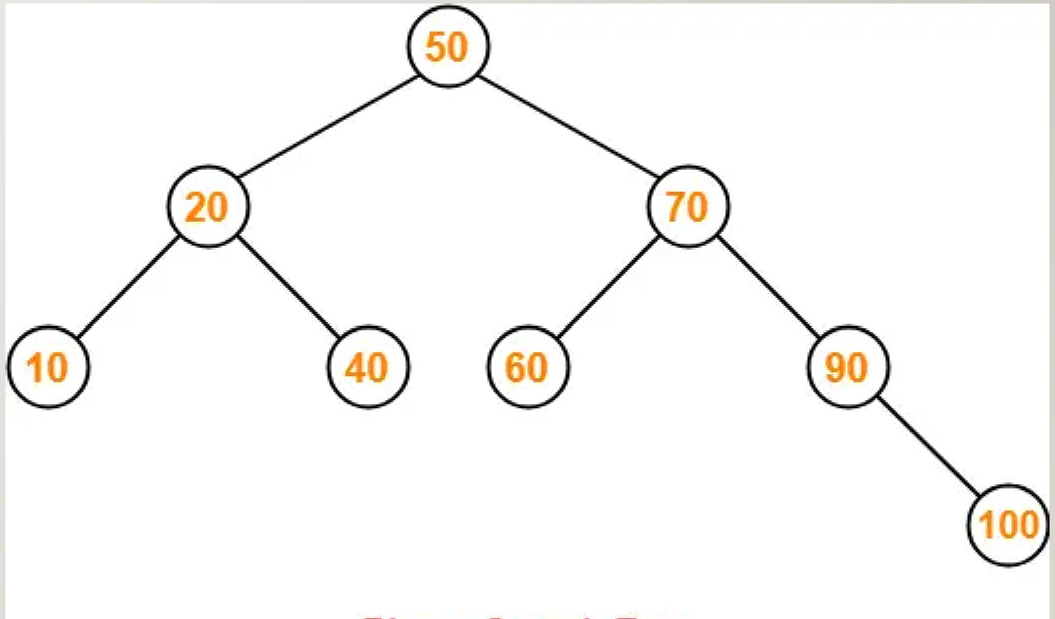
Binary Search Tree Construction

Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

When elements are given in a sequence,

•*Always consider the first element as the root node.*



Binary Search Tree

BST Traversal

A binary search tree is traversed in exactly the same way a binary tree is traversed

Preorder Traversal

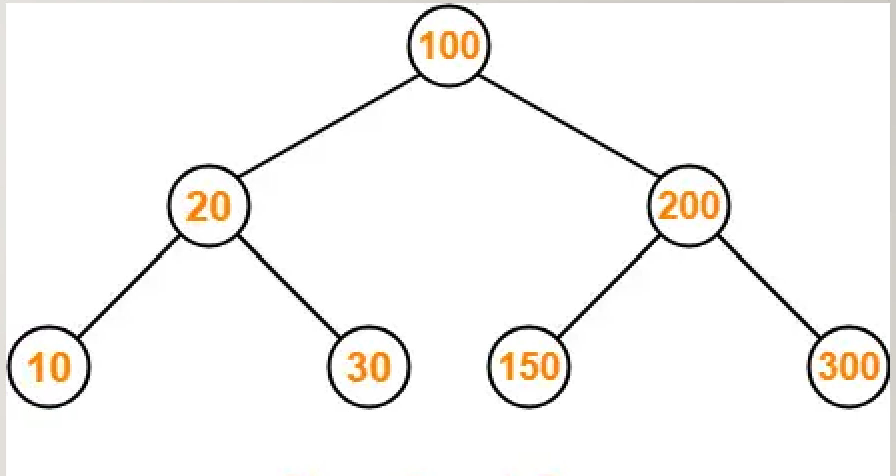
100, 20 , 10 , 30 , 200 , 150 , 300

Inorder Traversal-

10, 20 , 30 , 100 , 150 , 200 , 300

Postorder Traversal-

10, 30 , 20 , 150 , 300 , 200 , 100



Binary Search Tree

Construct a Binary Search Tree from given postorder

- The last element of postorder traversal is always root. We first construct the root.
- Then we find the index of last element which is smaller than root. Let the index be ‘i’.
- The values between 0 and ‘i’ are part of left subtree, and the values between ‘i+1’ and ‘n-2’ are part of right subtree.
- Divide given postorder at index “i” and recur for left and right sub-trees.

Example:

Consider a postorder sequence- {1, 7, 5, 50, 40, 10}

10 is the last element, so we make it root. Now we look for the last element smaller than 10, we find 5.

```
          10
        /    \
       /      \
{1, 7, 5}     {50, 40}
```

We recursively follow above steps for subarrays {1, 7, 5} and {40, 50}, and get the complete tree.

```
       10
      /   \
     5     40
    / \      \
   1   7      50
```

Construct BST from given preorder traversal

- The first element of preorder traversal is always the root.
- We first construct the root. Then we find the index of the first element which is greater than the root.
- Let the index be ‘i’. The values between root and ‘i’ will be part of the left subtree, and the values between ‘i’ and ‘n-1’ will be part of the right subtree.
- Divide the given pre[] at index “i” and recur for left and right sub-trees.

Example: Preorder- {10, 5, 1, 7, 40, 50}

10 is the first element, so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as follows.:

```
            10
           /   \
{5, 1, 7}     {40, 50}
```

We recursively follow the above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree

```
      10
     /    \
    5      40
   / \       \
  1   7       50
```

Construct BST from given inorder traversal

1. If the given array (traversal) is empty return null (emtpy tree)
2. Take the middle value of the given array (traversal) and create a node for it.
3. Take the subarray at the left of the selected array value, and perform the algorithm recursively for it. This returns a node (rooting a subtree), which should be attached as left child of the node created in step
4. Take the subarray at the right of the selected array value, and perform the algorithm recursively for it. This returns a node (rooting a subtree), which should be attached as right child of the node created in step
5. Return the node created in step 2.

Example: inorder traversal is 1,2,3,4,5,6

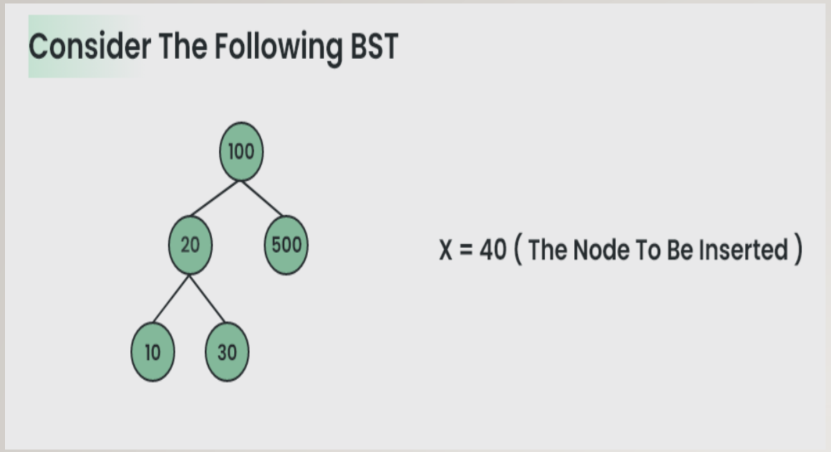
```
      3
     / \
    2   5
   /   / \
  1   4   6
```

Insertion in Binary Search Tree

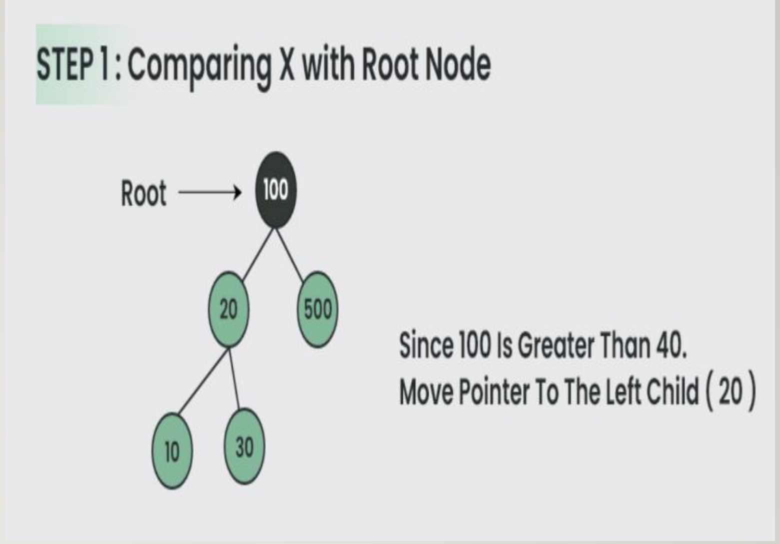
A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

- Check the value to be inserted (say **X**) with the value of the current node (say **val**) we are in:
  - If **X** is less than **val** move to the left subtree.
  - Otherwise, move to the right subtree.
- Once the leaf node is reached, insert **X** to its right or left based on the relation between **X** and the leaf node’s value.

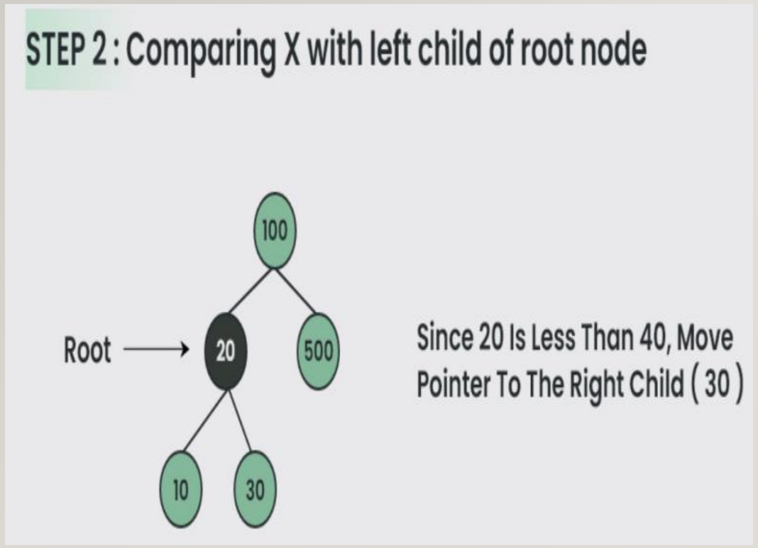
Example:


Consider The Following BST

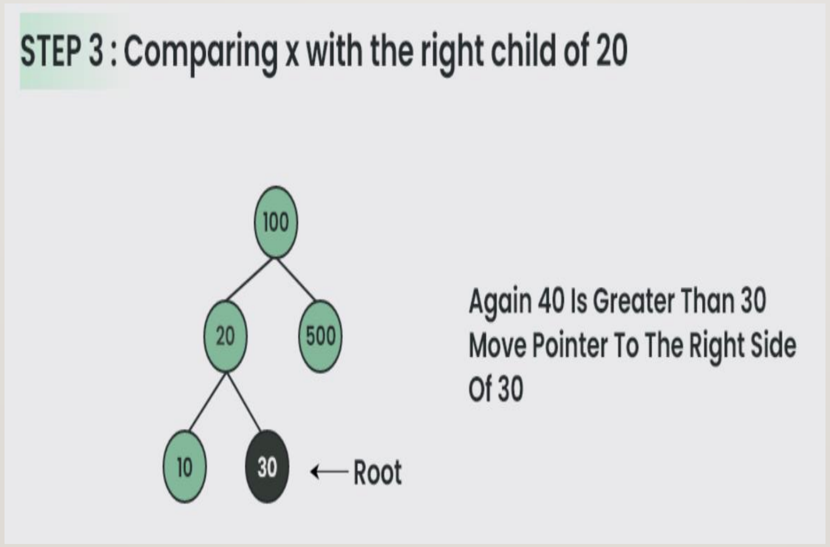
X = 40 (The Node To Be Inserted)


STEP 1 : Comparing X with Root Node

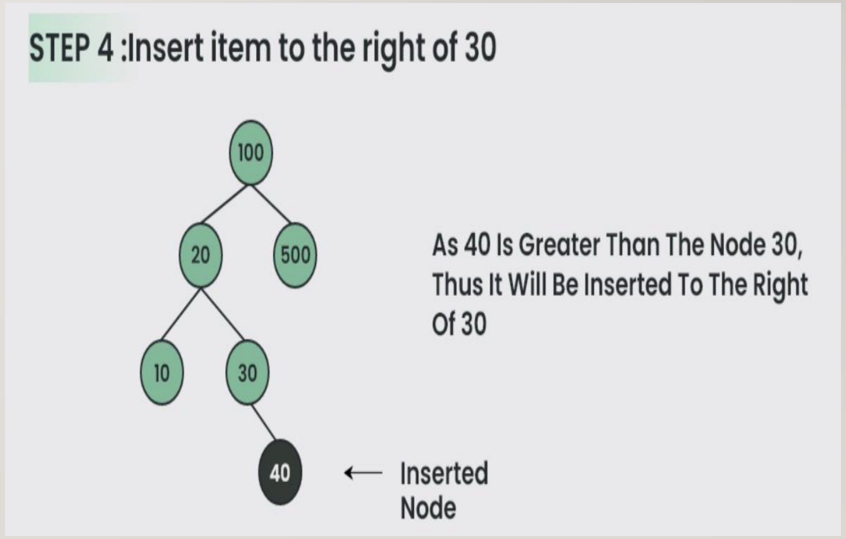
Since 100 Is Greater Than 40.
Move Pointer To The Left Child (20)


STEP 2 : Comparing X with left child of root node

Since 20 Is Less Than 40, Move
Pointer To The Right Child (30)


STEP 3 : Comparing x with the right child of 20

Again 40 Is Greater Than 30
Move Pointer To The Right Side
Of 30


STEP 4 :Insert item to the right of 30

As 40 Is Greater Than The Node 30,
Thus It Will Be Inserted To The Right
Of 30

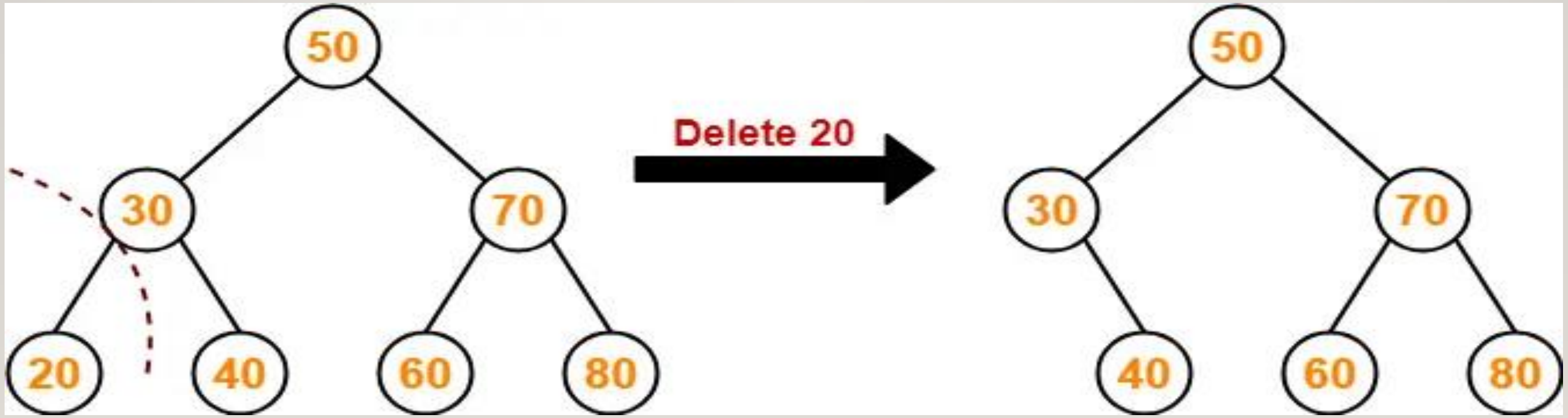
Deletion in Binary Search Tree

Case-01: Deletion Of A Node Having No Child (Leaf Node)-

Just remove / disconnect the leaf node that is to deleted from the tree.

Example-

Consider the following example where node with value = 20 is deleted from the BST-

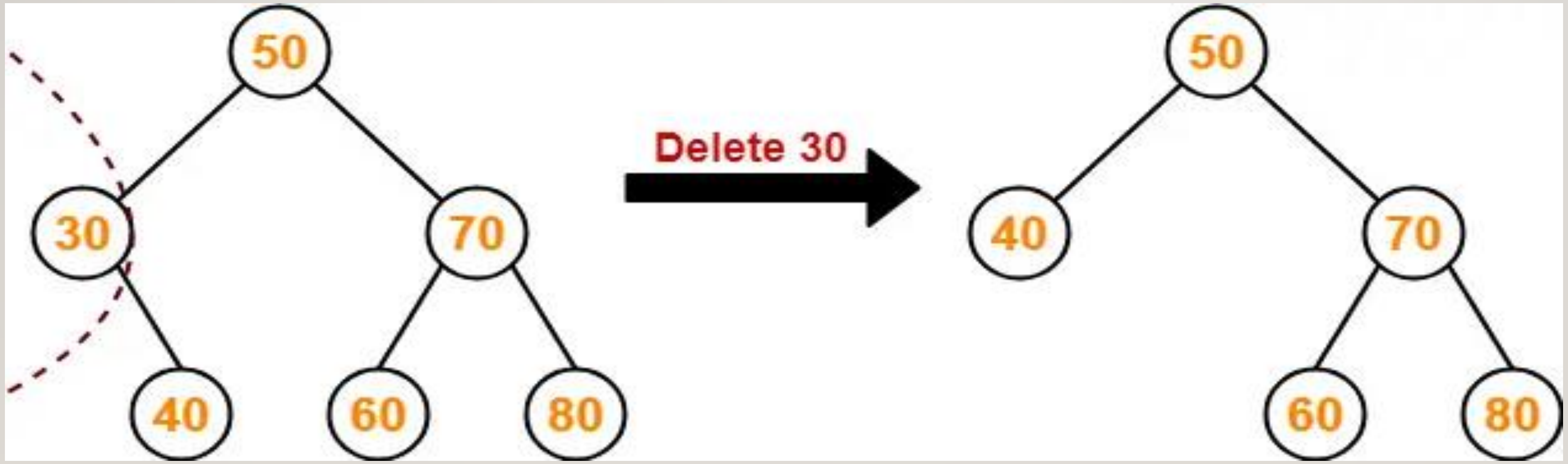


Case-02: Deletion Of A Node Having Only One Child-

Make the child of the deleting node, the child of its grandparent.

Example-

Consider the following example where node with value = 30 is deleted from the BST-



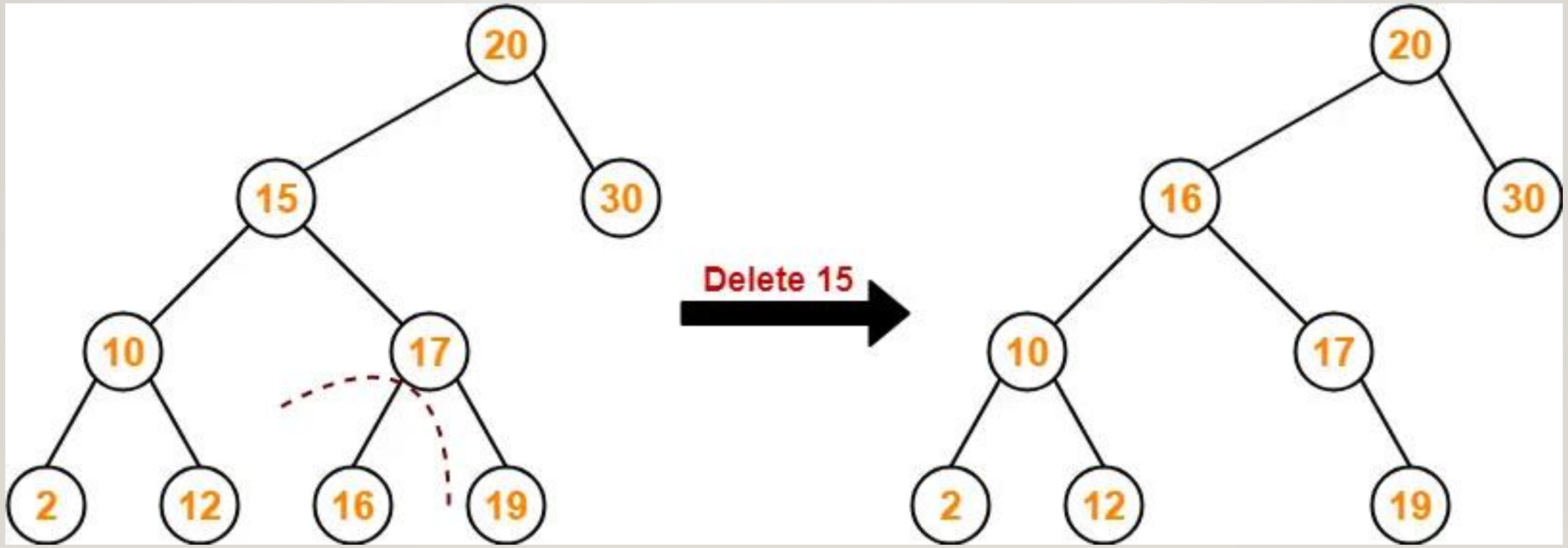
Case-03: Deletion Of A Node Having Two Children-

Method-01:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-

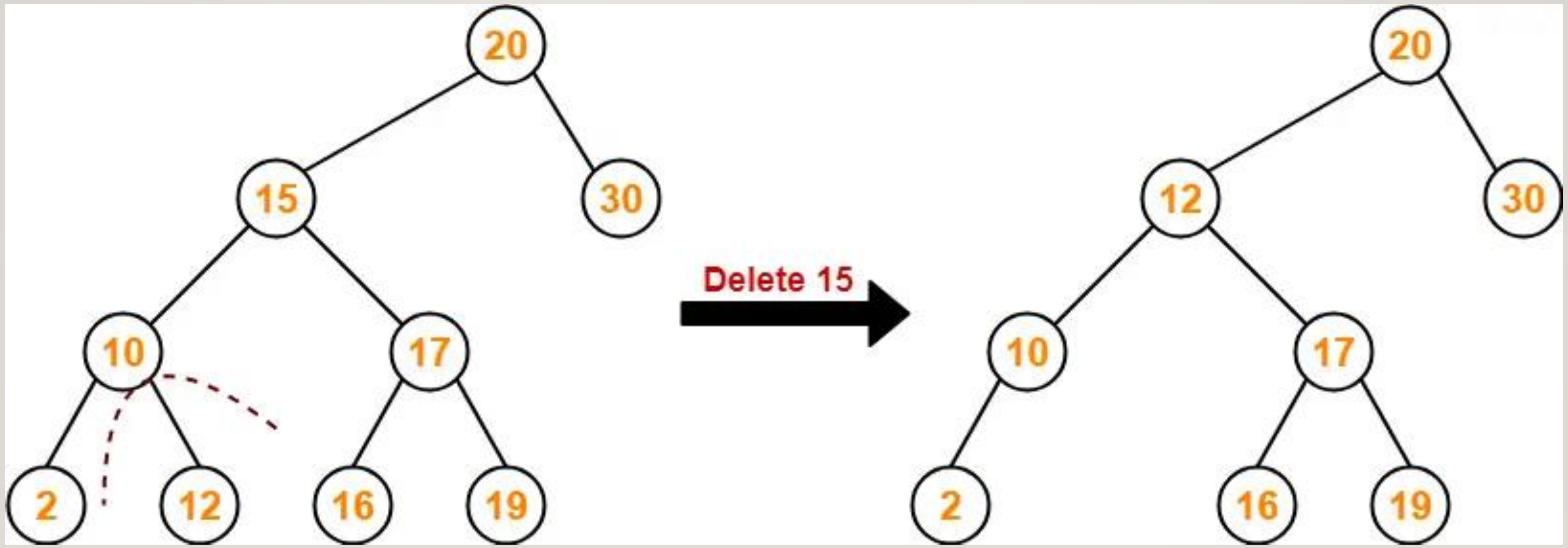


Method-02:

•Visit to the left subtree of the deleting node.
•Pluck the greatest value element called as inorder predecessor.
•Replace the deleting element with its inorder predecessor

Example-

Consider the following example where node with value = 15 is deleted from the BST-



Search Operation

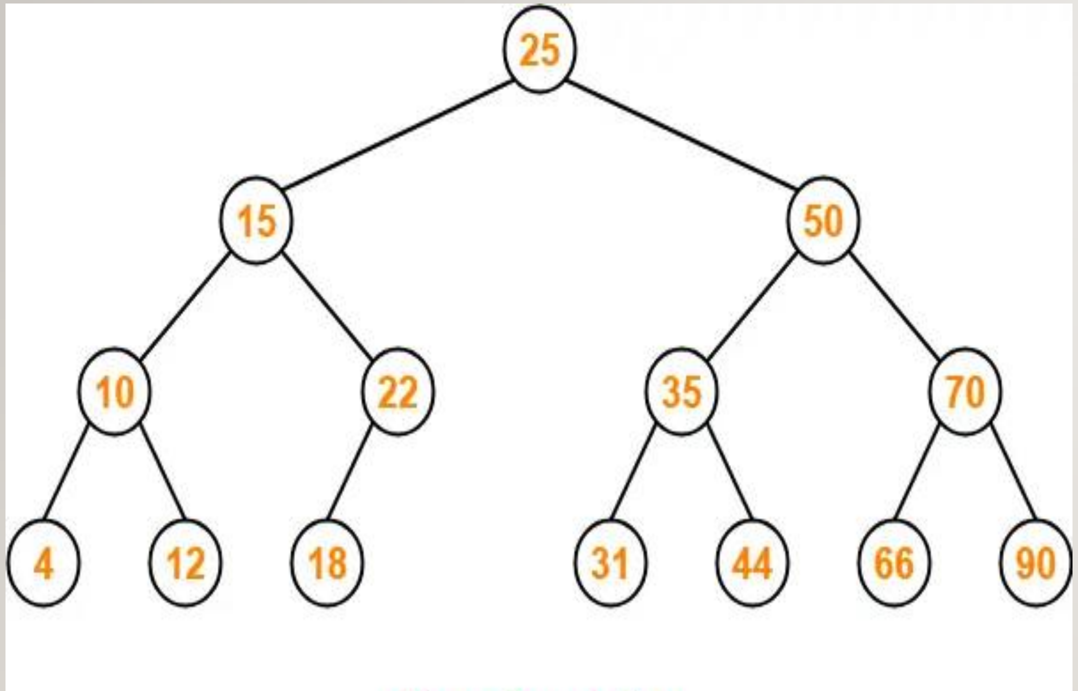
Search Operation is performed to search a particular element in the Binary Search Tree.

For searching a given key in the BST:

- Compare the key with the value of root node.
- If the key is present at the root node, then return the root node.
- If the key is greater than the root node value, then recur for the root node’s right subtree.
- If the key is smaller than the root node value, then recur for the root node’s left subtree.

Example:

Consider key = 45 has to be searched in the given BST-



Binary Search Tree

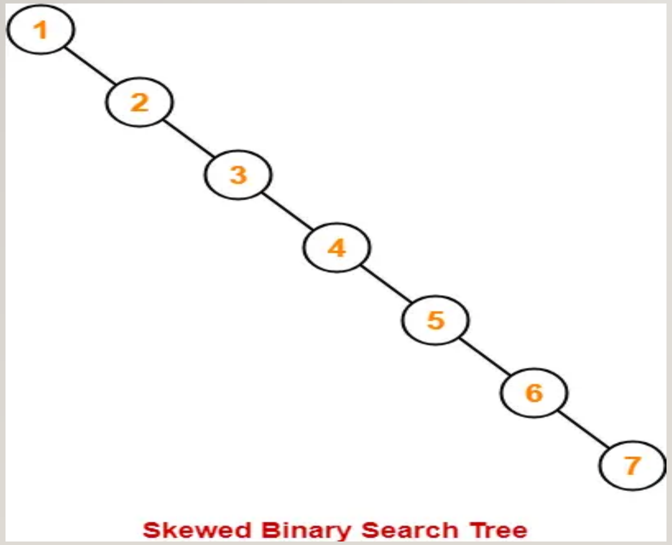
- We start our search from the root node 25.
- As 45 > 25, so we search in 25’s right subtree.
- As 45 < 50, so we search in 50’s left subtree.
- As 45 > 35, so we search in 35’s right subtree.
- As 45 > 44, so we search in 44’s right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

Time Complexity

Time complexity of all BST Operations = O(h).
Here, h = Height of binary search tree

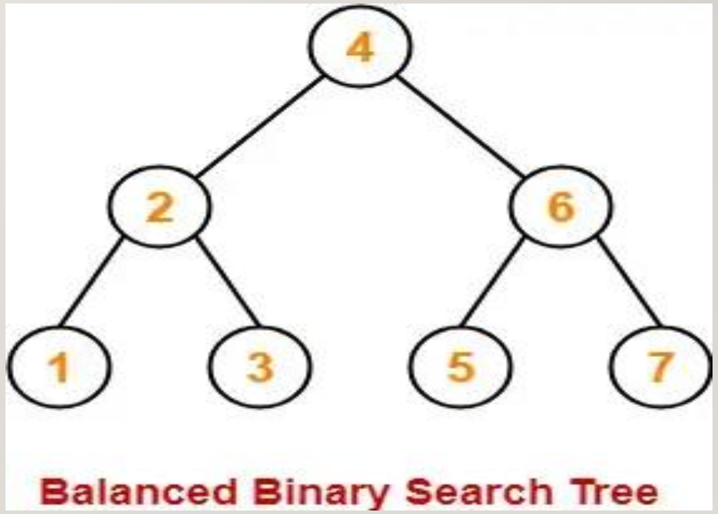
Worst Case-

In worst case,the binary search tree is a skewed binary search tree.
Height of the binary search tree becomes n.
So, Time complexity of BST Operations = O(n).



Best Case-

In best case,the binary search tree is a balanced binary search tree.
Height of the binary search tree becomes log(n).
So, Time complexity of BST Operations = O(logn).


4
2
6
1
3
5
7
Balanced Binary Search Tree