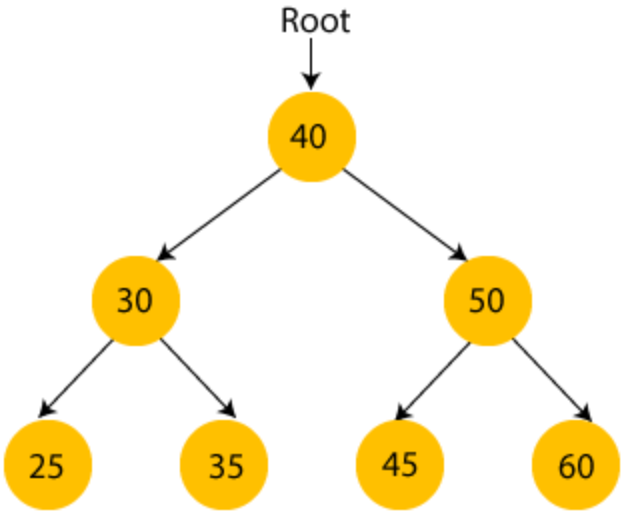
What is a Binary Search tree?

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right sub-trees of the root.

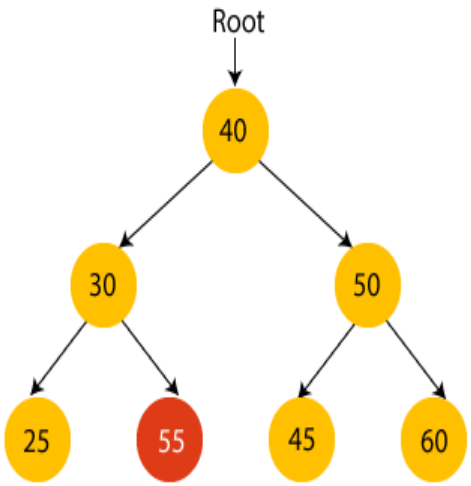
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left sub-tree are smaller than the root node, and all the nodes of the right sub-tree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55.

So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which sub-tree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.

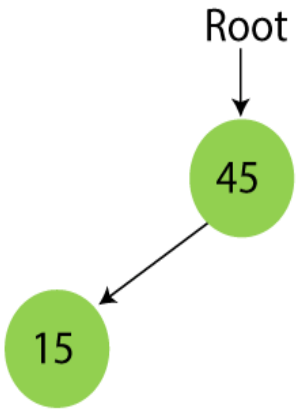
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left sub-tree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right sub-tree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below –

Step 1 - Insert 45.

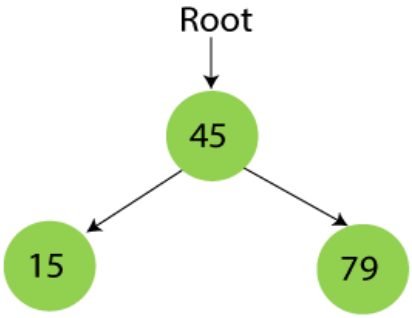
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left sub-tree.



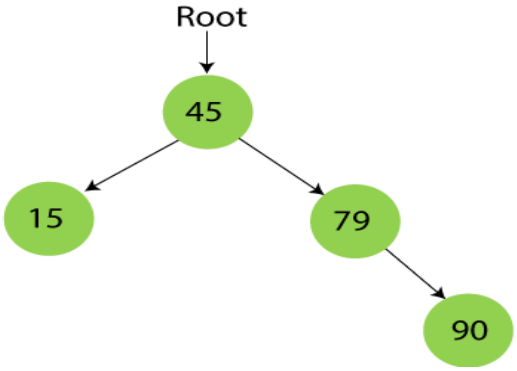
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right sub-tree.



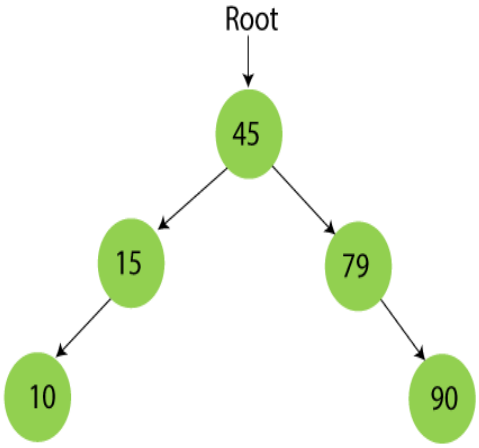
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right sub-tree of 79.



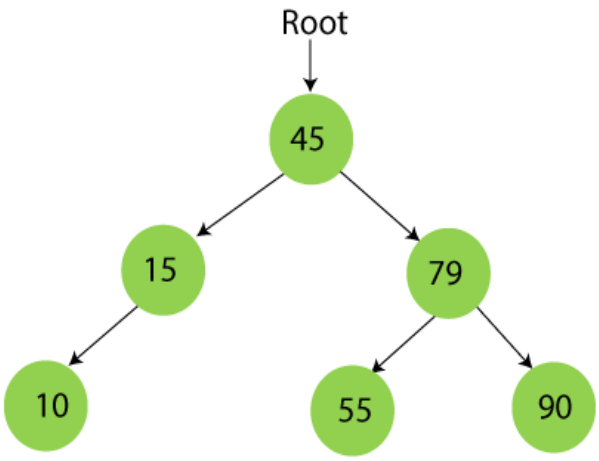
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left sub-tree of 15.



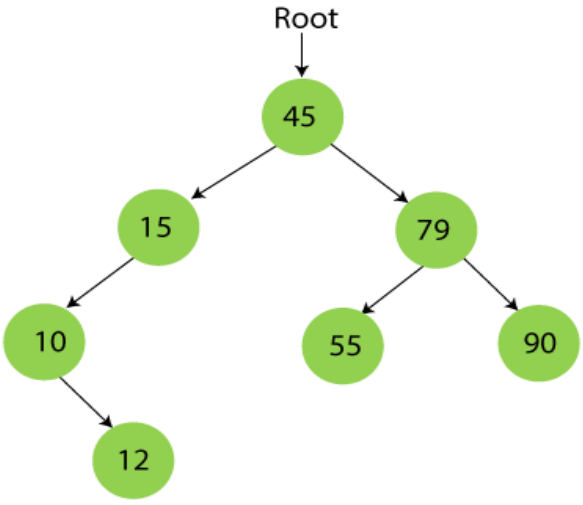
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left sub-tree of 79.



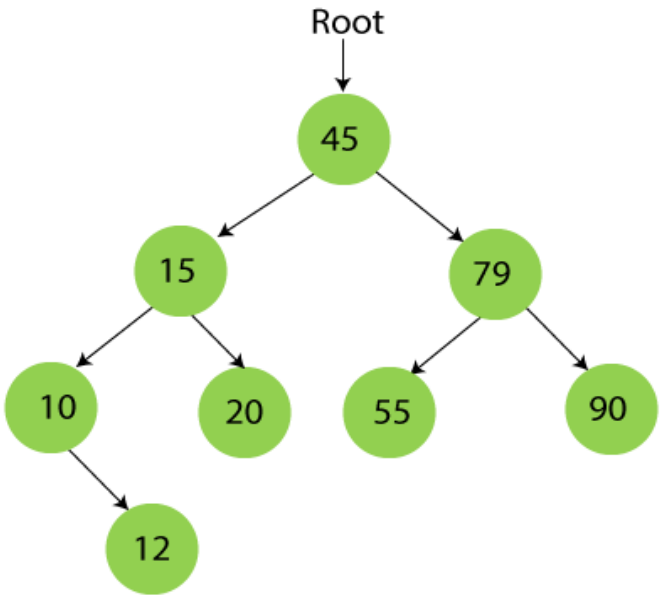
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right sub-tree of 10.



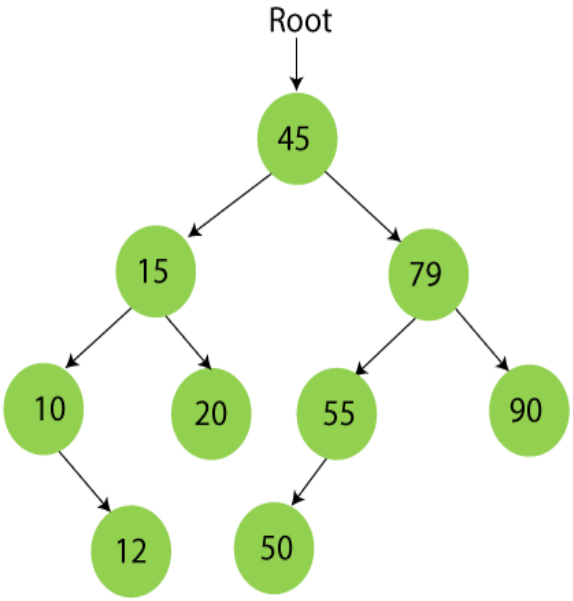
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right sub-tree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left sub-tree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of root node in the tree.
- Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
- Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
- Step 5 - If search element is smaller, then continue the search process in left sub-tree.
- Step 6- If search element is larger, then continue the search process in right sub-tree.
- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node
- Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
- Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Create a new-Node with given value and set its **left** and **right** to **NULL**.
- Step 2 - Check whether tree is Empty.
- Step 3 - If the tree is **Empty**, then set **root** to **new-Node**.
- Step 4 - If the tree is **Not Empty**, then check whether the value of new-Node is **smaller** or **larger** than the node (here it is root node).
- Step 5 - If new-Node is **smaller** than **or equal** to the node then move to its **left** child. If new-Node is **larger** than the node then move to its **right** child.
- Step 6- Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).
- Step 7 - After reaching the leaf node, insert the new-Node as **left child** if the new-Node is **smaller or equal** to that leaf node or else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- Step 1 - **Find** the node to be deleted using **search operation**
- Step 2 - Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- Step 1 - **Find** the node to be deleted using **search operation**
- Step 2 - If it has only one child then create a link between its parent node and child node.
- Step 3 - Delete the node using **free** function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

- Step 1 - **Find** the node to be deleted using **search operation**
- Step 2 - If it has two children, then find the **largest** node in its **left sub-tree** (OR) the **smallest** node in its **right sub-tree**.
- Step 3 - **Swap** both **deleting node** and node which is found in the above step.
- Step 4 - Then check whether deleting node came to **case 1** or **case 2** or else goto step 2
- Step 5 - If it comes to **case 1**, then delete using case 1 logic.
- Step 6- If it comes to **case 2**, then delete using case 2 logic.
- Step 7 - Repeat the same process until the node is deleted from the tree.

DIFFERENCES BETWEEN BINARY TREE & BINARY SEARCH TREE

BINARY TREE	BINARY SEARCH TREE
BINARY TREE is a nonlinear data structure where each node can have at most two child nodes.	BINARY SEARCH TREE is a node based binary tree that further has right and left sub-tree that too are binary search tree.
Types of Binary Trees are: Full binary tree Complete binary tree Extended Binary tree and more	AVL tree Splay Tree T-trees and more
In BINARY TREE there is no ordering in terms of how the nodes are arranged	In BINARY SEARCH TREE the left sub-tree has elements less than the nodes element and the right sub-tree has elements greater than the nodes element.
Binary trees allow duplicate values.	Binary Search Tree does not allow duplicate values.
Time complexity is usually O(n).	Time complexity is usually O(logn).
It is used for retrieval of fast and quick information and data lookup.	It works well at element deletion, insertion, and searching.