**Binary Search Tree** is a node-based binary tree data structure which has the following properties:

* 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.

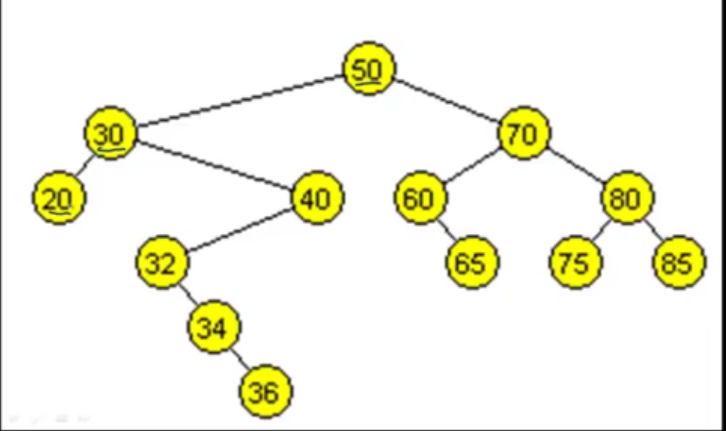
**#** A node with no left and right children is called a leaf.

**#** A node with child/children is called an ancestor.

There are three types of Binary Search Trees:

1. Full binary tree: every node other than **leaf** nodes has 2 child nodes.
2. Complete binary tree: all levels are filled except possibly the last one, and all nodes are filled in as far left as possible.
3. Perfect binary tree: all nodes have two children and all leaves are at the same level.

**A demonstration of a Complete Binary Search Tree :**



**Q** - How is Binary Search Tree better than Linked list?

**Ans -** Binary Search Trees help us to access the middle value faster than in a linked list, by always reducing the elements by half. In a linked list, getting the *middle position* can take

*O*(*n*) time. So the time complexity would go from *O*(log*n*) to *O*(*n*log*n*).

**Q**. When to use Binary Search Trees?

Ans - Implementing a **binary search tree** is useful in any situation where the elements can be compared in a less than / greater than manner.

**A Binary Search Tree should be able to do the following operations.**

* Insertion
* Deletion
* Traversal in pre, in and post orders
* Calculate height of the tree
* Check if a value exists in the tree

**#** To create a Binary Search Tree, we need two classes, Tree and Node. While the tree initiates the tree and forms operation on it, the Node class creates the node and co - operates with Tree class during method calls.

**Method - Insertion** - The idea to insert a value in a BST is that to form a root first, and then traverse left if the next value is smaller or right if the value is greater and then create a node and connect it with the previous value.

**Algorithm of insertion** -

1. Start from the root.

2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.

3. If an element to search is found anywhere, return true, else return false.

**Python Code for insertion -**

**def insert(self, value):**

**if self.value == value:**

**return**

**if self.value > value:**

**if self.left:**

**return self.left.insert(value)**

**else:**

**self.left = Node(value)**

**if self.value < value:**

**if self.right:**

**return self.right.insert(value)**

**else:**

**self.right = Node(value)**

**Method - Find** - similar to Insertion, we start from root and check if the value to find is greater, smaller or equal to the root value. If the value is smaller, traverse recursively left, if the value is greater traverse right.

**Python Code for Find -**

**found = False**

**if self.value == value:** #check to see if the current node's value matches the search value. NOTE - initiation self == root.

**found = True**

**elif self.value == None:**

**found = False**

**elif self.value < value:** #if the value is greater, check the right side of the tree if right side is there.

**if self.right:**

**return self.right.find(value)**

**else:**

**found = False**

**else:**

**if self.left:** #if the value is less, check the left side of the tree if left side is there.

**return self.left.find(value)**

**else:**

**Found = True**

**if found:**

**print("Value {} in the tree".format(value))**

**else:**

**print("value {} not in the tree".format(value))**

**Method - Traversals -**

**In order** - norder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)

**Python code for In order Traversal-**

**def in\_order(self):**

**if self:**

**if self.left:**

**self.left.in\_order()**

**if self != None:**

**print(self.value)**

**if self.right:**

**self.right.in\_order()**

**Pre Order-** Pre Order Traversal starts from the root and prints out values as it keeps going left. Once left check is finished, right values are checked and printed. It is used to create a copy of the tree.

**Algorithm Preorder(tree)**

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

**Python code for Pre Order Traversal -**

**def pre\_order(self):**

**if self and self != None:**

**print(self.value)**

**if self.left:**

**self.left.pre\_order()**

**if self.right:**

**self.right.pre\_order()**

**Post Order -** Post Order traversal only prints out a value if both of it left and right child values are checked. It is used to delete the tree.

**Algorithm Postorder(tree)**

1. Traverse the left subtree, i.e., call Postorder(left-subtree)

2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

**Python code for Post Order -**

def post\_order(self):

if self:

if self.left:

self.left.post\_order()

if self.right:

self.right.post\_order()

if self != None:

print(self.value)

**Method - Delete -**

**While deleting a node from a BST, 3 types of situations arise.**

1. The node to be deleted has both left and right child
2. The node to be deleted has either left or right child
3. The node is a leaf.

1 ) If the node has both left and right child, find the successor node, which is the smallest value of its right subtree and connect it in place of the deletion node. If the successor node has left and right children, connect it to its parent.

2) If the node has either left or right, simply direct the node’s parent to the node’s left or right child.

3) If the node is a leaf, simply nullify the node.

**Python code for Delete method -**

**def delete(self, value, par = None):**

**if self.value == value:**

**if self.right and self.left: ##Instance 1**

**parent = self**

**successor = self.right**

**while successor.left != None:**

**parent = successor**

**successor = successor.left**

**self.value = successor.value**

**successor.value = None**

**if successor.right:**

**parent.left = successor.right**

**elif self.right and not self.left: #Instance 2**

**temp = self.right**

**self.value = self.right.value**

**self.right = temp.right**

**self.left = temp.left**

**elif not self.right and self.left: #Instance 2**

**temp = self.left**

**self.value = self.left.value**

**self.right = temp.right**

**self.left = temp.left**

**else:**

**if self.value > par.value: # instance 3**

**par.right = None**

**else:**

**par.left = None**

**elif self.value < value:**

**if self.right:**

**par = self**

**cur = self.right**

**return self.right.delete(value, par)**

**else:**

**return**

**else:**

**if self.left:**

**par = self**

**return self.left.delete(value, par)**

**else:**

**return**

**Method - Getheight-**

This method helps us to find the maximum height of the tree by comparing the heights of left and right subtrees starting from the root. An empty tree has a height of -1, a tree with only the root has a height of 1 and each level of nodes has a height of 1.

**Python code for getheight() -**

**def getheight(self):**

**if self.left and self.right:**

**return 1 + max(self.left.getheight(), self.right.getheight())**

**elif self.left and not self.right:**

**return 1 + self.left.getheight()**

**elif not self.left and self.right:**

**return 1 + self.right.getheight()**

**else:**

**return 1**