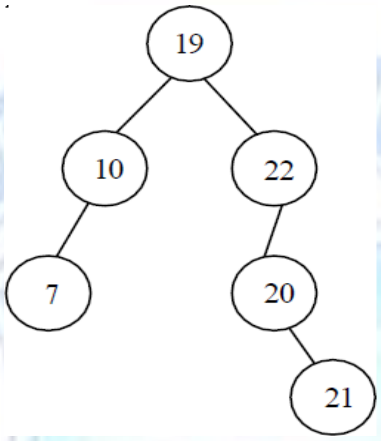
BINARY SEARCH TREE

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 subtrees of the root.



- Every BST is a Binary Tree
- But Not every binary tree is a BST

Declaration Routine for Binary Search Tree

```
class TreeNode:
        def __init__(self, key):
        self.key = key  # The value stored in the node
        self.left = None  # Reference to the left child
        node self.right = None  # Reference to the
        right child node
```

Insert : -

To insert the element X into the tree,

* Check with the root node T

* If it is less than the root,

Traverse the left subtree recursively until it reaches the T ->left equals to NULL. Then X is placed in T -> left.

* If X is greater than the root.

Traverse the right subtree recursively until it reaches the T ->

right equals to NULL. Then X is placed in T-> Right.

Routine to Insert into a Binary Search Tree

```
def insert(self, key):
      if self.root is
         None: self.root
         = Node(key)
      else:
         self._insert_recursive(self.root, key)
  def _insert_recursive(self, current_node,
      key): if key < current_node.key:
         if current_node.left is None:
            current_node.left =
            Node(key)
         else:
            self._insert_recursive(current_node.left
      , key) elif key > current_node.key:
         if current_node.right is
            None: current_node.right
            = Node(key)
         else:
            self._insert_recursive(current_node.right, key)
```

Find : -

- Check whether the root is NULL if so then return NULL.
- Otherwise, Check the value X with the root node value (i.e. T -> data)

  (1) If X is equal to T -> data, return T.

  (2) If X is less than T -> data, Traverse the left of T recursively.

  (3) If X is greater than T -> data, traverse the right of T recursively.

Routine for find Operation

```
def search_recursive(root, key):
   # Base Case: root is None or key is present
```

```
at root if root is None or root.key == key:
    return root
# Key is greater than root's key, search in right
subtree if root.key < key:
    return search_recursive(root.right, key)
# Key is smaller than root's key, search in left
subtree return search_recursive(root.left, key)
```

Delete :

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

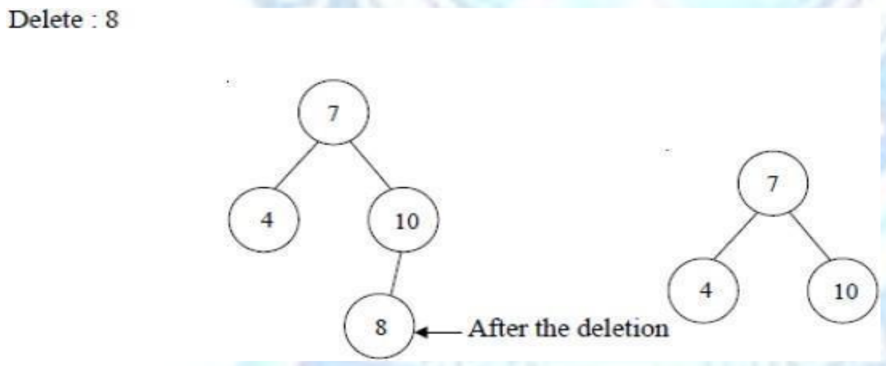
CASE 1: Node to be deleted is a leaf node (i.e) No children.

CASE 2: Node with one child.

CASE 3: Node with two children.

CASE 1 Node with no children (Leaf node)

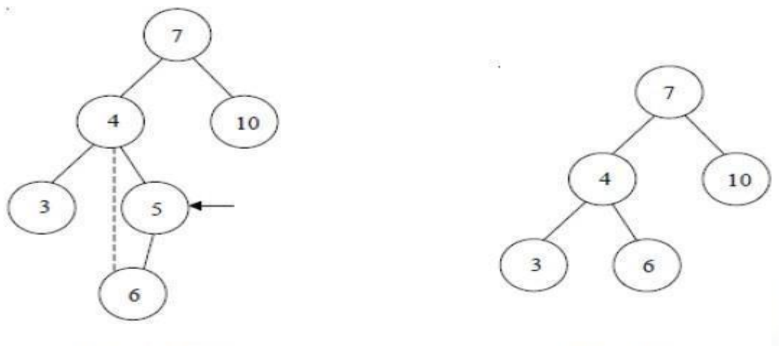
If the node is a leaf node, it can be deleted immediately.



CASE 2 : - Node with one child

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To Delete 5



before deletion After deletion

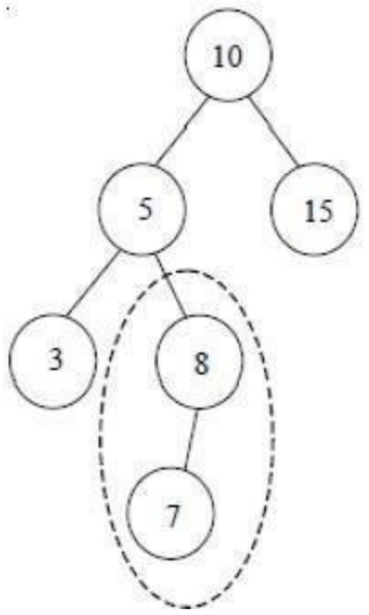
To delete 5, the pointer currently pointing the node 5 is now made to its child node 6.

Case 3: Node with two children

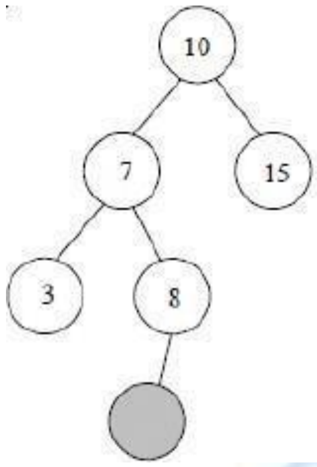
It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

Example 1 :

To Delete 5 :



* The minimum element at the right subtree is 7.



* Now the value 7 is replaced in the position of 5.

* Since the position of 7 is the leaf node delete immediately.

After deleting the node 5