Chapter 5: Binary Search Trees

Binary tree is a tree that each node in it has maximum of two children. Binary search tree (BST) is a binary tree which its elements positioned in special order. In each BST all values(i.e key) in left sub tree are less than values in right sub tree.

Section 5.1: Binary Search Tree - Insertion (Python)

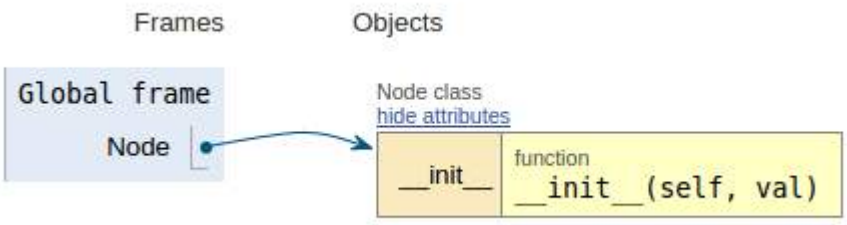
This is a simple implementation of Binary Search Tree Insertion using Python.

An example is shown below:

www.penjee.com

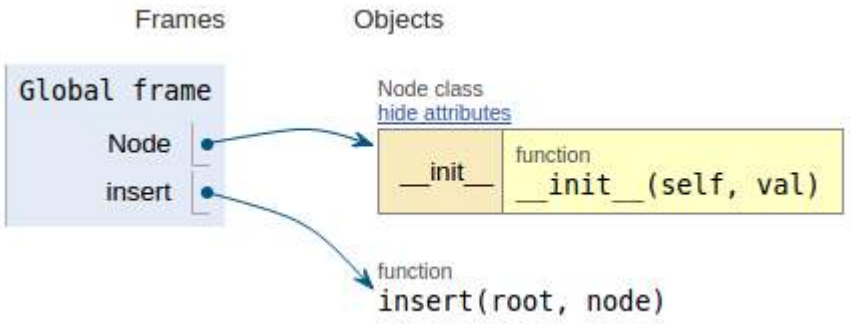
Following the code snippet each image shows the execution visualization which makes it easier to visualize how this code works.

```
class Node:
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.data = val
```

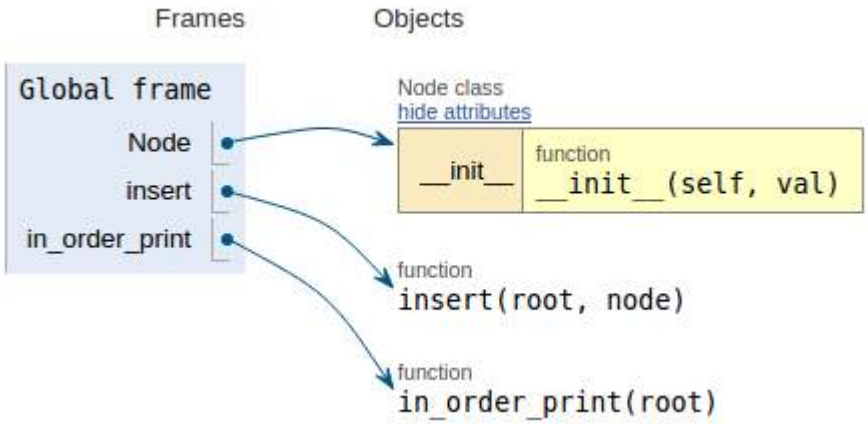



```
def insert(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.l_child is None:
                root.l_child = node
            else:
                insert(root.l_child, node)
        else:
            if root.r_child is None:
```

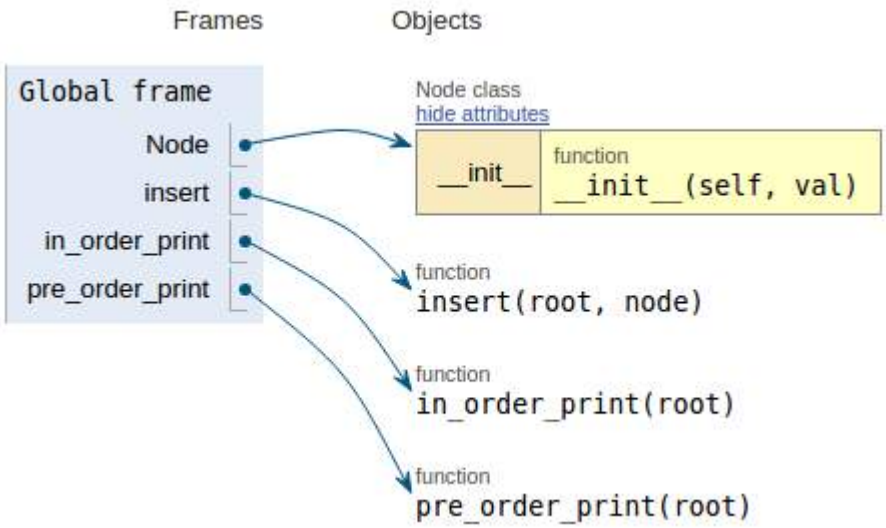
```
                root.r_child = node
            else:
                insert(root.r_child, node)
```




```
def in_order_print(root):
    if not root:
        return
    in_order_print(root.l_child)
    print root.data
    in_order_print(root.r_child)
```



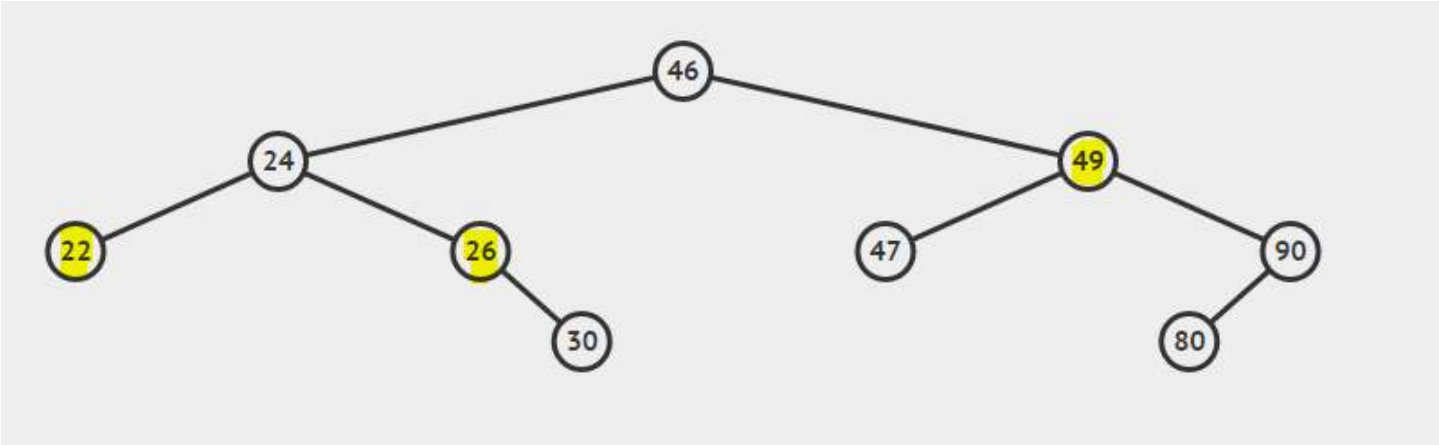

```
def pre_order_print(root):
    if not root:
        return
    print root.data
    pre_order_print(root.l_child)
    pre_order_print(root.r_child)
```



Section 5.2: Binary Search Tree - Deletion(C++)

Before starting with deletion I just want to put some lights on what is a Binary search tree(BST), Each node in a BST can have maximum of two nodes(left and right child).The left sub-tree of a node has a key less than or equal to its parent node's key. The right sub-tree of a node has a key greater than to its parent node's key.

Deleting a node in a tree while maintaining its **Binary search tree property.**



There are three cases to be considered while deleting a node.

- Case 1: Node to be deleted is the leaf node.(Node with value 22).
- Case 2: Node to be deleted has one child.(Node with value 26).
- Case 3: Node to be deleted has both children.(Node with value 49).

Explanation of cases:

1. When the node to be deleted is a leaf node then simply delete the node and pass `nullptr` to its parent node.
2. When a node to be deleted is having only one child then copy the child value to the node value and delete the child **(Converted to case 1)**
3. When a node to be delete is having two childs then the minimum from its right sub tree can be copied to the node and then the minimum value can be deleted from the node's right subtree **(Converted to Case 2)**

Note: The minimum in the right sub tree can have a maximum of one child and that too right child if it's having the left child that means it's not the minimum value or it's not following BST property.

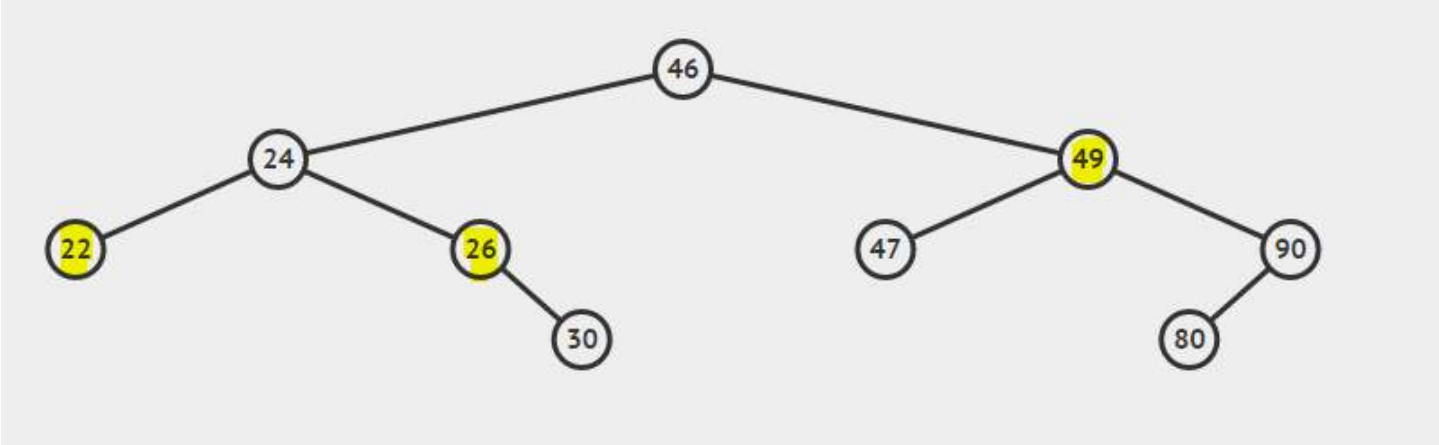
The structure of a node in a tree and the code for Deletion:

```
struct node
{
    int data;
    node *left, *right;
};

node* delete_node(node *root, int data)
{
  if(root == nullptr) return root;
  else if(data < root->data) root->left  = delete_node(root->left, data);
  else if(data > root->data) root->right = delete_node(root->right, data);

  else
  {
    if(root->left == nullptr && root->right == nullptr) // Case 1
    {
      free(root);
      root = nullptr;
    }
    else if(root->left == nullptr)       // Case 2
    {
       node* temp = root;
       root= root->right;
       free(temp);
    }
    else if(root->right == nullptr)      // Case 2
    {
       node* temp = root;
       root = root->left;
       free(temp);
    }
    else                                 // Case 3
    {
       node* temp = root->right;

       while(temp->left != nullptr) temp = temp->left;

       root->data = temp->data;
       root->right = delete_node(root->right, temp->data);
    }
  }
  return root;
}
```

Time complexity of above code is O(*h*), where *h* is the height of the tree.

Section 5.3: Lowest common ancestor in a BST

Consider the BST:



Lowest common ancestor of 22 and 26 is 24

Lowest common ancestor of 26 and 49 is 46

Lowest common ancestor of 22 and 24 is 24

Binary search tree property can be used for finding nodes lowest ancestor

Psuedo code:

```
lowestCommonAncestor(root,node1, node2){

if(root == NULL)
return NULL;



 else if(node1->data == root->data || node2->data== root->data)
    return root;

    else if((node1->data <= root->data && node2->data > root->data)
              || (node2->data <= root->data && node1->data > root->data)){

             return root;
    }

    else if(root->data > max(node1->data,node2->data)){
            return lowestCommonAncestor(root->left, node1, node2);
    }

        else {
            return lowestCommonAncestor(root->right, node1, node2);
      }
        }
```

Section 5.4: Binary Search Tree - Python

```
class Node(object):
    def __init__(self, val):
        self.l_child = None
        self.r_child = None
        self.val = val


class BinarySearchTree(object):
    def insert(self, root, node):
```

```
        if root is None:
            return node

        if root.val < node.val:
            root.r_child = self.insert(root.r_child, node)
        else:
            root.l_child = self.insert(root.l_child, node)

        return root


    def in_order_place(self, root):
        if not root:
            return None
        else:
            self.in_order_place(root.l_child)
            print root.val
            self.in_order_place(root.r_child)

    def pre_order_place(self, root):
        if not root:
            return None
        else:
            print root.val
            self.pre_order_place(root.l_child)
            self.pre_order_place(root.r_child)

    def post_order_place(self, root):
        if not root:
            return None
        else:
            self.post_order_place(root.l_child)
            self.post_order_place(root.r_child)
            print root.val
```

""" Create different node and insert data into it"""

```
r = Node(3)
node = BinarySearchTree()
nodeList = [1, 8, 5, 12, 14, 6, 15, 7, 16, 8]

for nd in nodeList:
    node.insert(r, Node(nd))

print "------In order ---------"
print (node.in_order_place(r))
print "------Pre order ---------"
print (node.pre_order_place(r))
print "------Post order ---------"
print (node.post_order_place(r))
```