Binary Seach Tree Implementation

```
class TreeNode:
    def __init__(self,key):
        self.key = key
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None
 #choose operation
    def operate(self,key,operation):
        if(operation == 'insert'):
            self.root = self.insert_recursive(self.root,key)
        elif(operation == 'inorder'):
            self.inorder_recursive(self.root)
        elif(operation  == 'preorder'):
            self.preorder_recursive(self.root)
        elif(operation == 'postorder'):
            self.postorder_recursive(self.root)
        elif(operation == 'find_min'):
            result = self.find_min(self.root)
            return result
        elif(operation == 'find_max'):
            result = self.find_max(self.root)
            return result
        elif(operation == 'find_ele'):
            key = int(input("Enter the element you want to search"))
            result = self.search_element(self.root,key)
            print("The element search status : ",result)
        elif(operation == 'levelorder'):
            self.levelorder_traversal(self.root)
        elif(operation == 'height'):
            result = self.height_tree(self.root)
            return result
        elif(operation == 'delete'):
            key = int(input("Enter the element you want to delete : "))
            self.inorder_recursive(self.root)
            self.delete_node(self.root,key)
            print("\n")
            self.inorder_recursive(self.root)
#insert elements
    def insert_recursive(self,root,key):
        if(root is None):
            return TreeNode(key)
        if(key < root.key):
            root.left = self.insert_recursive(root.left,key)
        elif(key > root.key):
            root.right = self.insert_recursive(root.right,key)
        return root
  #inorder traversal of BST
    def inorder_recursive(self,root):
        if(root is not None):
            self.inorder_recursive(root.left)
            print(root.key,end=" ")
            self.inorder_recursive(root.right)
#preorder traversal of BST
    def preorder_recursive(self,root):
        if(root is not None):
            print(root.key,end=" ")
            self.preorder_recursive(root.left)
            self.preorder_recursive(root.right)
#post order traversal of BST
    def postorder_recursive(self,root):
        if(root is not None):
            self.postorder_recursive(root.left)
            self.postorder_recursive(root.right)
            print(root.key,end=" ")
#level order traversal of BST
    def levelorder_traversal(self,root):
        if(root is None):
            return None
        queue = []
        queue.append(root)
        while(len(queue) != 0):
            ele = queue.pop(0)
            print(ele.key,end=" ")
            if(ele.left is not None):
                queue.append(ele.left)
            if(ele.right is not None):
                queue.append(ele.right)
#find minimum element in BST
    def find_min(self,root):
        if(root is None):
            return None
        while(root.left is not None):
            root = root.left
        return root.key
#find maximum element in BST
    def find_max(self,root):
        if(root is None):
            return None
        while(root.right is not None):
            root = root.right
        return root.key
```

#search for an element in BST

```
def search_element(self,root,key):
        if(root is None):
            return None
        if(root.key == key):
            return True
        elif(key<root.key):
            return self.search_element(root.left,key)
        else:
            return self.search_element(root.right,key)
        return False
```

#height of a BST

```
    def height_tree(self,root):
        if(root is None):
            return -1
        return(max(self.height_tree(root.left),self.height_tree(root.right))+1)
```

#delete a node from BST

```
   def delete_node(self,root,key):
        if(root is None):
            print("Tree is Empty")
            return
        if(key < root.key):
            root.left = self.delete_node(root.left,key)
        elif(key > root.key):
            root.right = self.delete_node(root.right,key)
        else:
            if(root.left is None):
                return root.right
            elif(root.right is None):
                return root.left
            root.key = self.find_min_node(root.right).key
            root.right = self.delete_node(root.right,root.key)
        return root
    #secondary function used to find the minimum value from RST in BST
    def find_min_node(self,root):
        current = root
        while(current.left is not None):
            current = current.left
        return current

bst = BST()
elements = [10,1,13,133,100,23,22]
for i in elements:
    bst.operate(i,"insert")

print("Inorder Traversal:")
bst.operate(None, 'inorder')

print("\nPreorder Traversal:")
bst.operate(None, 'preorder')

print("\nPostorder Traversal:")
bst.operate(None, 'postorder')
```

```
print("\nminimum element in the tree :")
print(bst.operate(None,'find_min'))

print("maximum element in the tree :")
print(bst.operate(None,'find_max'))

#bst.operate(None,'find_ele')

print("\nlevelorder Traversal:")
bst.operate(None, 'levelorder')

print("\nHeight of the tree : ")
print(bst.operate(None,'height'))
```