ADVANCE DATA STRUCTURES AND ALGORITHMS (ADSA)

Prabhnoor Singh

102115059

3NC3

ASSIGNMENT 1

Write a program for a Binary Search Tree (BST) having functions for the following operations:

Insert an element (no duplicates are allowed),
Delete an existing element,
Traverse the BST (in-order, pre-order, and post-order), Maximum depth, and Minimum depth.

CODE WRITTEN VSCODE (PYTHON) WITH OUTPUT :

Welcome | Untitled-1 | ass1,py

ass1,py > binary_search_tree > __init__

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

class binary_search_tree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self.inserting_element(self.root, key)

    def inserting_element(self, root, key):
        if not root:
            return Node(key)
        if key < root.key:
            root.left = self.inserting_element(root.left, key)
        elif key > root.key:
            root.right = self.inserting_element(root.right, key)
        return root

    def deleting_element(self, key):
        self.root = self._deleting_element(self.root, key)

    def _deleting_element(self, root, key):
        if not root:
            return root
        if key < root.key:
            root.left = self._deleting_element(root.left, key)
        elif key > root.key:
            root.right = self._deleting_element(root.right, key)
        else:
            if not root.left:
                return root.right
            elif not root.right:
                return root.left
            root.key = self.minimum_value_node(root.right).key
            root.right = self._deleting_element(root.right, root.key)
        return root

    def minimum_value_node(self, root):
        while root.left:
            root = root.left
        return root
```

Ln 8, Col 25 Spaces: 4 UTF-8 LF Python 3.9.6 64-bit Go Live

Welcome | Untitled-1 | ass1,py

ass1,py > binary_search_tree > inserting_element

```python
            self._traversal_postorder(root.left, result)
            self._traversal_postorder(root.right, result)
            result.append(root.key)

    def maximum_depth_of_tree(self):
        return self._maximum_depth_of_tree(self.root)

    def _maximum_depth_of_tree(self, root):
        if not root:
            return 0
        return 1 + max(self._maximum_depth_of_tree(root.left), self._maximum_depth_of_tree(root.right))

    def mminimum_depth_of_tree(self):
        return self._mminimum_depth_of_tree(self.root)

    def _mminimum_depth_of_tree(self, root):
        if not root:
            return 0
        if not root.left and not root.right:
            return 1
        if not root.left:
            return 1 + self._mminimum_depth_of_tree(root.right)
        if not root.right:
            return 1 + self._mminimum_depth_of_tree(root.left)
        return 1 + min(self._mminimum_depth_of_tree(root.left), self._mminimum_depth_of_tree(root.right))


# Example Usage:
binary_search_tree = binary_search_tree()
elements = [56, 23, 74, 52, 18, 7, 8]

for element in elements:
    binary_search_tree.insert(element)

print("Inorder Traversal of BST:", binary_search_tree.traveral_inorder())
print("Preorder Traversal of BST::", binary_search_tree.traversal_preorder())
print("Postorder Traversal of BST::", binary_search_tree.traversal_postorder())

print("Maximum Depth in BST:", binary_search_tree.maximum_depth_of_tree())
print("Minimum Depth in BST:", binary_search_tree.mminimum_depth_of_tree())

binary_search_tree.deleting_element(7)
print("After deleting 30:", binary_search_tree.traveral_inorder())

```

Ln 20, Col 20 Spaces: 4 UTF-8 LF Python 3.9.6 64-bit Go Live

Welcome | Untitled-1 | ass1,py

ass1,py > binary_search_tree > inserting_element

```python
        while root.left:
            root = root.left
        return root

    def traveral_inorder(self):
        result = []
        self._traveral_inorder(self.root, result)
        return result

    def _traveral_inorder(self, root, result):
        if root:
            self._traveral_inorder(root.left, result)
            result.append(root.key)
            self._traveral_inorder(root.right, result)

    def traversal_preorder(self):
        result = []
        self._traversal_preorder(self.root, result)
        return result

    def _traversal_preorder(self, root, result):
        if root:
            result.append(root.key)
            self._traversal_preorder(root.left, result)
            self._traversal_preorder(root.right, result)

    def traversal_postorder(self):
        result = []
        self._traversal_postorder(self.root, result)
        return result

    def _traversal_postorder(self, root, result):
        if root:
            self._traversal_postorder(root.left, result)
            self._traversal_postorder(root.right, result)
            result.append(root.key)

    def maximum_depth_of_tree(self):
        return self._maximum_depth_of_tree(self.root)

    def _maximum_depth_of_tree(self, root):
        if not root:
            return 0
        return 1 + max(self._maximum_depth_of_tree(root.left), self._maximum_depth_of_tree(root.right))
```

Ln 20, Col 20 Spaces: 4 UTF-8 LF Python 3.9.6 64-bit Go Live

OUTPUT GENERATED :

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    SQL CONSOLE

shailymandal@Shailys-MacBook-Air adsa %  cd /Users/shailymandal/Desktop/daa/adsa ; /usr/bin/env /usr/bin/python3 /Users/shailymandal/.vscode/ex
tensions/ms-python.python-2023.20.0/pythonFiles/lib/python/debugpy/adapter/../../debugpy/launcher 52552 -- /Users/shailymandal/Desktop/daa/adsa
/ass1,py
Inorder Traversal of BST: [7, 8, 18, 23, 52, 56, 74]
Preorder Traversal of BST:: [56, 23, 18, 7, 8, 52, 74]
Postorder Traversal of BST:: [8, 7, 18, 52, 23, 74, 56]
Maximum Depth in BST: 5
Minimum Depth in BST: 2
After deleting element: [8, 18, 23, 52, 56, 74]
shailymandal@Shailys-MacBook-Air adsa %

Python De...
Python

Ln 117, Col 30   Spaces: 4   UTF-8   LF   Python   3.9.6 64-bit   Go Live
```

CODE :

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

class binary_search_tree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self.inserting_element(self.root, key)

    def inserting_element(self, root, key):
        if not root:
            return Node(key)
        if key < root.key:
            root.left = self.inserting_element(root.left, key)
        elif key > root.key:
            root.right = self.inserting_element(root.right, key)
        return root

    def deleting_element(self, key):
        self.root = self._deleting_element(self.root, key)

    def _deleting_element(self, root, key):
        if not root:
            return root
        if key < root.key:
            root.left = self._deleting_element(root.left, key)
        elif key > root.key:
            root.right = self._deleting_element(root.right, key)
        else:
            if not root.left:
                return root.right
            elif not root.right:
                return root.left
            root.key = self.minimum_value_node(root.right).key
            root.right = self._deleting_element(root.right,
root.key)
```

```
        return root

    def minimum_value_node(self, root):
        while root.left:
            root = root.left
        return root

    def traveral_inorder(self):
        result = []
        self._traveral_inorder(self.root, result)
        return result

    def _traveral_inorder(self, root, result):
        if root:
            self._traveral_inorder(root.left, result)
            result.append(root.key)
            self._traveral_inorder(root.right, result)

    def traversal_preorder(self):
        result = []
        self._traversal_preorder(self.root, result)
        return result

    def _traversal_preorder(self, root, result):
        if root:
            result.append(root.key)
            self._traversal_preorder(root.left, result)
            self._traversal_preorder(root.right, result)

    def traversal_postorder(self):
        result = []
        self._traversal_postorder(self.root, result)
        return result

    def _traversal_postorder(self, root, result):
        if root:
            self._traversal_postorder(root.left, result)
            self._traversal_postorder(root.right, result)
            result.append(root.key)

    def maximum_depth_of_tree(self):
        return self._maximum_depth_of_tree(self.root)

    def _maximum_depth_of_tree(self, root):
        if not root:
            return 0
        return 1 + max(self._maximum_depth_of_tree(root.left),
self._maximum_depth_of_tree(root.right))

    def mminimum_depth_of_tree(self):
        return self._mminimum_depth_of_tree(self.root)
```

```
    def _mminimum_depth_of_tree(self, root):
        if not root:
            return 0
        if not root.left and not root.right:
            return 1
        if not root.left:
            return 1 + self._mminimum_depth_of_tree(root.right)
        if not root.right:
            return 1 + self._mminimum_depth_of_tree(root.left)
        return 1 + min(self._mminimum_depth_of_tree(root.left),
self._mminimum_depth_of_tree(root.right))


# Example Usage:
binary_search_tree = binary_search_tree()
elements = [56, 23, 74, 52, 18, 7, 8]

for element in elements:
    binary_search_tree.insert(element)

print("Inorder Traversal of BST:",
binary_search_tree.traveral_inorder())
print("Preorder Traversal of BST::",
binary_search_tree.traversal_preorder())
print("Postorder Traversal of BST::",
binary_search_tree.traversal_postorder())

print("Maximum Depth in BST:",
binary_search_tree.maximum_depth_of_tree())
print("Minimum Depth in BST:",
binary_search_tree.mminimum_depth_of_tree())

binary_search_tree.deleting_element(7)
print("After deleting 30:", binary_search_tree.traveral_inorder())
```