Assignment 7
Bridge Course: DSA

1. Write a program for AVL tree that has functions for the following operations:
    a. Insert an element (no duplicates are allowed).
    b. Delete an existing element,
    c. Traverse the AVL (in-order, pre-order, and post-order)

In the report, submit a **screenshot of the output tree from console** for each step while:

a. Constructing AVL tree for the following data 21,26,30,9,4,14,28,18,15,10,2,3,7
b. Deleting 18, 14, 28, 15, 3, and 30
c. Traversing in-order, pre-order, and post-order

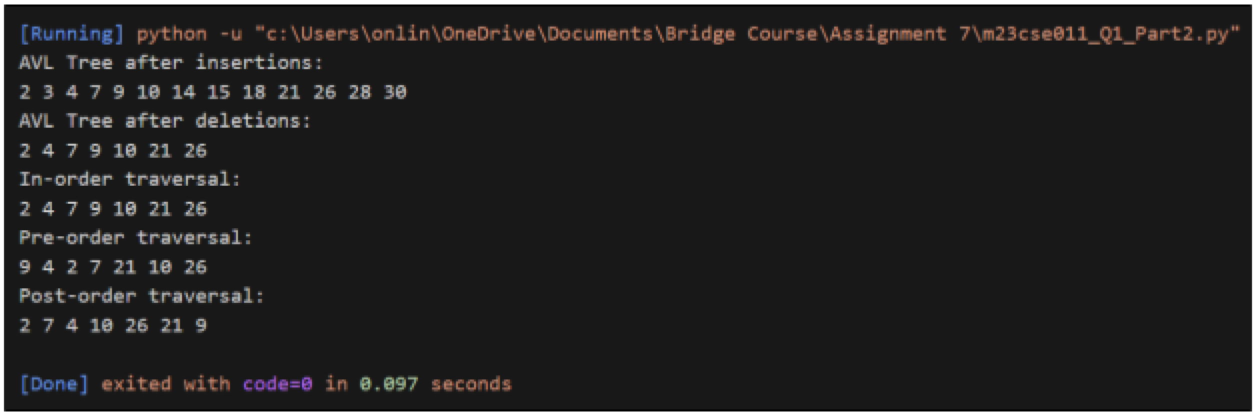
Solution

a. Insert an element
b. Delete an existing element
c. Traverse the AVL (In-order, Pre Order, Post Order)

Please Check CODE for the above solution

For part 2

a. Constructing AVL tree for the following data 21,26,30,9,4,14,28,18,15,10,2,3,7
b. Deleting 18, 14, 28, 15, 3, and 30
c. Traversing in-order, pre-order, and post-order



```
[Running] python -u "c:\Users\onlin\OneDrive\Documents\Bridge Course\Assignment 7\m23cse011_Q1_Part2.py"
AVL Tree after insertions:
2 3 4 7 9 10 14 15 18 21 26 28 30
AVL Tree after deletions:
2 4 7 9 10 21 26
In-order traversal:
2 4 7 9 10 21 26
Pre-order traversal:
9 4 2 7 21 10 26
Post-order traversal:
2 7 4 10 26 21 9

[Done] exited with code=0 in 0.097 seconds
```

2. Do the complexity analysis of the insert, delete, and traverse (In-order, pre-order and post-order) with proper explanation.

Solution

To insert a node into a tree we need to find the position for the new node based on its value. Here is the code and analysis of the complexity involved;

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

def insert(root, value):
    if root is None:
        return TreeNode(value)

    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)

    return root
```

Time Complexity: The time it takes to perform an insertion operation depends on the height of the tree. In the worst case scenario when the tree becomes unbalanced or skewed the height can be linear resulting in an insertion time complexity of O(n). However in cases where trees are balanced the time complexity is typically O(log n) where n represents the number of nodes.

2. Deleting a node from a tree involves locating and removing that node while reorganizing the rest of the tree accordingly. Here is the code and analysis regarding its complexity;

```
def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def delete(root, value):
    if root is None:
        return root

    if value < root.value:
        root.left = delete(root.left, value)
    elif value > root.value:
        root.right = delete(root.right, value)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        temp = minValueNode(root.right)
        root.value = temp.value
        root.right = delete(root.right, temp.value)

    return root
```

Time Complexity; Similar to insertion operations, deleting a node's time complexity depends on how balanced or unbalanced our tree's. In worst case scenarios deletion can take O(n) time. However when dealing with trees in cases deletion usually has a time complexity of O(log n).

3. Traversing a tree means visiting all nodes in an order. There are three types of tree traversal methods; in order pre order and post order traversal techniques.
Here is the code and analysis of complexity, for each type;

a. For Inorder traversal;

During Inorder traversal we first visit the subtree, the current node and finally the right subtree.

```
def inorderTraversal(root):
    result = []
    if root:
        result += inorderTraversal(root.left)
        result.append(root.value)
        result += inorderTraversal(root.right)
    return result
```

Time Complexity; The time complexity of, in order traversal is O(n) where n represents the number of nodes. This is because we need to visit all nodes

b. For Preorder traversal;

During pre order traversal we first visit the current node then the left subtree and finally the right subtree.

```
def preorderTraversal(root):
    result = []
    if root:
        result.append(root.value)
        result += preorderTraversal(root.left)
        result += preorderTraversal(root.right)
    return result
```

The time complexity of order traversal is O(n) just, like the time complexity of inorder traversal.

c. For Preorder traversal;

During pre order traversal we first visit the current node then the left subtree and finally the right subtree.

```
def postorderTraversal(root):
    result = []
    if root:
        result += postorderTraversal(root.left)
        result += postorderTraversal(root.right)
        result.append(root.value)
    return result
```

Time Complexity: Again, the time complexity of postorder traversal is O(n), similar to the previous traversals.