Name of Student: Pushkar Sane		
Roll Number: 45	**Lab Assignment Number: 4**	
Title of Lab Assignment: Implementation of binary tree and merkle tree.		
DOP: 13-08-2024	**DOS: 21-08-2024**	
CO Mapped:	**PO Mapped:**	**Signature:**

Practical No. 4

Aim: Implementation of binary tree and merkle tree

Theory:

1. **Binary Tree**

   A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. This structure is one of the most fundamental concepts in computer science and is widely used due to its efficient data organization and retrieval capabilities. The topmost node in a binary tree is known as the root, and each node, except the root, has a parent node. The nodes without any children are called leaf nodes, and nodes that have at least one child are known as internal nodes.

   Binary trees are primarily used to implement binary search trees (BSTs), where the left child of a node contains values smaller than the node's value, and the right child contains values greater than the node's value. This ordering property allows for efficient searching, insertion, and deletion operations, with an average-case time complexity of O(log n) in balanced trees. However, in the worst case, such as in an unbalanced tree that resembles a linked list, the time complexity can degrade to O(n).

   Binary trees are also foundational for various other tree-based data structures, including AVL trees, red-black trees, and binary heaps. For example, AVL trees and red-black trees are self-balancing binary search trees that automatically maintain a balanced structure during insertions and deletions to ensure optimal search times. Binary heaps, on the other hand, are specialized binary trees used to implement priority queues, where the parent node is always greater than or equal to (in max-heaps) or less than or equal to (in min-heaps) its children.

   Beyond searching and sorting, binary trees are also used in expression parsing, where each node represents an operator, and the children represent operands. This application is crucial in compilers and interpreters for evaluating mathematical expressions. Additionally, binary trees play a key role in Huffman coding, a compression algorithm that generates an optimal prefix code based on the frequency of characters in a data set.

2. Merkle Tree

A Merkle tree, also known as a hash tree, is a cryptographic data structure that extends the concept of a binary tree to efficiently and securely verify the integrity of large data sets. In a Merkle tree, each leaf node represents a hash of a block of data, while each non-leaf (or internal) node contains a hash of the concatenation of its child nodes' hashes. This hierarchical structure culminates in a single hash at the top, known as the root hash or Merkle root, which serves as a unique fingerprint for the entire dataset

.

The primary advantage of a Merkle tree is its ability to verify the integrity of a large set of data with minimal data transmission and computational effort. To verify whether a particular piece of data is part of the dataset, one only needs to check the hashes along the path from the relevant leaf node to the Merkle root. This process, known as a Merkle proof, involves comparing the computed hashes to the stored hashes, ensuring that any tampering or corruption of the data will be immediately detected.

Merkle trees are extensively used in blockchain technology to ensure the integrity and consistency of transactions within blocks. In a blockchain, each block contains a Merkle root that represents the hash of all the transactions within that block. This allows for efficient verification of individual transactions without needing to download the entire blockchain, which is crucial for lightweight clients (SPV clients) in a distributed network.

In addition to blockchains, Merkle trees are also used in distributed systems and peer-to-peer networks, where they help ensure data integrity across distributed nodes. For example, in distributed file systems like IPFS or in peer-to-peer protocols like BitTorrent, Merkle trees enable nodes to verify that they have received unaltered and complete data from other peers.

Merkle trees also play a critical role in optimizing network bandwidth. By enabling partial data verification, they allow systems to transmit only the necessary parts of the tree for verification purposes, reducing the amount of data that needs to be exchanged between nodes.

Overall, the Merkle tree is a powerful tool in cryptographic and distributed systems, offering a scalable and secure method for verifying large data sets, ensuring data integrity, and enabling efficient data transmission in decentralized environments.

Code:

#Binary Tree

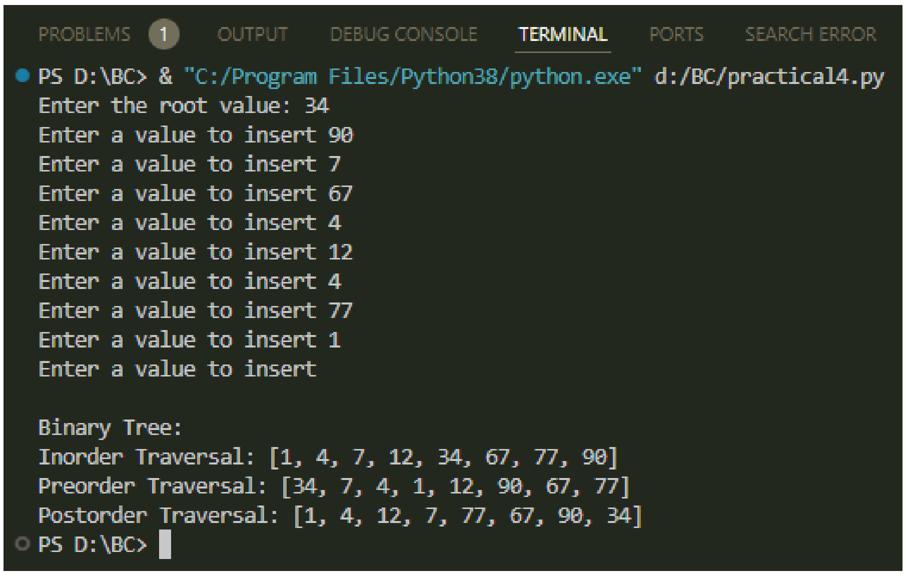
```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def insert(self, data):
       # Compare the new value with the parent node
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data

    # Print the tree
    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print(self.data)
        if self.right:
            self.right.PrintTree()
    # Inorder traversal (Left -> Root -> Right)
    def inorderTraversal(self, root):
        res = []
        if root:
            res = self.inorderTraversal(root.left)
            res.append(root.data)
            res = res + self.inorderTraversal(root.right)
        return res

    # Preorder traversal (Root -> Left -> Right)
    def preorderTraversal(self, root):
        res = []
        if root:
            res.append(root.data)
            res = res + self.preorderTraversal(root.left)
            res = res + self.preorderTraversal(root.right)
        return res

    # Postorder traversal (Left -> Right -> Root)
    def postorderTraversal(self, root):
        res = []
        if root:
            res = self.postorderTraversal(root.left)
            res = res + self.postorderTraversal(root.right)
            res.append(root.data)
        return res

# Take root input from the user
root_value = int(input("Enter the root value: "))
root = Node(root_value)

# Insert values based on user input
while True:
    data = input("Enter a value to insert ")
    if data == '':
        break
    root.insert(int(data))
# Print traversals
print("\nBinary Tree:")
print("Inorder Traversal:", root.inorderTraversal(root))
print("Preorder Traversal:", root.preorderTraversal(root))
print("Postorder Traversal:", root.postorderTraversal(root))
```

#Merkle Tree

```
from typing import List
import hashlib

class Node:
def __init__(self, left, right, value: str, content) -> None:
        self.left: Node = left
        self.right: Node = right
        self.value = value
        self.content = content

    @staticmethod
    def hash(val: str) -> str:
        return hashlib.sha256(val.encode("utf-8")).hexdigest()

    def __str__(self):
        return str(self.value)

class MerkleTree:
    def __init__(self, values: List[str]) -> None:
        self.__buildTree(values)

    def __buildTree(self, values: List[str]) -> None:
        leaves: List[Node] = [Node(None, None, Node.hash(e), e) for e in values]
        if len(leaves) % 2 == 1:
            leaves.append(leaves[-1:][0])
        self.root: Node = self.__buildTreeRec(leaves)

    def __buildTreeRec(self, nodes: List[Node]) -> Node:
        half: int = len(nodes) // 2

        if len(nodes) == 2:
            return Node(nodes[0], nodes[1], Node.hash(nodes[0].value + nodes[1].value), nodes[0].content + "+" + nodes[1].content)

        left: Node = self.__buildTreeRec(nodes[:half])
        right: Node = self.__buildTreeRec(nodes[half:])
        value: str = Node.hash(left.value + right.value)
        content: str = left.content + "+" + right.content
        return Node(left, right, value, content)

    def printTree(self) -> None:
        self.__printTreeRec(self.root)

    def __printTreeRec(self, node) -> None:
        if node is not None:
            if node.left is not None:
                print("Left: " + str(node.left))
                print("Right: " + str(node.right))
            else:
                print("Input")

            print("Value: " + str(node.value))
            print("Content: " + str(node.content))
            print("")
            self.__printTreeRec(node.left)
            self.__printTreeRec(node.right)

    def getRootHash(self) -> str:
        return self.root.value

def mixmerkletree() -> None:
    num_inputs = int(input("Enter the number of elements: "))
    elems = [input(f"Enter element {i + 1}: ") for i in range(num_inputs)]
    print("\nInputs: ")
    print(*elems, sep=" | ")
    print("")

    mtree = MerkleTree(elems)
    print("Root Hash: " + mtree.getRootHash() + "\n")
    mtree.printTree()

mixmerkletree()
```

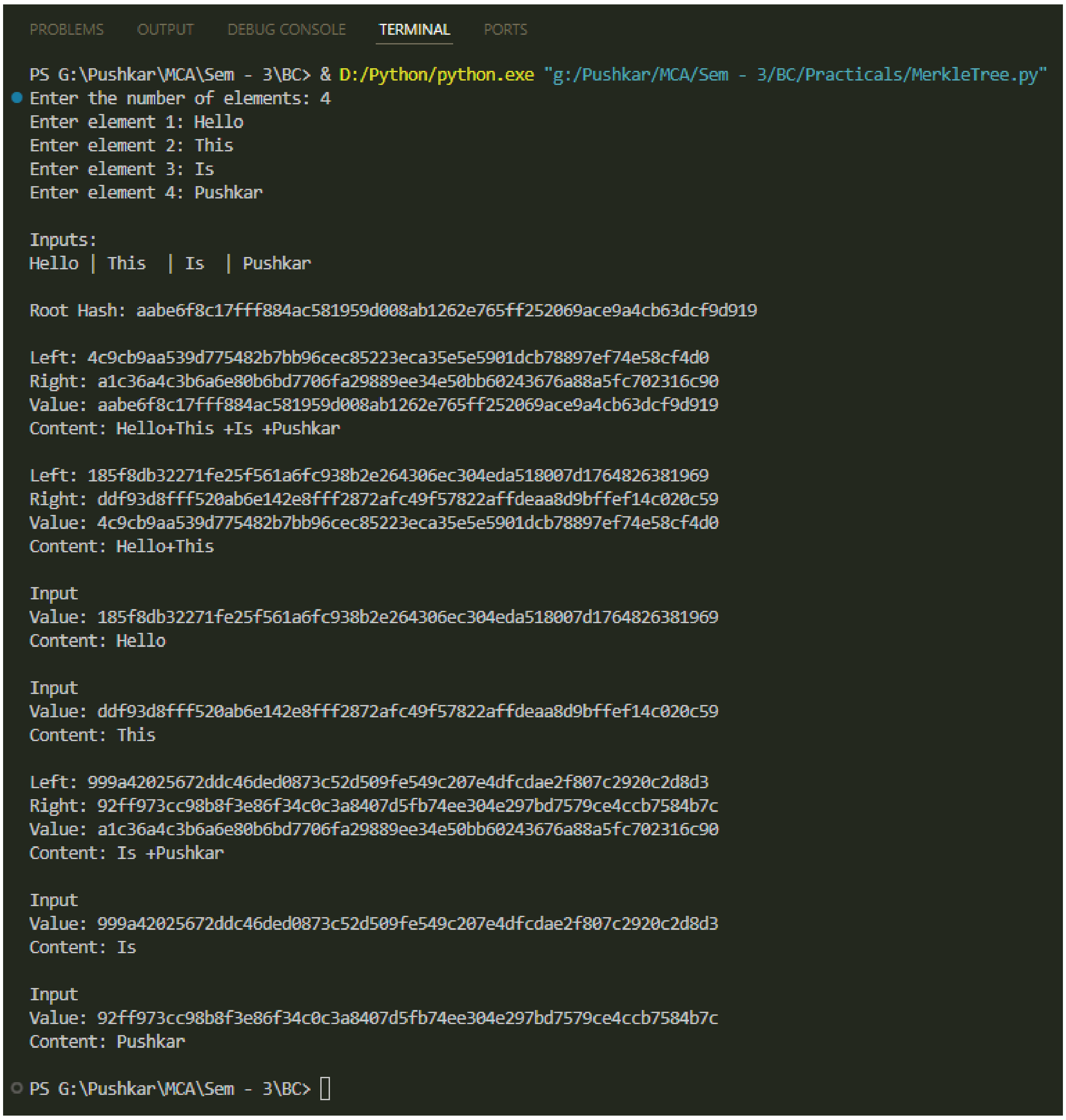
Output:

Binary Tree



```
PS D:\BC> & "C:/Program Files/Python38/python.exe" d:/BC/practical4.py
Enter the root value: 34
Enter a value to insert 90
Enter a value to insert 7
Enter a value to insert 67
Enter a value to insert 4
Enter a value to insert 12
Enter a value to insert 4
Enter a value to insert 77
Enter a value to insert 1
Enter a value to insert

Binary Tree:
Inorder Traversal: [1, 4, 7, 12, 34, 67, 77, 90]
Preorder Traversal: [34, 7, 4, 1, 12, 90, 67, 77]
Postorder Traversal: [1, 4, 12, 7, 77, 67, 90, 34]
PS D:\BC>
```

Merkle Tree



```
PS G:\Pushkar\MCA\Sem - 3\BC> & D:/Python/python.exe "g:/Pushkar/MCA/Sem - 3/BC/Practicals/MerkleTree.py"
Enter the number of elements: 4
Enter element 1: Hello
Enter element 2: This
Enter element 3: Is
Enter element 4: Pushkar

Inputs:
Hello | This  | Is  | Pushkar

Root Hash: aabe6f8c17fff884ac581959d008ab1262e765ff252069ace9a4cb63dcf9d919

Left: 4c9cb9aa539d775482b7bb96cec85223eca35e5e5901dcb78897ef74e58cf4d0
Right: a1c36a4c3b6a6e80b6bd7706fa29889ee34e50bb60243676a88a5fc702316c90
Value: aabe6f8c17fff884ac581959d008ab1262e765ff252069ace9a4cb63dcf9d919
Content: Hello+This +Is +Pushkar

Left: 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
Right: ddf93d8fff520ab6e142e8fff2872afc49f57822affdeaa8d9bffef14c020c59
Value: 4c9cb9aa539d775482b7bb96cec85223eca35e5e5901dcb78897ef74e58cf4d0
Content: Hello+This

Input
Value: 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
Content: Hello

Input
Value: ddf93d8fff520ab6e142e8fff2872afc49f57822affdeaa8d9bffef14c020c59
Content: This

Left: 999a42025672ddc46ded0873c52d509fe549c207e4dfcdae2f807c2920c2d8d3
Right: 92ff973cc98b8f3e86f34c0c3a8407d5fb74ee304e297bd7579ce4ccb7584b7c
Value: a1c36a4c3b6a6e80b6bd7706fa29889ee34e50bb60243676a88a5fc702316c90
Content: Is +Pushkar

Input
Value: 999a42025672ddc46ded0873c52d509fe549c207e4dfcdae2f807c2920c2d8d3
Content: Is

Input
Value: 92ff973cc98b8f3e86f34c0c3a8407d5fb74ee304e297bd7579ce4ccb7584b7c
Content: Pushkar

PS G:\Pushkar\MCA\Sem - 3\BC> 
```

Conclusion:

The implementation of the binary tree demonstrated the core functionalities of insertion and various tree traversal methods, including inorder, preorder, and postorder. The user input facilitated dynamic construction of the binary tree, allowing for a flexible and interactive demonstration of the tree's operations. The code effectively illustrates how binary trees can be used for efficient data organization and retrieval.