Department of Computer Engineering

Academic Term: July-November 2023

Class : B.E Computers-A Sem VII
Subject: Blockchain Technology Lab
Subject Code: CSDL7022

Practical No:	1
Title:	To implement Merkle Hash Tree
Date of Performance:	28/07/2023
Date of Submission:	28/07/2023
Roll No:	9427
Name of the Student:	Atharva Pawar

Evaluation:

Sr. No	Rubrics	Grades
1	Time Line (2)	
2	Output (3)	
3	Code optimization (2)	
4	Post lab (3)	

Signature of the Teacher :

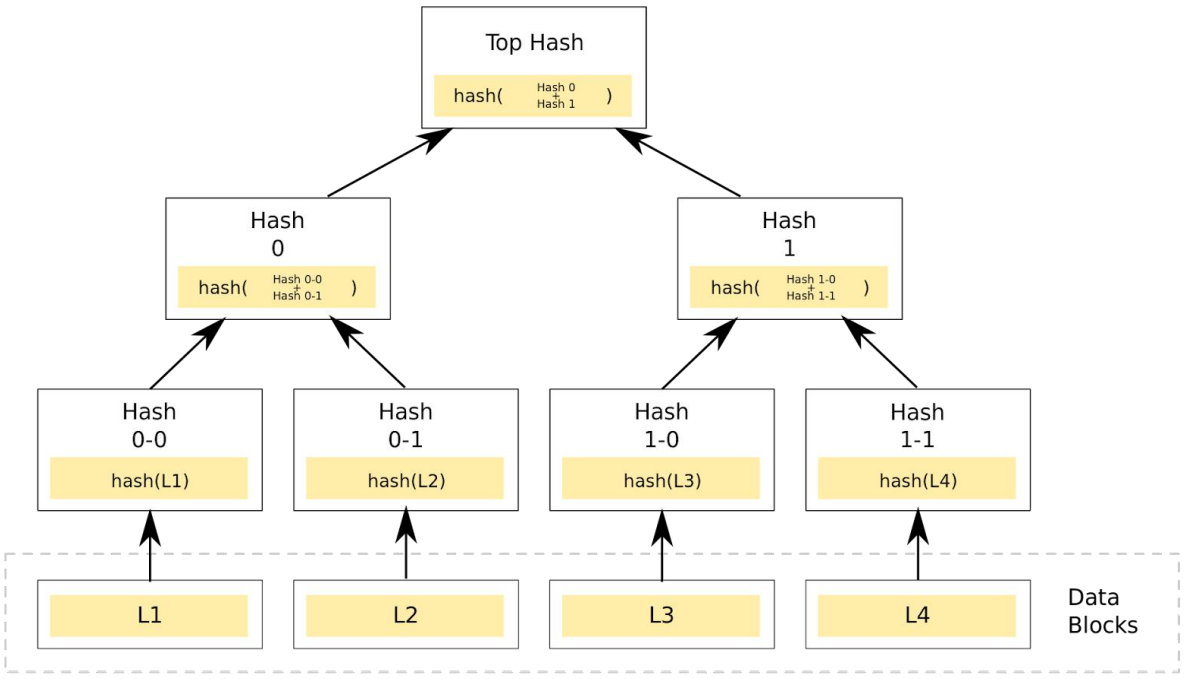
Experiment No. 1

Merkle Tree

Aim: To implement Merkle Hash Tree.

Theory:

Merkle tree also known as hash tree is a data structure used for data verification and synchronization. A Merkle tree is a hash-based data structure that is a generalization of the hash list. It is a tree structure in which each leaf node is a hash of a block of data, and each non-leaf node is a hash of its children. Typically, Merkle trees have a branching factor of 2, meaning that each node has up to 2 children.



Benefits and Protocol:

In various distributed and peer-to-peer systems, data verification is very important. This is because the same data exists in multiple locations. So, if a piece of data is changed in one location, it's important that data is changed everywhere. Data verification is used to make sure data is the same everywhere.

However, it is time-consuming and computationally expensive to check the entirety of each file whenever a system wants to verify data. So, this is why Merkle trees are used. Basically, we want to limit the amount of data being sent over a network (like the Internet) as much as

possible. So, instead of sending an entire file over the network, we just send a hash of the file to see if it matches. The protocol goes like this:

1. Computer A sends a hash of the file to computer B.
2. Computer B checks that hash against the root of the Merkle tree.
3. If there is no difference, we're done! Otherwise, go to step 4.
4. If there is a difference in a single hash, computer B will request the roots of the two subtrees of that hash.
5. Computer A creates the necessary hashes and sends them back to computer B.
6. Repeat steps 4 and 5 until you've found the data blocks(s) that are inconsistent. It's possible to find more than one data block that is wrong because there might be more than one error in the data.

Complexity:

Merkle trees have very little overhead when compared with hash lists. Binary Merkle trees, like the one pictured above, operate similarly to binary search trees in that their depth is bounded by their branching factor, 2. Included below is worst-case analysis for a Merkle tree with a branching factor of k.

	Average	Worst
Space	$O(n)$	$O(n)$
Search	$O(\log_2(n))$	$O(\log_k(n))$
Traversal	$*O(n)$	$*O(n)$
Insert	$O(\log_2(n))$	$O(\log_k(n))$
Delete	$O(\log_2(n))$	$O(\log_k(n))$

Applications:

1. Merkle trees are used in distributed systems for efficient data verification and to check inconsistencies from replicated locations. Apache Cassandra uses Merkle trees to detect inconsistencies between replicas of entire databases. They are efficient because they use hashes instead of full files. Hashes are ways of encoding files that are much smaller than the actual file itself.
2. Currently, their main uses are in peer-to-peer networks such as Tor, Bitcoin, and Git.

Code:

```
# Python code for implememenenting Merkle Tree
from typing import List
import hashlib

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

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

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

```
    def copy(self):
        """
        class copy function
        """
        return Node(self.left, self.right, self.value, self.content, True)

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].copy()) # duplicate last elem if odd number of elements
        self.root: Node = self.__buildTreeRec(leaves)
```

```
    def __buildTreeRec(self, nodes: List[Node]) -> Node:
        if len(nodes) % 2 == 1:
            nodes.append(nodes[-1].copy()) # duplicate last elem if odd number of elements
        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 = f'{left.content}+{right.content}'
        return Node(left, right, value, content)
```

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

    def __printTreeRec(self, node: Node) -> None:
        if node != None:
            if node.left != None:
                print("Left: "+str(node.left))
                print("Right: "+str(node.right))
            else:
```

```
                print("Input")

            if node.is_copied:
                print('(Padding)')
            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() -> List[str]:

    ## testcase:
```

```
    # testcase - 1
    elems = ["GeeksforGeeks", "Computer", "Good", "Morning"]

    # testcase - 2
    # elems = ["Geeksfor", "Geeks", "Computer", "Science","Good", "Morning", "Block", "Chain"]

    # as there are odd number of inputs, the last input is repeated
    output = []
```

```
    output.append("Inputs: ")
    output.append(" | ".join(elems))
    output.append("")
```

```
    mtree = MerkleTree(elems)
    output.append("Root Hash: " + mtree.getRootHash())
    output.append("")

    # Modified version of printTree to add lines to the output list
    def printTreeRec(node: Node, depth: int) -> None:
        if node is not None:
            if node.left is not None:
                output.append("     " * depth + "Left: " + str(node.left))
                output.append("     " * depth + "Right: " + str(node.right))
            else:
                output.append("     " * depth + "Input")
```

```
            if node.is_copied:
                output.append("     " * depth + '(Padding)')
```

```
            output.append("     " * depth + "Value: " + str(node.value))
            output.append("     " * depth + "Content: " + str(node.content))
            output.append("")
            printTreeRec(node.left, depth + 1)
            printTreeRec(node.right, depth + 1)
```

```
    printTreeRec(mtree.root, 0)
    return output


output_list = mixmerkletree()
# print(output_list)
```

```
temp = 0
for line in output_list:
    print(line)
```

Output:

Conclusion: We have successfully implemented Merkel tree.

Review Questions

Q. 1 What is another name for Merkle Tree?

Q. 2 Explain the approach to create Merkle tree.

Q.3 What are the advantages of Merkle tree?

Q. 4 What is Merkle root? Explain with example.

Q. 5 What is time and space complexity of Merkel tree?

Atharva Prashant Tawar (9427) - (Batch-D)

Blockchain Technology : Exp - 1 DATE:

Q1. What is another name for merkle tree?

=> another name for merkle tree is a "hash tree"

Q2. What is approch to create merkle tree?

=> The approach of creating merkle tree is taking a set of data casually, transactions or info., dividing in into smallest segments, then repeatedly hashing segments in pairs until a root hash is generated.

Q3. What are advantages of merkle tree?

=> The advantages of merkle trees include:

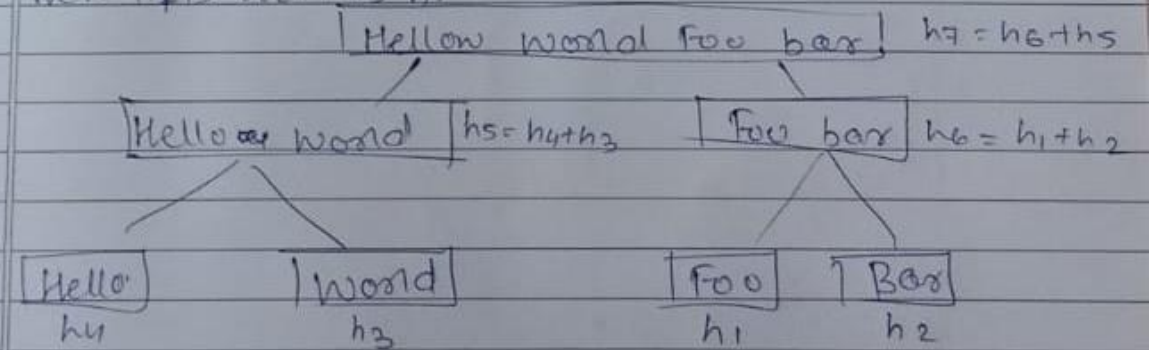
1. Integrity Verification.
They allow secure verification of data integrity in BC.
2. Space Efficiency: Despaite representing large datasets, we only require storage of root hash for verification.

Q4. What is merkle root? Explain with example?

=> Merkle root is a summarized hash of all data segments in merkle tree for eg.

Data segments "Hellow" | "world" | "Foo" | "bar"

Here H_7 is root has h.



DATE:

(Q5) What is time & space complexity of merkle tree?

=> * Time Complexity:

Creation of Merkle tree involves processing of n data items on leaf node level & they are halfed in subsequent levels.

So Time complexity :=

$$n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^{k-1}} + \frac{n}{2^k}$$

at root level, we have only one node so.

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n$$

$$k = \log_2 n$$

So total time is proportional to n So.

Time Complexity = $O(n)$

* Space Complexity:

On same lines as above we have space proportional to no of nodes processed

Space Complexity = $O(n)$