Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

Experiment No. 1

Aim:

Creating a Merkle Tree root hash Tree.

Theory:

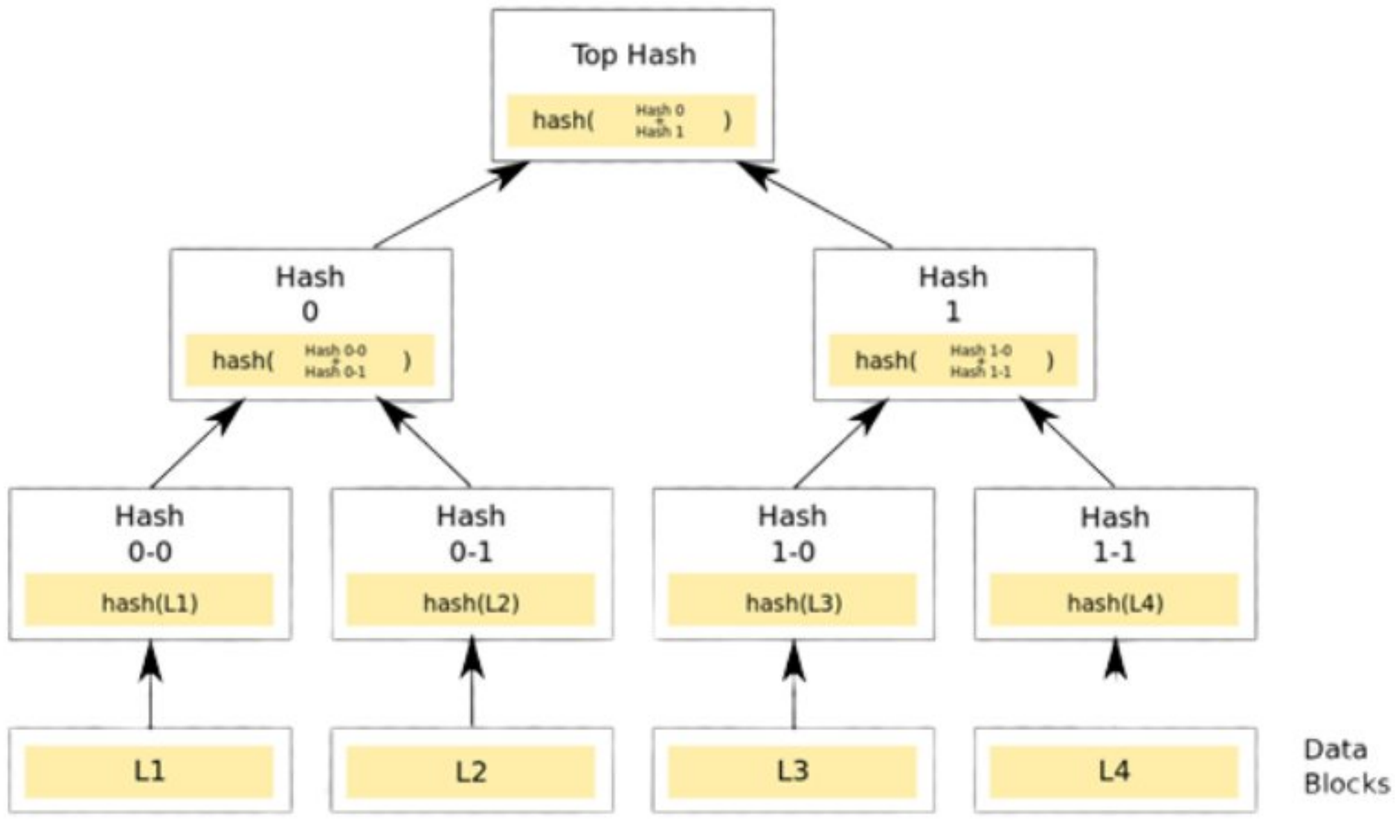
Merkle Tree is a special type of data structure which is completely built using Cryptographic Hash function. Before going deep into the Merkle Tree, let's have a glance at a hash function. The hash function is a function which converts the input data into a fixed length data regardless of the length of input data. The output of the hash function is called 'hash value', 'hashcode' or 'hash' in short. A hash function generates a completely unique hash with a fixed length for each input data. It is guaranteed that two hash functions will never collide for two or more different input data. Let's see an example of the hash function. When hashing the data 'Hash Me', you can see a hexadecimal code is generated which is 16 bytes long(32 characters). In the second example, the data to be hashed is 'HashMe'. The only change we made is the removal of space between the words. But in the result, you can see the hash code is entirely different. But with the same length of 16 bytes. Unlike normal encryption algorithms, no hash can be decoded into original data(plain text). Which means hashing is a one-way cryptographic method or we can say hashing is irreversible.

Input Data	Hash Code
Hash Me	644D45DD9DF9D3B92E5D773E13DF5215
HashMe	54FF7B00CC8493E7948F5DDF08396CBC

When hashing the data 'Hash Me', you can see a hexadecimal code is generated which is 16 bytes long(32 characters). In the second example, the data to be hashed is 'HashMe'. The only change we made is the removal of space between the words. But in the result, you can see the hash code is entirely different. But with the same length of 16 bytes. Unlike normal encryption algorithms, no hash can be decoded into original data(plain text). Which means hashing is a one-way cryptographic method or we can say hashing is irreversible.

Different popular hash functions are MD5, SHA256, SHA1, and SHA512. Each of them follows different cryptographic algorithms. SHA256 algorithm is the popular one which is followed by

most of the blockchains including Bitcoin. As already mentioned, the Merkle tree is totally builtusing a hashing function. In short, a Merkle tree formation is the process of making a single hash from a group of hashes. Let's see how it works with an illustration.



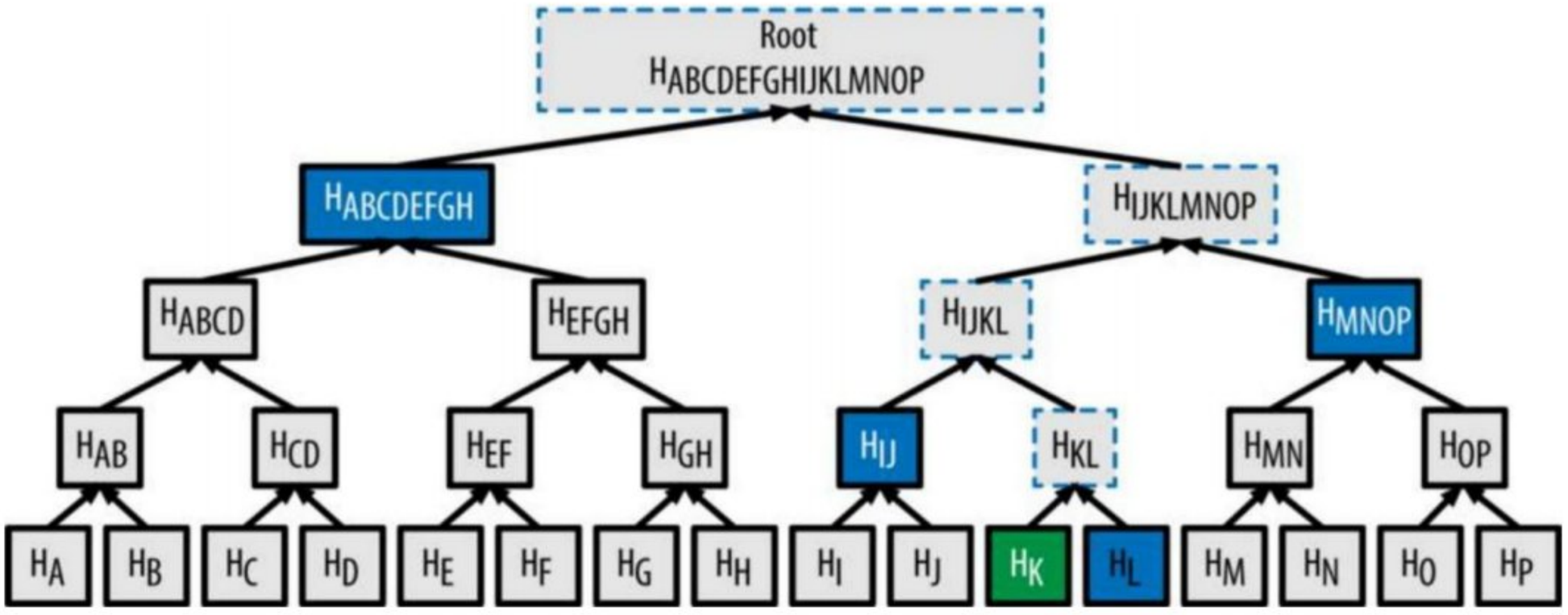
The bottom blocks L1, L2, L3, and L4 are the data blocks. The first step is to hash each data block using a specific hashing algorithm, say SHA256. So that the blocks Hash 0-0, Hash 0-1, Hash 1-0, Hash 1-1 are formed. This is the initial step for building a Merkle Tree. These blocks can be called Leaves of the Merkle Tree. The minimum number of Leaves should be two and there is no upper limit.

Next step is, hashing two adjacent hashes into a single hash. That is, Hash 0-0 is concatenated with Hash 0-1 and again hashed by SHA256 to get Hash 0. The same process is done for Hash 1-0 and Hash 1-1 blocks to produce Hash 1. The process will continue until a single hash is formed. The final hash is called Root of the Merkle Tree. It is called Merkle Root.

The validation of the existence of data in a Merkle Tree is an important process. The data to be validated is called Target Hash. As already mentioned, hashes are irreversible. That is, it is impossible to derive the target hashes from the Merkle Root. So, no data can be validated by decoding the Merkle root. The only way to validate data in a Merkle tree is to rebuild the tree. Target Hash alone is not enough to rebuild the tree. One method to rebuild the Merkle Tree is by collecting all the leaves and arranging them in the same order and building the tree again.

But this is a constraint for many applications and also time and storage consuming. If we study a Merkle Tree formation in a little more depth, we can see that all of the leaves are not required for

building the tree. Instead, a proof to reach at Merkle Root from Target Hash can be formed. This proof is called the Merkle Proof. Merkle Proof is nothing but a collection(array) of hashes which is explained below.



In this Merkle Tree, H_K(red block) is our target hash. HKL is formed by hashing HK with HL. H_{IJKL} is formed by hashing H_{IJ} with H_{KL} And so on to reach $H_{ABCDEFGHIJKLMNOP}$. It is clear that, to validate H_K in the Merkle tree, the blocks H_L, H_{IJ}, H_{MNOP}, $H_{ABCDEFGH}$ (yellow blocks) are the only required data instead of all the leaves H_A, H_B, H_C,. H_P. Here, the Merkle Proof of H_K in this tree is the array of hashes H_L, H_{IJ}, H_{MNOP}, and $H_{ABCDEFGH}$.

Program :

```
from typing import List
import typing
import hashlib

class Node:
    definit(self, left, right, value: str)-> None:
        self.left: Node = left
        self.right: Node =
        right self.value =
        value
    @staticmethod
    def hash(val: str)-> str:
        return hashlib.sha256(val.encode('utf-
```

```
    8')).hexdigest() @staticmethod
    def doubleHash(val: str)-> str:

    returnNode.hash(Node.hash(val))

class MerkleTree:
    definit(self,      values:      List[str])-
      >None: self.buildTree(values)
    defbuildTree(self, values: List[str])-> None:
      leaves: List[Node] = [Node(None, None, Node.doubleHash(e)) for e in
    values] if len(leaves) % 2 == 1:
      leaves.append(leaves[-1:][0]) # duplicate last elem if odd number of elements
    self.root: Node = self.buildTreeRec(leaves)
defbuildTreeRec(self, nodes: List[Node])-> Node:
    half: int = len(nodes) // 2
    if len(nodes) == 2:
      return      Node(nodes[0],      nodes[1],      Node.doubleHash(nodes[0].value      +
    nodes[1].value)) left: Node = self.buildTreeRec(nodes[:half])
    right:    Node    =    self.buildTreeRec(nodes[half:])
    value:    str    =    Node.doubleHash(left.value    +
    right.value) return Node(left, right, value)
def printTree(self)-> None:
    self.printTreeRec(self.root)
defprintTreeRec(self,  node)->None:
    if node != None:
      print("Node:",node.value)
      self.printTreeRec(node.left
      )
      self.printTreeRec(node.right
    ) def getRootHash(self)-> str:
      return self.root.value

def test()-> None:
    n=int(input("Enter        number        of
    elements:")) elems=[]
    for          i          in          range(n):
      elems.append(input("Enter
      element:"))
```

```
    mtree = MerkleTree(elems)
    print("Hash     Value:     ",mtree.getRootHash())
    print("Tree:")
    mtree.printTree()

    test()
```

Output:

```
PS D:\Isha> python bc_exp1.py
Enter number of elements:3
Enter element:Hello
Enter element:This is Blockchain
Enter element:Experiment 1
Hash Value:  89a9856a4f2591dc4c2e541aa98748a7ff8b2cf301827ad79123d73dcd6d3548
Tree:
Node: 89a9856a4f2591dc4c2e541aa98748a7ff8b2cf301827ad79123d73dcd6d3548
Node: 4b91920c19655a50051925562e4cfb36937a9be0030a34a083c2e9e54f91ba08
Node: 52c87cd40ccfbd7873af4180fced6d38803d4c3684ed60f6513e8d16077e5b8e
Node: 54c34b58ccb71be86980b9b530a97cd15a8dc55d8c0057141da6bcae22123903
Node: 98fd68d19f1b71a9df1d07c99de9a8638da41e67a41c7306bd80e86952b16aa6
Node: a2fdddf2a13748044fd60981394fbe7652ceab161c9f7c95db440950bd35ee85
Node: a2fdddf2a13748044fd60981394fbe7652ceab161c9f7c95db440950bd35ee85
PS D:\Isha>
```

Conclusion:

Q1. How do you create a Merkle Tree root hash for a set of data blocks, and what is its significance in data verification?