THEORY DA

CSE1007 Java Programming

(B.Tech. COMPUTER SCIENCE AND ENGINEERING)

FALL SEMESTER 2021-2022

Name:	**VASU JAISWAL**
Reg. No:	**20BCE2856**
Slot:	**B1+TB1**
Faculty:	**RA K Saravanaguru**

VIT – A Place to Learn; A Chance To Grow

INTRODUCTION TO MERKLE TREE :-

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.

Merkle trees are used in distributed systems for efficient data verification. 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. Currently, their main uses are in peer-to-peer networks such as Tor, Bitcoin, and Git.

The concept of Merkle Tree is named after Ralph Merkle, who patented the idea in 1979. Fundamentally, it is a data structure tree in which every leaf node labelled with the hash of a data block, and the non-leaf node labelled with the cryptographic hash of the labels of its child nodes. The leaf nodes are the lowest node in the tree.

Merkle trees are particularly effective in distributed systems where two separate systems can compare the data on each node via a Merkle tree and quickly determine which data sets (subtrees) are lacking on one or the other system. Then only the subset of missing data needs to be sent. Cassandra, based on Amazon's Dynamo, for example, uses Merkle trees as an anti-entropy measure to detect inconsistencies between replicas.

Architecture of Merkle Tree :-

Merkle trees have been an essential key to data verification throughout the history of computers. Their structure helps to verify the consistency of data content. Its architecture helps to speed up security authentication in big data applications. It is a complete binary tree, and each node is to hash the value from its child node.

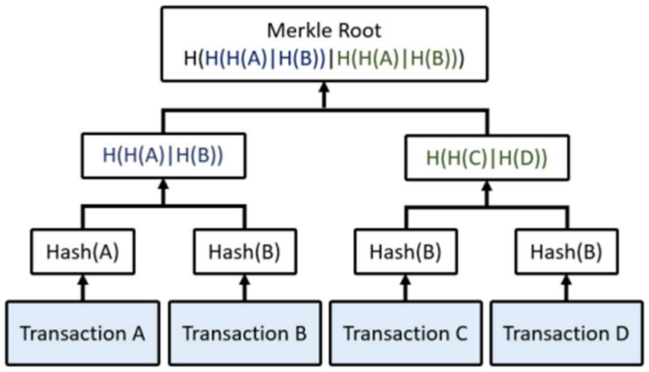


Figure 3. Merkle tree structure.

Merkle trees are also a fundamental part of blockchain technology. For a block, the Merkle root comes from a hashing transaction and pairing two transactions to hash and generate the upper level tree node. By doing so, it will get one hash to store that is deterministic based on the hashes of all the underlying transactions. This single hash is called the Merkle root.

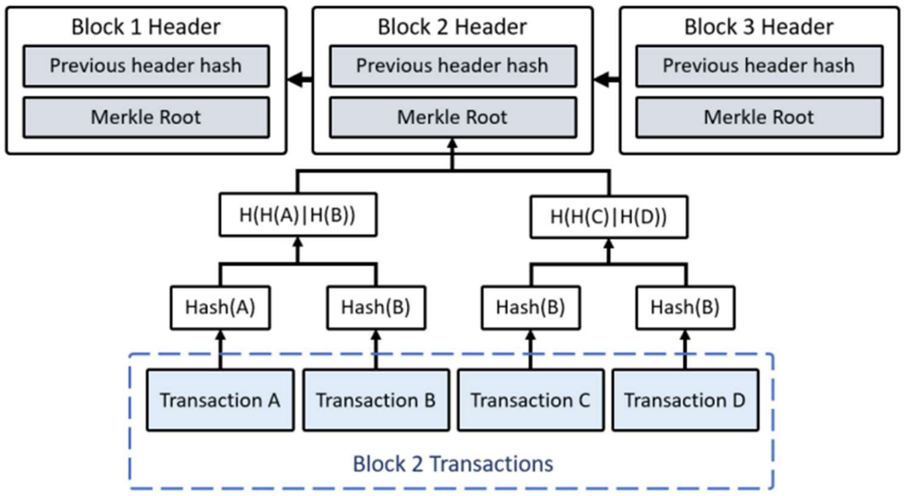


Figure 4. The architecture of Merkle tree in the blockchain.

In the blockchain, each block has a Merkle root stored in the block header. Merkle tree allows every node on the network to verify individual transaction without having to download and validate the entire block. If a copy of the block in the blockchain networks has the same Merkle root to another, then the transactions in that block are the same. Even a bit of incorrect data would lead to vastly different Merkle roots because of the properties of the hash. Therefore, it is not necessary to verify the amount of required information.

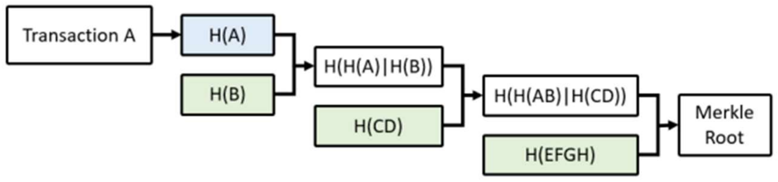


Figure 5. The verification of Merkle tree.

In a Merkle tree, transactions are grouped into pairs. The hash is computed for each pair and this is stored in the parent node. Now the parent nodes are grouped into pairs and their hash is stored one level up in the tree. This continues till the root of the tree. The different types of nodes in a Merkle tree are:
Root node: The root of merkle tree is known as merkle root and this merkle root is stored in the header of the block.
Leaf node: The leaf nodes contain the hash values of transaction data. Each transaction in the block has its data hashed and then this hash value (also known as transaction ID) is stored in leaf nodes.
Non-leaf node: The non-leaf nodes contain the hash value of their respective children. These are also called intermediate nodes because they contain the intermediate hash values and the hash process continues till the root of the tree.

The Merkle Tree ensures that the data is accurate. If any single transaction information or transaction sequence changes, the hash of this same transaction will reflect those changes. This update would propagate up the Merkle Tree until it reached the Merkle Root, altering the Merkle Root's value.

Algorithm used in MerkleTree :-

A Merkle tree adds up all of the operations in a block and creates an unique fingerprint of the complete set of functions, allowing users to check if the block contains a transaction.

- Merkle trees are created by continuously hashing sets of nodes so that only one hash exists, called as the Merkle Root or Root Hash.
- They are designed from the ground up, with Transaction IDs (hashes of individual transactions) as the foundation.
- That each non-leaf node hashes its preceding hash, while every leaf node hashes transactional data.

Algorithm : Merkle Tree Construction

```
$MerkleTree(T)\{$
1. $for\ each\ p\ in\ P_T\ d$
2.     $for\ each\ n\ in\ N_T\ do$
3.       $if(isChild(p,n))$
4.           $p\_material = Hash(p\_id, p\_publickey) + Hash(n\_id, n\_publickey)$
5.       $end\ if$
6.     $end\ for$
7. $end\ for$
 $\}$
$Main()\{$
1.   $for\ each\ T\ in\ L_T\ do$
2.      $MerkleTree(T)$
3.   $end\ for$
 $\}$
```

Consider the case below: Four transactions are done on the same block: H1, H2, H3, and H4. After that, each transaction is hashed, giving you:

- Hash 1 (H1)
- Hash 2 (H2)
- Hash 3 (H3)
- Hash 4 (H4)

Step 1: Each transaction's hash is calculated.

H1 stands for hash (T1)

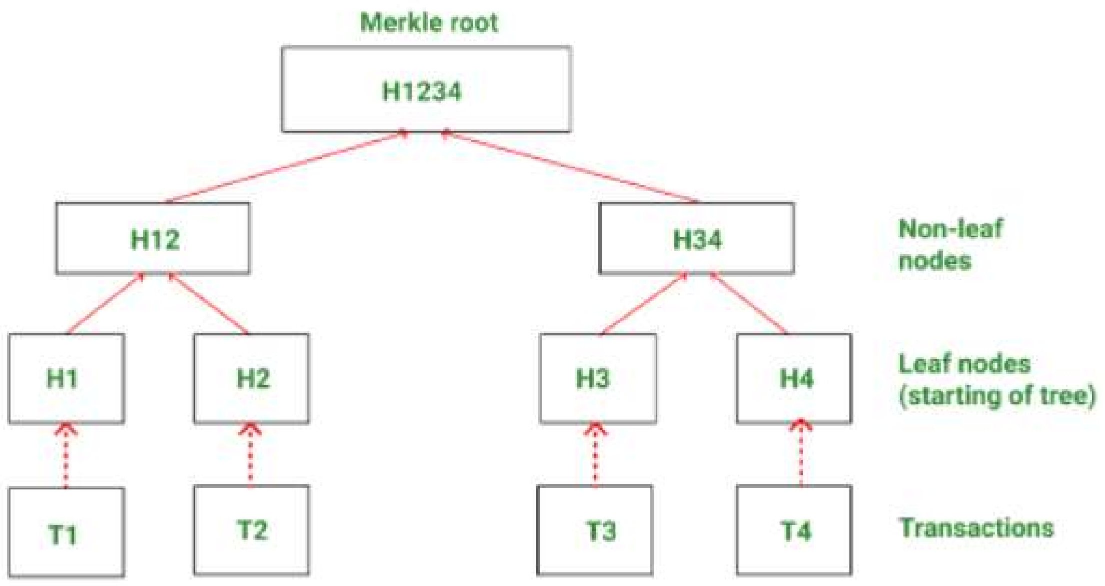
Step 2: Hashes are stored in the Merkle tree's leaf nodes.

Step 3: Non-leaf nodes will now be created. Leaf nodes will be put together from left to right to construct these nodes, and the hash of such pairs will be determined. To create H12, first compute the hash of H1 and H2. H34 is calculated in the same way. H12 and H34 are the parent nodes of H1, H2, H3, and H4, respectively. Non-leaf nodes are those that do not have any leaves.

Hash(H1 + H2) = H12

Hash(H3 + H4) = H34

Step 4: Finally, by combining H12 and H34, H1234 is calculated. H1234 seems to be hash remaining. This indicates that we have arrived at the root node and H1234 is the merkle root here.



Example of Merkle Tree

Implementation of Merkle Tree using Java :-

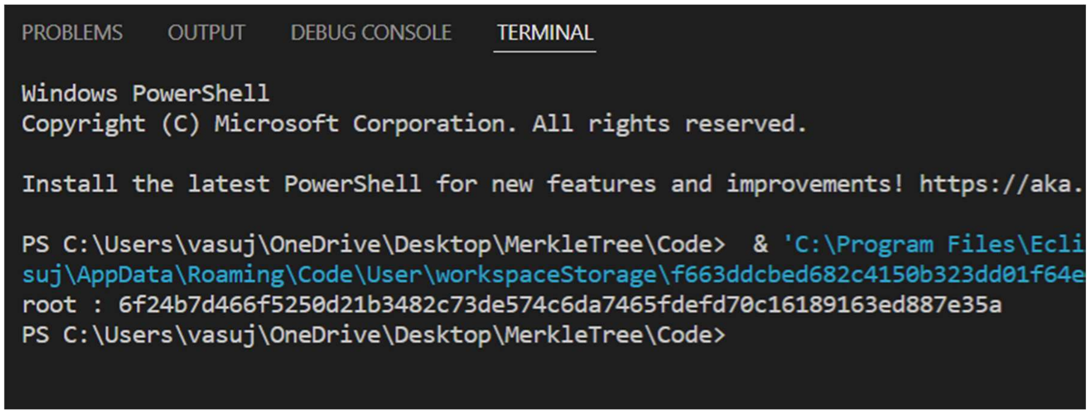
App.java :-

```
import java.util.ArrayList;
import java.util.List;

public class App {
    public static void main(String [] args) {
        List<String> tempTxList = new ArrayList<String>();
        tempTxList.add("v");
        tempTxList.add("a");
        tempTxList.add("s");
        tempTxList.add("u");
        tempTxList.add("j");

        MerkleTrees merkleTrees = new MerkleTrees(tempTxList);
        merkleTrees.merkle_tree();
        System.out.println("root : " + merkleTrees.getRoot());
    }
}
```

MerkleTress.java :-

```java
import java.security.MessageDigest;
import java.util.ArrayList;
import java.util.List;

public class MerkleTrees {
    List<String> txList;
    String root;


    public MerkleTrees(List<String> txList) {
        this.txList = txList;
        root = "";
    }

    public void merkle_tree() {

        List<String> tempTxList = new ArrayList<String>();

        for (int i = 0; i <this.txList.size(); i++) {
            tempTxList.add(this.txList.get(i));
        }

        List<String> newTxList = getNewTxList(tempTxList);

        while (newTxList.size() != 1) {
            newTxList = getNewTxList(newTxList);
        }

        this.root = newTxList.get(0);
    }

    private List<String> getNewTxList(List<String> tempTxList)
{

        List<String> newTxList = new ArrayList<String>();
        int index = 0;
        while (index <tempTxList.size()) {

            String left = tempTxList.get(index);
            index++;

            String right = "";
            if (index != tempTxList.size()) {
                right = tempTxList.get(index);
            }

            String sha2HexValue = getSHA2HexValue(left +
right);
            newTxList.add(sha2HexValue);
            index++;

        }

        return newTxList;
    }

    public String getSHA2HexValue(String str) {
        byte[] cipher_byte;
        try{
            MessageDigest md = MessageDigest.getInstance("SHA-
256");
            md.update(str.getBytes());
            cipher_byte = md.digest();
            StringBuilder sb = new StringBuilder(2 *
cipher_byte.length);
            for(byte b: cipher_byte) {
                sb.append(String.format("%02x", b&0xff) );
            }
            return sb.toString();
        } catch (Exception e) {
            e.printStackTrace();
        }

        return "";
    }

    public String getRoot() {
        return this.root;
    }
}
```

Output :-



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.

PS C:\Users\vasuj\OneDrive\Desktop\MerkleTree\Code>  & 'C:\Program Files\Ecli
suj\AppData\Roaming\Code\User\workspaceStorage\f663ddcbed682c4150b323dd01f64e
root : 6f24b7d466f5250d21b3482c73de574c6da7465fdefd70c16189163ed887e35a
PS C:\Users\vasuj\OneDrive\Desktop\MerkleTree\Code>
```

Merkle trees benefits:-

1. **Efficient verification-** Merkle trees offer efficient verification of integrity and validity of data and significantly reduce the amount of memory required for verification. The proof of verification does not require a huge amount of data to be transmitted across the blockchain network. Enable trustless transfer of cryptocurrency in the peer-to-peer, distributed system by the quick verification of transactions.
2. **No delay-** There is no delay in the transfer of data across the network. Merkle trees are extensively used in computations that maintain the functioning of cryptocurrencies.
3. **Less disk space-** Merkle trees occupy less disk space when compared to other data structures.
4. **Unaltered transfer of data-** Merkle root helps in making sure that the blocks sent across the network are whole and unaltered.
5. **Tampering Detection-** Merkle tree gives an amazing advantage to miners to check whether any transactions have been tampered with.
    a. Since the transactions are stored in a Merkle tree which stores the hash of each node in the upper parent node, any changes in the details of the transaction such as the amount to be debited or the address to whom the payment must be made, then the change will propagate to the hashes in upper levels and finally to the Merkle root.
    b. The miner can compare the Merkle root in the header with the Merkle root stored in the data part of a block and can easily detect this tampering.
6. **Time Complexity-** Merkle tree is the best solution if a comparison is done between the time complexity of searching a transaction in a block which as Merkle tree and another block that has transactions arranged in a linked list, then-

a. **Merkle Tree search-** O(logn)
b. **Linked List search-** O(n)
c. where n is the number of transactions in a block.

Reference:-

- https://en.wikipedia.org/wiki/Merkle_tree
- https://brilliant.org/wiki/merkle-tree/
- https://www.javatpoint.com/blockchain-tutorial
- https://www.geeksforgeeks.org/blockchain-merkle-trees