Sipna College of Engineering & Technology, Amravati.
Department of Computer Science & Engineering

Branch :- Computer Sci. & Engg. **Class :- Final Year**
Subject :-Block Chain Fundamentals Lab manual **Sem :- VII**
Teacher Manual

PRACTICAL NO 8

AIM: Create a smart contract for Merkle tree

S/W REQUIRED: Remix IDE

Merkle tree is a fundamental part of blockchain technology. It is a mathematical data structure composed of hashes of different blocks of data, and which serves as a summary of all the transactions in a block. It also allows for efficient and secure verification of content in a large body of data. It also helps to verify the consistency and content of the data. Both Bitcoin and Ethereum use Merkle Trees structure. Merkle Tree is also known as Hash Tree.

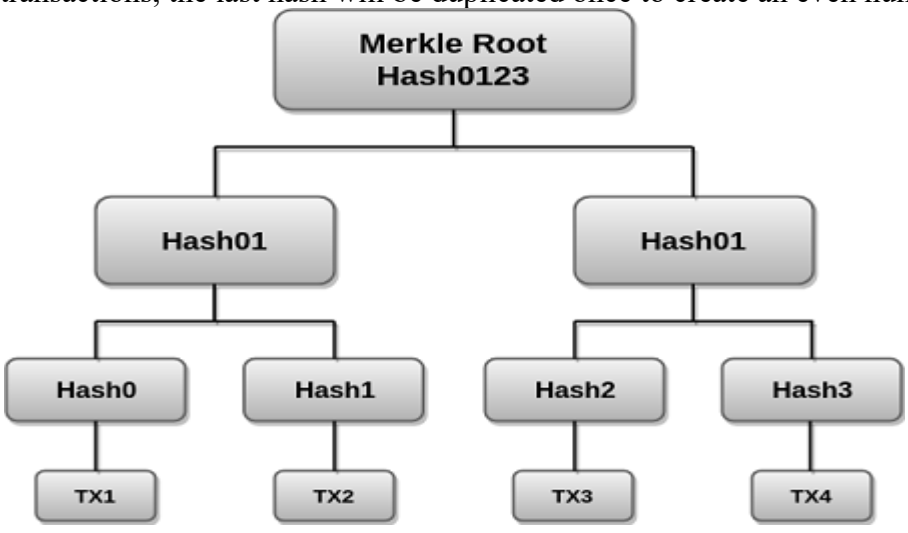
The concept of Merkle Tree is named after Ralph Merkle, who patented the idea in 1979. Fundamentally, it is a data structure tree in which every leaf node labelled with the hash of a data block, and the non-leaf node labelled with the cryptographic hash of the labels of its child nodes. The leaf nodes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

How does merkle tree works?

A Merkle tree stores all the transactions in a block by producing a digital fingerprint of the entire set of transactions. It allows the user to verify whether a transaction can be included in a block or not.

Merkle trees are created by repeatedly calculating hashing pairs of nodes until there is only one hash left. This hash is called the Merkle Root, or the Root Hash. The Merkle Trees are constructed in a bottom-up approach.

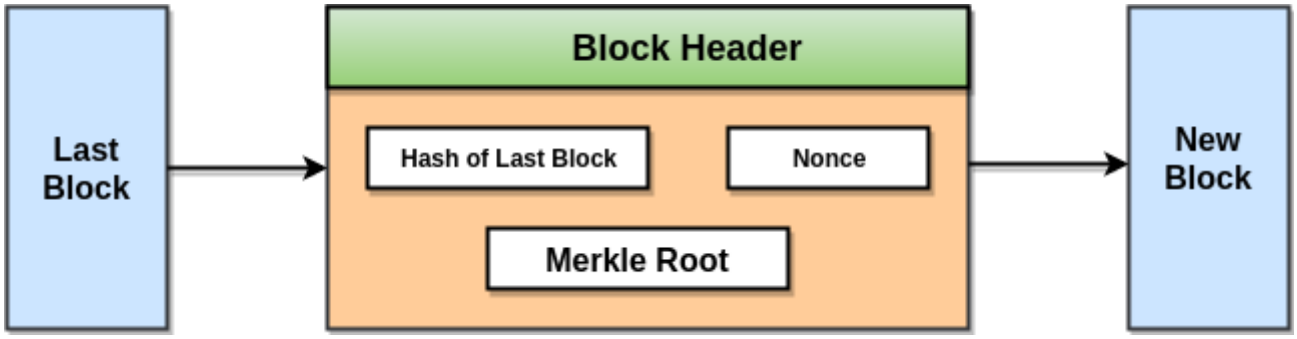
Every leaf node is a hash of transactional data, and the non-leaf node is a hash of its previous hashes. Merkle trees are in a binary tree, so it requires an even number of leaf nodes. If there is an odd number of transactions, the last hash will be duplicated once to create an even number of leaf nodes.



The above example is the most common and simple form of a Merkle tree, i.e., Binary Merkle Tree. There are four transactions in a block: TX1, TX2, TX3, and TX4. Here you can see, there is a top hash which is the

hash of the entire tree, known as the Root Hash, or the Merkle Root. Each of these is repeatedly hashed, and stored in each leaf node, resulting in Hash 0, 1, 2, and 3. Consecutive pairs of leaf nodes are then summarized in a parent node by hashing Hash0 and Hash1, resulting in Hash01, and separately hashing Hash2 and Hash3, resulting in Hash23. The two hashes (Hash01 and Hash23) are then hashed again to produce the Root Hash or the Merkle Root.

Merkle Root is stored in the block header. The block header is the part of the bitcoin block which gets hash in the process of mining. It contains the hash of the last block, a Nonce, and the Root Hash of all the transactions in the current block in a Merkle Tree. So having the Merkle root in block header makes the transaction tamper-proof. As this Root Hash includes the hashes of all the transactions within the block, these transactions may result in saving the disk space.



The Merkle Tree maintains the integrity of the data. If any single detail of transactions or order of the transaction's changes, then these changes reflected in the hash of that transaction. This change would cascade up the Merkle Tree to the Merkle Root, changing the value of the Merkle root and thus invalidating the block. So everyone can see that Merkle tree allows for a quick and simple test of whether a specific transaction is included in the set or not.

Merkle trees have three benefits:

- It provides a means to maintain the integrity and validity of data.
- It helps in saving the memory or disk space as the proofs, computationally easy and fast.
- Their proofs and management require tiny amounts of information to be transmitted across networks.

Implementation:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract MerkleProof {
    function verify(
        bytes32[] memory proof,
        bytes32 root,
        bytes32 leaf,
        uint index
    ) public pure returns (bool) {
        bytes32 hash = leaf;

        for (uint i = 0; i < proof.length; i++) {
            bytes32 proofElement = proof[i];
```

```
            if (index % 2 == 0) {
                hash = keccak256(abi.encodePacked(hash, proofElement));
            } else {
                hash = keccak256(abi.encodePacked(proofElement, hash));
            }

            index = index / 2;
        }

        return hash == root;
    }
}

contract TestMerkleProof is MerkleProof {
    bytes32[] public hashes;

    constructor() {
        string[4] memory transactions = [
            "alice -> bob",
            "bob -> dave",
            "carol -> alice",
            "dave -> bob"
        ];

        for (uint i = 0; i < transactions.length; i++) {
            hashes.push(keccak256(abi.encodePacked(transactions[i])));
        }

        uint n = transactions.length;
        uint offset = 0;

        while (n > 0) {
            for (uint i = 0; i < n - 1; i += 2) {
                hashes.push(
                    keccak256(
                        abi.encodePacked(hashes[offset + i], hashes[offset + i + 1])
                    )
                );
            }
            offset += n;
            n = n / 2;
        }
    }

    function getRoot() public view returns (bytes32) {
        return hashes[hashes.length - 1];
    }

    /* verify
    3rd leaf
    0xdca3326ad7e8121bf9cf9c12333e6b2271abe823ec9edfe42f813b1e768fa57b

    root
```

```
    0xcc086fcc038189b4641db2cc4f1de3bb132aefbd65d510d817591550937818c7

    index
    2

    proof
    0x8da9e1c820f9dbd1589fd6585872bc1063588625729e7ab0797cfc63a00bd950
    0x995788ffc103b987ad50f5e5707fd094419eb12d9552cc423bd0cd86a3861433
    */
}
```

Output: Byte code generated as well as API generated

API code

```
[
	{
		"inputs": [
			{
				"internalType": "bytes32[]",
				"name": "proof",
				"type": "bytes32[]"
			},
			{
				"internalType": "bytes32",
				"name": "root",
				"type": "bytes32"
			},
			{
				"internalType": "bytes32",
				"name": "leaf",
				"type": "bytes32"
			},
			{
				"internalType": "uint256",
				"name": "index",
				"type": "uint256"
			}
		],
		"name": "verify",
		"outputs": [
			{
				"internalType": "bool",
				"name": "",
				"type": "bool"
			}
		],
		"stateMutability": "pure",
		"type": "function"
	}
]
```

CONCLUSION: Thus we have created a smart contract for Merkle tree