Experiment No.3
To implement the concept of Merkle root
Date of Performance: 24/08/2023
Date of Submission: 24/08/2023

AIM: To implement the concept of Merkle root

Objective: To develop a program to create a cryptographic hash using the concept of merkle tree

Theory:

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.

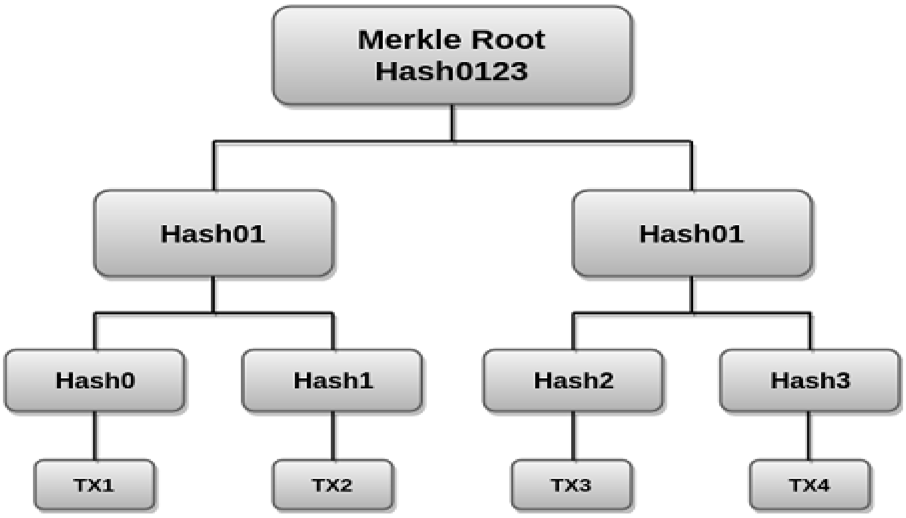

Fig.3.1 Merkle Root Tree Structure

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.

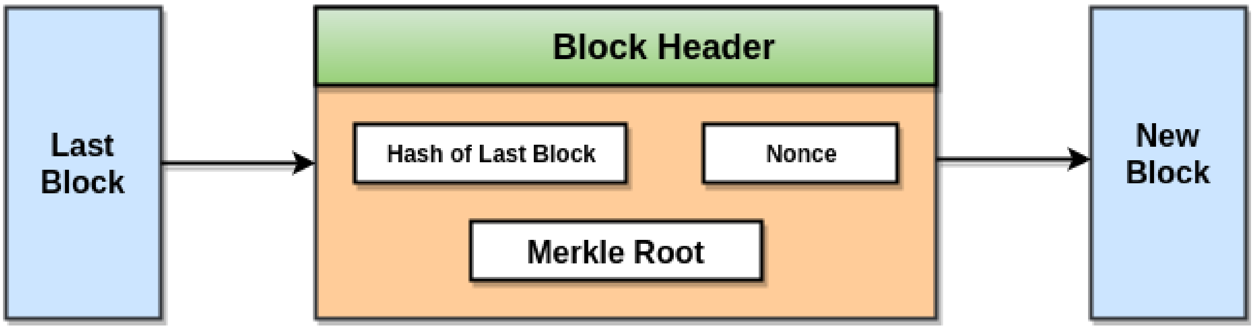

Fig.3.2 Merkle Root in Block

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.

Process:

Step 1. The transaction represents the original data blocks which are hashed to produce transaction hashes (transaction id) which form the leaf nodes.

Step 2. The leaf nodes have to be even in number for a binary hash tree to work so if the number of leaf nodes is an odd number, then the last leaf node is duplicated to even the count.

Step 3. Each pair of leaf nodes is concatenated and hashed to form the second row of hashes.

Step 4. The process is repeated until a row is obtained with only two hashes

Step 5. These last two hashes are concatenated to form the Merkle root.

Code:

App.java

```
package exp3;

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("a");

      tempTxList.add("b");

      tempTxList.add("c");

      tempTxList.add("d");

      tempTxList.add("e");



      MerkleTree merkleTrees = new MerkleTree(tempTxList);

      merkleTrees.merkle_tree();

      System.out.println("root : " + merkleTrees.getRoot());

   }

}
```

MerkleTree.java

```
package wxp3;

import java.security.MessageDigest;

import java.util.ArrayList;

import java.util.List;

public class MerkleTree {

  //transaction List

   List<String> txList;

  //Merkle Root

   String root;

   public MerkleTree(List<String> txList) {

       this.txList = txList;

       root = "";

   System.out.println("Transaction List"+this.txList);

   }

   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);

    //Execute the loop until only one hash value is left
    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()) {
      //left
        String left = tempTxList.get(index);
        System.out.print("Left--> \t"+left+ "\t");
```

```
            index++;

            //right
            String right = "";
            if (index != tempTxList.size()) {
                right = tempTxList.get(index);
                System.out.print("Right--> \t"+right+"\t");
                System.out.println("");
            }

            //sha2 hex value
            String sha2HexValue = getSHA2HexValue(left + right);
            System.out.println("sha2HexValue \t"+sha2HexValue);
            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) );
    //      System.out.println("appended SHA-256 hash is"+sb.toString());
            }
            return sb.toString();
        } catch (Exception e) {
            e.printStackTrace();
        }

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

Output:

```
Transaction List[a, b, c, d, e]
Left-->         a       Right-->        b
sha2HexValue    fb8e20fc2e4c3f248c60c39bd652f3c1347298bb977b8b4d5903b85055620603
Left-->         c       Right-->        d
sha2HexValue    21e721c35a5823fdb452fa2f9f0a612c74fb952e06927489c6b27a43b817bed4
Left-->         e       sha2HexValue    3f79bb7b435b05321651daefd374cdc681dc06faa65e374e38337b88ca046dea
Left-->         fb8e20fc2e4c3f248c60c39bd652f3c1347298bb977b8b4d5903b85055620603        Right-->        21e721c35a5823fdb452fa2f9f0a612c74fb952e06927489c6b27a43b817bed4
sha2HexValue    12a40550c10c6339bf6f271445270e49b844d6c9e8abc36b9b642be532befe94
Left-->         3f79bb7b435b05321651daefd374cdc681dc06faa65e374e38337b88ca046dea        sha2HexValue    ef5960718ca91ca07e63f1d1cf5320ad3c8f923d481c1f8c873aa987e1d6e1f6
Left-->         12a40550c10c6339bf6f271445270e49b844d6c9e8abc36b9b642be532befe94        Right-->        ef5960718ca91ca07e63f1d1cf5320ad3c8f923d481c1f8c873aa987e1d6e1f6
sha2HexValue    3b7e1e6ba3b82975d7802511d8c7fabbe7a5d112d0dd112fbcfbb7e6417a3214
root : 3b7e1e6ba3b82975d7802511d8c7fabbe7a5d112d0dd112fbcfbb7e6417a3214
BUILD SUCCESSFUL (total time: 0 seconds)
```

Conclusion: The Merkle root is a linchpin for transaction integrity within a block. By hashing all transactions in a tree structure, any modification ripples up, changing the root, thus detecting tampering. This succinct hash enables swift verification, vital for scalability. Its role in the block header streamlines proof-of-work mining and bolsters security against hash collisions. Additionally, it empowers proof of transaction inclusion, upholding blockchain data integrity and enabling trustless validation. The Merkle root safeguards the reliability and efficiency of blockchain transactions, underpinning the entire network's integrity.