**Aim : -**

To implement a Merkle Tree in Java that generates a root hash by recursively hashing transaction data, ensuring secure and efficient verification of data integrity.

**Procedure : -**

A **Merkle Tree** (also known as a **Hash Tree**) is a binary tree data structure in which:

* **Leaf nodes** contain the hash of individual data elements (e.g., transactions).
* **Non-leaf (internal) nodes** contain the hash of the concatenation of their child node hashes.

In this implementation:

* The input is a list of transactions (strings).
* Each transaction is hashed using **SHA-256** to create the leaf nodes.
* Pairs of hashes are concatenated and then hashed again to form parent nodes.
* This process continues recursively until a single root hash (Merkle Root) is produced.

**Working Steps: -**

1. Accept a list of transactions as input.
2. Compute the SHA-256 hash of each transaction.
3. Group adjacent hashes in pairs.
4. Concatenate each pair and compute the SHA-256 hash again.
5. Repeat the process level by level until only one hash remains – the **Merkle Root**.
6. Store the entire Merkle Tree structure in a list for reference.

**Source Code : -**

import java.nio.charset.StandardCharsets;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

public class MerkleTree

{

private List<String> transactions;

private List<String> merkleTree;

public MerkleTree(List<String> transactions)

{

this.transactions = buildInitialHashes(transactions);

this.merkleTree = buildMerkleTree(this.transactions);

}

private List<String> buildInitialHashes(List<String> rawTransactions)

{

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

for (String tx : rawTransactions)

{

hashedTransactions.add(calculateHash(tx));

}

return hashedTransactions;

}

private String calculateHash(String data)

{

try

{

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] hashBytes = digest.digest(data.getBytes(StandardCharsets.UTF\_8));

StringBuilder hexString = new StringBuilder();

for (byte hashByte : hashBytes)

{

String hex = Integer.toHexString(0xff & hashByte);

if (hex.length() == 1) hexString.append('0');

hexString.append(hex);

}

return hexString.toString();

}

catch (NoSuchAlgorithmException e)

{

e.printStackTrace();

}

return null;

}

private List<String> buildMerkleTree(List<String> hashedLeaves)

{

List<String> merkleTree = new ArrayList<>(hashedLeaves);

int levelOffset = 0;

for (int levelSize = hashedLeaves.size(); levelSize > 1; levelSize = (levelSize + 1) / 2)

{

for (int left = 0; left < levelSize; left += 2)

{

int right = Math.min(left + 1, levelSize - 1);

String leftHash = merkleTree.get(levelOffset + left);

String rightHash = merkleTree.get(levelOffset + right);

String parentHash = calculateHash(leftHash + rightHash);

merkleTree.add(parentHash);

}

levelOffset += levelSize;

}

return merkleTree;

}

public List<String> getMerkleTree()

{

return merkleTree;

}

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

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

System.out.print("Enter the number of transactions: ");

int n = scanner.nextInt();

scanner.nextLine();

for (int i = 1; i <= n; i++)

{

System.out.print("Enter transaction " + i + ": ");

transactions.add(scanner.nextLine());

}

MerkleTree tree = new MerkleTree(transactions);

System.out.println("\nMerkle Tree Hashes:");

for (String hash : tree.getMerkleTree())

{

System.out.println(hash);

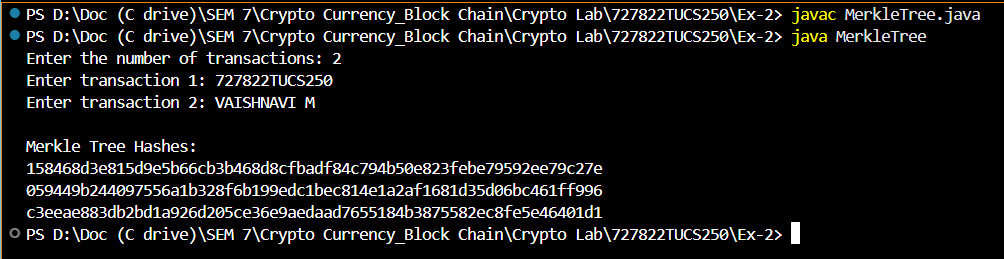
}

scanner.close();

}

}

**Output : -**



**AIM:**  
To create a block structure that securely stores transaction data in a blockchain system. Each block holds a set of recent transactions that are yet to be validated. Once validated, the block is closed and linked to the previous block, forming a chain. New blocks are generated when validators or miners successfully verify the encrypted information in the block header.

**Procedure : -**

A **Block** is a fundamental component of a blockchain. It is a data structure that securely stores information such as transactions, timestamps, and a reference to the previous block. Each block is uniquely identified using a cryptographic hash.

In this implementation:

* Each block has an **index**, **timestamp**, **data**, **previous hash**, and a **nonce** (used in mining).
* The **SHA-256** hashing algorithm is used to calculate the block’s hash based on its contents.
* A **mining** process is included to simulate Proof-of-Work, where the hash must meet a specified difficulty (number of leading zeros).

**Working Steps : -**

1. Initialize the block with index, previous hash, and data.
2. Generate a timestamp and set nonce to 0.
3. Calculate the initial hash using SHA-256.
4. Mine the block by adjusting the nonce until the hash starts with the required number of zeros (difficulty).
5. Print the mined block details: index, timestamp, previous hash, current hash, and data.

**Source Code : -**

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.util.Date;

import java.util.Scanner;

public class Block

{

private int index;

private long timestamp;

private String previousHash;

private String hash;

private String data;

private int nonce;

public Block(int index, String previousHash, String data)

{

this.index = index;

this.timestamp = new Date().getTime();

this.previousHash = previousHash;

this.data = data;

this.nonce = 0;

this.hash = calculateHash();

}

public String calculateHash()

{

try

{

MessageDigest digest = MessageDigest.getInstance("SHA-256");

String input = index + timestamp + previousHash + data + nonce;

byte[] hashBytes = digest.digest(input.getBytes());

StringBuilder hexString = new StringBuilder();

for (byte hashByte : hashBytes)

{

String hex = Integer.toHexString(0xff & hashByte);

if (hex.length() == 1) hexString.append('0');

hexString.append(hex);

}

return hexString.toString();

}

catch (NoSuchAlgorithmException e)

{

e.printStackTrace();

}

return null;

}

public void mineBlock(int difficulty)

{

String target = new String(new char[difficulty]).replace('\0', '0');

while (!hash.substring(0, difficulty).equals(target))

{

nonce++;

hash = calculateHash();

}

System.out.println("Block mined: " + hash);

}

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

System.out.print("Enter Block Index: ");

int index = scanner.nextInt();

scanner.nextLine();

System.out.print("Enter Previous Hash: ");

String prevHash = scanner.nextLine();

System.out.print("Enter Data: ");

String data = scanner.nextLine();

Block block = new Block(index, prevHash, data);

System.out.println("\nCalculating Hash...");

System.out.println("Initial Hash: " + block.calculateHash());

System.out.println("Mining Block...");

block.mineBlock(1);

System.out.println("\nBlock Details:");

System.out.println("Block Index: " + block.index);

System.out.println("Timestamp: " + block.timestamp);

System.out.println("Previous Hash: " + block.previousHash);

System.out.println("Current Hash: " + block.hash);

System.out.println("Data: " + block.data);

scanner.close();

}

}

**Output : -**

****

**Result : -**

Thus, the implementation of the **Merkle Tree** and **Block** has been successfully completed.