**Course Code: CSE1007**

**Java Programming**

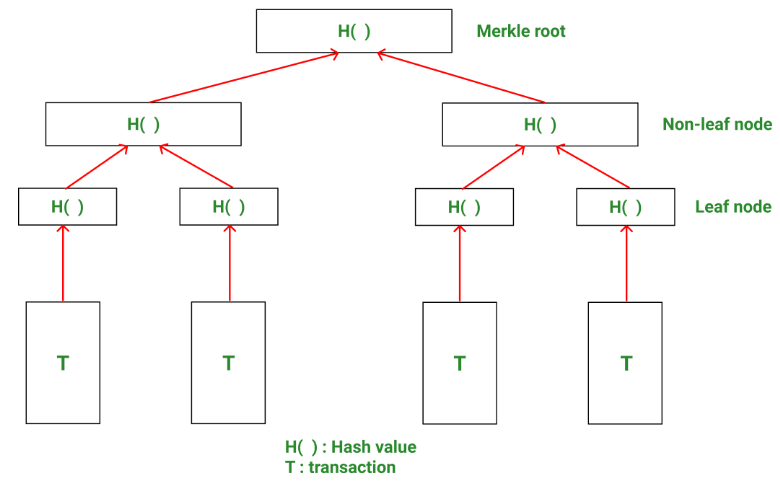
**Topic : Merkle Tree**

**Introduction to Merkle Tree**

Merkle trees are a critical component of blockchain technology. It's a mathematical data structure made up of hashes of various data blocks that provides as an overview of almost all of the operations in a block. It also enables quick and secure content verification across big datasets. It also aids in the verification of the data's accuracy and quality. Merkle Trees are used in both Bitcoin and Ethereum. Hash Tree and Merkle Tree are two names for the same plant.

Ralph Merkle, who patented the idea in 1979, is the creator of the Merkle Tree concept. It's simply a data structure tree with each and every leaf node marked with the hash of a data block and every non-leaf node marked with the cryptographic hash of its child nodes' labels.  Leaf nodes are also the tree's lowest nodes.

**Architecture of Merkle Tree**



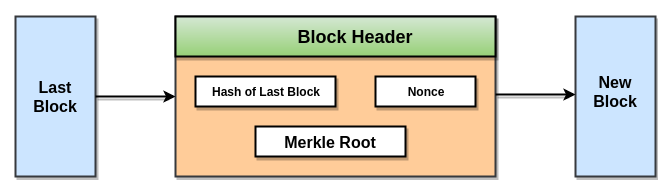
***Architecture diagram of Merkle Tree***

* A blockchain might include thousands of blocks, each containing thousands of transactions. As a result, memory space and computational power are two of the most significant obstacles.
* It would be ideal to use as few available data for validating transactions, since this would save CPU processing and improve security, and Merkle trees give just that.
* Transactions are arranged into pairs in a Merkle tree. For each pair, a hash is generated and saved on the parent node. The parent nodes have now been linked into pairs, and their hashes have been put one level higher in the tree. This process continues until the tree's root is reached. In a Merkle tree, there are several distinct sorts of nodes:

1. Root node: The merkle tree's root is termed as the merkle root, and it is saved in the block's header.
2. Leaf Node: Hash values of transactional data which are stored in the leaf nodes. Each block's transactions are hashed, and the hash value is subsequently saved on leaf nodes.
3. Non-Leaf Node: The hash value of the relevant offspring is stored in the non-leaf nodes. Because they contain intermediary hash values and the hash process persists until the tree's root, these are also known as intermediate nodes.

* Bitcoin hashes transaction data using the SHA-256 hash algorithm until the Merkle root is found.
* In nature, a Merkle tree is binary. This indicates that the Merkle tree must have an even number of leaf nodes to be created correctly.

The block header contains Merkle Root. Block header is the section of the bitcoin block that is hashed throughout the mining process. In a Merkle Tree, it holds the hash of the previous block, a Nonce, and Root Hash of most of the transactions in the current block. As a result, putting the Merkle root in the block header secures the transaction. Because this Root Hash contains the hashes of all operations inside the block, these operations can result in disc space savings.



***Working of Block Header***

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 Merkle Tree**

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.

Take a look at a Merkle Tree in Blockchain as an example to better grasp the idea.

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)**

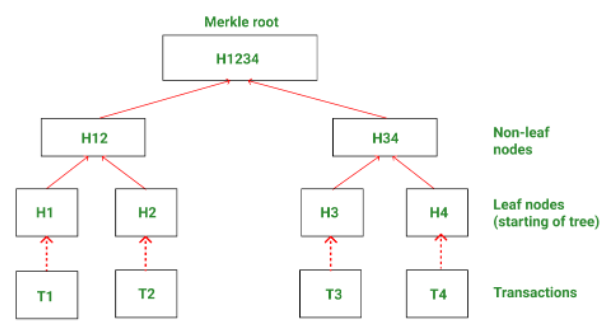
**Step 2**: Hashes are stored in the Merkle tree's leaf nodes.

**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***

A Merkle Tree is far more intricate in reality.  Nonetheless, this example provides an excellent understanding of how algorithms function and why they have been so powerful.

**Implementation of Merkle Tree using Java**

The binary merkle tree will be constructed in this sample implementation. Let's start with defining the node. It features a data field to hold the hash and left and right pointers to link to the binary tree's left and right children, much like a conventional tree.

**Node.java**

public class Node {

private Node left;

private Node right;

private String hash;

public Node(Node left, Node right, String hash) {

this.left = left;

this.right = right;

this.hash = hash;

}

public Node gettingLeft() {

return left;

}

public void settingLeft(Node left) {

this.left = left;

}

public Node gettingRight() {

return right;

}

public void settingRight(Node right) {

this.right = right;

}

public String gettingHash() {

return hash;

}

public void settingHash(String hash) {

this.hash = hash;

}

}

For the java representations of cryptographic algorithms, we'll need the prerequisite mentioned below. However, for our implementation, we will use keccak 256.

[**BouncyCastle.xml**](https://gist.github.com/pranaybathini/5001c7a23e5100b26ae78a333fcb263d#file-bouncycastle-xml)

<--The Bouncy Castle Crypto package is a Java implementation of cryptographic algorithms -->

<dependency>

<groupId>org.bouncycastle</groupId>

<artifactId>bcprov-jdk15on</artifactId>

<version>1.68</version>

</dependency>

After that, we have to specify the **Hash Algorithm** file which we have used for generating the Hash values for all the nodes

**HashAlgorithm.java**

import org.bouncycastle.jcajce.provider.digest.Keccak;

import org.bouncycastle.util.encoders.Hex;

import java.nio.charset.StandardCharsets;

public class HashAlgorithm {

public static String generateHash(String originalString) {

Keccak.Digest256 digest256 = new Keccak.Digest256();

byte[] hashedByteArray = digest256.digest(

originalString.getBytes(StandardCharsets.UTF\_8));

return new String(Hex.encode(hashedByteArray));

}

}

A full binary tree will be created. We shall examine the odd node twice when generating the hash of its parent in the event of odd nodes.

**MerkleTree.java**

package com.merkle.tree.implementation;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

public class MerkleTree {

public static Node Generation\_of\_Tree(ArrayList<String> DataBlocks) {

ArrayList<Node> childNode = new ArrayList<>();

for (String message : DataBlocks) {

childNode.add(new Node(null, null, HashAlgorithm.generateHash(message)));

}

return Building\_of\_Tree(childNode);

}

private static Node Building\_of\_Tree(ArrayList<Node> children) {

ArrayList<Node> parents= new ArrayList<>();

while (children.size()!=1)

{

int index=0,length=children.size();

while (index<length)

{

Node leftChild = children.get(index);

Node rightChild = null;

if ((index + 1)<length)

{

rightChild = children.get(index + 1);

} else

{

rightChild = new Node(null, null, leftChild.getHash());

}

String parentHash= HashAlgorithm.generateHash(leftChild.gettingHash()+rightChild.gettingHash());

parents.add(new Node(leftChild,rightChild, parentHash));

index+=2;

}

children=parents;

parents=new ArrayList<>();

}

return children.get(0);

}

private static void levelorderoftraversal(Node root)

{

if (root == null)

{

return;

}

if ((root.gettingLeft()==null && root.gettingRight()==null))

{

System.out.println(root.gettingHash());

}

Queue<Node> queue=new LinkedList<>();

queue.add(root);

queue.add(null);

while(!queue.isEmpty())

{

Node node = queue.poll();

if (node != null)

{

System.out.println(node.gettingHash());

}

else

{

System.out.println();

if (!queue.isEmpty())

{

queue.add(null);

}

}

if (node != null && node.gettingLeft() != null)

{

queue.add(node.gettingLeft());

}

if (node != null && node.gettingRight() != null)

{

queue.add(node.gettingRight());

}

}

}

public static void main(String[] args)

{

ArrayList<String> DataBlocks = new ArrayList<>();

DataBlocks.add("Hardeep");

DataBlocks.add("Robin");

DataBlocks.add("Kumar");

DataBlocks.add("Simran");

Node root =Generation\_of\_Tree(DataBlocks);

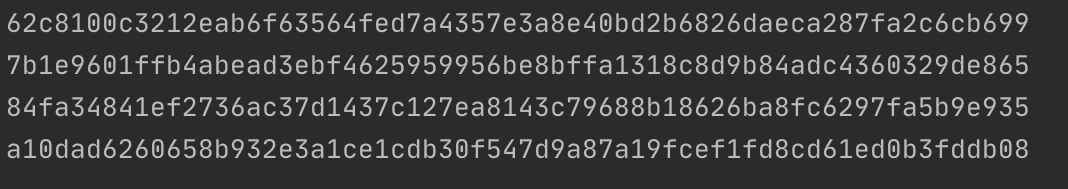
levelorderoftraversal(root);

}

}

We'll conduct traversal after creating the output to print the hashes in level order.

The output of the above shown implementation will be like:



**Benefits of Merkle Tree in Blockchain**

Merkle trees offer some of the benefits are as follows:

* Validate data integrity: It may be used to properly validate data integrity.
* The Merkle tree takes up extremely minimal disc space as compared to other data structures.
* Merkle trees may be split down into little chunks of information for verification, allowing for the transfer of minuscule amounts of data over networks.
* Verification that is both efficient and effective: The data format is fast, and it just takes a few seconds to validate the data's integrity.

**Reference:**

* <https://www.javatpoint.com/blockchain>
* <https://www.simplilearn.com/tutorials/blockchain-tutorial>
* <https://www.geeksforgeeks.org/blockchain>