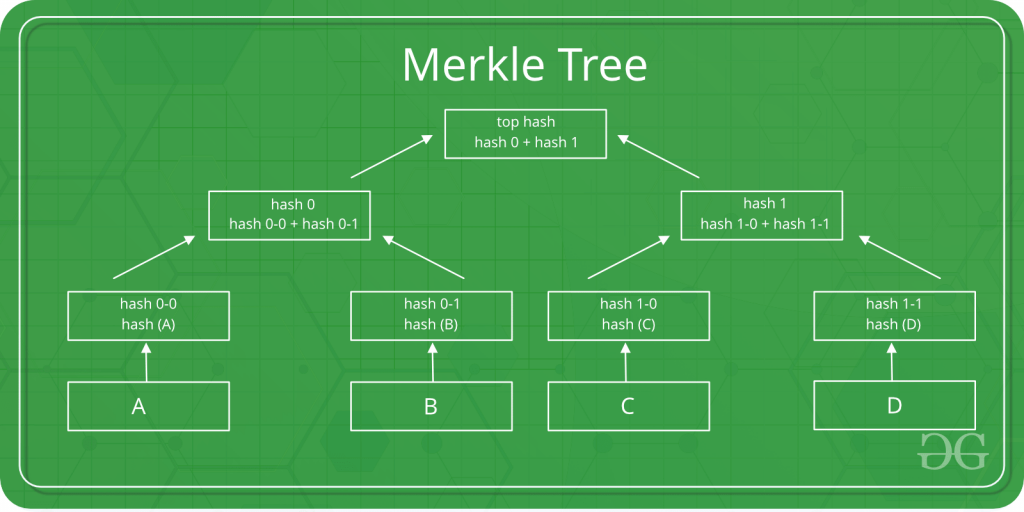
**CSE1007 – JAVA PROGRAMMING**

**THEORY ASSIGNMENT 1**

**MERKLE TREE**

**INTRODUCTION:**

* 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.
* Hash trees can be used to verify any kind of data stored, handled and transferred in and between computers. Currently the main use of hash trees is to make sure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks. Suggestions have been made to use hash trees in trusted computing systems.

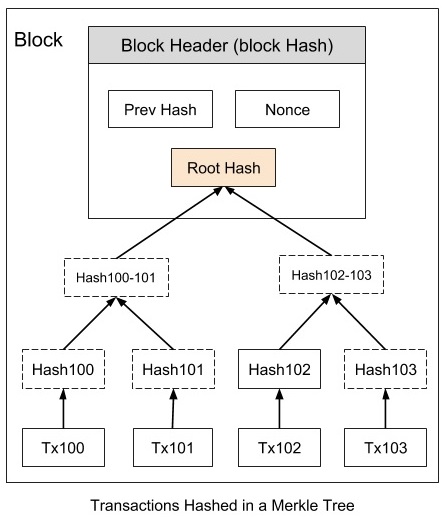


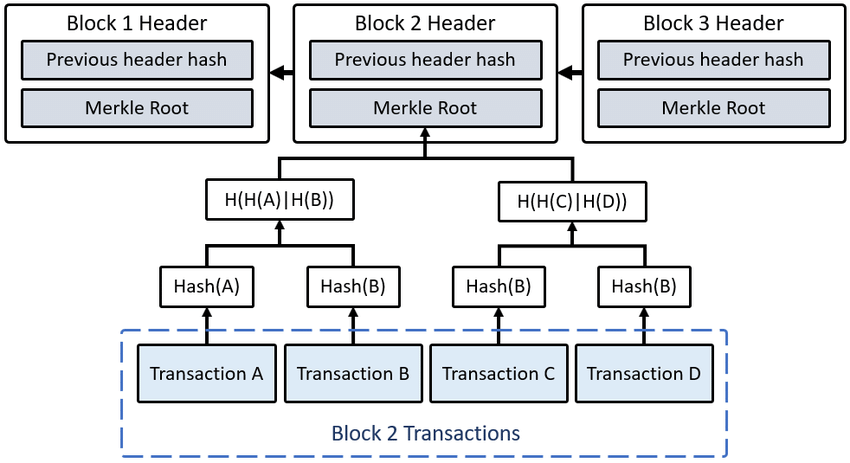
**Why is Merkle tree better than binary tree?**

* It provides both integrity and validity of data.
* The memory needed to store the data is significantly reduced.
* It only needs small amounts of information to be transmitted across networks.
* It provides a way of verifying transactions in a block without downloading an entire block.

**Why Merkle tree is used in blockchain?**

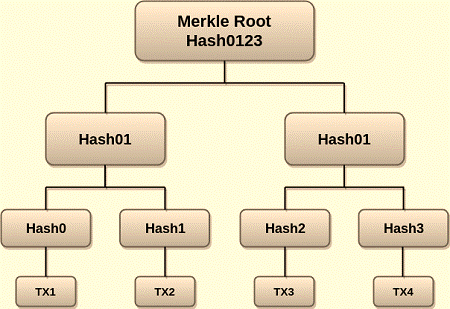
* A blockchain is comprised of various blocks that are linked with one another, hence the name blockchain. Merkle tree stores the blockchain data in an efficient and secure way.
* It can verify blockchain data very quickly. It also provides quick movement of large amounts of data from one computer node to the other blockchain network.
* Every transaction that occurs on the blockchain network has a hash associated with it. These hashes are stored in the form of a tree-like structure such that each hash is linked to its parent.
* Since numerous transactions are stored on a particular block, all the transaction hashes that are present in the block are also hashed, which results in a Merkle root.



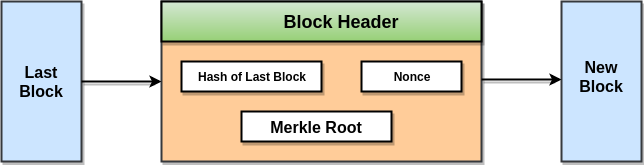


**ARCHITECTURE OF MERKLE TREE:**

* 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.
* A **hash function** maps any type of arbitrary data of any length to a fixed-size output. It is commonly used in cryptography since it is a cryptographic function. They are efficient and are well-known for one property: they are irreversible. It's a one-way function that's only meant to work in one direction.
* 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.

**ALGORITHM OF MERKLE TREE:**

1. **Structure of the node of binary Merkle tree**
2. It contains four variables:

* It contains a key variable.
* It contains value variable.
* It contains two links.

1. **Find operation in Merkle tree**

This function is used to check whether the given key is present in the merkle tree or not. If it is present, then it will return that node else it will return null.

1. Algorithm of find function in Merkle tree**:**

* We will take tree and key as parameters.
* If the tree is null, then we will return null.
* If the tree->key is equal to the key, we will return the tree.
* If the key is smaller than tree->key then we will return find(tree->left, key)
* else return find(tree->right, key)

1. **Add operation in Merkle tree**

This function is used to add a node into the merkle tree. This function is used to insert source code into a git repository when we commit a change to a repository.

1. Algorithm to add a node in Merkle tree:

* We will take key and value as parameters and take the hash(key) and store it in a variable called index.
* Store (struct node\*) arr[index]. Head in a pointer called tree of datatype node.
* Create new node with its key as key and value as value and both links as null.
* If tree is null, then assign the new node to head and increment the size by 1.
* If the tree is not null, then we will check if the key is already present in the tree using the find function.
* If the key is already present in the tree, then we will update the value.
* If it is not present in the tree, then we will use the insert function to insert the element.

1. Algorithm of insert function:

* It will take tree and item pointers of node data type as parameters.
* If item->key is smaller than tree->key and tree->left is null, then assign the item to tree->left.
* If item->key is smaller than tree->key and tree->left is not null, then call insert function with tree->left and item as parameters.
* If item->key is greater than tree->key and tree->right is null, then assign the item to tree->right.
* If item->key is greater than tree->key and tree->right is not null, then call insert function with tree->right and item as parameters.

1. **Delete a node from merkle tree**

This function is used to delete a node from Merkle tree. If the key given is present in the merkle tree, then it will delete the node from the tree. Git remembers all the files you have staged and stores them in a tree structure inside the commit. The nodes of this tree represent your files and directories. This function is used to delete these nodes.

1. Algorithm to delete a node in Merkle tree:

* We will take a key as a parameter.
* Take the hash(key) and store it in a variable called index.
* store (struct node\*) arr[index] and head in a pointer called tree of datatype node.
* If the tree is null, then the key is not present.
* If the tree is not null, then we will check if the key is already present in the tree using the find function.
* If the find function returns null, then the key is not present in the tree.
* If it is not null, then we will use the remove function to delete the element.

1. Algorithm of remove function:

* It will take tree and key as parameters.
* If the tree is null, then return null.
* If the key is smaller than the tree->key then tree->left is equal to remove(tree->left, key) and return tree.
* If the key is greater than the tree->key then tree->right is equal to remove(tree->right, key) and return tree.
* else if the tree->left is equal to null and the tree->right is equal to null then decrement the size and return tree->left.
* else if the tree->left is not equal to null and the tree->right is equal to null then decrement the size and return tree->left.
* else if tree->left is equal to null and tree->right is not equal to null then decrement the size and return tree->right.
* else assign tree->left to a pointer called left of data type node.
* While left->right is not equal to null, left is equal to left->right.
* tree->key is equal to left->key.
* tree->value is equal to left->value.
* tree->left is equal to remove(tree->left, tree->key).
* Return tree.

**CODE IN JAVA:**

**REFERENCES:**

<https://www.geeksforgeeks.org/introduction-to-merkle-tree/>

<https://www.simplilearn.com/tutorials/blockchain-tutorial/merkle-tree-in-blockchain>

<https://www.investopedia.com/terms/m/merkle-tree.asp>

<https://brilliant.org/wiki/merkle-tree/#:~:text=A%20Merkle%20tree%20is%20a,has%20up%20to%202%20children>.

<https://iq.opengenus.org/merkle-tree/>

<https://brilliant.org/wiki/merkle-tree/>

<https://www.javatpoint.com/blockchain-merkle-tree>