Assignment No: 2
Date: 18/03/2023

Title: Blockchain Hashing using Merkle Tree

Assignment Type of Submission:			
Group	Yes	**List all group members' details:**	**% Contribution Assignment Workload**
		Vivek Murarka 22200673	20%
		Purvish Shah 22200112	20%
		Ravi Raj Pedada 22200547	20%
		Meghana KR 22200568	20%
		Nikhitha Grace Josh 22200726	20%

1. Problem Domain Description:

Blockchain

The concept of a blockchain is based on a distributed database or ledger that is shared among the nodes of a computer network, which means that no central authority controls the information recorded on the blockchain. When a new block is added to the chain, it contains a hash of the previous block, along with other information such as a timestamp and a Merkle root. This hash ensures the integrity of the chain, as any attempt to tamper with a block will cause the hash to change, making it obvious that the chain has been compromised.

Because of this immutable nature, a blockchain can serve as a secure and transparent ledger for recording transactions, with each block containing a set of validated transactions. In Bitcoin, for example, each block has a size limit of 1MB and can contain around 4000 transactions, depending on their size. Miners play a critical role in maintaining the security of the blockchain by verifying transactions and adding new blocks to the chain through a process called proof of work. This process involves solving complex mathematical puzzles, which not only validates transactions but also ensures the authenticity and integrity of the blockchain.

Problem Statement

The time and resources required to verify the integrity of the data stored on the blockchain increase as the size of the blockchain grows. This is because the blockchain is a continuously growing ledger that records every transaction made on it. As more transactions are added to the blockchain, the amount of data that needs to be verified also increases, which can make the process slower and more resource-intensive.

This is where Merkle trees come in. By using a tree structure to represent the data stored on the blockchain, Merkle trees allow for a more efficient verification process. Instead of having to verify every single transaction in the blockchain, Merkle trees allow for a subset of the data to be verified. Since the hash values of the transactions are organized in a tree structure, the integrity of the entire block can be confirmed by verifying only the hash values at certain points in the tree.

The first use of Merkle trees in digital currencies can be traced back to the original Bitcoin whitepaper, published in 2008 by the pseudonymous creator Satoshi Nakamoto. The whitepaper describes how a Merkle tree is used in the Bitcoin blockchain to verify the transactions in a block, with the root of the tree being included in the block header.

Merkle trees improve the efficiency and scalability of the blockchain while maintaining the security and integrity of the data stored on it. It also ensures the integrity and authenticity of transactions in a distributed system without relying on a central authority.

The challenge in using the Merkle tree in blockchain is to ensure that the Merkle root hash is consistent across all nodes in the network, so that all nodes can independently verify the transactions in a block. The process of constructing the Merkle tree must also be efficient and scalable, as the number of transactions in a block can be very large. Furthermore, any attempt to modify a transaction in the block will result in an invalid Merkle proof, thereby preventing malicious actors from tampering with the transactions in the block.

Principles/Properties:

In this section we discuss certain terms important to using Merkle Trees in blockchain.

1. Hash Functions

   Hashing is a fundamental component of blockchain technology and is used to ensure the integrity and security of data stored on the blockchain.

A one-way hash function is a mathematical algorithm that takes input data and generates a fixed-size output, known as a hash.

The function is designed in such a way that it is easy to compute the hash from the input data, but computationally infeasible to generate the original input data from the hash. Once the hash is generated, it cannot be reversed, making it a useful tool for ensuring the security of digital information.

Say x is the data to be hashed and F is the one way mathematical hash function, then,

y = F(x) such that it is near to impossible to find x such that F(x) = y.

In this project, we have used the SHA-256 hashing algorithm which is considered to be a secure and reliable algorithm and is widely used as a standard for digital signature algorithms and protocols. SHA-256 (Secure Hash Algorithm 256) is a cryptographic hash function that generates a fixed-size, 256-bit hash value from any input data. It works by taking an input message and processing it through a series of mathematical operations to produce the output hash value. The resulting hash is unique to the input message, and any small change to the input message will produce a completely different hash value. This property makes SHA-256 useful for verifying the integrity of data, as even a minor alteration to the original message will result in a different hash value.

In blockchain systems, Merkle trees are utilized to compute the hash value of a block, which involves hashing the complete set of transactions within the block to obtain a singular hash value. In Merkle trees, to compute the final hash or the Merkle root, the individual data elements are first hashed, then pairs of hashes are combined into a new hash through a process of repeated hashing until a single hash, the Merkle root, is produced. This is further explained in section 2.

Using Merkle trees for hashing provides several advantages for blockchain systems.

One significant advantage is the reduction in storage requirements. With Merkle trees, the entire set of transactions in a block can be represented compactly, reducing the storage space needed. This is in contrast to using hashing alone, which would require each transaction or block to be stored individually, leading to impractical storage requirements as the blockchain grows.

Another advantage of using Merkle trees is the increased efficiency of transaction verification. Verifying each transaction individually without Merkle trees would be resource-intensive and time-consuming. However, Merkle trees allow for efficient verification of transactions

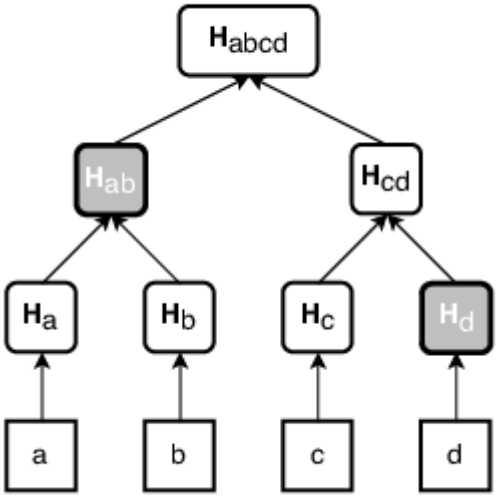
by only requiring a small subset of transactions to be verified, saving time and resources.

Moreover, Merkle trees provide a means of quickly detecting any tampering or corruption in transactions or blocks. Any changes to a transaction or block would result in a different Merkle root, which can be easily identified, making it challenging for malicious actors to tamper with the blockchain without being detected.

2. Merkle Path/Proof

In a blockchain, each block contains a set of transactions. To verify that a particular transaction is included in a block, one would typically need to download the entire block and check that the transaction is present. But in a Merkle tree implementation, a participant in the blockchain network needs to download only a small portion of the Merkle tree, known as a Merkle path, that proves that a particular transaction is included in the block.

A Merkle proof is a cryptographic proof that allows a user to verify that a specific piece of data is included in a larger dataset, without having to download or inspect the entire dataset.



Merkle proof for Data node C

To create a Merkle proof, a user starts at the root node, and recursively computes the hashes of each node down the relevant branch of the tree until they reach the node containing the desired data. They then include in the proof the hashes of all the sibling nodes on the path from the root to the leaf node. The resulting proof consists of the hash of the desired data and the sibling hashes, and can be used by anyone to verify that the data is indeed part of the original dataset, by hashing

each pair of sibling hashes and working their way up to the root node. If the resulting root hash matches the one provided in the proof, and the leaf hash is one of the sibling hashes, then the data is deemed to be part of the original dataset.

For example, in the above figure, to validate the existence of Data node C, the user can do it using the Merkle path [H_d, H_{ab}].

1. Calculate $H_c = F(c)$
2. Calculate H_{cd} by combining H_c (Step 1) and H_d (from the Merkle path)
3. Calculate Habcd by combining H_{cd} (Step 2) and H_{ab} (from the Merkle path)

In this way, the user is able to calculate the Merkle root by using just the Merkle path instead of the entire list of transactions.

Merkle proofs are commonly used in blockchain technology to allow nodes to efficiently verify that a particular transaction or block is part of the blockchain without having to download and store the entire chain.

3. Merkle Trees are Tamper Proof

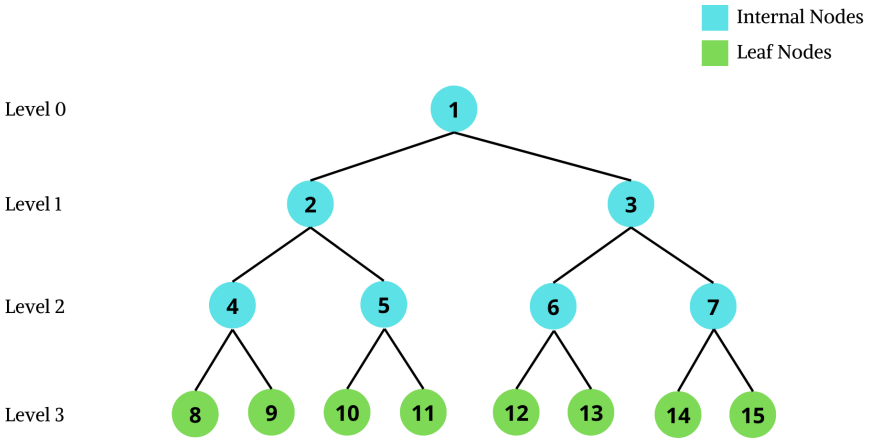
Merkle trees also help to ensure the security of the blockchain by making it difficult to modify or tamper with past transactions. Because each block's Merkle tree includes the hash of the previous block's Merkle tree, any modification to a past transaction would require the modification of all subsequent blocks' Merkle trees as well. This makes it very difficult to modify past transactions without being detected.

2. Theoretical Foundations of the Data Structure(s) utilised

A Merkle tree is a specific type of binary tree data structure that is commonly used in cryptography and data verification applications.

Binary Tree

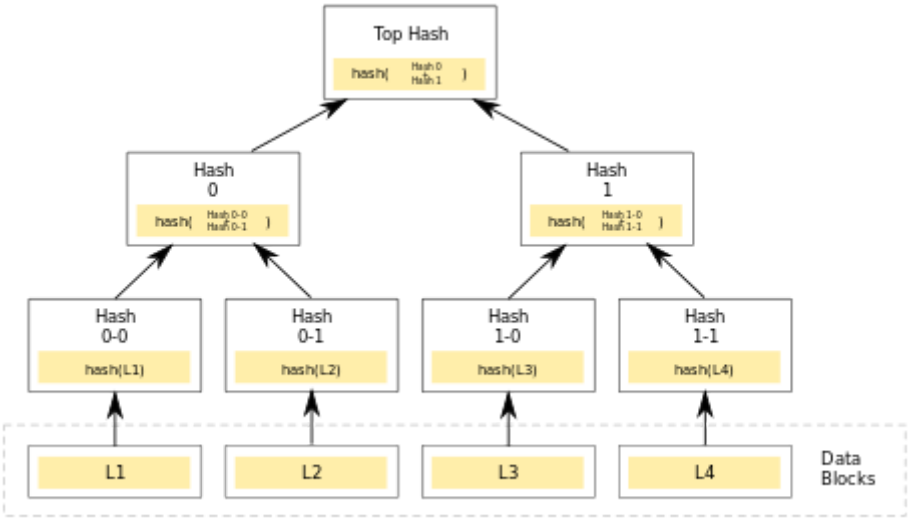
A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child. The topmost node of the tree is called the root, and each node in the tree represents a sub-tree, with the left child representing the left sub-tree and the right child representing the right sub-tree. Binary trees are commonly used in computer science for efficient searching and sorting algorithms, as well as in database indexing and file systems.



Binary Tree

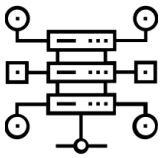
Merkle Tree

The concept of a Merkle tree was first introduced in a 1987 paper by Ralph Merkle, titled "A Digital Signature Based on a Conventional Encryption Function". This paper proposed the use of a binary hash tree (which later became known as a Merkle tree) to create a secure digital signature scheme.



Merkle Tree

In the paper, Merkle described how the hash tree could be used to create a digital signature that would be secure against attacks by a computationally bounded adversary. The basic idea was to hash the data to be signed and then recursively hash the resulting hashes until a single hash value was obtained.



The root hash of the resulting hash tree would then serve as the signature of the data.

Each leaf node, called data node, represents a specific transaction in the block, and each non-leaf node is the hash of its two child nodes. This process is repeated until a single root node, called Merkle root is created, which is the hash of the two child nodes that represent the entire block.

Hash(i) = F(F(left child), F(right child)), where
i is a non leaf node

Hash(i) = F(x), where
x is a transaction in the block, and
i is a leaf node

The root node is typically included in the header of a block in a blockchain, and it serves as a unique identifier for the block. By comparing the Merkle root of a block received from a peer with the Merkle root computed locally, a node in the network can verify the authenticity and integrity of the transactions in the block.

To verify the authenticity and integrity of a transaction, a node can compute the Merkle root by recursively hashing pairs of nodes starting with the leaf nodes, until the root node is reached. The resulting hash value can then be compared with the Merkle root stored in the blockchain to ensure that the transaction has not been tampered with.

Merkle Tree Implementation

Merkle Tree is a special variant of Binary Tree with different attributes. A Merkle tree is a type of hash tree where each node represents the hash of its child nodes, with the root node representing the hash of all the data in the tree. While a Merkle tree is typically represented as a binary tree, it is not a binary search tree.

A binary search tree is a specific type of binary tree where the left child of each node has a smaller key value than the node itself, and the right child has a greater key value. This property makes binary search trees useful for searching and sorting data.

In contrast, a Merkle tree is used for verifying data integrity in a distributed system. Each leaf node in the tree represents a block of data, and the hash of each block is calculated and stored in the leaf node. The non-leaf nodes in the tree represent the hashes of their child nodes, with the root node representing the hash of all the data in the tree.

While it is possible to represent a Merkle tree as a binary tree, the key values assigned to the nodes are derived from the hashes of the data blocks rather

than being explicitly assigned. This means that the nodes in a Merkle tree are not ordered in the same way as they are in a binary search tree.

In summary, while a Merkle tree can be represented as a binary tree, it is not a binary search tree and is used for a different purpose, namely, verifying the integrity of data in a distributed system.

Let us now see how the Merkle tree is constructed.

1. To begin, calculate the hash value for each individual data item. These hash values will serve as the leaf nodes of the Merkle tree.
2. Organize the leaf nodes into pairs, starting from the leftmost node and proceeding to the right.
3. Create a parent node for each pair by combining the hash values of the two child nodes and applying the hash function to the combined value. This resulting hash value will be the label of the parent node.
4. Repeat this process for the newly created parent nodes by grouping them into pairs and calculating the hash value for each pair.
5. Continue pairing and calculating hash values until only one node is left, which will be the Merkle root or root hash of the tree.

Time Complexity

The time complexity of a Merkle tree depends on the number of leaf nodes or individual data elements to be hashed, as well as the desired depth of the tree.

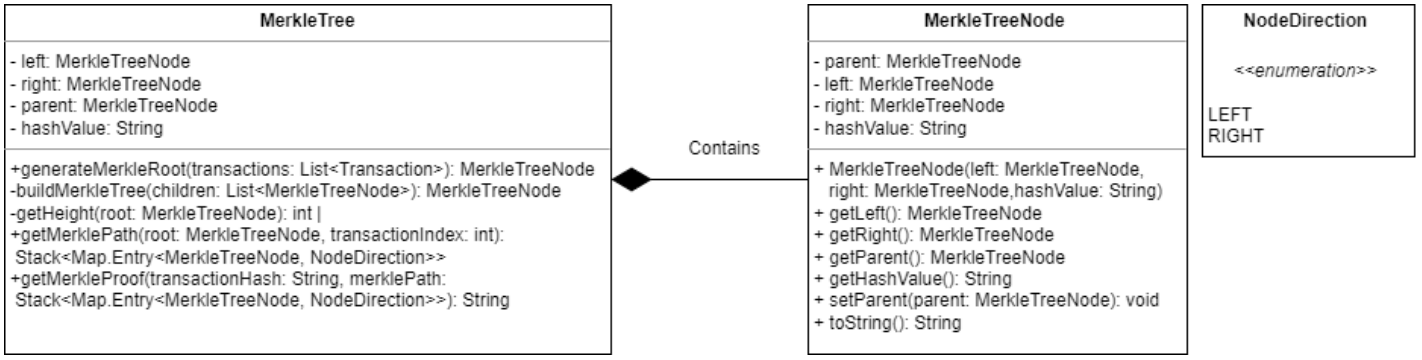
Building a Merkle tree from n leaf nodes requires O(n) hash computations, and verifying a proof for a single leaf node requires O(log n) hash computations. Thus, the time complexity for building a Merkle tree is O(n), while the time complexity for verifying a proof is O(log n).

Space Complexity

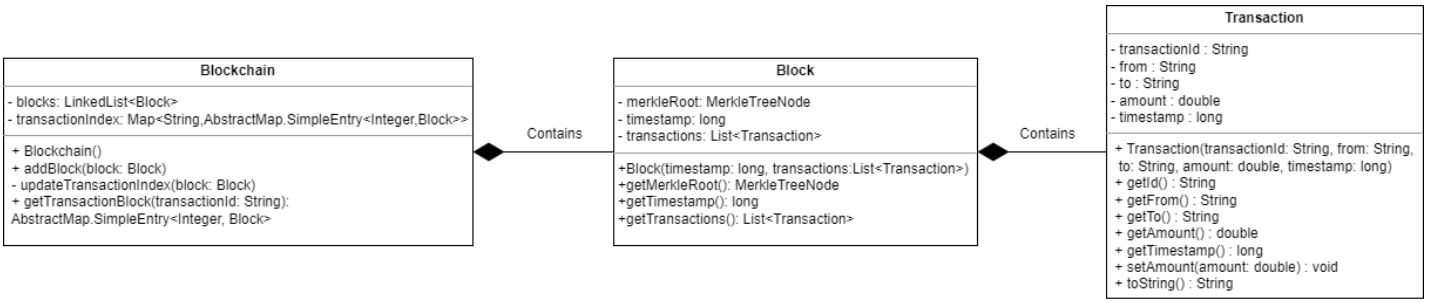
The space complexity of a Merkle tree depends on the number of leaf nodes or individual data elements to be hashed, as well as the desired depth of the tree.

Building a Merkle tree from n leaf nodes requires storage for the n leaf nodes themselves, as well as for the intermediate hashes that are computed during the tree-building process. The number of intermediate hashes is equal to the number of nodes in the tree minus the number of leaf nodes, and thus the space complexity of a Merkle tree is O(n) in terms of the number of leaf nodes.

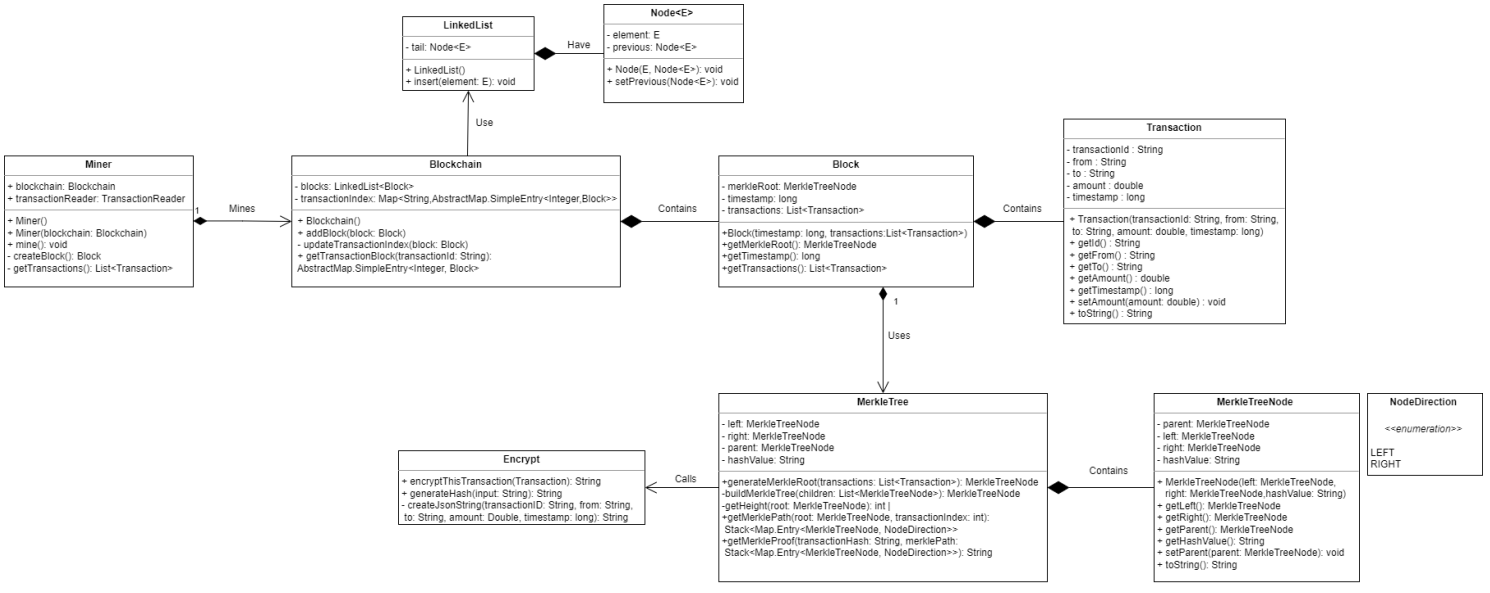
3. Analysis/Design (UML Diagram(s))



Merkle tree composition



Relation between Transactions, Block and Blockchain



System Architecture Diagram

4. Code Implementation

GitHub (link):
https://github.com/black-hawks/merkle_tree

Set of Experiments run and results:

1. Verification of a block in a blockchain

   When a block is added to the blockchain, it needs to be validated to ensure that the transactions within it are valid and the block has not been tampered with. To validate a block, we need to validate the transactions within the block.

   To check if transactions in the block are valid, we have considered few scenarios to demonstrate tampering situations:

   a. Valid block:
      Before starting the test, we fetched transactions generated in a one-second period and created a block by computing the Merkle root hash for them. In the test, we fetched the list of transactions from the block and regenerated the Merkle root, and compared it with the block's Merkle root to check if the block is valid.

   b. Block tampered by appending an invalid transaction:
      In this scenario, we created an invalid transaction and added it to the block's transaction list. Regenerating the Merkle root hash based on the manipulated transactions list results in a different hash than the Merkle root hash of the block, indicating that the block was tampered.

   c. Block tampered by replacing a valid transaction:
      In the block, there is a single transaction that has been replaced by an invalid transaction. When regenerating the merkle root, we will receive a different hash as a result.

   d. Block tampered by changing a valid transaction:
      In this case, we change the amount of existing transactions in the block. The result of this is that a different Merkle root hash is generated.

2. Verification of the presence/integrity of a transaction in a block

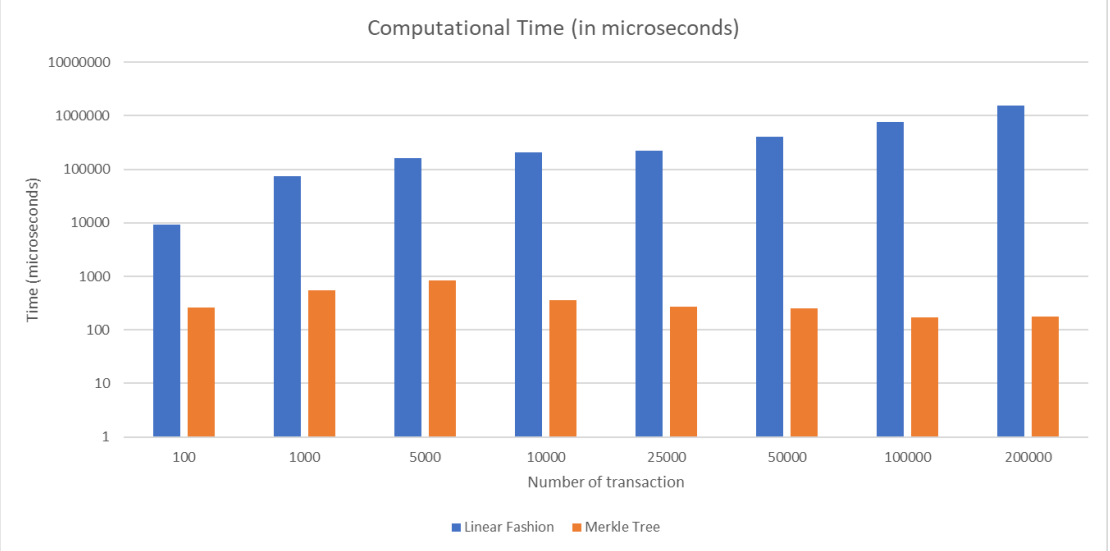
In order to verify the presence/integrity of a transaction in a block, we have considered 3 test scenarios:

a. Transaction is valid:
This is the ideal case where we use a valid transaction (a transaction that is actually present in the block) as the input.

b. Transaction has been tampered with:
In this case, we provide an invalid transaction as input.

c. Transaction is valid but block is invalid:
In this case, we provide a valid transaction but an invalid block (another transaction in the block has been tampered with)

We have compared the time taken to verify transactions with and without a Merkle tree. From the graph below, it can be seen that using Merkle trees reduces the time taken considerably.



Analyzing the Efficiency of Merkle Trees: A Comparison of Transaction Verification Times with and without Merkle Trees

Consider a block, B with 4 transactions (T1, T2, T3 and T4). Consider that we are verifying the presence of transaction T4. Consider $Hash_{actual}$ to be the hash of block B that was obtained from a trusted source.

Without Merkle trees, we will need to use the entire set of transactions (here, T1, T2, T3 and T4) and compute the hash in a linear fashion as follows:

$Hash_B = H(H(H(H(T1), H(T2)), H(T3)), H(T4))$

Then, we compare the computed $Hash_B$ with $Hash_{actual}$ to verify the presence of T4.

But using Merkle Trees, we do not need the entire set of transactions, but instead need only the Merkle path.

Here,
Merkle path = $[H_3, H_{12}]$, *where,*
$H_1 = H(T1)$, $H_2 = H(T2)$, $H_3 = H(T3)$, $H_4 = H(T4)$
$H_{12} = H(H_1, H_2)$
$H_{34} = H(H_3, H_4)$
$H_{1234} = H(H_{12}, H_{34})$

Thus we can calculate the block B's hash using just the Merkle path as follows:

$Hash_B = H(H_{12}, H(H_3, H(T4)))$

As it can be seen, the computation is considerably reduced from O(n) to O(log n).

5. Video of the Implementation running

Zoom link:
https://ucd-ie.zoom.us/rec/share/aJnfzaHBG3CMJOc7nX8AjfhuEs9NjorkScq8U8F2hU9RXRhan2iYrwzNwAy-7geE.pEIjQbDl8lmExO0Q

Zoom password:
70FM+zr4

6. References

[1] *Merkle, R.C. (1988). A Digital Signature Based on a Conventional Encryption Function. In: Pomerance, C. (eds) Advances in Cryptology — CRYPTO '87. CRYPTO 1987. Lecture Notes in Computer Science, vol 293. Springer, Berlin, Heidelberg.* https://doi.org/10.1007/3-540-48184-2_32
[2] *H. Liu, X. Luo, H. Liu and X. Xia, "Merkle Tree: A Fundamental Component of Blockchains," 2021 International Conference on Electronic Information Engineering and Computer Science (EIECS), Changchun, China, 2021, pp. 556-561,* https://doi.org/10.1109/EIECS53707.2021.9588047
[3] *Nakamoto, Satoshi, "Bitcoin: A Peer-to-Peer Electronic Cash System", October, 2008.* https://bitcoin.org/bitcoin.pdf