Manorath Ital D20A/25

Blockchain Experiment 1

Aim : Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root tree hash

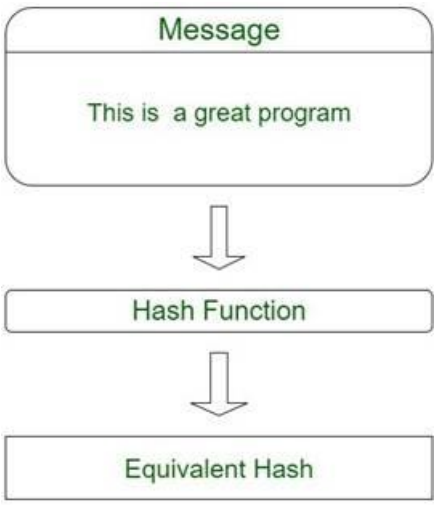
Theory :

1. Cryptographic Hash Function in Blockchain

A cryptographic hash function is a fundamental concept in blockchain technology that provides security and trust in a decentralized environment. It is a mathematical algorithm that takes input data of any size and produces a fixed-length output called a hash value or digest. This hash value uniquely represents the original data.

One of the most important properties of a cryptographic hash function is determinism, which means the same input will always generate the same hash output. Another key property is pre-image resistance, which ensures that it is practically impossible to determine the original input data from its hash value. Collision resistance guarantees that two different inputs cannot generate the same hash, thereby preventing data forgery. Hash functions are also designed for fast computation, making them efficient even when processing large volumes of data. The avalanche effect ensures that even a small change in the input data produces a completely different hash output.

In blockchain, cryptographic hash functions ensure data integrity by detecting any unauthorized changes in transactions or blocks. They securely link blocks together by including the previous block's hash in the current block, forming a tamper-proof chain. Hash functions are extensively used in Merkle Trees, block headers, mining, and digital signature verification. Therefore, cryptographic hash functions form the core security mechanism of blockchain systems.



2. Merkle Tree

A Merkle Tree is a specialized tree data structure used to efficiently organize and verify large sets of data in a secure and reliable manner. It is widely used in blockchain systems to manage and validate transactions within a block.

In a Merkle Tree, the leaf nodes contain the cryptographic hashes of individual transactions. These hashes are then paired, concatenated, and hashed again to form the parent nodes. This process continues recursively until a single hash remains at the top of the tree, known as the Merkle Root. The Merkle Root acts as a compact representation of all transactions in the block.

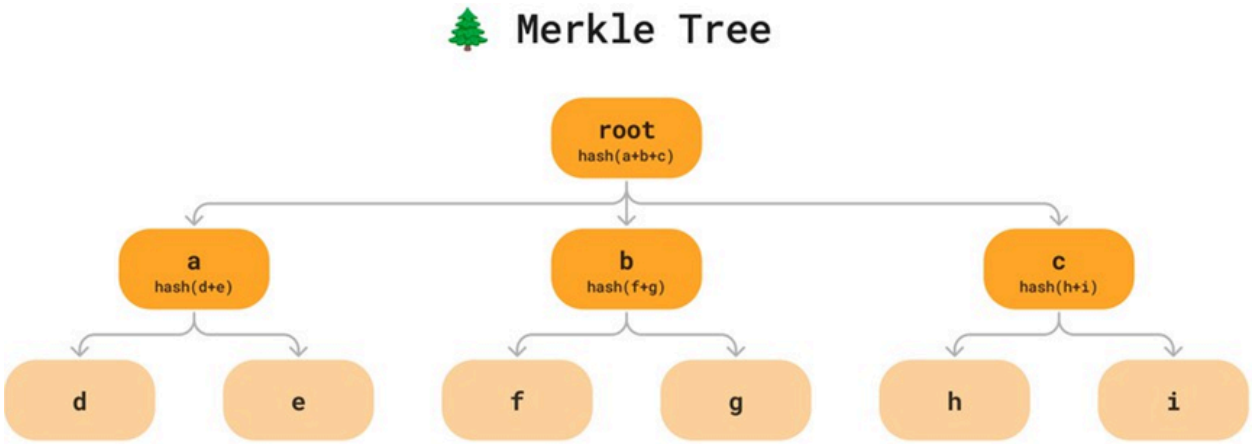
The primary purpose of a Merkle Tree is to ensure data integrity while reducing computational overhead. It allows quick verification of individual transactions without requiring access to the entire dataset. Because of these advantages, Merkle Trees are an essential component of blockchain platforms such as Bitcoin and Ethereum.

3. Structure of Merkle Tree

The structure of a Merkle Tree consists of multiple hierarchical levels of hash nodes arranged in a binary tree format. Each level of the tree plays a specific role in maintaining data integrity.

At the lowest level, the leaf nodes store the hashes of individual transactions. These leaf hashes are paired and combined to form intermediate nodes, where each intermediate node contains the hash of its two child nodes. At the top of the tree lies the root node, known as the Merkle Root, which represents the combined hash of all transactions in the tree.

If the number of transactions is odd, the last transaction hash is duplicated to maintain a balanced tree structure. This structured approach ensures consistency, efficient verification, and secure data representation across the blockchain.

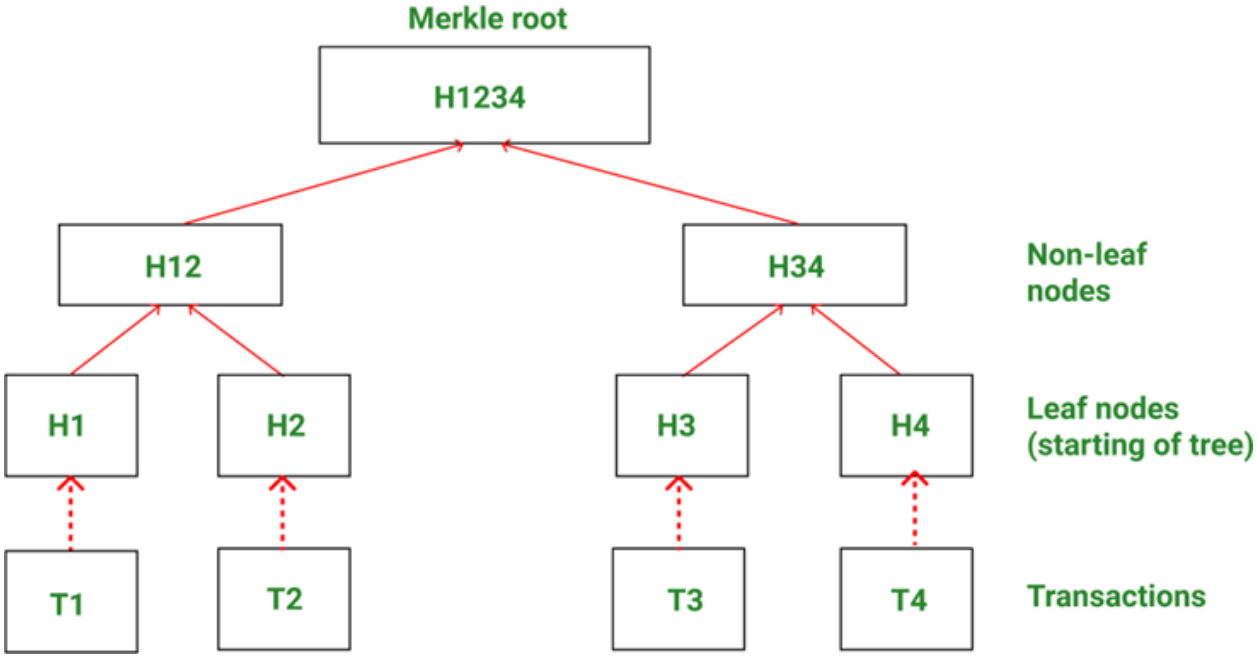


4. Merkle Root

The Merkle Root is the topmost hash in a Merkle Tree and serves as a single cryptographic summary of all transactions within a block. It is generated by repeatedly hashing pairs of transaction hashes until only one hash remains.

The Merkle Root is stored in the block header, making it a critical component during block validation and mining. Any modification in even a single transaction will result in a completely different Merkle Root, immediately signaling data tampering. This property

makes the Merkle Root highly effective for ensuring transaction integrity and trust in blockchain networks.



By representing thousands of transactions with a single hash, the Merkle Root enables efficient storage, fast verification, and secure block validation.

5. Working of Merkle Tree

The working of a Merkle Tree is based on a step-by-step hierarchical hashing process. Initially, each transaction is individually hashed using a cryptographic hash function. These transaction hashes are then grouped in pairs, concatenated, and hashed again to form the next level of the tree.

This pairing and hashing process continues level by level until only one hash remains. The final hash obtained is known as the Merkle Root. If there is an odd number of transaction hashes at any level, the last hash is duplicated to ensure pairing.

This mechanism allows efficient verification because only a small number of hashes are needed to verify a transaction instead of processing the entire dataset. As a result, Merkle Trees significantly improve performance and security in blockchain systems.

6. Benefits of Merkle Tree

Merkle Trees offer numerous benefits, especially in distributed and decentralized environments. They enable fast and efficient transaction verification by minimizing the amount of data required for validation. Storage requirements are reduced because only hash values are stored instead of entire datasets.

Merkle Trees make it easy to detect data tampering, as any change in transaction data alters the hash values up to the Merkle Root. They also support lightweight or Simplified Payment Verification (SPV) nodes, which can verify transactions without downloading the entire blockchain. Additionally, Merkle Trees enhance scalability and overall system security, making them ideal for large blockchain networks.

7. Use of Merkle Tree in Blockchain

In blockchain technology, Merkle Trees are used to organize and manage transactions efficiently within a block. All transaction hashes are structured into a Merkle Tree, and the resulting Merkle Root is stored in the block header.

This structure enables fast transaction verification and allows nodes to confirm the presence of a transaction without accessing all block data. Merkle Trees also support Simplified Payment Verification, reduce network bandwidth consumption, and improve computational efficiency. By ensuring data integrity and immutability, Merkle Trees play a crucial role in maintaining trust in blockchain systems.

8. Use Cases of Merkle Tree

Merkle Trees are used in a wide range of applications beyond blockchain. In cryptocurrencies, they help verify transactions securely. In distributed file systems such as IPFS, Merkle Trees ensure file integrity and efficient data sharing. They are also used in peer-to-peer networks for secure data validation.

Version control systems like Git rely on Merkle Trees to track changes and maintain file history. Additionally, Merkle Trees are used in secure databases and cloud storage systems to verify data consistency and prevent unauthorized modifications. These diverse applications highlight the importance of Merkle Trees in modern computing systems.

Manorath Ital D20A/25

Code and output :

Hash Generation using SHA-256: Developed a Python program to compute a SHA-256 hash for any given input string using the hashlib library.

import hashlib

# Take input from user
input_string = input("Enter a string to hash: ")

# Generate SHA-256 hash
sha256_hash = hashlib.sha256(input_string.encode())

# Display the hash value
print("SHA-256 Hash:", sha256_hash.hexdigest())

[1]:
```
import hashlib

# Take input from user
input_string = input("Enter a string to hash: ")

# Generate SHA-256 hash
sha256_hash = hashlib.sha256(input_string.encode())

# Display the hash value
print("SHA-256 Hash:", sha256_hash.hexdigest())
```

```
Enter a string to hash:  Manorath Ital
SHA-256 Hash: d9d0533386d35cd9328193346c1217cf305170c5b53f25d4dde6cd6272c00780
```

Manorath Ital D20A/25

Target Hash Generation with Nonce: Created a program to generate a hash code by concatenating a user input string and a nonce value to simulate the mining process.

import hashlib

Take input from user

input_string = input("Enter the data string: ")
nonce = input("Enter nonce value: ")

# Concatenate input string and nonce
combined_data = input_string + nonce

Generate SHA-256 hash

hash_result = hashlib.sha256(combined_data.encode())

Display the hash

print("Generated Hash:", hash_result.hexdigest())

```python
import hashlib

# Take input from user
input_string = input("Enter the data string: ")
nonce = input("Enter nonce value: ")

# Concatenate input string and nonce
combined_data = input_string + nonce

# Generate SHA-256 hash
hash_result = hashlib.sha256(combined_data.encode())

# Display the hash
print("Generated Hash:", hash_result.hexdigest())
```

```
Enter the data string:  Manorath
Enter nonce value:  1
Generated Hash: 628a1df6da68e36dea14231fb2ceea7b8540eade0404b105650ffd907321661f
```

Manorath Ital D20A/25

Proof-of-Work Puzzle Solving: Implemented a program to find the nonce that, when combined with a given input string, produces a hash starting with a specified number of leading zeros.

```
import hashlib

# Take input from user
data = input("Enter data string: ")
difficulty = int(input("Enter difficulty level (number of leading zeros): "))

nonce = 0
target = '0' * difficulty

print("Mining started...")

while True:
    # Combine data and nonce
    combined_data = data + str(nonce)

    # Generate SHA-256 hash
    hash_result = hashlib.sha256(combined_data.encode()).hexdigest()

    # Check if hash meets difficulty requirement
    if hash_result.startswith(target):
        print("\nValid Hash Found!")
        print("Nonce:", nonce)
        print("Hash:", hash_result)
        break

    nonce += 1
```

```
Enter data string:  Manorath Ital
Enter difficulty level (number of leading zeros):  3
Mining started...

Valid Hash Found!
Nonce: 1938
Hash: 000777f56becf308b84faff808402b5fc163eab0c3950b19948d23c0f65bb74b
```

Manorath Ital D20A/25

Merkle Tree Construction: Built a Merkle Tree from a list of transactions by recursively hashing pairs of transaction hashes, doubling up last nodes if needed, and generated the Merkle Root hash for blockchain transaction integrity.

```
import hashlib

def sha256_hash(data):
    """
    Generates SHA-256 hash of given data
    """
    return hashlib.sha256(data.encode()).hexdigest()

def merkle_root(transactions):
    """
    Generates the Merkle Root from a list of transactions
    """
    # Step 1: Hash all transactions
    hashes = [sha256_hash(tx) for tx in transactions]

    # Step 2: Build Merkle Tree
    while len(hashes) > 1:

        # If odd number of hashes, duplicate the last hash
        if len(hashes) % 2 != 0:
            hashes.append(hashes[-1])

        new_level = []

        for i in range(0, len(hashes), 2):
            combined_hash = hashes[i] + hashes[i + 1]
            new_hash = sha256_hash(combined_hash)
            new_level.append(new_hash)

        hashes = new_level

    # Step 3: Final hash is the Merkle Root
    return hashes[0]

# Main Program
transactions = [
    "Transaction 1",
```

```
    "Transaction 2",
    "Transaction 3",
    "Transaction 4",
    "Transaction 5"
]

print("Transactions:")
for tx in transactions:
    print(tx)

print("\nMerkle Root Hash:")
print(merkle_root(transactions))
```

```
Transactions:
Transaction 1
Transaction 2
Transaction 3
Transaction 4
Transaction 5

Merkle Root Hash:
2c2c4cdf817ca1233db4784bb8752eddca8428c5c88ad7fad7e7235532e33c3c
```

Conclusion

Cryptographic hash functions and Merkle Trees form the backbone of blockchain security. Hash functions ensure data integrity, immutability, and secure linking of blocks, while Merkle Trees enable efficient verification of large sets of transactions with minimal storage and computation. Together, they make blockchain systems secure, tamper-proof, scalable, and reliable for decentralized applications such as cryptocurrencies and distributed data storage systems.