Shreya Sawant
Roll no - 55

Experiment No: 1

Aim: Cryptography in Blockchain, Merkle root Tree Hash

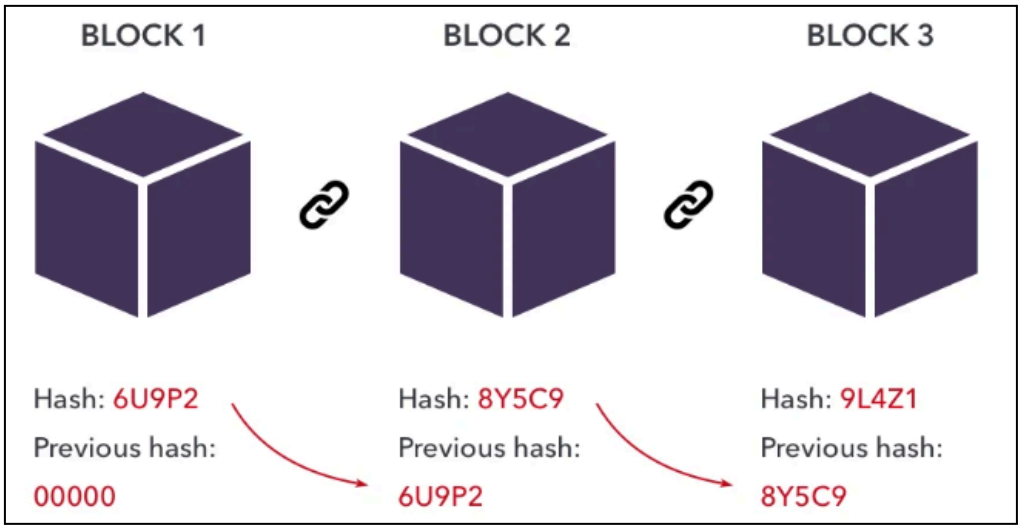
Theory:

1. Cryptographic Hash Functions in Blockchain

Cryptographic hash functions play a fundamental role in the functioning and security of blockchain technology. A cryptographic hash function is a mathematical algorithm that takes an input of any size and produces a fixed-length output known as a hash value or digest. In blockchain systems, commonly used hash functions such as SHA-256 are designed to be deterministic, collision-resistant, and computationally efficient.

In a blockchain, hash functions ensure data integrity by converting transaction data and block information into unique hash values. Even a small change in the input data results in a completely different hash, making unauthorized data modification easily detectable. Hash functions also link blocks together, as each block contains the hash of the previous block, forming an immutable chain. This chaining mechanism ensures that altering one block would require recalculating hashes for all subsequent blocks, which is practically infeasible in a decentralized network.

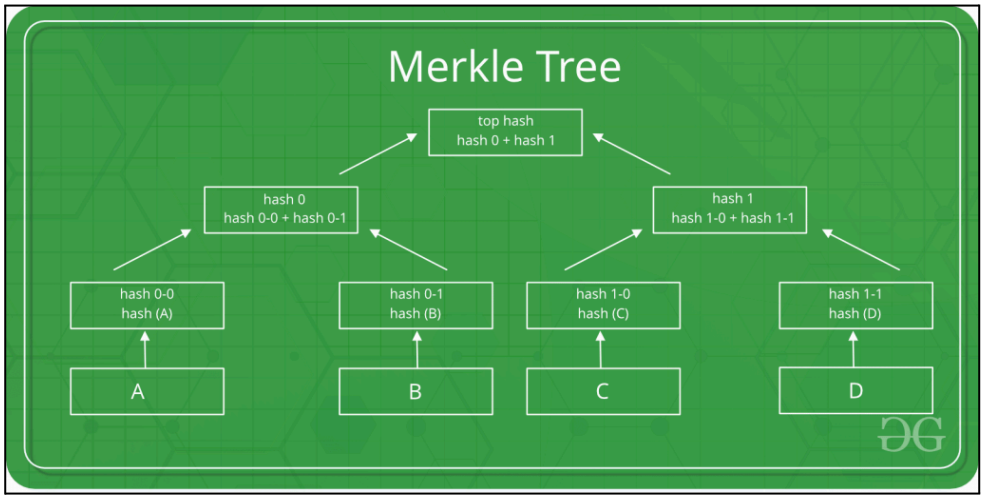
Additionally, cryptographic hash functions are used in mining and consensus mechanisms, where miners compete to find a valid hash that satisfies predefined conditions. Overall, hash functions provide security, transparency, and trust in blockchain networks without relying on a central authority.



2. What is a Merkle Tree?

A Merkle Tree, also known as a hash tree, is a hierarchical data structure used in blockchain and distributed systems to efficiently verify and organize large sets of data. It consists of leaf nodes, which store the hash values of individual data blocks or transactions, and non-leaf nodes, which store the hash of their child nodes. The root of the tree, known as the Merkle Root, represents a single hash that summarizes all the transactions within a block.

The Merkle Tree structure allows blockchains to verify the integrity of transaction data efficiently without requiring access to the entire dataset. Because the Merkle Root is stored in the block header, it provides a compact and secure representation of all transactions in that block. Any change in a transaction will alter its hash, which propagates up the tree and changes the Merkle Root, making tampering immediately evident.



3. What is a Cryptographic Puzzle and the Golden Nonce?

A cryptographic puzzle is a computational challenge used in blockchain systems, particularly in Proof of Work consensus mechanisms, to regulate block creation and maintain network security. The puzzle requires miners to find a specific input value that, when combined with block data and passed through a cryptographic hash function, produces a hash that meets predefined difficulty criteria, such as having a certain number of leading zeros.

The Golden Nonce refers to the specific nonce value that successfully solves the cryptographic puzzle. A nonce is a random number that miners repeatedly change in order to generate different hash outputs. When a miner finds the correct nonce that results in a valid hash, that nonce is called the Golden Nonce. This discovery allows the miner to propose a new block to the network and receive a reward.

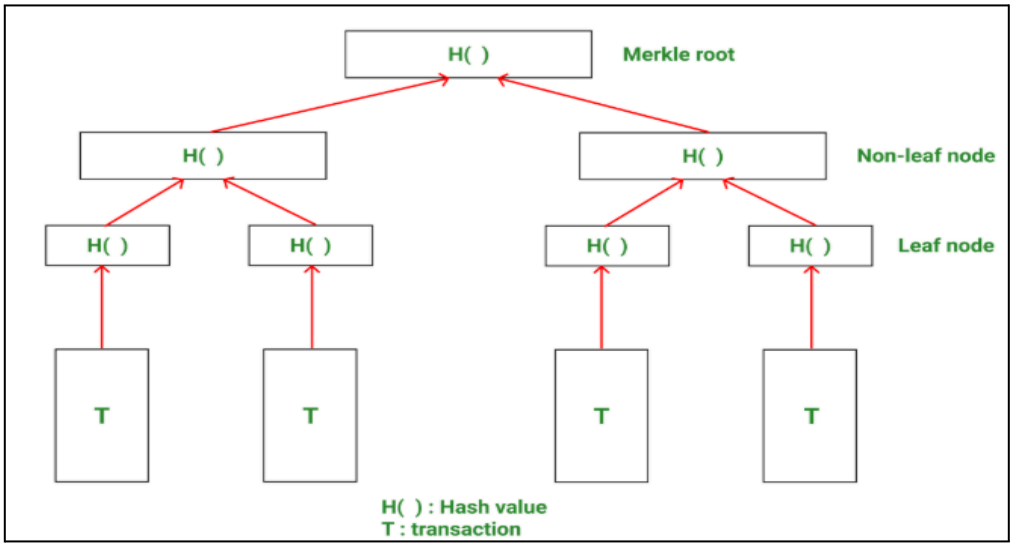
The process of solving cryptographic puzzles ensures that adding new blocks requires significant computational effort, making it difficult for malicious actors to manipulate the blockchain. At the same time, verification of the solution by other nodes is fast and

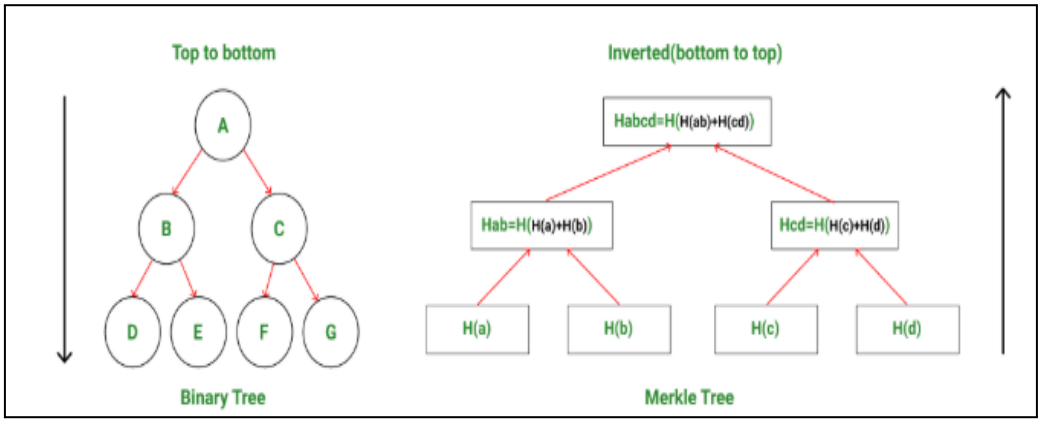
efficient.

4. How Does a Merkle Tree Work?

A Merkle Tree works by recursively hashing data in pairs until a single root hash is produced. Initially, each transaction in a block is hashed individually to form the leaf nodes. These hashes are then grouped in pairs, and each pair is combined and hashed again to form the next level of nodes. This process continues layer by layer until only one hash remains at the top, known as the Merkle Root.

If the number of transactions is odd, the last hash is duplicated to ensure pairing. The Merkle Root is stored in the block header and acts as a fingerprint for all transactions in that block. To verify a specific transaction, only a small number of hashes, known as a Merkle Proof, are required instead of the entire dataset. This makes transaction verification fast and resource-efficient, especially for lightweight nodes.





5. Benefits of Merkle Tree

Merkle Trees provide several important advantages in blockchain systems. One of the key benefits is efficient data verification, as only a small portion of the tree is needed to verify the authenticity of a transaction. This significantly reduces computational and storage requirements.

- Efficient Data Verification
  Merkle Trees allow transactions to be verified efficiently without accessing the entire dataset. Only a limited number of hashes from the transaction to the Merkle Root are required, which reduces computational effort and speeds up verification.
- Enhanced Data Security and Integrity
  Each node in a Merkle Tree is created using cryptographic hash functions. Any modification to a transaction changes its hash and subsequently alters the Merkle Root, making data tampering easy to detect.
- Support for Scalability
  Merkle Trees enable blockchains to handle a large number of transactions efficiently. Even as transaction volume increases, verification remains fast because only a small number of hashes are involved.
- Reduced Storage Requirements
  Nodes do not need to store all transaction data to verify blocks. Lightweight nodes can store only block headers containing the Merkle Root, saving significant storage space.
- Simplified Payment Verification (SPV)
  Merkle Trees support Simplified Payment Verification, allowing lightweight clients to confirm transaction inclusion without downloading the full blockchain, making them suitable for resource-limited devices.

6. Use Cases of Merkle Tree

Merkle Trees are widely used in blockchain systems such as Bitcoin and Ethereum to organize and verify transactions within a block. They enable nodes to confirm transaction inclusion efficiently, which is essential for maintaining decentralized trust.

- Blockchain Transaction Verification
  Merkle Trees are extensively used in blockchain platforms such as Bitcoin and Ethereum to organize transactions within a block. They allow nodes to efficiently verify whether a specific transaction is included in a block, which is essential for maintaining trust in decentralized systems.

- Distributed Databases
  In distributed databases, Merkle Trees are used to verify data consistency across multiple servers. They help quickly identify differences between datasets stored on different nodes, ensuring synchronization and integrity.

- File Systems
  Merkle Trees are used in secure and distributed file systems to detect data

corruption and unauthorized modifications. By comparing Merkle Roots, systems can efficiently verify file integrity without scanning the entire file structure.

- Peer-to-Peer (P2P) Networks
  In peer-to-peer networks, Merkle Trees are used to ensure secure and reliable data transmission. They allow nodes to verify the correctness of data chunks received from other peers.

Output-

```python
import hashlib

def create_hash(string):
    # Create a hash object using SHA-256 algorithm
    hash_object = hashlib.sha256()
    # Convert the string to bytes and update the hash object
    hash_object.update(string.encode('utf-8'))
    # Get the hexadecimal representation of the hash
    hash_string = hash_object.hexdigest()
    # Return the hash string
    return hash_string

# Example usage
input_string = input("Enter a string: ")
hash_result = create_hash(input_string)
print("Hash:", hash_result)
```

```
Enter a string: Shreya
Hash: d5d9dc386fe4dd2dae4cca1edf2d599782d035d6cdba4ab2ddf615acb3853e30
```

Program #2: Program to generate required target hash with input string and nonce

```python
import hashlib

# Get user input
input_string = input("Enter a string: ")
nonce = input("Enter the nonce: ")

# Concatenate the string and nonce
hash_string = input_string + nonce

# Calculate the hash using SHA-256
hash_object = hashlib.sha256(hash_string.encode('utf-8'))
hash_code = hash_object.hexdigest()

# Print the hash code
print("Hash Code:", hash_code)
```

```
Enter a string: Shreya
Enter the nonce: 3
Hash Code: d27cf6ad1394d3702d239518ab209fc256e0b04a9f7594053e6f678ee5694db7
```

Program #3: Python code for Solving Puzzle for leading zeros with expected nonce and given string

```python
import hashlib

def find_nonce(input_string, num_zeros):
    nonce = 0
    hash_prefix = '0' * num_zeros

    while True:
        # Concatenate the string and nonce
        hash_string = input_string + str(nonce)
        # Calculate the hash using SHA-256
        hash_object = hashlib.sha256(hash_string.encode('utf-8'))
        hash_code = hash_object.hexdigest()

        # Check if the hash code has the required number of leading zeros
        if hash_code.startswith(hash_prefix):
            print("Hash:",hash_code )
            return nonce

        nonce += 1

# Get user input
input_string = "Hello Shreya"
num_zeros = 1

# Find the expected nonce
expected_nonce = find_nonce(input_string, num_zeros)

# Print the expected nonce
print("Input String:", input_string)
print("Leading Zeros:", num_zeros)
print("Expected Nonce:", expected_nonce)
```

```
Hash: 08fab5d4b70930b4093b80eb6884e8dcb06b175af48a4e1411fac415815618c4
Input String: Hello Shreya
Leading Zeros: 1
Expected Nonce: 29
```

Program 4: Generating Merkle Tree for given set of Transactions

```python
import hashlib

def build_merkle_tree(transactions):
    if len(transactions) == 0:
        return None

    if len(transactions) == 1:
        return transactions[0]

    # Recursive construction of the Merkle Tree
    while len(transactions) > 1:
        if len(transactions) % 2 != 0:
            transactions.append(transactions[-1])

        new_transactions = []
        for i in range(0, len(transactions), 2):
            combined = transactions[i] + transactions[i+1]
            hash_combined = hashlib.sha256(combined.encode('utf-8')).hexdigest()
            new_transactions.append(hash_combined)

        transactions = new_transactions

    return transactions[0]

# Example usage
transactions = ["Transaction 1", "Transaction 2", "Transaction 3", "Transaction 4", "Transaction 5"]

merkle_root = build_merkle_tree(transactions)
print("Merkle Root:", merkle_root)
```

```
Merkle Root: a4a18941de1162b17a46c4f8c87d8a0850b46fad17ac881340061d9233785077
```

Conclusion- In this experiment, we implemented cryptographic hashing, simulated mining using a nonce, solved a Proof-of-Work puzzle to find the Golden Nonce, and constructed a Merkle Tree from given transactions by hashing and combining them level by level to obtain the Merkle Root. We learned how SHA-256 ensures data integrity, how nonce and mining demonstrate blockchain security, and how Merkle Trees efficiently verify and protect transaction data in a blockchain.