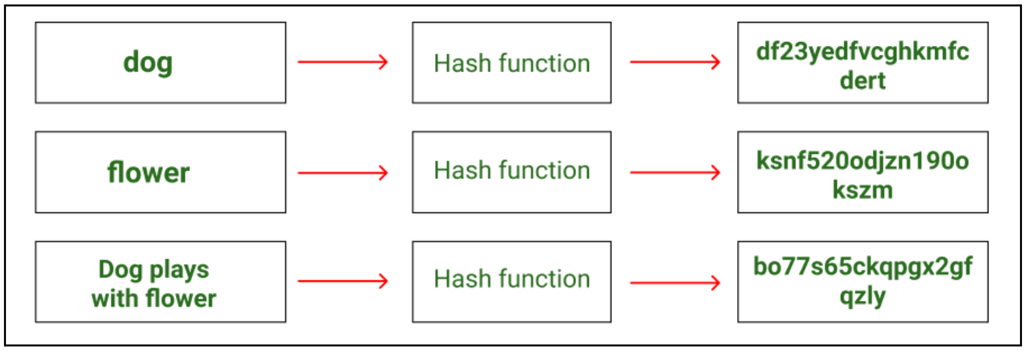
EXPERIMENT NO: 01

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.



Characteristics:

1. **Deterministic** – Same input always produces the same hash
2. **Pre-image resistance** – Original data cannot be derived from hash
3. **Collision resistance** – Two different inputs cannot have same hash
4. **Fast computation**
5. **Avalanche effect** – Small change in input causes large change in hash

Role in Blockchain:

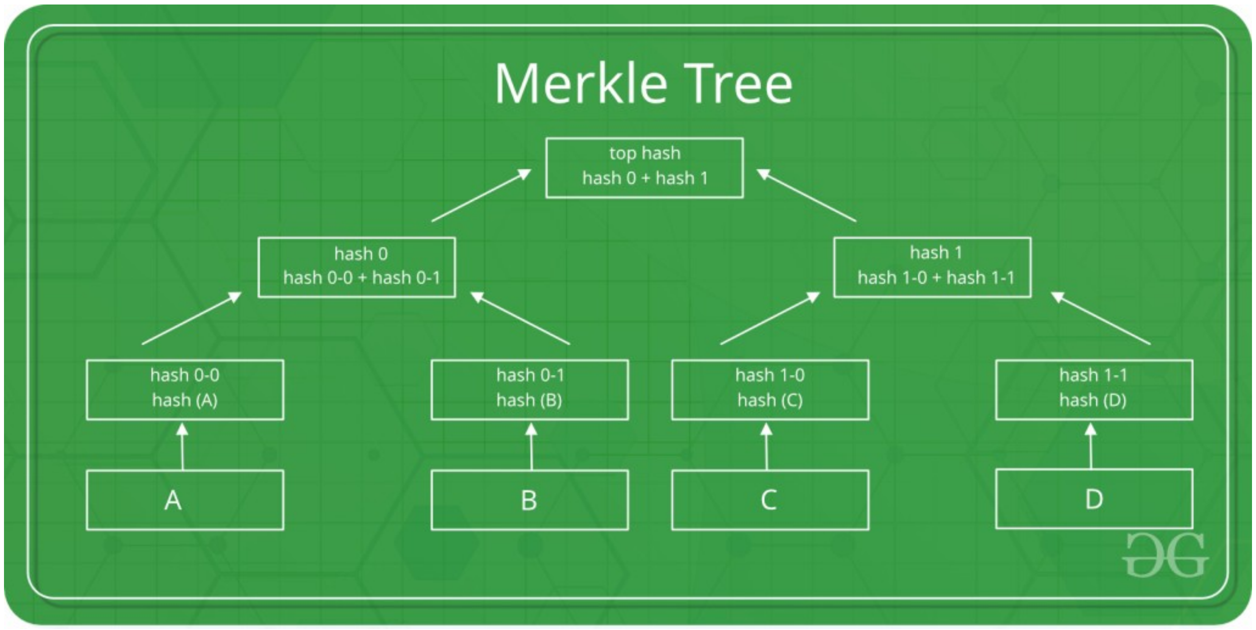
- Ensures **data integrity**
- Links blocks securely using hashes
- Used in **Merkle Trees**, block headers, and digital signatures

Thus, cryptographic hash functions form the **security backbone of blockchain**.

2. Merkle Tree

A **Merkle Tree** is a **binary tree data structure** used to efficiently verify large amounts of data.

Merkle tree also known as hash tree is a data structure used for data verification and synchronization.
It is a tree data structure where each non-leaf node is a hash of it's child nodes. All the leaf nodes are at the same depth and are as far left as possible.
It maintains data integrity and uses hash functions for this purpose.



Explanation:

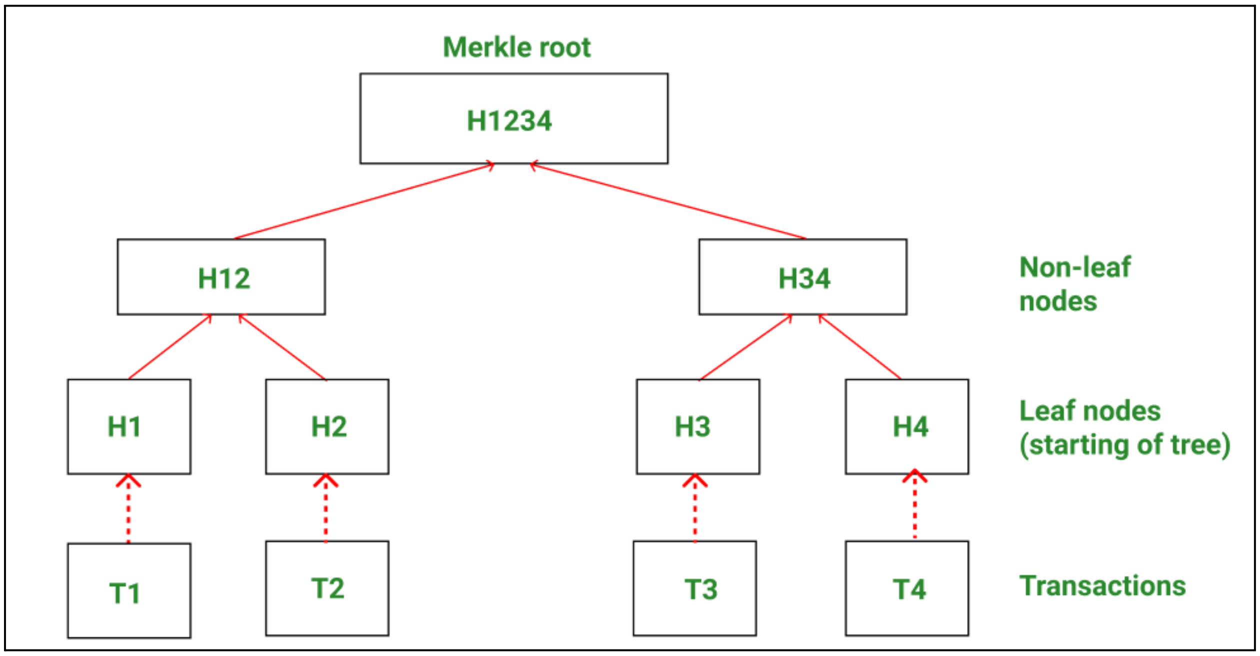
- Leaf nodes store **hashes of transactions**
- Parent nodes store **hash of concatenated child hashes**
- The topmost node is called the **Merkle Root**

Purpose:

- Ensures **data integrity**
- Allows fast verification of transactions
- Used in blockchain systems like **Bitcoin and Ethereum**

Merkle Trees reduce computation and improve security in distributed systems.

3. Structure of Merkle Tree



1. A blockchain can potentially have thousands of blocks with thousands of transactions in each block. Therefore, memory space and computing power are two main challenges.

2. It would be optimal to use as little data as possible for verifying transactions, which can reduce CPU processing and provide better securityy, and this is exactly what Merkle trees offer.

3. In a Merkle tree, transactions are grouped into pairs. The hash is computed for each pair and this is stored in the parent node. Now the parent nodes are grouped into pairs and their hash is stored one level up in the tree. This continues till the root of the tree. The different types of nodes in a Merkle tree are:

- **Root node:** The root of the Merkle tree is known as the Merkle root and this Merkle root is stored in the header of the block.
- **Leaf node:** The leaf nodes contain the hash values of transaction data. Each transaction in the block has its data hashed and then this hash value (also known as transaction ID) is stored in leaf nodes.

- **Non-leaf node:** The non-leaf nodes contain the hash value of their respective children. These are also called intermediate nodes because they contain the intermediate hash values and the hash process continues till the root of the tree.

4. Bitcoin uses the SHA-256 hash function to hash transaction data continuously till the Merkle root is obtained.

5. Further, a Merkle tree is **binary in nature**. This means that the **number of leaf nodes needs to be even** for the Merkle tree to be constructed properly. In case there is an odd number of leaf nodes, the tree duplicates the last hash and makes the number of leaf nodes even.

---

4. Merkle Root

The Merkle Root is the topmost hash value obtained from a Merkle Tree and it represents all the transactions contained in a block in the blockchain.

Explanation:

- Each transaction in a block is first converted into a hash
- These hashes are combined pairwise and hashed repeatedly
- The final single hash generated at the top of the tree is called the Merkle Root

Importance of Merkle Root:

1. It acts as a unique digital fingerprint of all transactions in a block
2. It is stored in the block header, not the entire transaction list
3. If any single transaction is modified, the Merkle Root changes completely
4. It helps in efficient verification of transactions without downloading the full block
5. It ensures data integrity, security, and immutability of the blockchain

Role in Blockchain:

- Used by miners during block validation
- Enables Simplified Payment Verification (SPV) for lightweight nodes
- Links transaction data securely with the block structure

Thus, the Merkle Root is a critical component of blockchain architecture that provides **efficient, secure, and tamper-proof transaction verification**.

---

5. Working of Merkle Tree

The working of a Merkle Tree involves hierarchical hashing.

Steps:

1. Each transaction in a block is hashed individually using a cryptographic hash function.
2. The transaction hashes are paired and concatenated.
3. The concatenated hashes are again hashed to form parent nodes.
4. This process is repeated level by level until only one hash remains.
5. The final hash obtained at the top is called the Merkle Root.
6. If the number of transactions is odd, the last hash is duplicated to maintain pairing

Example: Consider a block having 4 transactions- T1, T2, T3, T4. These four transactions have to be stored in the Merkle tree and this is done by the following steps-

Step 1: The hash of each transaction is computed.

H1 = Hash(T1).

Step 2: The hashes computed are stored in leaf nodes of the Merkle tree.

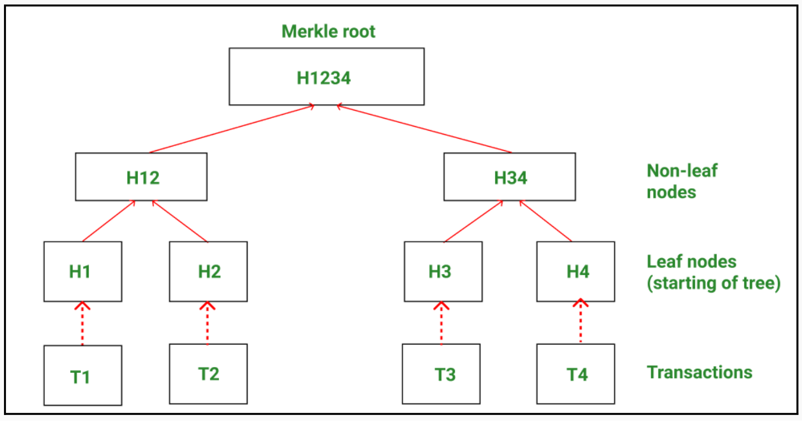
Step 3: Now non-leaf nodes will be formed. In order to form these nodes, leaf nodes will be paired together from left to right, and the hash of these pairs will be calculated. Firstly hash of H1 and H2 will be computed to form H12. Similarly, H34 is computed. Values H12 and H34 are parent nodes of H1, H2, and H3, H4 respectively. These are non-leaf nodes.

H12 = Hash(H1 + H2)

H34 = Hash(H3 + H4)

Step 4: Finally H1234 is computed by pairing H12 and H34. H1234 is the only hash remaining. This means we have reached the root node and therefore H1234 is the Merkle root.

H1234 = Hash(H12 + H34)



This process allows efficient verification of large transaction sets.

---

6. Benefits of Merkle Tree

1. **Efficient verification:** Merkle trees offer efficient verification of integrity and validity of data and significantly reduce the amount of memory required for verification. The proof of verification does not require a huge amount of data to be transmitted across the blockchain network. Enable trustless transfer of cryptocurrency in the peer-to-peer, distributed system by the quick verification of transactions.
2. **No delay:** There is no delay in the transfer of data across the network. Merkle trees are extensively used in computations that maintain the functioning of cryptocurrencies.
3. **Less disk space:** Merkle trees occupy less disk space when compared to other data structures.
4. **Unaltered transfer of data:** Merkle root helps in making sure that the blocks sent across the network are whole and unaltered.
5. **Tampering Detection:** Merkle tree gives an amazing advantage to miners to check whether any transactions have been tampered with.

---

7. Use of Merkle Tree in Blockchain

- In a centralized network, data can be accessed from one single copy. This means that nodes do not have to take the responsibility of storing their own copies of data and data can be retrieved quickly.
- However, the situation is not so simple in a distributed system.
- Let us consider a scenario where blockchain does not have Merkle trees. In this case, every node in the network will have to keep a record of every single transaction that has occurred because there is no central copy of the information.

- This means that a huge amount of information will have to be stored on every node and every node will have its own copy of the ledger. If a node wants to validate a past transaction, requests will have to be sent to all nodes, requesting their copy of the ledger. Then the user will have to compare its own copy with the copies obtained from several nodes.
- Any mismatch could compromise the security of the blockchain. Further on, such verification requests will require huge amounts of data to be sent over the network, and the computer performing this verification will need a lot of processing power for comparing different versions of ledgers.
- Without the Merkle tree, the **data itself has to be transferred all over the network** for verification.
- Merkle trees allow comparison and verification of transactions with **viable computational power and bandwidth**. Only a small amount of information needs to be sent, hence compensating for the huge volumes of ledger data that had to be exchanged previously.

---

8. Use Cases of Merkle Tree (5 Marks)

A Merkle Tree is widely used in systems that require efficient, secure, and reliable data verification. It allows verification of large data sets with minimal computation and storage.

1. Blockchain and Cryptocurrencies

In blockchain systems like **Bitcoin and Ethereum**, Merkle Trees are used to:

- Organize transactions within a block
- Generate the Merkle Root, which is stored in the block header
- Verify transactions efficiently without downloading the entire block
  This ensures data integrity, security, and immutability of transactions.

2. Distributed File Systems (IPFS)

Merkle Trees are used in distributed storage systems such as **InterPlanetary File System (IPFS)** to:

- Verify file integrity across distributed nodes
- Detect corrupted or modified data
- Ensure reliable file sharing in decentralized networks

3. Peer-to-Peer (P2P) Networks

In P2P networks, Merkle Trees help to:

- Verify the correctness of data received from multiple peers
- Reduce bandwidth by verifying small hash values instead of full data
- Prevent malicious nodes from sending tampered data

4. Version Control Systems (Git)

Git uses Merkle Trees to:

- Track changes in files and directories
- Detect unauthorized or accidental modifications
- Maintain integrity of project history
  Each commit generates a hash that depends on previous commits, ensuring **tamper-proof version tracking**.

5. Secure Databases and Cloud Storage

Merkle Trees are used in databases and cloud platforms to:

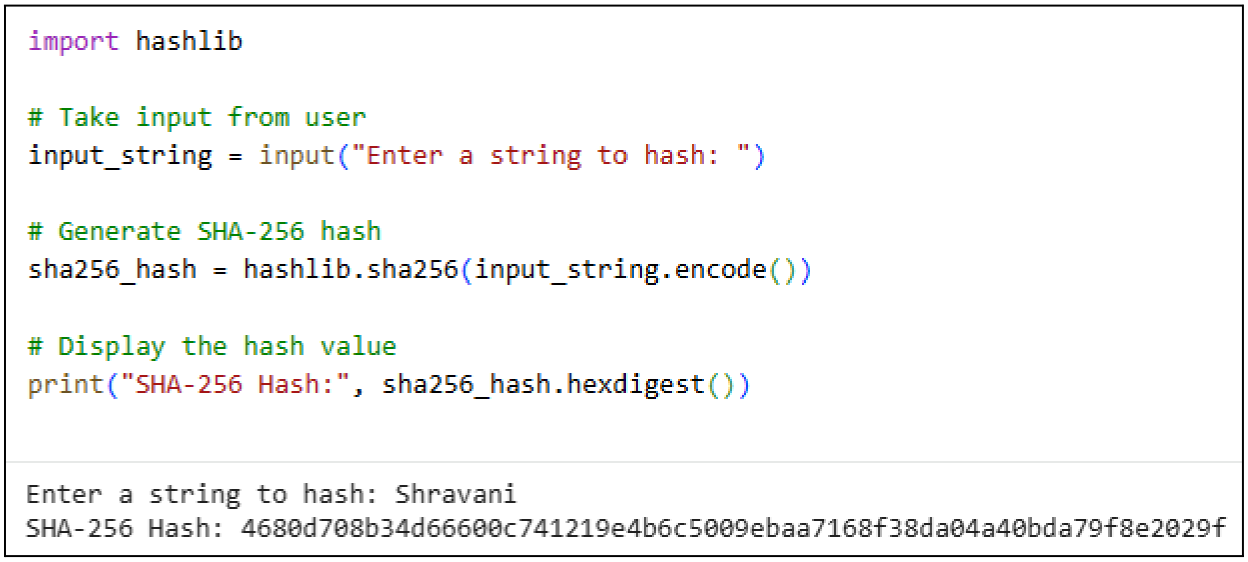
- Verify integrity of large datasets
- Perform efficient audits and data validation
- Ensure that stored data has not been altered

CODE:

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

```
CODE:
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())
```

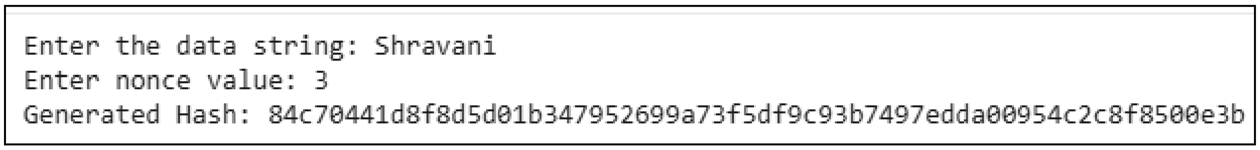


```
import hashlib

# 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: Shravani
SHA-256 Hash: 4680d708b34d66600c741219e4b6c5009ebaa7168f38da04a40bda79f8e2029f
```

2. 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 proc

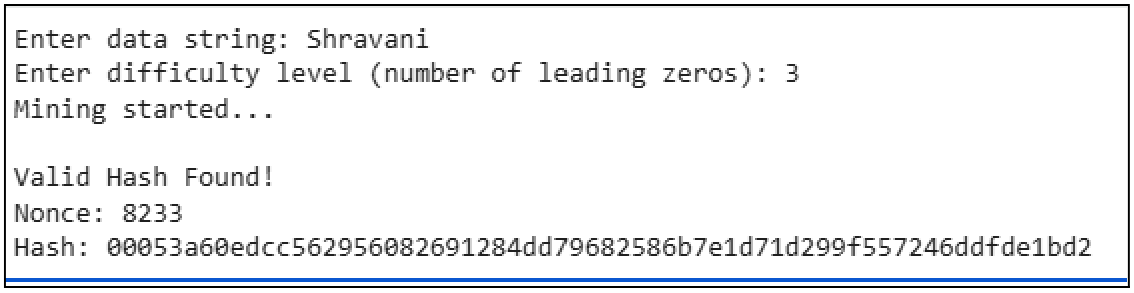
```
hashlib

# 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: Shravani
Enter nonce value: 3
Generated Hash: 84c70441d8f8d5d01b347952699a73f5df9c93b7497edda00954c2c8f8500e3b
```

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

```python
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
```

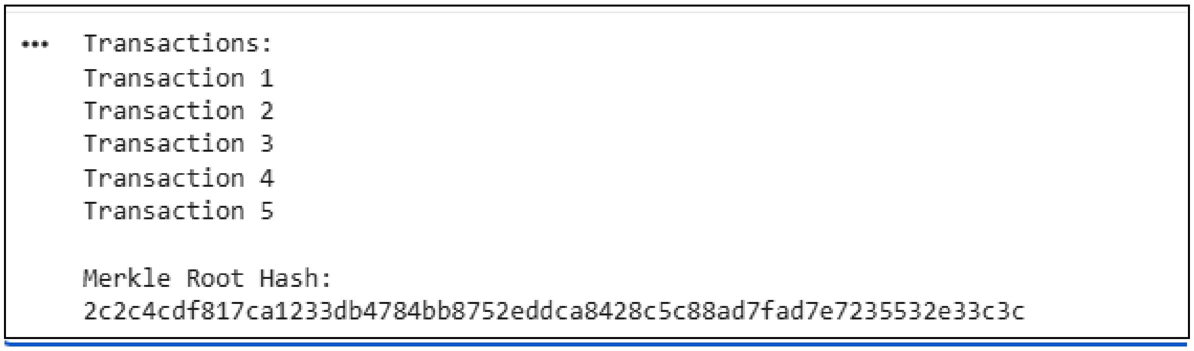


```
Enter data string: Shravani
Enter difficulty level (number of leading zeros): 3
Mining started...

Valid Hash Found!
Nonce: 8233
Hash: 00053a60edcc562956082691284dd79682586b7e1d71d299f557246ddfde1bd2
```

4. 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 Merkl 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 = [
```

```python
    "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:

This experiment helped in understanding the core cryptographic concepts used in blockchain technology. SHA-256 ensures data integrity, Proof of Work provides security through computational effort, and Merkle Trees enable efficient transaction verification. These concepts collectively form the foundation of secure and decentralized blockchain networks.