Ansh Sarfare **D20A - 51**

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 mathematical function that takes an input string of any length and converts it into a fixed-length output string known as a **hash value**. These hash functions are designed to be secure, fast, and irreversible, making them suitable for blockchain applications.

To be cryptographically secure and useful, a hash function must satisfy the following properties:

- **Collision Resistance:**
  It should be computationally difficult to find two different messages m_1 and m_2 such that hash(k, m_1) = hash(k, m_2) where *k* is a key value.

- **Preimage Resistance:**
  Given a hash value *h*, it should be infeasible to determine the original input *m* such that h = hash(k, m). The original data cannot be derived from the hash.

- **Second Preimage Resistance:**
  Given a message m_1, it should be difficult to find another message m_2 such that both produce the same hash.

- **Large Output Space:**
  Hash functions produce a large number of possible outputs, making brute-force collision attacks impractical.

- **Deterministic:**
  For the same input, the hash function must always generate the same output.

- **Avalanche Effect:**
  A very small change in input results in a completely different hash output.

- **Puzzle Friendliness:**
  Even if a portion of the hash is known, it should be impossible to predict the remaining bits.

- **Fixed-Length Mapping:**
  Regardless of input size, the output hash is always of fixed length.

Role of Hash Functions in Blockchain

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

2. Merkle Tree (Hash Tree)

A Merkle Tree, also known as a **Hash Tree**, is a data structure used for efficient data verification and synchronization. It is a binary tree in which each non-leaf node is the cryptographic hash of its child nodes.

All leaf nodes are located at the same depth and positioned as far left as possible. The hash at the top of the tree is called the **Merkle Root**, which represents the entire dataset.

Key characteristics:

- Enables efficient handling of large datasets
- Any change in data can be easily detected
- The root hash acts as a fingerprint for the complete data

3. Structure of a Merkle Tree

A Merkle Tree is an **inverted binary tree** constructed from the bottom up using cryptographic hashes. It consists of the following components:

- **Leaf Nodes:**
  The lowest level of the tree. Each leaf node contains the hash of an individual data block or transaction.

- **Internal Nodes:**
  Each internal node is generated by concatenating the hashes of its two child nodes and hashing the combined value.

- **Merkle Root:**
  The topmost node of the tree. It uniquely represents all the data below it. Any modification in the underlying data results in a different Merkle Root, indicating

tampering.

4. Merkle Root

The **Merkle Root** is the final hash at the top of a Merkle Tree. It serves as a concise summary of all transactions within a block. In blockchains like Bitcoin, the Merkle Root is stored in the block header.

Importance of Merkle Root

1. Acts as a unique digital fingerprint of all transactions in a block
2. Stored in the block header instead of the full transaction list
3. Any transaction modification changes the Merkle Root
4. Enables fast and efficient transaction verification
5. Ensures data integrity, security, and immutability

5. Working of Merkle Tree

A Merkle Tree is built in a **bottom-up manner** by repeatedly hashing pairs of nodes until a single hash remains.

Example:
Consider four transactions: T_1, T_2, T_3, T_4

Step 1: Compute individual hashes

- $H_1 = \text{Hash}(T_1)$
- $H_2 = \text{Hash}(T_2)$
- $H_3 = \text{Hash}(T_3)$
- $H_4 = \text{Hash}(T_4)$

Step 2: Form parent nodes

- $H_{12} = \text{Hash}(H_1 + H_2)$
- $H_{34} = \text{Hash}(H_3 + H_4)$

Step 3: Compute Merkle Root

- $H_{1234} = \text{Hash}(H_{12} + H_{34})$

The final hash H_{1234} is the **Merkle Root**.

6. Use of Merkle Tree in Blockchain

In distributed blockchain networks, every node maintains its own copy of the ledger. Without Merkle Trees, validating transactions would require transferring and comparing entire ledgers, which is inefficient and resource-intensive.

Merkle Trees solve this problem by:

- Allowing verification using only a small subset of hashes
- Reducing network bandwidth usage
- Eliminating the need to download full transaction data

Thus, Merkle Trees enable scalable, secure, and efficient blockchain operation.

Program and Output:

```
import hashlib

# 1. SHA-256 Hash Function
def generate_hash(message):
    return hashlib.sha256(message.encode()).hexdigest()

user_text = input("Enter a message to hash: ")
hash_result = generate_hash(user_text)
print("Generated Hash:", hash_result)
```

```
Enter a message to hash: Hello Ansh
Generated Hash: c35f50e7afce646b6211b5487fdae273159b01ce3d7cdd864aff00aa86e914c5
```

```python
#2. Execution: Hash with Nonce
def generate_hash_with_nonce(message, nonce):
    combined = message + str(nonce)
    return generate_hash(combined)

nonce_value = int(input("Enter a nonce value: "))
nonce_hash = generate_hash_with_nonce(user_text, nonce_value)
print("Hash with Nonce:", nonce_hash)
```

```
Enter a nonce value: 3
Hash with Nonce: d51352c6698752911f09fff6252c715e6271a5ee599605aeddff69e1659a3c7c
```

```python
# 3. Proof of Work
def proof_of_work(message, difficulty):
    print("\nMining started...")
    prefix = "0" * difficulty
    nonce = 0

    while True:
        new_hash = generate_hash_with_nonce(message, nonce)
        if new_hash.startswith(prefix):
            return nonce, new_hash
        nonce += 1

difficulty_level = int(input("Enter difficulty (number of leading zeros): "))
valid_nonce, valid_hash = proof_of_work(user_text, difficulty_level)

print("Nonce Found:", valid_nonce)
print("Valid Hash:", valid_hash)
```

```
Enter difficulty (number of leading zeros): 4

Mining started...
Nonce Found: 67746
Valid Hash: 00008d13c735d52906bf7e7ab2c8953e1cc80276305bae1d36baaebd506f0a8f
```

```python
# 4. Merkle Tree & Merkle Root
def calculate_merkle_root(transactions):
    hash_list = [generate_hash(tx) for tx in transactions]

    while len(hash_list) > 1:
        if len(hash_list) % 2 != 0:
            hash_list.append(hash_list[-1])
        temp_list = []
        for i in range(0, len(hash_list), 2):
            combined_hash = hash_list[i] + hash_list[i + 1]
            temp_list.append(generate_hash(combined_hash))
        hash_list = temp_list
    return hash_list[0]

print("Enter transactions (type 'stop' to finish):")
transactions = []
while True:
    tx = input()
    if tx.lower() == "stop":
        break
    transactions.append(tx)
if len(transactions) == 0:
    print("⚠️ No transactions entered.")
else:
    root_hash = calculate_merkle_root(transactions)
    print("Merkle Root Hash:")
    print(root_hash)
```

```
Enter transactions (type 'stop' to finish):
1
2
stop
Merkle Root Hash:
33b675636da5dcc86ec847b38c08fa49ff1cace9749931e0a5d4dfdbdedd808a
```

Conclusion

This experiment successfully demonstrated the foundational cryptographic concepts used in blockchain technology. SHA-256 hashing and Proof-of-Work illustrated how data integrity and consensus are achieved, while the Merkle Tree demonstrated efficient and secure verification of large transaction datasets. Together, these mechanisms ensure the security, immutability, and scalability of blockchain systems.