Name : Himesh Pathai D20A/39

Lab - 1: **Experiment No: 1**

AIM: Cryptography in Blockchain, Merkle root Tree Hash

Lab Objectives: To explore Blockchain concepts.

Lab Outcomes (LO)**:** Creating Cryptographic Hash using Merkle Tree (LO1)

Task to be performed :

1. Make a copy of this **Google Colab Notebook**
2. Try to solve the errors in each of the 4 Programs
3. In the 4th Program - Constructing a Merkle Tree Root Hash, modify the code as follows:
   - Update the transactions list with valid entries.
     eg : transactions = ['A -> B : $10', 'B -> C : $5,'C -> A : $2'] Sample Transactions to be considered
     T1 : Alice → Bob : $200; T2 : Bob → Dave : $500; T3 : Dave → Eve : $100 T4 : Eve → Alice : $300; T5 : Roo → Bob : $50
   - Hash the transactions before combining them in the for loop
   - Print all the intermediate hash during the construction of the Merkle Tree Root Hash
4. Upload your working code with Input & Output used for execution in the **Google Form**

Tools & Libraries used :

- Python Libraries: **hashlib**

Instructions : (Prepare for viva for the following topics)

1. Cryptographic Hash functions in Blockchain
2. What is a Merkle Tree?
3. What is a Cryptographic Puzzle and explain the Golden Nonce
4. How does a Merkle Tree work?
5. Benefits of Merkle Tree
6. Use cases of Merkle Tree

Outcome :

1. Understood the concept of hashing, Cryptographic Puzzle to find nonce, Merkle Tree and its relevance.
2. Implemented programs
   - To find the Golden Nonce while solving the Cryptographic Puzzle
   - To construct a Merkle Tree for the given transactions
3. Prepare a document with Aim, Tasks performed, Program, Output and Conclusion.
4. Submit the hardcopy **by the 1st week of Sem**
   (As per the instructions, submit a hard copy of the same).

Aim

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

Tasks Performed:

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

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 process.

3. 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.

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 Merkle Root hash for blockchain transaction integrity.

Theory

Cryptographic Hash Functions in Blockchain

Cryptographic hash functions are the foundation of blockchain security. A hash function takes an input of any size and produces a fixed-length output known as a hash value. In blockchain systems, SHA-256 (Secure Hash Algorithm) is commonly used due to its strong security properties such as determinism, collision resistance, and irreversibility. Even a small change in input data results in a completely different hash, ensuring data integrity and tamper detection. Hash functions link blocks together and protect transaction data from unauthorized modification.

Merkle Tree

A Merkle Tree (also known as a hash tree) is a data structure used in blockchain to efficiently organize and verify large sets of transactions. It allows multiple transaction hashes to be combined into a single hash called the Merkle Root. Merkle Trees play a crucial role in ensuring data integrity and efficient verification of transactions within a block.

Structure of a Merkle Tree

A Merkle Tree is a **binary tree structure** used to organize and verify data efficiently. Its structure consists of multiple levels:

1. **Leaf Nodes**
    - The bottom level of the tree
    - Each leaf node contains the **hash of a transaction or data block**

2. **Intermediate (Parent) Nodes**
    - Formed by **combining and hashing two child node hashes**
    - Each parent node stores the hash of its children

3. **Handling Odd Number of Nodes**
    - If there is an odd number of hashes at any level, the **last hash is duplicated**
    - This ensures every node has a pair for hashing

4. **Root Node (Merkle Root)**
    - The topmost node of the tree
    - Represents the **single hash value summarizing all transactions**

Merkle Root

The Merkle Root is the final hash obtained at the top of the Merkle Tree. It represents a single cryptographic summary of all transactions in a block. Any modification to a transaction changes its hash, which in turn changes all parent hashes up to the Merkle Root. Therefore, the Merkle Root guarantees the integrity and authenticity of all transactions in a block.

Working of Merkle Tree

The working of a Merkle Tree involves the following steps:

1. Each transaction is hashed using a cryptographic hash function.
2. Hashes are paired and concatenated.
3. The concatenated values are hashed again to form parent nodes.
4. This process repeats recursively until one hash remains.
5. The final hash obtained is the Merkle Root.

This hierarchical hashing enables efficient verification of transactions without needing to process all data.

Benefits of Merkle Tree

Merkle Trees offer several advantages:

- Efficient verification of large data sets
- Reduced storage requirements
- Fast transaction validation
- Improved data integrity and security
- Enables lightweight clients to verify transactions without downloading the entire blockchain

Uses of Merkle Tree in Blockchain

In blockchain systems, Merkle Trees are used to:

- Organize and validate transactions within a block
- Detect data tampering quickly
- Support Simplified Payment Verification (SPV)
- Reduce bandwidth and storage usage
- Maintain trustless verification in decentralized networks

Use Cases of Merkle Tree

Merkle Trees are widely used in:

- Bitcoin and other cryptocurrency blockchains
- Distributed systems for data verification
- Secure file systems
- Peer-to-peer networks
- Version control systems (e.g., Git)
- Data synchronization and integrity checking

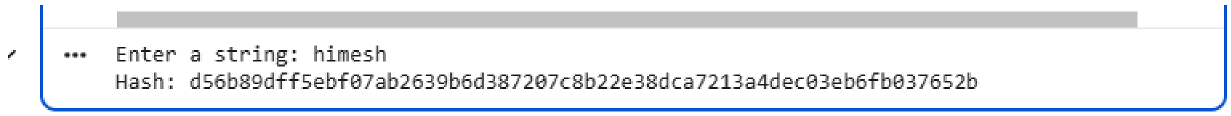
Code

Part 1: SHA-256 Hash Generation

```python
import hashlib

def create_hash(string):
    hash_object = hashlib.sha256()
    hash_object.update(string.encode('utf-8'))
    hash_string = hash_object.hexdigest()
    return hash_string
```

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

Output:

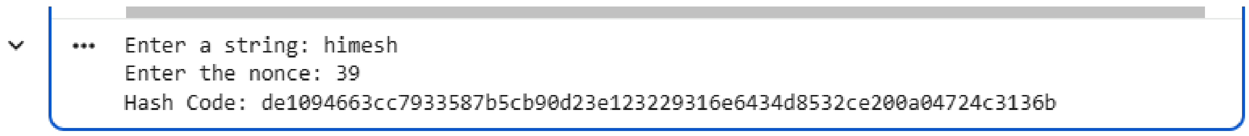


```
Enter a string: himesh
Hash: d56b89dff5ebf07ab2639b6d387207c8b22e38dca7213a4dec03eb6fb037652b
```

Part 2: Hash Generation with Nonce

```
import hashlib

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

hash_string = input_string + nonce

hash_object = hashlib.sha256(hash_string.encode('utf-8'))
hash_code = hash_object.hexdigest()

print("Hash Code:", hash_code)
```

Output:



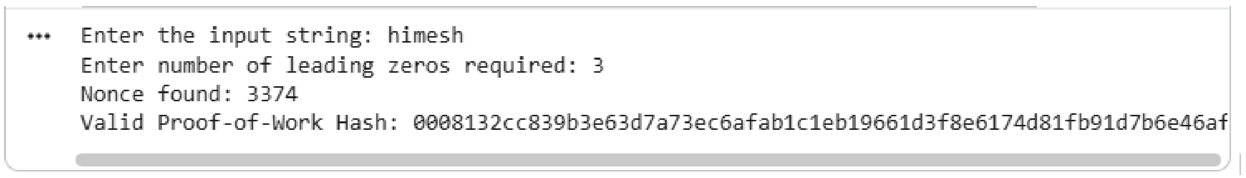
```
Enter a string: himesh
Enter the nonce: 39
Hash Code: de1094663cc7933587b5cb90d23e123229316e6434d8532ce200a04724c3136b
```

Part 3: Proof-of-Work Puzzle Solving

```
import hashlib

data = input("Enter the input string: ")
difficulty = int(input("Enter number of leading zeros required: "))
```

```
prefix = '0' * difficulty
nonce = 0

while True:
    text = data + str(nonce)
    hash_result = hashlib.sha256(text.encode()).hexdigest()
    if hash_result.startswith(prefix):
        break
    nonce += 1

print("Nonce found:", nonce)
print("Valid Proof-of-Work Hash:", hash_result)
```

Output:

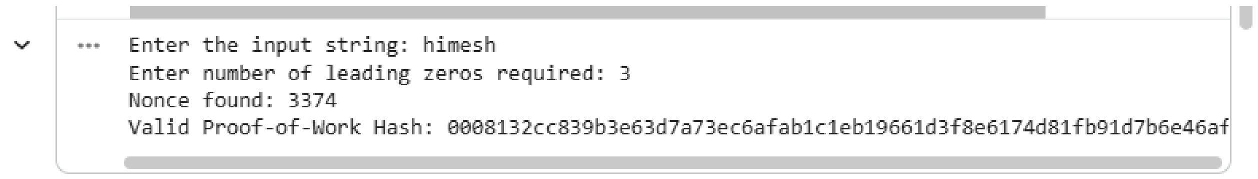


```
Enter the input string: himesh
Enter number of leading zeros required: 3
Nonce found: 3374
Valid Proof-of-Work Hash: 0008132cc839b3e63d7a73ec6afab1c1eb19661d3f8e6174d81fb91d7b6e46af
```

Part 4: Merkle Tree Construction

```
import hashlib

def sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()

def merkle_root(transactions):
    hashes = [sha256(tx) for tx in transactions]

    while len(hashes) > 1:
        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_level.append(sha256(combined_hash))

        hashes = new_level
    return hashes[0]

transactions = [
    "Himesh pays Bob 10 BTC",
    "Bob pays Prem 5 BTC",
    "Prem pays Dave 2 BTC",
    "Dave pays Eve 1 BTC"
]

print("Transactions:")
for tx in transactions:
    print(tx)
print("Merkle Root Hash:", merkle_root(transactions))
```

Output:



```
Enter the input string: himesh
Enter number of leading zeros required: 3
Nonce found: 3374
Valid Proof-of-Work Hash: 0008132cc839b3e63d7a73ec6afab1c1eb19661d3f8e6174d81fb91d7b6e46af
```

Conclusion

This study demonstrates fundamental cryptographic concepts and their applications within blockchain technology:

- **SHA-256 Hashing** provides a secure and unique fingerprint for any input, ensuring data integrity.
- **Nonce and Target Hashing** allow the simulation of proof-of-work consensus, emphasizing the difficulty of mining blocks by meeting hash conditions (leading zeros).
- **Proof-of-Work Puzzle Solving** shows how mining requires computational effort to find a nonce making the hash satisfy network difficulty.
- **Merkle Tree Construction** efficiently summarizes and validates large numbers of transactions with a single Merkle root hash, which is essential in blockchain for fast verification and ensuring the integrity of data blocks.

Together, these implementations capture the essence of how cryptography underpins blockchain's security, immutability, and efficiency, especially through hash functions and structures like Merkle Trees that enable trustless and decentralized transaction management.