Snehal Patil D20A 41

Blockchain Exp:1

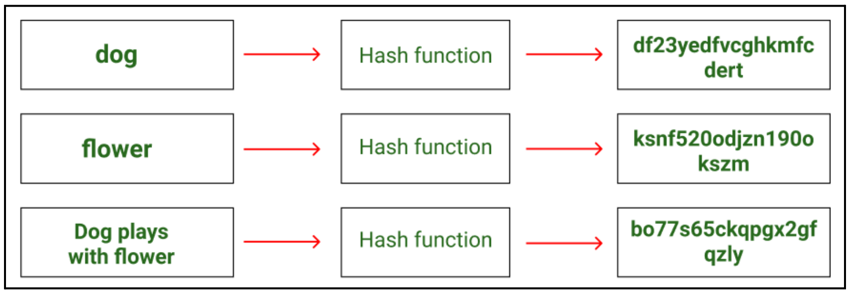
Aim: Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root tree hash

Theory:

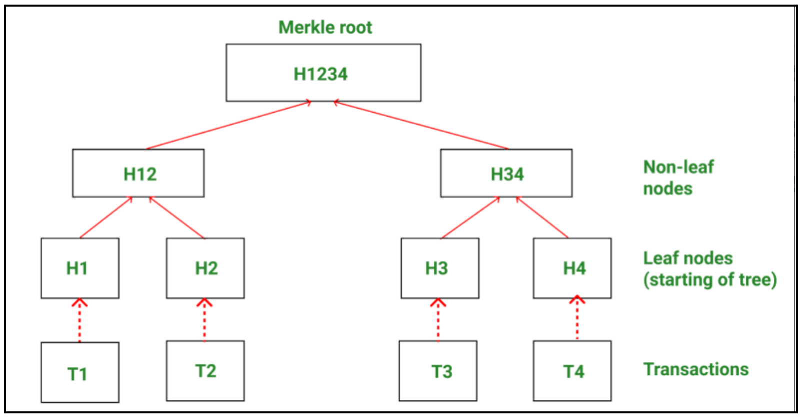
1. Cryptography

Cryptography is the science of securing information by transforming readable data (plaintext) into an unreadable format (ciphertext) using mathematical algorithms and keys to protect it from unauthorized access. It ensures secure communication by achieving key security goals such as confidentiality, integrity, authentication, and non-repudiation. Cryptography involves techniques like encryption and decryption, where data is encrypted before transmission and decrypted by authorized users at the receiving end. Cryptographic hash functions, such as SHA-256, generate fixed-length hash values that help verify data integrity and detect any unauthorized modifications. In modern technologies like blockchain, cryptography plays a crucial role in securing transactions, validating data, and maintaining a decentralized and tamper-proof system.

Cryptographic Hash:



2. Merkle Tree



A **Merkle Tree** is a cryptographic data structure used to efficiently verify the integrity and authenticity of large datasets. It is also known as a **hash tree**, where data is stored in the form of hashes rather than raw information. Merkle Trees use cryptographic hash functions such as SHA-256 to ensure data security and tamper detection.

3. Merkle Tree Structure

The structure of a Merkle Tree is a **binary tree** in which:

- Leaf nodes store the hash of individual data blocks or transactions.
- Internal nodes store the hash of the concatenation of their child nodes.
- The tree is built from the bottom up until a single root hash is obtained.

This hierarchical structure allows efficient verification of data integrity.

4. Merkle Root

The **Merkle Root** is the top-most hash value of a Merkle Tree. It represents the combined hash of all transactions or data blocks in the tree. Any modification in the underlying data causes a change in the Merkle Root, making it a reliable indicator of data integrity and consistency.

5. Working of Merkle Tree

The working of a Merkle Tree involves hashing individual data blocks, pairing and re-hashing them repeatedly until a single hash is produced. If the number of data blocks is odd, the last block is duplicated to maintain the tree structure. This process ensures that verification can be performed efficiently without accessing the entire dataset.

6. Benefits of Merkle Tree

Merkle Trees provide several important benefits:

- Efficient verification of large datasets
- Strong data integrity and tamper detection
- Reduced storage requirements

- Faster synchronization in distributed systems
- Secure data validation using cryptographic hashes

7. Use of Merkle Tree in Blockchain

In blockchain technology, Merkle Trees are used to store and verify transactions within a block. Each transaction is hashed and included in the tree, while the Merkle Root is stored in the block header. This allows blockchain nodes to verify individual transactions efficiently without downloading the entire block, improving scalability and performance.

8. Use Cases of Merkle Tree

Merkle Trees are widely used in various applications, including:

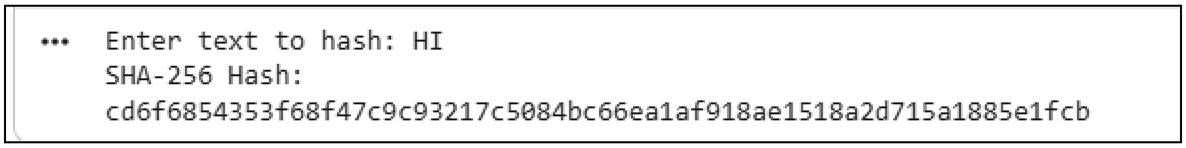
- Blockchain and cryptocurrency systems
- Distributed databases
- Peer-to-peer networks
- Version control systems
- Secure file verification
- Digital signature systems

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.

Code:

```python
import hashlib
# Take input from user
text = input("Enter text to hash: ")
# Encode the text to bytes
encoded_text = text.encode('utf-8')
# Create SHA-256 hash
sha256_hash = hashlib.sha256(encoded_text)
# Get hexadecimal representation
hash_result = sha256_hash.hexdigest()

print("SHA-256 Hash:")
print(hash_result)
```

Output:

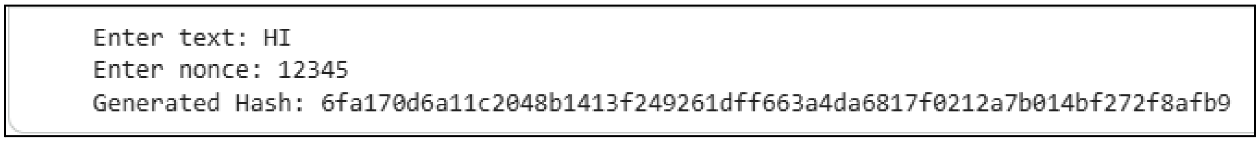


```
•••   Enter text to hash: HI
      SHA-256 Hash:
      cd6f6854353f68f47c9c93217c5084bc66ea1af918ae1518a2d715a1885e1fcb
```

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.

```
import hashlib

def hash_with_nonce(text, nonce):
    combined = text + str(nonce)
    return hashlib.sha256(combined.encode('utf-8')).hexdigest()

# Example
data = input("Enter text: ")
nonce = int(input("Enter nonce: "))

print("Generated Hash:", hash_with_nonce(data, nonce))
```

Output:

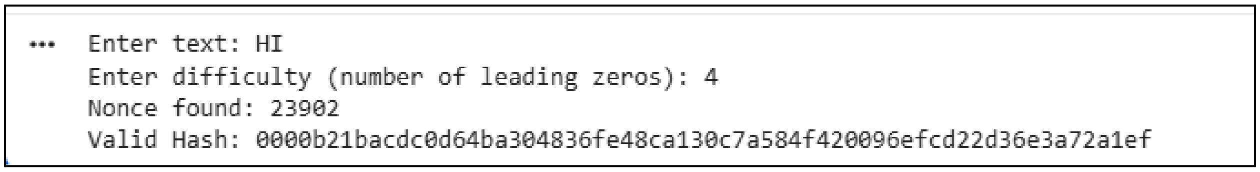


```
Enter text: HI
Enter nonce: 12345
Generated Hash: 6fa170d6a11c2048b1413f249261dff663a4da6817f0212a7b014bf272f8afb9
```

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.

```
import hashlib

def proof_of_work(text, difficulty):
    nonce = 0
    target = "0" * difficulty

    while True:
        combined = text + str(nonce)
        hash_result = hashlib.sha256(combined.encode('utf-8')).hexdigest()
        if hash_result.startswith(target):
            return nonce, hash_result
        nonce += 1

# Example
data = input("Enter text: ")
difficulty = int(input("Enter difficulty (number of leading zeros): "))

nonce, hash_result = proof_of_work(data, difficulty)
print("Nonce found:", nonce)
print("Valid Hash:", hash_result)
```

Output:



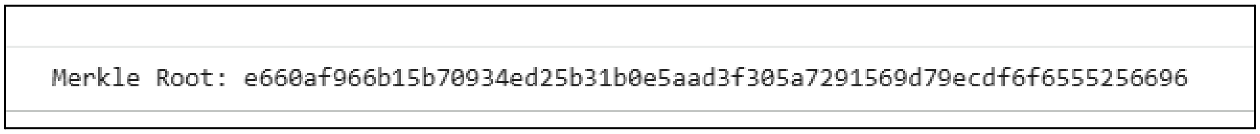
```
Enter text: HI
Enter difficulty (number of leading zeros): 4
Nonce found: 23902
Valid Hash: 0000b21bacdc0d64ba304836fe48ca130c7a584f420096efcd22d36e3a72a1ef
```

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.

Code:

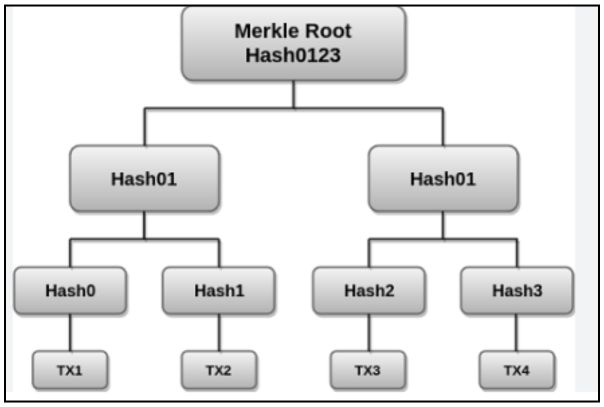
```
import hashlib

def hash_data(data):
    return hashlib.sha256(data.encode('utf-8')).hexdigest()

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

    if len(transactions) % 2 != 0:
        transactions.append(transactions[-1])  # duplicate last if odd

    new_level = []
    for i in range(0, len(transactions), 2):
        combined = transactions[i] + transactions[i + 1]
        new_level.append(hash_data(combined))

    return merkle_root(new_level)

# Example
transactions = [
    "Alice pays Bob",
    "Bob pays Charlie",
    "Charlie pays Dave",
    "Dave pays Eve"
]
hashed_transactions = [hash_data(tx) for tx in transactions]
root = merkle_root(hashed_transactions)

print("Merkle Root:", root)
```

Output:



```
Merkle Root: e660af966b15b70934ed25b31b0e5aad3f305a7291569d79ecdf6f6555256696
```

Merkle tree:



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.