Mohit Patil **D20A** **4**

Blockchain Experiment 1

<u>AIM:</u> Cryptography in Blockchain, Merkle root Tree Hash

<u>THEORY:</u>

Q1: Cryptographic Hash functions in Blockchain

Cryptographic hash functions are mathematical algorithms that convert input data of any size into a fixed-length string of characters, known as a hash value. These functions are fast, deterministic, and one-way, meaning that the original input cannot be retrieved from the hash.

Working of Cryptographic Hash Functions

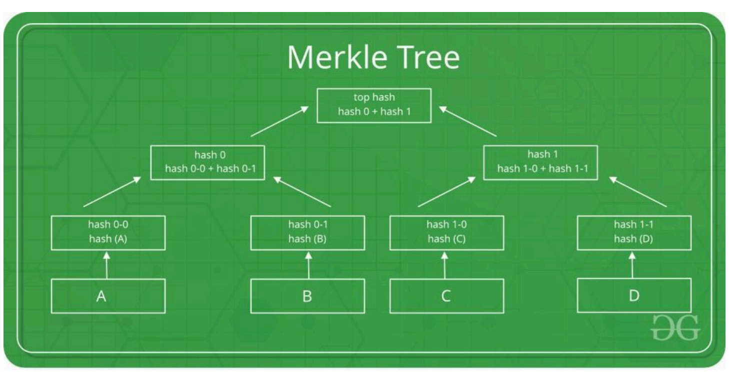
- **Input Processing**
  The function accepts input data of any length, such as text, files, or data streams, and processes it using mathematical operations.
- **Fixed-Size Output Generation**
  Regardless of input size, the output hash has a fixed length, usually represented as a hexadecimal string.
- **Deterministic Operation**
  The same input always produces the same hash, enabling reliable data verification.
- **Avalanche Effect**
  A minor change in input, even a single bit, results in a completely different hash output.
- **One-Way Computation**
  It is computationally infeasible to reverse the hash to obtain the original input.
- **Collision Resistance**
  The probability of two different inputs generating the same hash value is extremely low.

Popular Cryptographic Hash Algorithms

- MD5 (Message Digest Algorithm 5)
- SHA-1 (Secure Hash Algorithm 1)
- SHA-2 Family (SHA-256, SHA-512)
- SHA-3 (Keccak)
- BLAKE2 and BLAKE3

Q2: What is a Merkle Tree?

A Merkle Tree, also known as a Hash Tree, is a tree-based data structure in which each leaf node stores the hash of a data block, and each non-leaf node stores the hash of its child nodes. The final hash at the top of the tree is called the Merkle Root.



Structure of a Merkle Tree

- **Leaf Nodes:**
  Contain hash values of individual transactions or data blocks.
- **Internal Nodes:**
  Store hashes generated by combining and hashing child nodes.
- **Merkel Root:**
  The topmost node representing the combined hash of all underlying data.

Q3: What is a Cryptographic Puzzle and explain the Golden Nonce
Cryptographic Puzzle
A cryptographic puzzle is a problem that miners must solve to:

- Add a new block to the blockchain
- Prove computational work (Proof-of-Work)

The puzzle requires finding a hash that satisfies a difficulty condition.

Golden Nonce

- A nonce is a random number added to block data
- Miners repeatedly change the nonce and recompute the hash
- The Golden Nonce is the specific nonce value that produces a hash meeting the required difficulty (e.g., hash starts with a certain number of zeros)

Q4: How does a Merkle Tree work?

Step-by-Step Working

a. Each transaction is hashed using a cryptographic hash function
b. Hashes are paired and combined
c. Combined hashes are hashed again
d. If the number of hashes is odd, the last hash is duplicated
e. This process continues until one hash remains
f. The final hash is the Merkle Root

Q5: Benefits of Merkle Tree

- Reduces the amount of data required for verification
- Improves data integrity and security
- Enables fast synchronization between nodes
- Suitable for large-scale systems

Q6: Use cases of Merkle Tree

- **Blockchain Technology**
  Used to store and verify transactions in a block.
- **Distributed Systems**
  Ensures consistency of data across nodes.
- **Version Control Systems**
  Used in systems like Git for tracking file changes.

- **Database Verification**
  Detects unauthorized data modification.

<u>TASKS PERFORMED:</u>

Task 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.

```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)
```

Output:

```
...   Enter a string: Mohit
      Enter the nonce: 1
      Hash Code: c6c672adb6cd1e798e2d03306ea3fe2f&cd2b35eda
                 23a02ca38a64dd6660bc58a6
```

Task 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.

```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)
```

Output:

```
...   Enter a string: Mohit
      Enter the nonce: 1
      Hash Code: c6c672adb6cd1e798e2d03306ea3fe2f&cd2b35eda
```

Task 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 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 = "Mohit"
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)
```

```
...  Input String: Mohit
     Leading Zeros: 1
     Expected Nonce: 3
     Hash: 0a74ef1bca
     5574860bcd141665468433f64C3c231fabfeded22df8bfd2d8a7f196e5ab39
```

Task 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.

```
import hashlib
def build_merkle_tree(transactions):
    if len(transactions) == 0:
        return None
    # Hash each transaction initially
    hashed_transactions = [
        hashlib.sha256(t.encode('utf-8')).hexdigest()
        for t in transactions
    ]
    print("Initial Hashed Transactions:", hashed_transactions)
    if len(hashed_transactions) == 1:
        return hashed_transactions[0]
    # Recursive construction of the Merkle Tree
    current_level = hashed_transactions
    level = 1
    while len(current_level) > 1:
        print(f"\nLevel {level} Hashes:")
        # If odd number of nodes, duplicate last
        if len(current_level) % 2 != 0:
            current_level.append(current_level[-1])
        new_level = []

        for i in range(0, len(current_level), 2):
            combined = current_level[i] + current_level[i + 1]
            hash_combined = hashlib.sha256(
                combined.encode('utf-8')
            ).hexdigest()
            new_level.append(hash_combined)
            print(
                f"  Combining {current_level[i][:6]}... and "
```

```python
                f"{current_level[i+1][:6]}... to get
{hash_combined[:6]}..."
            )
        current_level = new_level
        level += 1
    return current_level[0]

# Example usage
transactions = [
    "Anuprita -> Sneha : 800",
    "Sneha -> Shreya : 540",
    "Shreya -> Ria : 100",
    "Eesha -> Anuprita : 500",
    "Ria -> Sneha : 680"
]

print("\nMerkle Tree Generation")
merkle_root = build_merkle_tree(transactions)

print("\nMerkle Root:", merkle_root)
```

```
...
Merkle Tree Generation
Initial Hashed Transactions: ['4d5a82f66fccc5af28031a82a5ceb39dc3fe23097c56c8dff7720841a02aef1f', 'fd0b9eafb0c7f5403

Level 1 Hashes:
  Combining 4d5a82... and fd0b9e... to get 9cd41e...
  Combining 43dd57... and 2ed5c8... to get 9f37c0...
  Combining 4313e3... and 4313e3... to get bc2172...

Level 2 Hashes:
  Combining 9cd41e... and 9f37c0... to get 6fa7f1...
  Combining bc2172... and bc2172... to get ba86e9...

Level 3 Hashes:
  Combining 6fa7f1... and ba86e9... to get c5db5f...

Merkle Root: c5db5fb4092cf71773baf667101b0158168f44d9a99579c5385353c80ef38aff
```

CONCLUSION:

In this experiment, cryptographic hash functions and Merkle Tree construction were successfully implemented using Python. SHA-256 hashing ensured data integrity, while nonce-based Proof-of-Work and Merkle Root generation demonstrated secure and tamper-resistant transaction verification in blockchain systems.