Ronak Katariya
D20A / 29

Exp 1

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:

1. Cryptographic Hashing

Cryptographic hashing is a process of converting an input (message, file, or data) of any size into a **fixed-length string**, called a **hash value** or **digest**, using a **hash function**.

Characteristics of Cryptographic Hash Functions

1. **Deterministic** – Same input always produces the same hash.
2. **Fixed output length** – Regardless of input size.
3. **Fast computation** – Efficient to compute hash.
4. **Pre-image resistance** – Hard to find original input from hash.
5. **Collision resistance** – Hard to find two different inputs with the same hash.
6. **Avalanche effect** – Small change in input results in a large change in hash.

Examples of Hash Functions

- SHA-256 (used in Bitcoin)
- SHA-1
- MD5 (not secure now)

Purpose of Cryptographic Hashing

- Data integrity verification
- Digital signatures
- Password storage
- Blockchain security

2. Merkle Tree (also called Hash Tree)

A **Merkle Tree** is a **binary tree data structure** in which:

- **Leaf nodes** store hashes of actual data blocks.

- **Non-leaf (parent) nodes** store hashes of their child nodes.
- The **topmost hash** is called the **Merkle Root**.

It is widely used in **blockchain and distributed systems** to ensure **data integrity and efficient verification**.

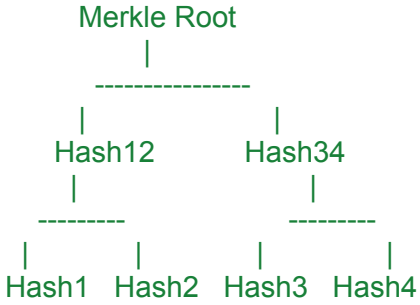
Components of a Merkle Tree

1. **Leaf Nodes**
   - Contain hash of individual data blocks.
   - Example:
     H1 = Hash(Data1)

2. **Intermediate (Internal) Nodes**
   - Store hash of concatenation of child hashes.
   - Example:
     H12 = Hash(H1 + H2)

3. **Merkle Root**
   - The final top hash representing all data.

Example Structure



4. Merkle Root

The **Merkle Root** is the **topmost hash** of a Merkle Tree.
Key Points

- It uniquely represents all transactions or data blocks.
- Any change in even **one transaction** changes the Merkle Root.
- Stored in the **block header** in blockchain.

Importance

- Ensures data integrity.
- Enables quick verification of data.

5. Working of Merkle Tree

Step-by-Step Working

1. **Divide data** into blocks (transactions).
2. **Hash each data block** to form leaf nodes.
3. **Pair adjacent hashes** and hash them together.

4. Repeat until **one hash remains** (Merkle Root).
5. If odd number of nodes, the last hash is **duplicated**.

Example

- Data: D1, D2, D3, D4
- Leaf hashes: H1, H2, H3, H4
- Parent hashes:
  H12 = Hash(H1 + H2)
  H34 = Hash(H3 + H4)
- Merkle Root:
  Hash(H12 + H34)

6. Benefits of Merkle Tree

1. **Efficient verification**
   - Only few hashes needed to verify a transaction.
2. **Data integrity**
   - Any modification is easily detected.
3. **Reduced storage**
   - No need to store all data for verification.
4. **Scalability**
   - Works efficiently with large datasets.
5. **Secure**
   - Based on cryptographic hash functions.

7. Use of Merkle Tree in Blockchain

Merkle Trees are used to store **transactions inside a block**.
Role in Blockchain

- All transactions are hashed and organized into a Merkle Tree.
- The **Merkle Root** is stored in the **block header**.
- Helps in **transaction verification** without downloading full block.

In Bitcoin

- Uses **SHA-256** hashing.
- Supports **Simplified Payment Verification (SPV)** for light nodes.

8. Use Cases of Merkle Tree

1. Blockchain Systems
   - Bitcoin, Ethereum
   - Transaction validation and integrity

2. Distributed Systems
   - Data synchronization

Ronak Katariya
D20A / 29

- File integrity verification

3. Databases
   - Efficient consistency checking

4. Version Control Systems
   - Git uses Merkle trees for tracking file changes

5. Cloud Storage
   - Ensuring stored data integrity

Code:

Hash Generation using SHA-256:

```
import hashlib

data_string = 'Hello, Blockchain! Ronak Here'
encoded_data = data_string.encode('utf-8')
sha256_hash = hashlib.sha256()
sha256_hash.update(encoded_data)
hex_digest = sha256_hash.hexdigest()

print(f"Original string: '{data_string}'")
print(f"SHA-256 Hash: {hex_digest}")
```

Output:

```
Original string: 'Hello, Blockchain! Ronak Here'
SHA-256 Hash: 9e2e7a60bcd2026881f3fb9b69be214879a2cf6f775009d4adde56010d59df7a
```

Target Hash Generation with Nonce:
Code:

Ronak Katariya
D20A / 29

```python
import hashlib

def generate_hash_with_nonce(data, nonce):
    """
    Generates a SHA-256 hash by concatenating data and a nonce.
    """
    # Convert data and nonce to string, then encode to bytes
    combined_data = str(data) + str(nonce)
    encoded_data = combined_data.encode('utf-8')
    return hashlib.sha256(encoded_data).hexdigest()

# Example usage:
user_data = 'Hello, Blockchain! Ronak Here'
example_nonce = 12345

hash_output = generate_hash_with_nonce(user_data, example_nonce)

print(f"Original Data: '{user_data}'")
print(f"Nonce: {example_nonce}")
print(f"Combined String (data + nonce): '{user_data}{example_nonce}'")
print(f"Generated Hash: {hash_output}")
```

```
Original Data: 'Hello, Blockchain! Ronak Here'
Nonce: 12345
Combined String (data + nonce): 'Hello, Blockchain! Ronak Here12345'
Generated Hash: aa923f358d76e0e4a3fee0a3cd04271cc835ebff2692a77f13e9061e0653df68
```

Proof-of-Work Puzzle Solving:
Code:

```
import hashlib
import time

def proof_of_work(data, difficulty):
    """
     Finds a nonce such that the SHA-256 hash of (data + nonce) starts with 'difficulty' leading zeros.
    """
    nonce = 0
    prefix = '0' * difficulty
    print(f"Searching for a hash starting with '{prefix}'...")

    start_time = time.time()
    while True:
        combined_data = str(data) + str(nonce)
        current_hash = hashlib.sha256(combined_data.encode('utf-8')).hexdigest()

        if current_hash.startswith(prefix):
```

```
            end_time = time.time()
            elapsed_time = end_time - start_time
            print(f"\nProof of Work found in {elapsed_time:.4f} seconds!")
            return nonce, current_hash

        nonce += 1
        # Optional: Print progress for long computations
        # if nonce % 100000 == 0:
        #     print(f"Tried {nonce} nonces, current hash: {current_hash}")

# Example Usage:
block_data = "This is some block data for the PoW puzzle."
difficulty_level = 4  # Number of leading zeros required

print(f"Block Data: '{block_data}'")
print(f"Difficulty (leading zeros): {difficulty_level}")

found_nonce, final_hash = proof_of_work(block_data, difficulty_level)

print(f"Nonce found: {found_nonce}")
print(f"Final Hash: {final_hash}")
print(f"Verification: Does hash start with '{'0' * difficulty_level}'? {final_hash.startswith('0' * difficulty_level)}")
```

Output:

```
Block Data: 'This is some block data for the PoW puzzle.'
Difficulty (leading zeros): 4
Searching for a hash starting with '0000'...

Proof of Work found in 0.0300 seconds!
Nonce found: 11481
Final Hash: 00003e17edd1474c4caa2304dcbd1e6b6ec4ad06f48709d9691425d51171cde9
Verification: Does hash start with '0000'? True
```

Merkle Tree Construction:
Code:

```
import hashlib

def hash_data(data):
    """
    Hashes input data using SHA-256.
    """
    return hashlib.sha256(str(data).encode('utf-8')).hexdigest()

def build_merkle_tree(transactions):
    """
    Constructs a Merkle tree from a list of transactions and returns the Merkle root.
    """
    leaf_nodes = [hash_data(tx) for tx in transactions]
```

Ronak Katariya
D20A / 29

```
    if not leaf_nodes:
        return ""
    current_level = list(leaf_nodes)
    while len(current_level) > 1:
        next_level = []

        if len(current_level) % 2 != 0:
            current_level.append(current_level[-1])

        for i in range(0, len(current_level), 2):
            hash1 = current_level[i]
            hash2 = current_level[i+1]
            combined_hash = hash_data(hash1 + hash2)
            next_level.append(combined_hash)
        current_level = next_level
    return current_level[0]

transactions = ['Alice sends 10 BTC to Bob', 'Bob sends 5 BTC to Carol', 'Carol sends 2 BTC to
David', 'David sends 1 BTC to Eve']

merkle_root = build_merkle_tree(transactions)

print(f"Sample Transactions: {transactions}")
print(f"Calculated Merkle Root: {merkle_root}")
```

Output:

```
Sample Transactions: ['Alice sends 10 BTC to Bob', 'Bob sends 5 BTC to Carol', 'Carol sends 2 BTC to David', 'David sends 1 BTC to Eve']
Calculated Merkle Root: 877a67e259432f93d0601d4f6c199f860d18d0a9eb7fd999633a29a2a2b8ad31
```

Conclusion:

Cryptographic hashing ensures data integrity by converting data into fixed-length hash values, while Merkle Trees organize these hashes in a hierarchical structure for efficient and secure verification. The Merkle Root represents the integrity of the entire dataset, and any change in data results in a different root hash. Merkle Trees reduce storage and verification overhead by allowing partial verification instead of checking all data. Due to these advantages, they are widely used in blockchain technology for secure, scalable, and reliable transaction validation.