Blockchain Exp - 1

Prajjwal Pandey D20A / Batch A / 37

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

Theory: SHA and Merkle Tree in Blockchain

1. Cryptographic Hash Functions in Blockchain

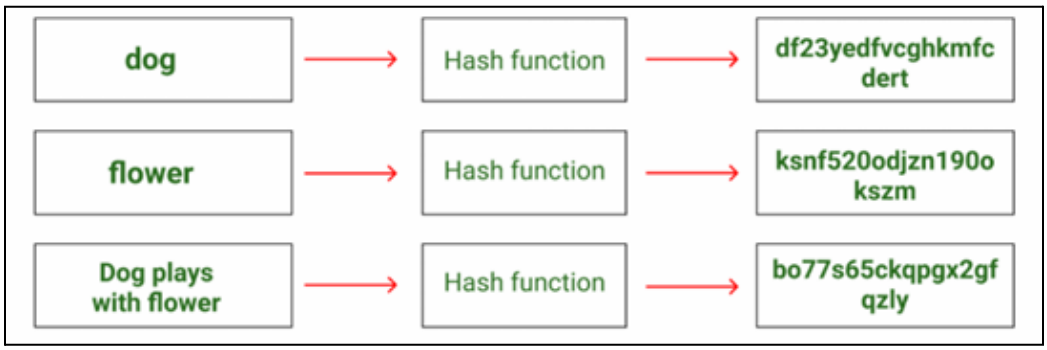
A cryptographic hash function is a fundamental building block of blockchain technology. It accepts input data of arbitrary size and produces a fixed-size output known as a **hash digest**. These functions are designed to be fast to compute but extremely hard to reverse or manipulate.

For a hash function to be secure and effective in blockchain systems, it must satisfy the following properties:

- **Collision Resistance**: It should be computationally infeasible to find two different inputs that generate the same hash value.
- **Preimage Resistance**: Given a hash output, it must be practically impossible to determine the original input.
- **Second Preimage Resistance**: Given an input and its hash, it should be difficult to find another input that produces the same hash.
- **Deterministic Nature**: The same input must always produce the same hash output.
- **Avalanche Effect**: A small change in the input should cause a drastic and unpredictable change in the output hash.
- **Fixed Output Length**: Regardless of input size, the hash output length remains constant.
- **Computational Infeasibility**: Guessing the output without computing the hash is practically impossible.

Role in Blockchain:

- Ensures immutability of transaction data
- Links blocks together through hash pointers
- Used in Proof-of-Work mechanisms
- Forms the basis of Merkle Trees and digital signatures

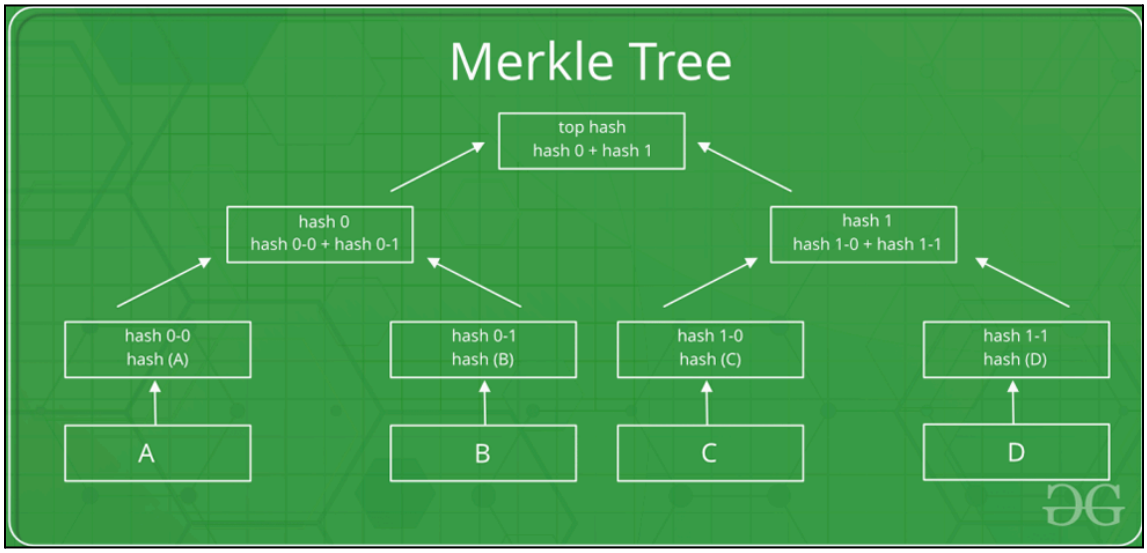


2. Merkle Tree (Hash Tree)

A Merkle Tree, also known as a **hash tree**, is a hierarchical data structure that enables efficient and secure verification of large datasets. It is constructed by recursively hashing pairs of data until a single hash value remains.

Each **leaf node** represents the hash of a data block (transaction), while each **internal node** stores the hash of its child nodes. The final hash at the top is called the **Merkle Root**, representing the integrity of the entire dataset.

Merkle Trees are widely used in blockchains to verify transactions efficiently without requiring access to all stored data.

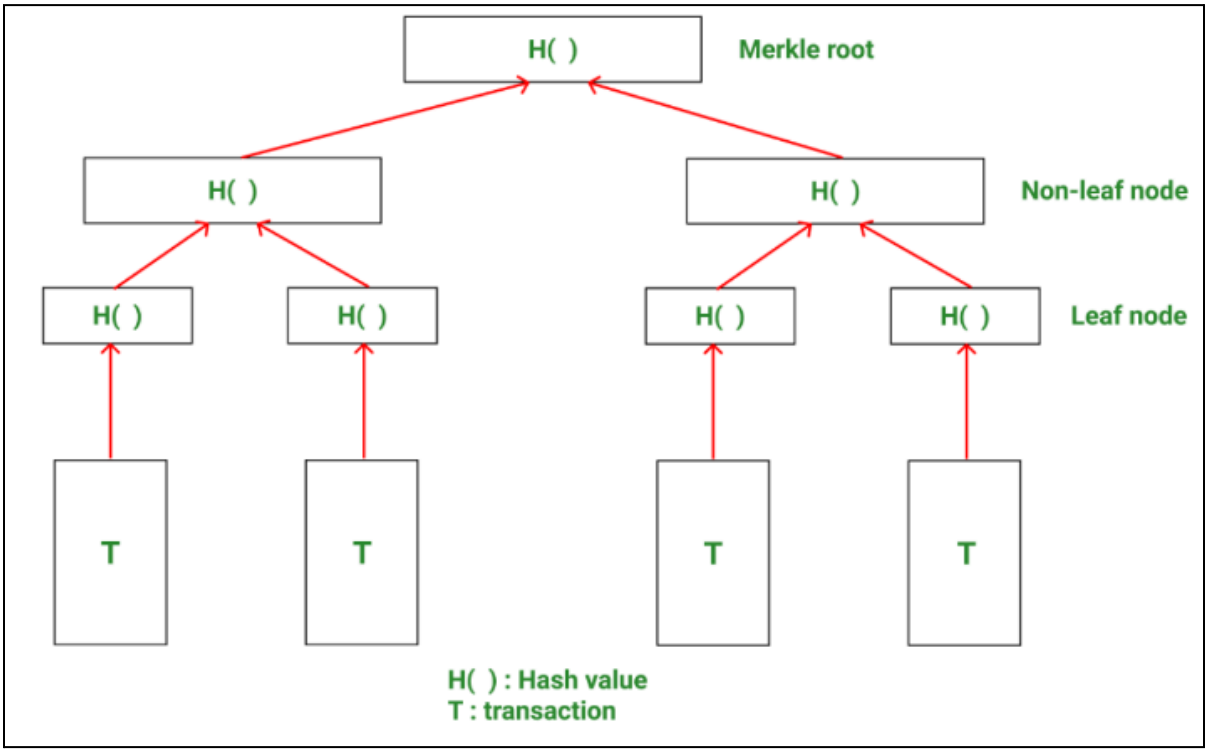


3. Structure of a Merkle Tree

A Merkle Tree consists of the following components:

- **Leaf Nodes**: Store the cryptographic hashes of individual transactions or data blocks.
- **Intermediate Nodes**: Generated by concatenating and hashing the hashes of child nodes.
- **Merkle Root**: The topmost node that acts as a unique digital fingerprint for all underlying data.

If even a single transaction changes, the resulting change propagates upward, altering the Merkle Root and indicating data tampering.



4. Merkle Root

The Merkle Root is the final hash produced after repeatedly hashing all transaction pairs in a Merkle Tree. It compactly represents the entire set of transactions in a block.

Importance of Merkle Root:

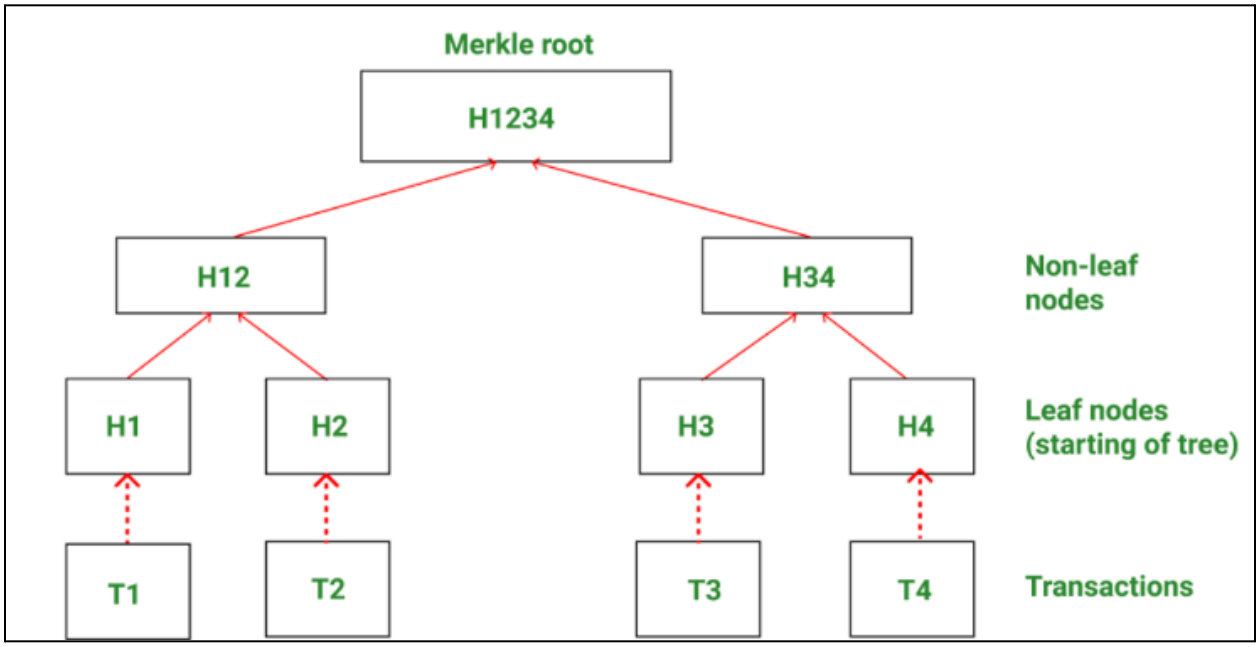
1. Acts as a single integrity check for all transactions
2. Stored in the block header instead of full transaction data
3. Any modification in a transaction changes the Merkle Root
4. Enables fast verification using Merkle Proofs
5. Strengthens security and immutability of the blockchain

5. Working of a Merkle Tree

The construction of a Merkle Tree follows a **bottom-up approach**:

1. Each transaction is hashed individually.
2. Hashes are grouped in pairs and concatenated.
3. The concatenated values are hashed again to form parent nodes.
4. This process repeats until only one hash remains — the Merkle Root.

If the number of nodes at any level is odd, the last hash is duplicated to maintain a complete binary structure.



Example: Consider a block having 4 transactions- T1, T2, T3, T4. These four transactions have to be stored in the Merkle tree and this is done by the following steps

Step 1: The hash of each transaction is computed.

H1 = Hash(T1)

Step 2: The hashes computed are stored in leaf nodes of the Merkle tree.

Step 3: Now non-leaf nodes will be formed. In order to form these nodes, leaf nodes will be paired together from left to right, and the hash of these pairs will be calculated. Firstly hash of H1 and H2 will be computed to form H12. Similarly, H34 is computed. Values H12 and H34 are parent nodes of H1, H2, and H3, H4 respectively. These are non-leaf nodes.

H12 = Hash(H1 + H2)

H34 = Hash(H3 + H4)

Step 4: Finally H1234 is computed by pairing H12 and H34. H1234 is the only hash remaining.

This means we have reached the root node and therefore H1234 is the Merkle root.

H1234 = Hash(H12 + H34)

6. Advantages of Merkle Trees

- **Efficient Verification**: Requires minimal data to verify transaction authenticity
- **Reduced Bandwidth Usage**: Only hash values are transmitted instead of full data
- **Tamper Detection**: Even minor data changes are immediately detectable
- **Storage Optimization**: Occupies less space than storing full datasets
- **Supports Trustless Systems**: Enables verification without centralized authority

7. Role of Merkle Trees in Blockchain

In decentralized systems like blockchain, each node maintains a copy of the ledger. Without Merkle Trees, validating transactions would require sharing entire datasets between nodes.

Merkle Trees solve this problem by:

- Allowing verification using small Merkle Proofs
- Reducing computation and communication overhead
- Enabling nodes to validate transactions independently
- Improving scalability and efficiency of blockchain networks

8. Applications of Merkle Trees

1. **Blockchain Light Clients (SPV)**
   Enables mobile wallets to verify transactions using only block headers and Merkle proofs.
2. **Version Control Systems (Git)**
   Tracks file changes efficiently by hashing directory structures.
3. **Distributed Databases**
   Used in systems like Cassandra for data synchronization and repair.

4. **Peer-to-Peer File Sharing**
   Ensures integrity of downloaded file chunks in BitTorrent and IPFS.
5. **Certificate Transparency**
   Verifies SSL certificates without downloading massive certificate logs.

PROGRAMS & OUTPUT :

Program 1: Python program that uses the hashlib library to create the hash of a given string:

```
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)
```

```
Enter a string: prajjwal
Hash: b2d3977033871cb96477ac0cdc4bee8442df9e14eed9c68a9174ee43cac8267c
```

Program 2: Program to generate required target hash with input string and nonce

```
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)
```

```
Enter a string: PRAJJWAL
Enter the nonce: 1
Hash Code: 3919c66c5bbe2c20311e2324140351f413827d4aabae929c5c6a038927c084d9
```

Program 3: Python code for Solving Puzzle for leading zeros with expected nonce and given string

```
import hashlib

def find_nonce(input_string, num_zeros):
    nonce = 9
    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 = "A"
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)
```

```
Hash: 06663975f75f189f1d70bd14d8f22df264cd6a5c7575d6875183d0ec76432fbb
Input String: A
Leading Zeros: 1
Expected Nonce: 9
```

Program 4: Generating Merkle Tree for given set of Transactions

```python
import hashlib

def build_merkle_tree(transactions):
    if len(transactions) == 0:
        return None

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

    # Recursive construction of the Merkle Tree
    while len(transactions) > 1:
        if len(transactions) % 2 != 0:
            transactions.append(transactions[-1])

        new_transactions = []
        for i in range(0, len(transactions), 2):
            combined = transactions[i] + transactions[i+1]
            hash_combined = hashlib.sha256(combined.encode('utf-8')).hexdigest()
            new_transactions.append(hash_combined)

        transactions = new_transactions

    return transactions[0]

# Example usage
transactions = ["Transaction 1", "Transaction 2", "Transaction 3", "Transaction 4", "Transaction 5"]

merkle_root = build_merkle_tree(transactions)
print("Merkle Root:", merkle_root)
```

```
Merkle Root: a4a18941de1162b17a46c4f8c87d8a0850b46fad17ac881340061d9233785077
```

Conclusion

This experiment demonstrates how cryptographic hash functions and Merkle Trees form the backbone of blockchain security. Hashing ensures data integrity and immutability, while Merkle Trees enable efficient verification of large transaction sets. Together, these mechanisms make blockchain systems secure, scalable, and resistant to tampering, reinforcing trust in decentralized networks.