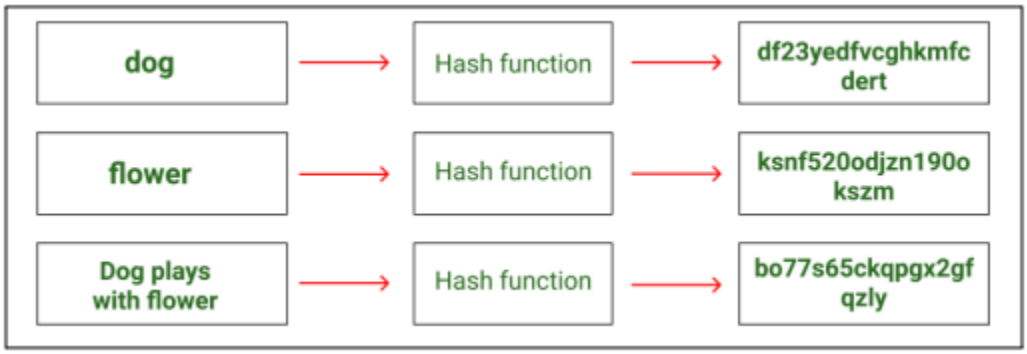
Blockchain Lab Experiment 01

Aim: To develop Python programs to understand SHA-based cryptography used in blockchain technology and to study the construction and working of Merkle Root hash trees.

Theory:

Cryptographic Hash Function in Blockchain

A cryptographic hash function plays a crucial role in ensuring security within blockchain systems. It is a one-way mathematical function that converts input data of any length into a fixed-size string known as a hash. This hash uniquely represents the original input data.



Key Properties:

- Deterministic – The same input always results in the same hash output
- One-way function – It is computationally infeasible to retrieve the original data from the hash
- Collision resistant – Two different inputs should not produce identical hashes
- Efficient computation – Hash generation is fast
- Avalanche effect – A minor input change causes a significant change in hash output

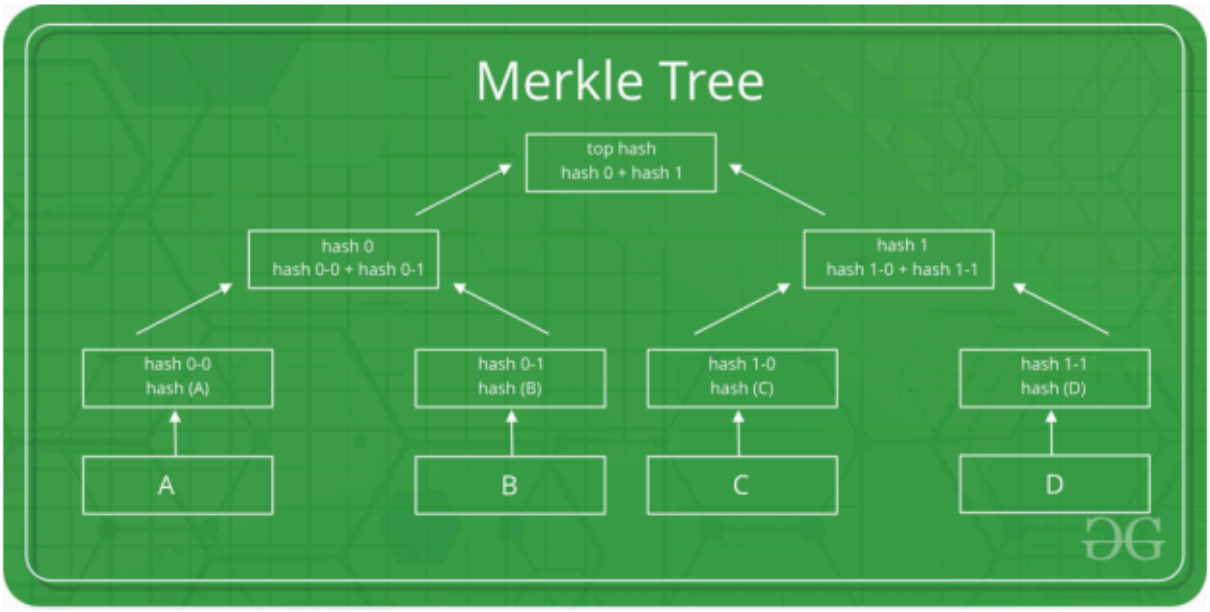
Application in Blockchain:

- Maintains integrity of transaction data
- Securely links blocks using hash pointers
- Forms the basis of Merkle Trees, block headers, and digital signatures

Thus, hash functions act as the foundation for blockchain security.

Merkle Tree

A Merkle Tree, also referred to as a hash tree, is a hierarchical data structure used to verify large datasets efficiently. It uses cryptographic hashes to summarize and validate data.



Description:

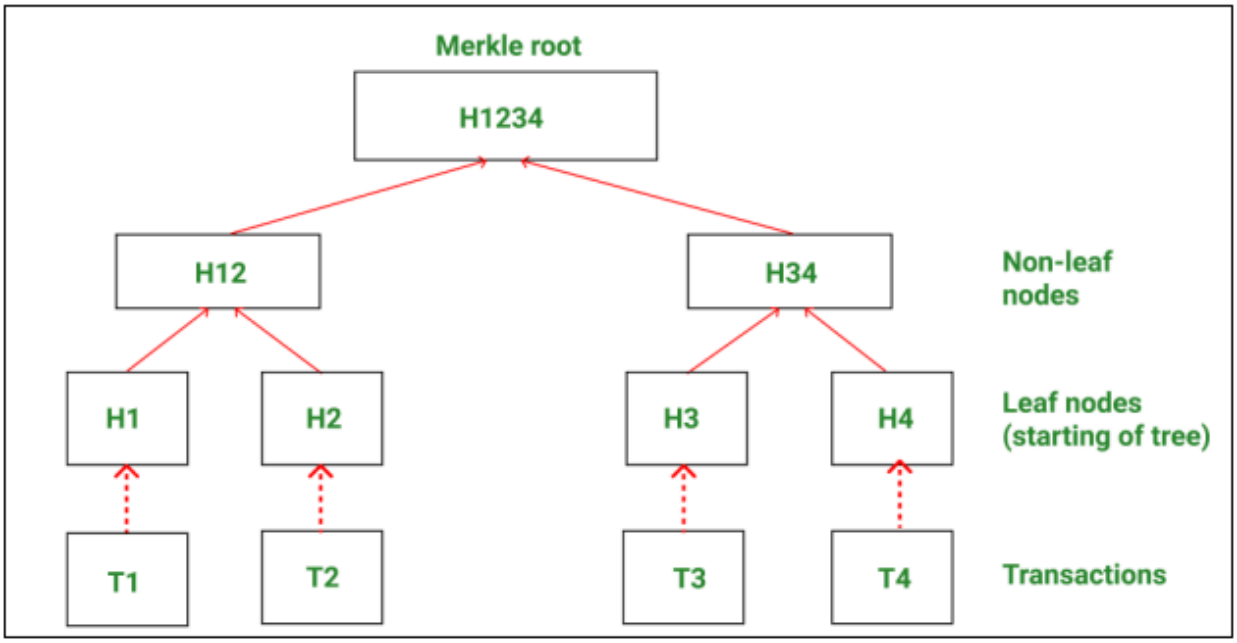
- Transaction hashes are stored in leaf nodes
- Parent nodes store hashes of combined child nodes
- The top node is called the Merkle Root

Purpose:

- Verifies transaction integrity
- Enables faster validation
- Reduces data transfer and storage requirements

Merkle Trees enhance both performance and security in distributed networks.

Structure of Merkle Tree



Blockchain networks handle massive transaction volumes, which require optimized memory usage and processing efficiency. Merkle Trees address this challenge effectively.

Working Structure:

- Transactions are paired and hashed
- Each parent node stores the hash of its child nodes
- Hashing continues upward until a single root hash is produced

Node Types:

- Root Node – Stores the Merkle Root in the block header
- Leaf Nodes – Store hashes of individual transactions
- Intermediate Nodes – Store combined hashes during tree construction

Bitcoin uses the SHA-256 algorithm to generate hashes throughout the Merkle Tree. If the number of transactions is odd, the final hash is duplicated to maintain a balanced binary tree.

Merkle Root

The Merkle Root is the final hash obtained from the Merkle Tree and represents all transactions in a block.

Explanation

- Individual transactions are hashed
- Hashes are paired and repeatedly re-hashed
- The final hash at the top is called the Merkle Root

Significance

- Acts as a digital fingerprint of the block's transactions
- Stored in the block header instead of full transaction data
- Any change in transaction data alters the Merkle Root
- Enables efficient transaction verification

Role in Blockchain

- Used by miners for block validation
- Supports Simplified Payment Verification (SPV)
- Ensures immutability and integrity

Working of Merkle Tree

Merkle Trees operate using layered hashing.

Steps:

1. Each transaction is hashed
2. Hashes are paired and concatenated
3. Combined hashes are re-hashed
4. Process continues until one hash remains
5. Final hash represents the Merkle Root
6. Last hash is duplicated if transaction count is odd

This method allows verification without processing the entire dataset.

Benefits of Merkle Tree

- Faster and efficient data verification
- Reduced memory and bandwidth usage
- Minimal data transfer for validation
- Detects data tampering quickly
- Supports trustless blockchain operations

Use of Merkle Tree in Blockchain

Without Merkle Trees, each blockchain node would need to store and compare full transaction records. This would consume excessive bandwidth and processing power.

Merkle Trees simplify verification by:

- Allowing validation using hash values only
- Minimizing data transmission
- Reducing computational overhead

Applications of Merkle Tree

1. Blockchain and Cryptocurrencies
   - Organizes transactions
   - Generates Merkle Root
   - Ensures secure and immutable records

2. Distributed File Systems (IPFS)
   - Verifies file integrity
   - Detects data corruption
   - Enables decentralized storage

3. Peer-to-Peer Networks
   - Validates shared data
   - Prevents malicious modifications

4. Version Control Systems (Git)
   - Tracks file changes
   - Ensures history integrity

5. Cloud Storage and Secure Databases
   - Validates large datasets
   - Enables efficient audits

CODE:

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

```
import hashlib

# Take input from user
input_string = input("Enter a string to hash: ")

# Generate SHA-256 hash
sha256_hash = hashlib.sha256(input_string.encode())

# Display the hash value
print("SHA-256 Hash:", sha256_hash.hexdigest())
```

```
Enter a string to hash: Sannidhi
SHA-256 Hash: 2f4002a07f5420bdd12ee6b8adb952e2aa30cf9175415531da2f662d2a5607f2
```

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

# Take input from user
input_string = input("Enter the data string: ")
nonce = input("Enter nonce value: ")

# Concatenate input string and nonce
combined_data = input_string + nonce

# Generate SHA-256 hash
hash_result = hashlib.sha256(combined_data.encode())

# Display the hash
print("Generated Hash:", hash_result.hexdigest())
```

```
Enter the data string: Sannidhi
Enter nonce value: 4
Generated Hash: d3b11dd6405d97a1425f42bebc63adc1cefbcf0e8b53edd00211aa4be4445d8c
```

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.

```python
import hashlib
# Take input from user
data = input("Enter data string: ")
difficulty = int(input("Enter difficulty level (number of leading zeros):
"))

nonce = 0
target = '0' * difficulty

print("Mining started...")

while True:
    # Combine data and nonce
    combined_data = data + str(nonce)

    # Generate SHA-256 hash
    hash_result = hashlib.sha256(combined_data.encode()).hexdigest()

    # Check if hash meets difficulty requirement
    if hash_result.startswith(target):
        print("\nValid Hash Found!")
        print("Nonce:", nonce)
        print("Hash:", hash_result)
        break

    nonce += 1
```

```
Enter data string: Sannidhi
Enter difficulty level (number of leading zeros): 3
Mining started...

Valid Hash Found!
Nonce: 659
Hash: 0001acd2267d3eb32717537dcc31d4e10e65ed4dedb0551b97f7590626abe318
```

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.

```python
import hashlib

def sha256_hash(data):
    """
    Generates SHA-256 hash of given data
    """
    return hashlib.sha256(data.encode()).hexdigest()


def merkle_root(transactions):
    """
    Generates the Merkle Root from a list of transactions
    """
    # Step 1: Hash all transactions
    hashes = [sha256_hash(tx) for tx in transactions]

    # Step 2: Build Merkle Tree
    while len(hashes) > 1:
        # If odd number of hashes, duplicate the last hash
        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_hash = sha256_hash(combined_hash)
            new_level.append(new_hash)

        hashes = new_level

    # Step 3: Final hash is the Merkle Root
    return hashes[0]

    # ---------------- Main Program ----------------
    transactions = [

        "Transaction 1",
        "Transaction 2",
        "Transaction 3",
        "Transaction 4",
        "Transaction 5"
    ]

    print("Transactions:")
    for tx in transactions:
        print(tx)
    print("\nMerkle Root Hash:")
    print(merkle_root(transactions))
```

```
Transactions:
Transaction 1
Transaction 2
Transaction 3
Transaction 4
Transaction 5

Merkle Root Hash:
2c2c4cdf817ca1233db4784bb8752eddca8428c5c88ad7fad7e7235532e33c3c
```

Conclusion:

This experiment provided practical understanding of SHA-256 hashing, Proof-of-Work concepts, and Merkle Tree structures. These mechanisms collectively ensure data security, integrity, and efficient verification in blockchain networks.