

Aim: Cryptographic hash functions in blockchain, merkle tree

Theory:

1. Cryptographic Hash Functions in Blockchain

A **cryptographic hash function** is a mathematical algorithm that takes an input of any size and produces a **fixed-length output**, called a **hash value** or **digest**. In blockchain technology, cryptographic hash functions are fundamental for ensuring **data integrity, security, immutability, and trust** in a decentralized environment.

Common hash functions used in blockchains include **SHA-256** (Bitcoin), **Keccak-256 / SHA-3** (Ethereum), and **RIPEMD-160**.

2. Characteristics of Cryptographic Hash Functions

A secure cryptographic hash function must satisfy the following properties:

1. Deterministic

- The same input always produces the same hash.
- Ensures consistency across all nodes in the blockchain.

2. Fixed Output Length

- Regardless of input size, the output hash has a fixed length.
- Example: SHA-256 always produces a 256-bit hash.

3. Pre-image Resistance

- Given a hash, it is computationally infeasible to find the original input.
- Protects transaction data from reverse engineering.

4. Second Pre-image Resistance

- Given an input and its hash, it is infeasible to find another input with the same hash.
- Prevents data substitution attacks.

5. Collision Resistance

- It is extremely difficult to find two different inputs that produce the same hash.
- Ensures uniqueness of transaction identifiers.

6. Avalanche Effect

- A small change in input results in a completely different hash.
Helps detect even minor data tampering.

3. Role of Cryptographic Hash Functions in Blockchain

3.1 Transaction Hashing

- Each transaction is hashed to create a **transaction ID (TXID)**.
- Any modification to a transaction changes its hash.

3.2 Block Hashing

- Each block contains:
 - Transaction data
 - Previous block hash
 - Timestamp
 - Nonce
- All these fields are hashed to generate a **block hash**, linking blocks together.
This creates a **chain of blocks**, ensuring immutability.

4. Hash Functions in Proof of Work (PoW)

In Proof of Work:

Miners repeatedly hash block data with different nonce values.

The goal is to find a hash that satisfies a **difficulty target** (e.g., leading zeros).

Example:

$\text{Hash}(\text{Block Data} + \text{Nonce}) < \text{Target}$

This process:

- Requires computational effort
Prevents spam and malicious attacks
- Secures the blockchain against tampering

5. Hash Functions in Merkle Trees

- Transactions are hashed and organized into a **Merkle Tree**.
- The final hash (Merkle Root) summarizes all transactions.
Only the Merkle Root is stored in the block header.

Benefits:

- Efficient transaction verification
- Reduced storage
Enables **Simplified Payment Verification (SPV)**

2. What is a Merkle Tree?

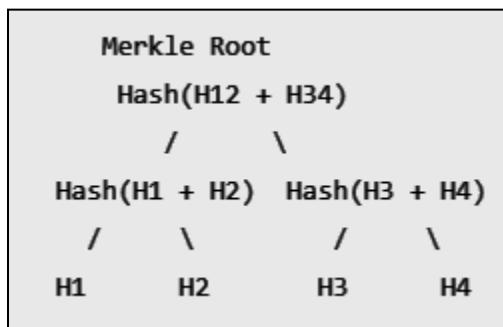
A **Merkle Tree** (also called a **Hash Tree**) is a **binary tree data structure** in which:

- Each **leaf node** contains the hash of a data block (e.g., transaction).
- Each **non-leaf node** contains the hash of its two child nodes.
- The **top hash** is called the **Merkle Root**.

It is used to **efficiently verify data integrity** in distributed systems like blockchains

3. Structure of a Merkle Tree

- **Leaf Nodes:** Hashes of individual transactions/data blocks.
- **Intermediate Nodes:** Hashes of concatenated child hashes.
- **Root Node (Merkle Root):** Final hash representing all data



4. Merkle Tree Rules

1. Every **transaction is hashed** first.
2. Hashes are combined **in pairs**.
3. If the number of hashes is **odd**, the last hash is **duplicated**.

4. Parent node hash = Hash(left child + right child)
5. Process continues until **only one hash remains** (Merkle Root).

5. Working of Merkle Tree (Step-by-Step)

1. Start with a list of transactions.
2. Generate hash for each transaction.
3. Pair adjacent hashes and hash their concatenation.
4. Repeat the pairing and hashing at each level.
5. Continue until a single hash (Merkle Root) is produced.

6. Benefits of Merkle Tree

- **Data Integrity** – Detects tampering instantly.
- **Efficient Verification** – Verifies large data with small hash proofs.
- **Reduced Storage** – Only hashes are stored, not full data.
- **Scalability** – Works efficiently with millions of transactions.
- **Security** – Cryptographic hashes prevent forgery.

7. Uses of Merkle Tree

- Blockchain transaction verification
- Distributed systems
- File integrity checking
- Version control systems
- Peer-to-peer (P2P) networks

8. Use Cases of Merkle Tree

1. Blockchain (Bitcoin, Ethereum)

- Stores transaction summaries in each block.
- Enables **light clients (SPV)** to verify transactions without full blockchain.

2. Cryptocurrency Mining

- Merkle Root is included in the block header.
- Changing a transaction changes the block hash.

3. Distributed Databases

- Efficient data synchronization between servers.

4. Git Version Control

- Tracks file changes and ensures data integrity.

5. Cloud Storage Systems

- Verifies file integrity without downloading entire files.

9. Why Merkle Trees Are Important

Merkle Trees allow **secure, fast, and efficient verification of large datasets**, making them essential for **blockchain technology and distributed computing**.

Implementation:

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

```
import hashlib

def sha256_hash(data):
    """
    Generates SHA-256 hash for the given input string.
    """
    sha = hashlib.sha256()
    sha.update(data.encode('utf-8'))
    return sha.hexdigest()

# User input
message = input("Enter a string to hash using SHA-256: ")

# Generate hash
hash_value = sha256_hash(message)

print("\nSHA-256 Hash:")
print(hash_value)
```

```
Enter a string to hash using SHA-256: rujuta
```

```
SHA-256 Hash:
```

```
fb95267b17e4b69adfed9e84b2075e8a7eec20cb67645021fdb8d3c23e8ef556
```

2. **Target Hash Generation with Nonce:** Create 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 sha256_hash(data):
    """
    Returns SHA-256 hash of the given data
    """
    return hashlib.sha256(data.encode('utf-8')).hexdigest()

# User input
message = input("Enter the data to be mined: ")

# Difficulty level (number of leading zeros)
difficulty = int(input("Enter difficulty level : "))

target_prefix = "0" * difficulty
nonce = 0

while True:
    combined_data = message + str(nonce)
    hash_result = sha256_hash(combined_data)

    if hash_result.startswith(target_prefix):
        print("Data      :", message)
        print("Nonce     :", nonce)
        print("Hash Result:", hash_result)
        break

    nonce += 1
```

```

Enter the data to be mined: rujuta
Enter difficulty level : 2
2
Data      : rujuta
Nonce     : 766
Hash Result : 00bd9c61ebfdb50a96eac836ebc6b421310660a6b88d81fa2dd30420380ae
          a3

```

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(data, difficulty):
    nonce = 0
    target = "0" * difficulty

    while True:
        combined = data + str(nonce)
        hash_result = hashlib.sha256(combined.encode()).hexdigest()

        if hash_result.startswith(target):
            return nonce, hash_result
        nonce += 1

# Example usage
data = input("Enter data: ")
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)

```

```

Enter data: h
Enter difficulty (number of leading zeros): 1
Nonce found: 26
Valid Hash: 0d368145227c320365c84b2120a560309a5fab745b05270a632d8128fb1ddf7b

```

```
Enter data: h
Enter difficulty (number of leading zeros): 2
Nonce found: 513
Valid Hash: 0032c8b348d54d996bf5926715b00732e8a0afbbea49c7180bdf9d42a50f9995
```

```
Enter data: h
Enter difficulty (number of leading zeros): 3
Nonce found: 1103
Valid Hash: 000b469ca1628133ccaf405ae947e57b5191484d16da7d34996513473c82aa42
== Code Execution Successful ==
```

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 sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()

def merkle_root(transactions):
    # Hash all transactions
    hashes = [sha256(tx) for tx in transactions]

    # Build the Merkle Tree
    while len(hashes) > 1:
        if len(hashes) % 2 != 0:
            hashes.append(hashes[-1]) # Duplicate last hash if odd

        new_level = []
        for i in range(0, len(hashes), 2):
            combined_hash = sha256(hashes[i] + hashes[i + 1])
            new_level.append(combined_hash)

        hashes = new_level

    return hashes[0]

# Example usage
```

```
transactions = [  
    "Tx1: Alice pays Bob",  
    "Tx2: Bob pays Charlie",  
    "Tx3: Charlie pays Dave",  
    "Tx4: Dave pays Eve"  
]  
  
print("Merkle Root:", merkle_root(transactions))
```

```
Merkle Root: 8fc5b2e4190bad429227fcd3d63a35aebbbb72ab6ee0fb6425c05b7e9d0316  
9
```