Blockchain Lab Exp 1

Name: Rohan Lalchandani **Class:** D20A
Roll no: 31 **Batch:** A

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

Theory:

Cryptographic Hash Function in Blockchain:

Cryptographic Hash Function in Blockchain A cryptographic hash function is a special mathematical technique that converts any kind of input data—such as transaction details, files, or text—into a fixed-size string of characters known as a hash. In blockchain technology, this hash works like a unique digital identity for the data stored inside a block. Even the smallest modification in the input, like changing one letter, produces a completely new hash value. This makes hashing extremely useful for ensuring blockchain security and trust. Blockchains such as Bitcoin and Ethereum commonly rely on hashing algorithms like SHA-256 to secure transaction records. Simply put, hashing guarantees that once information is stored on a blockchain, it cannot be secretly changed without everyone noticing.

Features of Cryptographic Hash Functions:

- **Consistency (Deterministic):** The same input always results in the exact same hash.
- **Constant Length Output:** Regardless of how big or small the input is, the hash output remains the same fixed size (for example, 256 bits in SHA-256).
- **Quick Processing:** Hash calculations are efficient and can be done rapidly, which is necessary for blockchain systems handling many transactions.
- **One-Way Nature (Pre-image Resistance):** It is nearly impossible to retrieve the original input from its hash.
- **Strong Uniqueness (Collision Resistance):** Two different inputs should not generate an identical hash.
- **Avalanche Property:** A minor input change leads to a drastically different hash result.

Importance of Hashing in Blockchain:

- Ensures **data integrity** by detecting any change in block data
- Provides **immutability**, as altering a block changes its hash
- Secures blocks through **cryptographic hash functions** like SHA-256
- Links blocks together using **previous block hash**
- Enables **Proof-of-Work mining** by validating computational effort
- Helps in **transaction verification** using Merkle Trees
- Prevents **tampering and fraud** in the blockchain network
- Ensures fast and efficient **data verification**
- Supports **decentralization and trustless systems**

Properties of SHA-256:

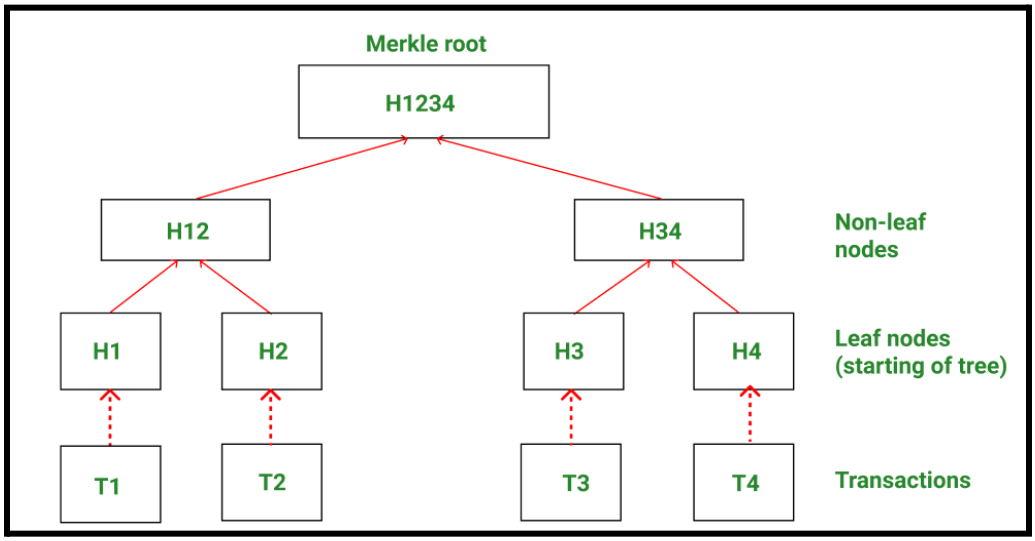
- **Deterministic**: The same input always produces the same 256-bit hash value.
- **Fixed Output Length**: Regardless of input size, SHA-256 always generates a 256-bit (64-hex character) hash.
- **Pre-image Resistance**: It is computationally infeasible to determine the original input from its hash.
- **Second Pre-image Resistance**: Given an input and its hash, finding another input with the same hash is practically impossible.
- **Collision Resistance**: Two different inputs producing the same hash is extremely unlikely.
- **Avalanche Effect**: A small change in input results in a completely different hash output.
- **Fast Computation**: Efficient to compute, making it suitable for real-time applications like blockchain.

What is a Merkle Tree (Hash Tree)?

A **Merkle Tree**, also known as a **Hash Tree**, is a **binary tree data structure** used to efficiently store and verify large sets of data. In a Merkle Tree, each **leaf node** contains the hash of a data block (such as a transaction), and each **non-leaf (parent) node** contains the hash of the concatenation of its child nodes. This hashing process continues recursively until a single hash, called the **Merkle Root**, is obtained at the top of the tree.

The Merkle Root uniquely represents all underlying data. Any change in even one data block results in a completely different Merkle Root, making data tampering easy to detect. Merkle Trees are widely used in **blockchain systems** to ensure transaction integrity, reduce storage requirements, and enable fast and secure verification of data.

A Merkle Tree follows a layered hierarchical arrangement. It begins with individual transaction hashes at the bottom level. These hashes are merged pairwise to form parent nodes, continuing upward through multiple levels until reaching the topmost hash, the Merkle Root, which summarizes and secures all transactions within the block.



1. **Leaf Nodes:** These are the hashes of individual transactions in a block. Each transaction is first hashed to form a leaf node.
2. **Intermediate (Parent) Nodes:** Pairs of leaf nodes are combined and hashed again to form parent nodes. This process continues upward, combining child nodes at each level.
3. **Root Node (Merkle Root):** The topmost node of the tree, representing all transactions in the block. If any transaction changes, the Merkle Root changes, making tampering detectable.

2. Merkle Root:

The Merkle Root is the topmost hash of the Merkle Tree. It acts as a summary of all transactions in a block. If any transaction is changed, its hash changes, which affects the Merkle Root. Since the Merkle Root is stored in the block header, this makes tampering easy to detect.

3. Working of Merkle Tree

The Merkle Tree works by organizing and summarizing all transactions in a block into a single hash (the Merkle Root) so that data can be verified efficiently and securely.

The process follows these steps:

- **Hashing Transactions:** Each transaction in the block is first hashed using a cryptographic hash function, usually SHA-256. These hashes form the leaf nodes of the Merkle Tree.

  Example: Transactions: T1, T2, T3, T4 Hashes: H(T1), H(T2), H(T3), H(T4)

- **Pairing and Hashing:** The leaf nodes are grouped in pairs. Each pair of hashes is concatenated (joined together) and then hashed again to create a parent node. If the number of transactions is odd, the last hash is duplicated to form a pair. This ensures that every level of the tree has an even number of nodes.

  Example:

  - Pair 1: H(T1) + H(T2) → Hash = H12
  - Pair 2: H(T3) + H(T4) → Hash = H34

- **Building the Tree:** The pairing and hashing process continues up the tree, combining parent nodes to create new higher-level nodes.

  Example: H12 + H34 → Hash = H1234

- **Creating the Merkle Root:** This process repeats until only one hash remains at the top of the tree. This top-level hash is called the Merkle Root, which represents all transactions in the block.

- **Verification:** To verify that a transaction exists in a block, a Merkle Proof is used. Instead of checking every transaction, only a small number of hashes along the path from the transaction leaf to the Merkle Root are needed. This makes verification fast and efficient, even for large blocks of data. Example: To verify T1: Use H(T2) and H34 along with H(T1) to recalculate H1234. If it matches the Merkle Root, T1 is valid.

- **Tamper Detection:** If any transaction is modified, its hash changes. This change propagates up the tree and alters the Merkle Root. Since the Merkle Root is stored in the block header, any tampering becomes immediately obvious. Example: If T3 changes → H(T3) changes → H34 changes → H1234 changes → Merkle Root mismatch.

4. Benefits of Merkle Tree

- **Efficient Verification:** Only a small number of hashes are needed to verify a transaction using a Merkle Proof, so checking data is fast even for large blocks.
- **Data Integrity:** Any change in a transaction alters the Merkle Root, making it easy to detect tampering.
- **Reduced Storage:** Instead of storing all transaction data in the block header, only the Merkle Root is stored, saving space.
- **Scalability:** Merkle Trees allow blockchains to handle a large number of transactions without slowing down verification.
- **Security:** The structure ensures that transactions cannot be altered without being noticed, keeping the blockchain secure.

5. Use of Merkle Tree in Blockchain:

- **Efficient Transaction Verification:** Nodes can verify a single transaction without downloading the entire block, using a Merkle Proof.
- **Ensures Data Integrity:** Any change in a transaction immediately changes the Merkle Root, helping detect tampering.
- **Reduces Storage Needs:** Only the Merkle Root is stored in the block header, instead of all transaction data, saving space.
- **Supports Lightweight Nodes:** Simple Payment Verification (SPV) nodes can confirm transactions securely without holding the full blockchain.

Applications of Merkle Tree:

Merkle Trees are an important data structure used not only in blockchain but also in many digital security systems. Their main advantage is that they can verify huge amounts of data quickly without needing to store or process everything. Below are some common practical uses:

1. Blockchain Block Transaction Management

In cryptocurrencies such as Bitcoin, thousands of transactions may exist inside a single block. Merkle Trees help compress this information efficiently.

- Example: Instead of keeping the entire transaction list in the block header, only one hash value—the Merkle Root—is stored.
- If any transaction is modified, the Merkle Root changes instantly.

Why useful: Ensures block integrity and makes transaction verification simpler.

2. Lightweight Wallet Verification (SPV Method)

Many mobile wallets and small devices cannot store complete blockchain data. They use Merkle Trees for fast checking.

- Example: A phone wallet can verify a payment by downloading only the transaction proof path, not the entire block.

Benefit: Saves internet bandwidth, storage space, and still maintains trust.

3. Software Development and File Tracking

Merkle Trees are widely used in systems that need reliable tracking of file changes.

- Example: In software platforms, every file version can be hashed, and any update changes the root hash, clearly showing that the content was altered.

Advantage: Helps in managing project history and preventing unnoticed modifications.

4. Cloud Storage Validation

Cloud providers must guarantee that stored files are not corrupted over time.

- Example: A Merkle Tree can be built from file chunks, allowing users to verify that their uploaded data remains unchanged.

Result: Secure storage and easy corruption detection.

5. Secure Data Sharing in P2P Systems

Peer-to-peer networks often share data in fragments, making verification necessary.

- Example: When downloading a large video split into parts, Merkle Trees allow users to confirm each part is authentic before assembling the full file.

Benefit: Prevents fake or tampered file pieces from spreading.

6. Digital Auditing and Record Security

Merkle Trees are also useful in financial audits and secure record-keeping.

- Example: Banks or companies can store transaction logs in a Merkle Tree so that auditors can later confirm no records were changed.

Importance: Ensures transparency and long-term trust in stored logs.

Colab Link:

https://colab.research.google.com/drive/1jNFwWGu97HtYjrz6tlHUh1nYGFr0aZRq?usp=sharing

Output and Screenshots:

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.

Code:

```
import hashlib
# Take input from user
data = input("Enter a string: ")

# Create SHA-256 hash
hash_object = hashlib.sha256(data.encode())
hash_value = hash_object.hexdigest()

print("SHA-256 Hash:", hash_value)
```

Output:

```
Enter a string: Rohan Blockchain
SHA-256 Hash: 2b969d0ab6cdb2894503ecaf30e22fe67d64b977c8aa5ddb95b0ff733040c2d7
```

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.

Code:

```
import hashlib

data = input("Enter input string: ")
target_prefix = "0000"
nonce = 0

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

```
    if hash_value.startswith(target_prefix):
        break
    nonce += 1

print("Input Data:", data)
print("Nonce Found:", nonce)
print("Target Hash:", hash_value)
```

Output:

```
Enter input string: This is life
Input Data: This is life
Nonce Found: 29674
Target Hash: 00002278f2f3518f7cd630efb5d76d22820a1f1da8ad325bd1d87a352001f37d
```

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

data = input("Enter input string: ")
difficulty = int(input("Enter number of leading zeros:
"))

prefix = "0" * difficulty
nonce = 0

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

    if hash_value.startswith(prefix):
        print("Proof-of-Work Solved!")
        print("Nonce:", nonce)
        print("Hash:", hash_value)
        break

    nonce += 1
```

Output:

```
Enter input string: Birds are flying
Enter number of leading zeros: 5
Proof-of-Work Solved!
Nonce: 1710266
Hash: 00000bc68bf85693e62e45918aaeb15feab9a3963a316666de8bcfa1c6f850fd
```

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.

Code:

```
import hashlib

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

def build_merkle_tree(transactions):
    hashes = [sha256(tx) for tx in transactions]

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

        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]

# Input transactions
transactions = [
    "Tx1: Alice pays Bob",
    "Tx2: Bob pays Charlie",
    "Tx3: Charlie pays Dave",
    "Tx4: Dave pays Eve"
]
```

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

Output:

```
Merkle Root Hash: 8fc5b2e4190bad429227fcd3d63a35aebbbbd72ab6ee0fb6425c05b7e9d03169
```

Conclusion:

In this practical, we successfully studied and implemented the core cryptographic concepts used in blockchain technology. We generated SHA-256 hashes to ensure data integrity, explored the role of nonce in producing different hash outputs, and simulated the Proof-of-Work mechanism by finding a valid nonce that satisfies a target difficulty condition.

Additionally, we constructed a Merkle Tree to securely summarize multiple transactions into a single Merkle Root. These experiments demonstrate how hashing, mining, and Merkle Trees collectively provide security, immutability, and efficient verification in blockchain systems.