EXPERIMENT NO 01

Name: Atharva Prabhu **Class:** D20A

Roll: No: 45 **Batch:** B

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

Theory:

Cryptographic Hash Functions in Blockchain

A cryptographic hash function is a mathematical algorithm that converts input data of any size into a fixed-length output known as a hash value or digest. In blockchain systems, hash functions play a critical role in maintaining data integrity, security, immutability, and trust within decentralized networks.

Widely used hash functions in blockchain include:

- SHA-256 (Bitcoin)
- Keccak-256 / SHA-3 (Ethereum)
- RIPEMD-160

These algorithms ensure that blockchain data remains tamper-proof and verifiable.

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 output, ensuring consistency across all blockchain nodes.

2. Fixed Output Length

Regardless of input size, the hash output has a constant length.

Example: SHA-256 always generates a 256-bit hash.

3. Pre-image Resistance

Given a hash value, it is computationally infeasible to determine the original input, protecting transaction data from reverse engineering.

4. Second Pre-image Resistance

Given an input and its hash, it is extremely difficult to find another input that produces the same

hash, preventing data replacement attacks.

5. Collision Resistance

It is highly improbable to find two different inputs that generate identical hashes, ensuring uniqueness of transactions.

6. Avalanche Effect
A small change in input produces a drastically different hash output, allowing immediate detection of data tampering.

Properties of SHA-256

SHA-256 (Secure Hash Algorithm – 256-bit) belongs to the SHA-2 family and is one of the most commonly used hashing algorithms in blockchain technology. It converts any input data into a secure 256-bit hash value, regardless of the input's original size.

- **Fixed 256-bit Output:** SHA-256 always produces a hash of exactly 256 bits, usually shown as 64 hexadecimal characters, no matter how large the input is.
- **High Security Strength:** It is widely trusted for modern cryptography and is a standard choice in blockchain networks for strong protection.
- **Collision Resistance:** The chance that two different pieces of data generate the same hash is extremely rare, ensuring reliable data integrity.
- **Avalanche Property:** Even the smallest change in input data results in a completely different hash output, making unauthorized modifications easy to detect.
- **Irreversible (One-Way Function):** It is practically impossible to reverse the hash and recover the original input data.

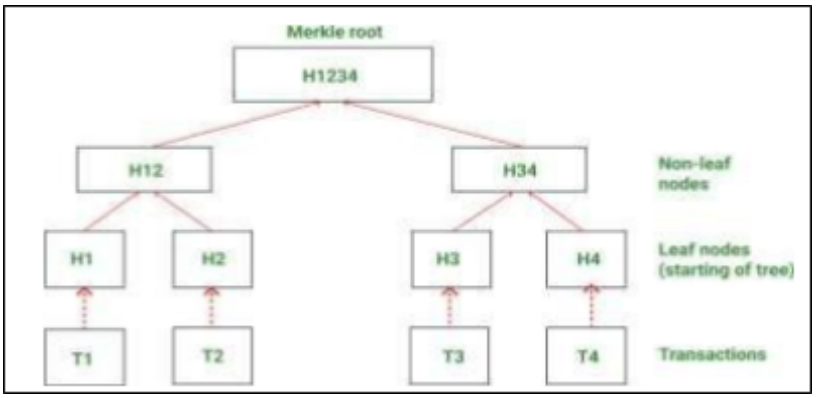
Merkle Tree (Hash Tree)

A Merkle Tree is a specialized binary tree structure used in blockchain to store and verify a large number of transactions efficiently. In this structure, every transaction is first converted into a hash, forming the leaf nodes. These hashes are then combined in pairs, hashed again, and repeated layer by layer until one final hash remains, called the **Merkle Root**.

The Merkle Root represents the complete set of transactions in a block. By storing only this single root hash inside the block header, blockchain systems can confirm that all transactions remain unchanged. If even one transaction is altered, the Merkle Root changes instantly, making fraud or tampering easy to detect.

Structure of a Merkle Tree

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 Notebook:
Code Link

Code & Output:

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
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)
```

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
def find_nonce(input_string, num_zeros):
    nonce = 0
    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
```

```
Enter a string: Atharva
Enter the nonce: 4
Hash Code: c18c0ab360b4b8f433845a7df2c266d5a358929bedb633b424514d1b99ea1b80
```

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 find_nonce(input_string, num_zeros):
nonce = 0
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)
```

```
 Enter the input string: Atharva
Enter the number of leading zeros: 4
Hash: 0000161e30b1ef5493a7a64d6b8f7e4c11423737eb8406e8dec44f9ee7ebd48c
Input String: Atharva
Leading Zeros: 4
Expected Nonce: 97093
```

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

    while len(hashes) > 1:
        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_level.append(sha256(combined_hash)
        hashes = new_level
    return hashes[0]
transactions = [
    "User1 sends 12 coins to User7",
    "Miner42 rewards itself 6.25 coins",
    "WalletA transfers 0.8 coins to WalletB",
    "ContractX releases 15 coins to Vault9"
]

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

Merkle Root Hash: 6f3d19cf44fda0e0a0af16ec5c6d00db12d88eeabc4b875b105db2d87b6b4eb

Conclusion:

Cryptographic hash functions like **SHA-256** help keep blockchain data safe, secure, and unchangeable. **Merkle Trees** organize transactions in a way that makes it easy to verify them quickly and detect any tampering. Together, they make blockchain trustworthy, capable of handling large amounts of data without compromising security. These concepts help us understand how blockchain maintains security, efficiency even with large amounts of data.