EXPERIMENT NO 01

Name: Tejas Gunjal **Class:** D20A

Roll: No: 24 **Batch:** B

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

Theory:

1. Cryptographic Hash Function in Blockchain

A cryptographic hash function is a mathematical algorithm that takes an input of any size (such as text, a file, or transaction data) and converts it into a fixed-length output called a hash value. In blockchain, this hash acts like a digital fingerprint of the data. Whenever data inside a block changes, even by a single character, the generated hash changes completely. Because of this behavior, hash functions help in maintaining data integrity and security in blockchain systems. Popular blockchains like Bitcoin and Ethereum use SHA-256 as their hashing algorithm.

In simple terms, hashing ensures that once data is recorded on the blockchain, it becomes extremely difficult to alter without being detected.

Characteristics of Cryptographic Hash Function

1. **Deterministic:** The same input will always produce the same hash output.
2. **Fixed Output Size**: No matter how large or small the input is, the output hash will always be of fixed length (256 bits in SHA-256).
3. **Fast Computation:** Hash values can be generated quickly, which is important for processing large numbers of transactions.
4. **Pre-image Resistance:** It is practically impossible to reverse a hash and find the original.
5. **Collision Resistance:** Two different inputs should not produce the same hash value.
6. **Avalanche Effect:** A small change in input results in a completely different hash.

Role of Cryptographic Hash Function in Blockchain

- It connects all blocks together by storing the previous block's hash in the next block, forming a secure chain.
- It keeps blockchain data safe and unchangeable, because any change in data changes the hash immediately.
- It is used in mining and verification to validate transactions and add new blocks securely.
- It protects wallet addresses, digital signatures, and user data from tampering and fraud.

2. Properties of SHA 256

SHA-256 (Secure Hash Algorithm – 256 bit) is part of the SHA-2 family and is widely used in blockchain for secure hashing. It generates a 256-bit (32-byte) hash value for any input data.

- **256-bit Output:** SHA-256 always generates a fixed-length hash of 256 bits, represented as 64 hexadecimal characters, regardless of the input size.
- **Strong Security Level:** It is considered highly secure and is widely used in blockchain systems and modern digital security applications.
- **Collision Resistant:** The probability of two different inputs producing the same hash value is extremely low, making it reliable for data integrity.
- **Avalanche Effect:** Even a very small change in the input results in a completely different hash output, which enhances security.
- **One-Way Nature:** It is practically impossible to retrieve the original input data from the generated hash value.

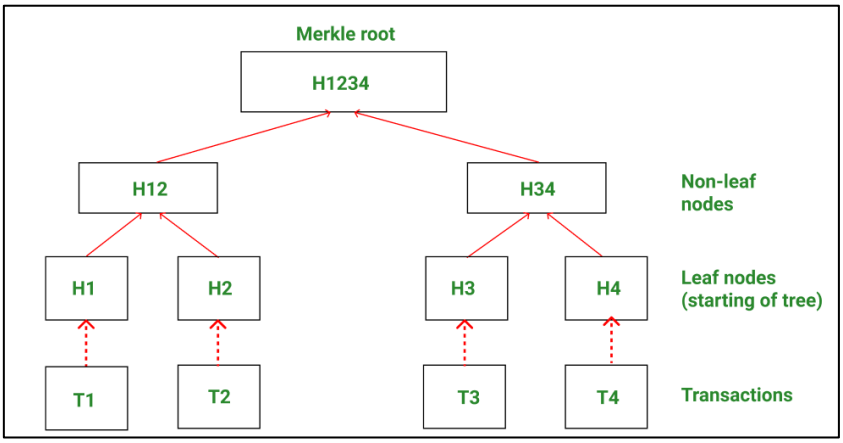
3. Merkle Tree [Hash Tree]

A Merkle Tree is a binary tree used in blockchain to organize and verify large sets of data efficiently. Each transaction is first hashed to form leaf nodes, then pairs of hashes are combined and hashed repeatedly to form parent nodes, until a single hash called the Merkle Root is created.

The Merkle Root represents all transactions in a block, so storing just this root in the block header ensures data integrity. If any transaction changes, the Merkle Root changes, making tampering easy to detect. Merkle Trees also allow fast verification using a small number of hashes, reducing storage and improving efficiency.

4. Structure of Merkle Tree

A Merkle Tree is organized in a hierarchical structure that allows efficient verification of transactions in a blockchain. It starts from individual transaction hashes and combines them in pairs, creating multiple levels of nodes until reaching a single top-level hash, known as the Merkle Root.



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.

5. 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.

6. 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.

7. 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.

8. 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.

9. Use Cases of Merkle Tree

Merkle Trees are widely used in blockchain and other data systems because they allow efficient verification of data integrity while reducing storage and computational requirements. Some important use cases are:

- **Blockchain and Cryptocurrencies:**
  In blockchain networks like Bitcoin and Ethereum, Merkle Trees are used to summarize all transactions in a block.
  - Example: Instead of storing every transaction in the block header, only the Merkle Root is stored. This allows nodes to verify transactions quickly using a Merkle Proof.
  - Benefits: Faster transaction verification, less storage, and tamper detection.

- **Efficient Transaction Verification (SPV Nodes):**
  Lightweight blockchain nodes, also called SPV (Simple Payment Verification) nodes, do not store the full blockchain.
  - They rely on Merkle Trees to confirm if a transaction is part of a block by only downloading the Merkle Path, not the entire block.
  - This reduces bandwidth and memory requirements while keeping security intact.

- **Distributed File Systems and Version Control:**
  Systems like Git or distributed storage platforms use Merkle Trees to track changes and ensure integrity.
  - Example: Git uses Merkle Trees to store snapshots of files. Any change in a file updates the hashes, making it easy to detect modifications.

- **Data Integrity Verification:**
  Merkle Trees can verify large sets of data in a secure way.
  - Example: Cloud storage providers or P2P networks can use Merkle Trees to ensure uploaded files are not corrupted or altered during transfer.

- **Peer-to-Peer Networks:**
  In P2P networks like BitTorrent, Merkle Trees help verify that each chunk of a downloaded file matches the original.
  - Each piece is hashed, and the combined hashes form a Merkle Tree, allowing clients to check integrity efficiently without downloading the full file first.

- **Secure Auditing and Logging:**
  Merkle Trees are used in audit trails to verify the integrity of logs over time.
  - Example: A company can store hashes of logs in a Merkle Tree so auditors can confirm that no logs have been tampered with, even years later.

Colab Notebook:
https://colab.research.google.com/drive/1M8mXKUs_rtgINWuT3S7DpAG8CHMCXazl?usp=sharing

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

data_string = 'Hello, I am Tejas'
encoded_data = data_string.encode('utf-8')
sha256_hash = hashlib.sha256()
sha256_hash.update(encoded_data)
hex_digest = sha256_hash.hexdigest()

print(f"Original string: '{data_string}'")
print(f"SHA-256 Hash: {hex_digest}")
```

```
Original string: 'Hello, I am Tejas'
SHA-256 Hash: ec39ad1714531280a8d32487eed9e6ee2fe6dde3ed419325a3c50ad4db486d33
```

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
data = input("Enter the input string: ")
nonce = int(input("Enter nonce value: "))
combined_data = data + str(nonce)
hash_with_nonce = hashlib.sha256(combined_data.encode()).hexdigest()
print("Hash with Nonce:", hash_with_nonce)
```

```
Enter the input string: Blockchain Practical
Enter nonce value: 5
Hash with Nonce: d0876b935820341da4a9dcd3c187b0d9aca39b1e1e5b3b681497d2a15c179b84
```

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 the input string: ")
difficulty = int(input("Enter number of leading zeros required: "))
prefix = '0' * difficulty
nonce = 0
while True:
```

```
        text = data + str(nonce)
        hash_result = hashlib.sha256(text.encode()).hexdigest()
        if hash_result.startswith(prefix):
            break
        nonce += 1
    print("Nonce found:", nonce)
    print("Valid Proof-of-Work Hash:", hash_result)
```

```
Enter the input string: Information Technology
Enter number of leading zeros required: 6
Nonce found: 2637081
Valid Proof-of-Work Hash: 000000850dd34ea6cce8c1c6ab23e661104b242ae749527abbe23a0da86a3bfe
```

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: 6f3d19cf44fda0e0a0af16ec5c6d00db12d88eeabc4b875b105db2d87b6b4eb1
```

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, and efficiency even with large amounts of data.