EXPERIMENT NO.1

Name: Aryan Anil Patankar
Class:D20A
Roll No:38
Batch:B

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

1. Cryptographic Hash Functions in Blockchain

Cryptographic hash functions are mathematical algorithms that take an input of any length and produce a fixed-length output string known as a hash value. In blockchain technology, they play a crucial role in ensuring data integrity, security, and immutability. These functions must be cryptographically secure, meaning they exhibit specific properties that make them resistant to attacks.

Properties of Cryptographic Hash Functions

- **Collision Resistant**: It is computationally infeasible to find two different inputs (m1 and m2) that produce the same hash output.
- **Preimage Resistant**: Given a hash value h, it is difficult to find the original input m that produced h.
- **Second Preimage Resistant**: Given an input m1, it is hard to find another input m2 that produces the same hash as m1.
- **Large Output Space**: Brute-force attacks require checking an enormous number of possibilities.
- **Deterministic**: The same input always produces the same output.
- **Avalanche Effect**: A small change in the input causes a significant change in the output.
- **Puzzle Friendliness**: Even if part of the output is known, predicting the rest is impossible.

- **Fixed-Length Mapping**: Outputs are always of a consistent length regardless of input size.

How Hash Functions Work in Blockchain

Hash functions convert variable-length data (e.g., transactions) into fixed-size outputs. In blockchain, this process involves encryption (generating the hash) and potential decryption for verification. Common algorithms include SHA-256, which always outputs 256 bits (32 bytes).

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.

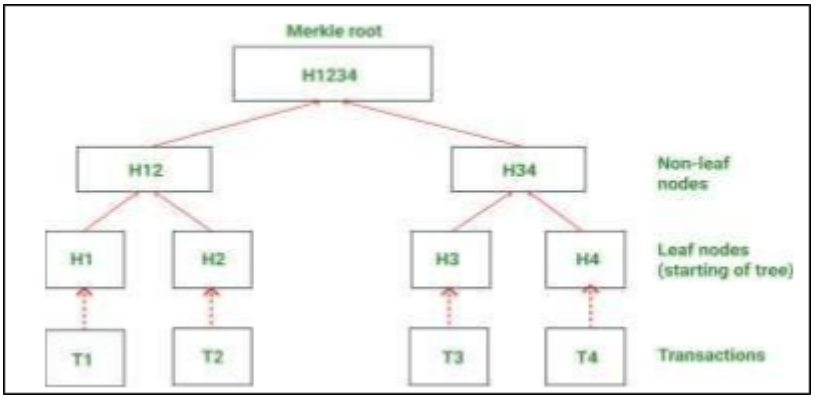
2. What is a Merkle Tree?

A Merkle Tree, also known as a hash tree, is a binary tree data structure used for efficient data verification and synchronization. Each leaf node represents the hash of a data block (e.g., a transaction), while each non-leaf node is the hash of its child nodes. The tree culminates in a single root hash (Merkle root), which serves as a digital fingerprint of the entire dataset.

In blockchain, Merkle Trees organize transactions per block, replacing less efficient structures like linked lists. They leverage cryptographic hashes and hash pointers (pointers that include both data location and its hash for verification).

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. **<u>Merkle Root</u>**

   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. What is a Cryptographic Puzzle and Explanation of the Golden Nonce

A cryptographic puzzle in blockchain refers to a complex mathematical problem that miners must solve to validate transactions and add new blocks. It is central to the Proof of Work (PoW) consensus mechanism, particularly in networks like Bitcoin. The puzzle involves finding a specific hash value that meets the network's difficulty criteria, such as being below a certain target threshold. This requires intensive computations, making it resource-heavy to solve but easy to verify.

Key Aspects of Cryptographic Puzzles

- **Purpose**: Secures the network by preventing easy tampering, double-spending, or counterfeiting. It ensures consensus without trusting any single party.
- **Process**: Miners hash block data (including transactions and a nonce) repeatedly until the output satisfies conditions. The difficulty adjusts dynamically (e.g., every 2016 blocks in Bitcoin) to maintain a ~10-minute block time.
- **Computational Intensity**: Involves brute-force trials; with millions of miners,

it takes significant power and time.
- **Security**: Limits total coin supply (e.g., 21 million Bitcoins) and makes illicit changes expensive.

The "Golden Nonce" refers to the specific nonce value that, when combined with the block data and hashed, produces a valid hash solving the puzzle. It is the "winning" nonce that allows a miner to claim the block reward (e.g., 6.25 Bitcoins). Miners iterate through nonce values (a 32-bit number) until this golden nonce is found, after which the block is broadcast for verification

4. 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 = H123

- **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.

5. 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.

6. Use Cases of Merkle Tree

Merkle Trees are versatile, with primary applications in blockchain and beyond:

- **Blockchain Transactions**: Organize and verify transactions in blocks; Merkle root in headers ensures integrity.
- **Proof of Membership**: Miners or users prove a transaction's inclusion using a Merkle proof (path of hashes), requiring only O(log n) data.
- **Simple Payment Verification (SPV)**: Lightweight clients verify transactions without the full blockchain.
- **Coinbase Transactions**: Included in every block's tree for new coin creation.
- **Distributed Systems**: Ensure data consistency across nodes; detect inconsistencies in replicas (e.g., Apache Cassandra databases).
- **File Systems**: Verify content in systems like IPFS or BitTorrent.
- **Inconsistency Detection**: Compare root hashes to pinpoint data differences quickly.

In Bitcoin and other blockchains, they enable trustless, efficient operations in peer-to-peer networks.

Colab Notebook:
Blockchain Exp1

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.

```python
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)
```

```
Enter a string: aryanpatankar
Hash: a224b5173a85e4214fbf6d88600d3b4dca7b9523032095bbdb0584ed4e73fa87
```

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.

```python
# Program 2: Generating Target Hash with Input String and Nonce
import hashlib

# Get user input
input_string = input("Enter a string: ")
nonce = input("Enter the nonce: ")

# Concatenate the string and nonce
hash_string = input_string + nonce

# Calculate the hash using SHA-256
hash_object = hashlib.sha256(hash_string.encode('utf-8'))
hash_code = hash_object.hexdigest()

# Print the hash code
print("Hash Code:", hash_code)
```

```
Enter a string: Aryan
Enter the nonce: 1
Hash Code: 52d2de03394c757ee98f7cd8b73a72c99bfa8d953cbfde1905f948b6fc51c61d
```

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
# Program 3: Solving Cryptographic Puzzle for 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 = "Aryan"
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)
```

```
Hash: 043a2485b4ad07329119b33b48423d5477bf17a2afab3fe342b67cd0a727dde5
Input String: Aryan
Leading Zeros: 1
Expected Nonce: 20
```

4.**Merkle Tree Constructio**n: 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 build_merkle_tree(transactions):
    if len(transactions) == 0:
        return None

    # Hash each transaction initially
    hashed_transactions = [
        hashlib.sha256(t.encode('utf-8')).hexdigest()
        for t in transactions
    ]
    print("Initial Hashed Transactions:", hashed_transactions)

    # If only one transaction, it is the Merkle Root
    if len(hashed_transactions) == 1:
        return hashed_transactions[0]

    # Iterative construction of the Merkle Tree
    current_level = hashed_transactions
    level = 1

    while len(current_level) > 1:
        print(f"\nLevel {level} Hashes:")

        # If odd number of hashes, duplicate the last one
        if len(current_level) % 2 != 0:
            current_level.append(current_level[-1])

        new_level = []

        for i in range(0, len(current_level), 2):
            combined = current_level[i] + current_level[i + 1]
            hash_combined = hashlib.sha256(
                combined.encode('utf-8')
```

```
        ).hexdigest()

        new_level.append(hash_combined)

        print(
            f"  Combining {current_level[i][:6]}... and "
            f"{current_level[i + 1][:6]}... to get "
            f"{hash_combined[:6]}..."
        )

    current_level = new_level
    level += 1

# Final remaining hash is the Merkle Root
return current_level[0]
```

```
Initial Hashed Transactions: ['4d5a82f66fccc5af28031a82a5ceb39dc3fe23097c56c8dff7720841a02aef1f', 'fd0b9eafb0c7f54035c2f21ff45dbbf165440d5931ad

Level 1 Hashes:
  Combining 4d5a82... and fd0b9e... to get 9cd41e...
  Combining 43dd57... and 2ed5c8... to get 9f37c0...
  Combining 4313e3... and 4313e3... to get bc2172...

Level 2 Hashes:
  Combining 9cd41e... and 9f37c0... to get 6fa7f1...
  Combining bc2172... and bc2172... to get ba86e9...

Level 3 Hashes:
  Combining 6fa7f1... and ba86e9... to get c5db5f...

Merkle Root: c5db5fb4092cf71773baf667101b0158168f44d9a99579c5385353c80ef38aff
```

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.