Experiment 1

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

Theory:

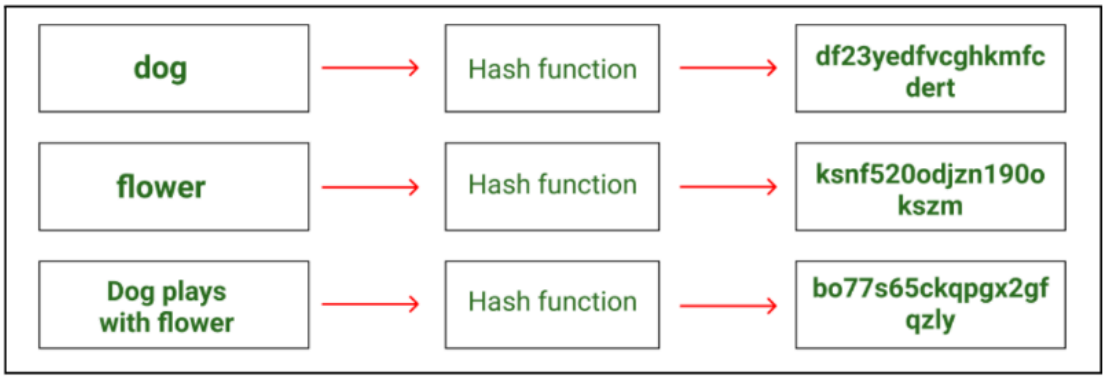
1. Cryptographic Hash Function in Blockchain

A hash function is a mathematical function that takes an input string of any length and converts it to a fixed-length output string. The fixed-length output is known as the hash value. To be cryptographically secure and useful, a hash function should have the following properties:

- Collision resistant: Given two messages m1 and m2, it is difficult to find a hash value such that hash(k, m1) = hash(k, m2) where k is the key value.
- Preimage resistance: Given a hash value h, it is difficult to find a message m such that h = hash(k, m). Original data cannot be derived from hash
- Second preimage resistance: Given a message m1, it is difficult to find another message m2 such that hash(k, m1) = hash(k, m2).
- Large output space: The only way to find a hash collision is via a brute force search, which requires checking as many inputs as the hash function has possible outputs.
- Deterministic: A hash function must be deterministic, which means that for any given input a hash function must always give the same result.
- Avalanche Effect: This means for a small change in the input, the output will change significantly.
- Puzzle Friendliness: This means even if one gets to know the first 200 bytes, one cannot guess or determine the next 56 bytes.
- Fixed-length Mapping: For any input of fixed length, the hash function will always generate the output of the same length.

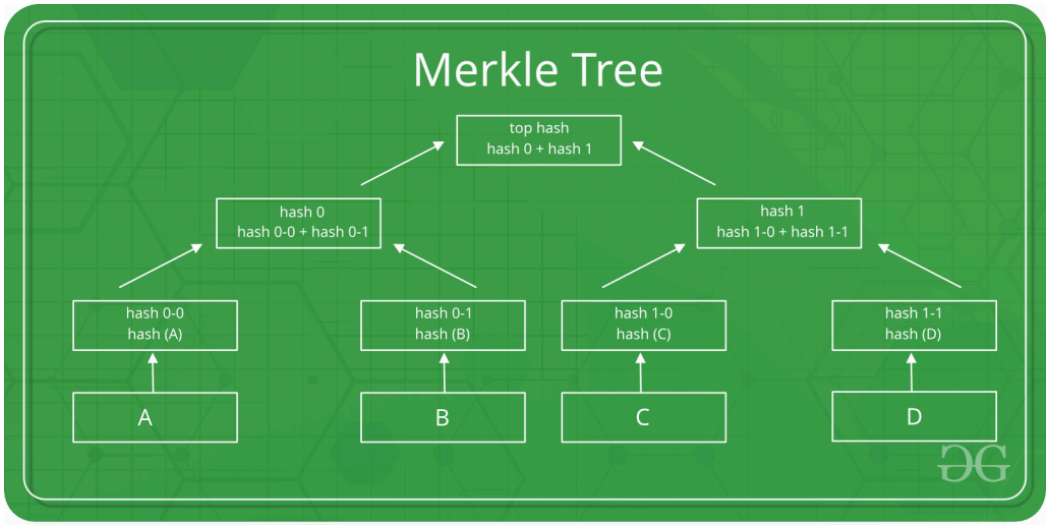
Role in Blockchain:

- Ensures data integrity
- Links blocks securely using hashes
- Used in Merkle Trees, block headers, and digital signatures



2. Merkle Tree (Hash Tree)

Merkle tree also known as hash tree is a data structure used for data verification and synchronization. It is a tree data structure where each non-leaf node is a hash of it's child nodes. All the leaf nodes are at the same depth and are as far left as possible. It maintains data integrity and uses hash functions for this purpose.



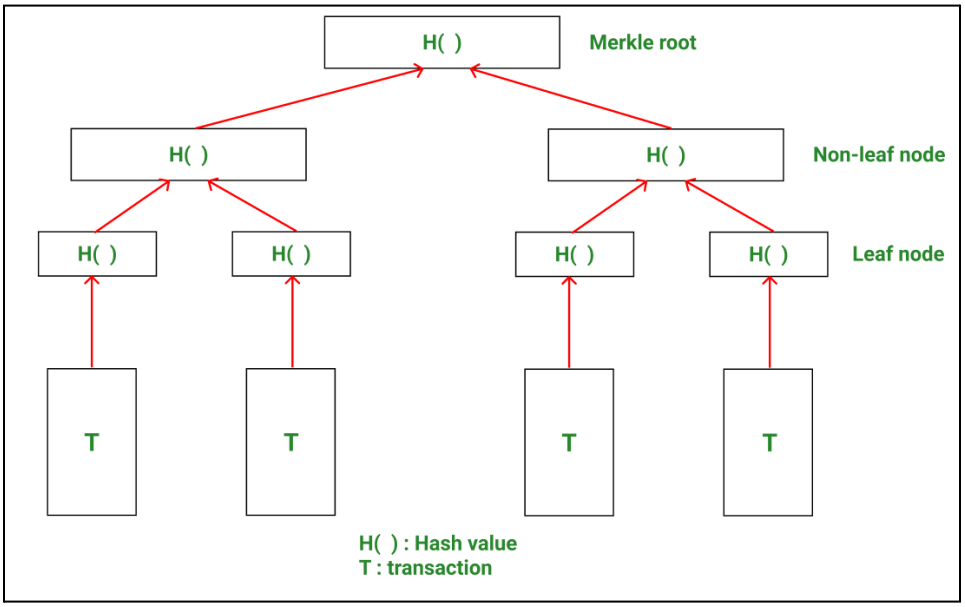
This is a binary merkel tree, the top hash is a hash of the entire tree.

- This structure of the tree allows efficient mapping of huge data and small changes made to the data can be easily identified.
- If we want to know where data change has occurred then we can check if data is consistent with root hash and we will not have to traverse the whole structure but only a small part of the structure.
- The root hash is used as the fingerprint for the entire data.

3. Structure of a Merkle Tree

A Merkle tree, also known as a hash tree, is an inverted binary tree data structure used for efficient and secure verification of large datasets. Its structure is built from the bottom up using cryptographic hashes, with three main components:

- **Leaf Nodes:** The bottom layer of the tree. Each leaf node contains the cryptographic hash of an individual data block (e.g., a transaction in a blockchain or a file chunk in a distributed file system).
- **Internal Nodes:** The middle layers of the tree. Each internal node is created by taking the hashes of its two child nodes, concatenating them, and then hashing the combined result. This process is repeated up the tree.
- **Merkle Root:** The single, topmost node of the tree. It is the final hash value that acts as a unique digital fingerprint for the entire dataset below it. If even a single piece of the original data is altered, the hashes of its parent nodes will change all the way up to the Merkle Root, thus indicating tampering



4. Merkle Root

A Merkle root is the final, single hash at the top of a Merkle tree (or hash tree), representing a summary of all transactions in a block, commonly used in blockchains like Bitcoin to efficiently verify data integrity without downloading the whole chain. It's generated by repeatedly hashing pairs of transaction hashes until only one hash remains, making it a "digital fingerprint" that ensures data hasn't been tampered with, as any change would alter the root.

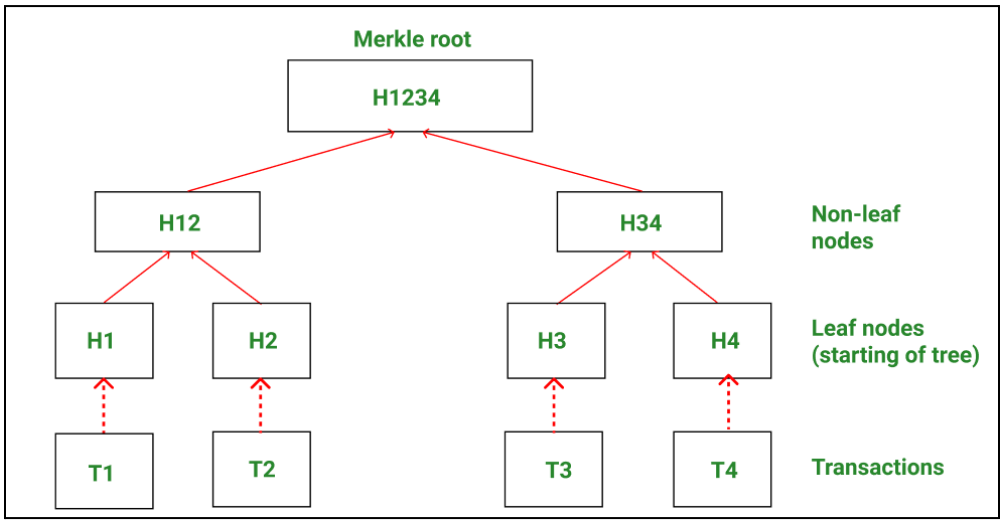
Importance of Merkle Root:

1. It acts as a unique digital fingerprint of all transactions in a block
2. It is stored in the block header, not the entire transaction list
3. If any single transaction is modified, the Merkle Root changes completely
4. It helps in efficient verification of transactions without downloading the full block
5. It ensures data integrity, security, and immutability of the blockchain

5. Working of Merkle Tree

A Merkle tree is constructed from the leaf nodes level all the way up to the Merkle root level by grouping nodes in pairs and calculating the hash of each pair of nodes in that particular level. This hash value is propagated to the next level. This is a bottom-to-up type of construction where the hash values are flowing from down to up direction.

Hence, by comparing the Merkle tree structure to a regular binary tree data structure, one can observe that Merkle trees are actually inverted down.



Example: Consider a block having 4 transactions- T1, T2, T3, T4. These four transactions have to be stored in the Merkle tree and this is done by the following steps

Step 1: The hash of each transaction is computed.

H1 = Hash(T1)

Step 2: The hashes computed are stored in leaf nodes of the Merkle tree.

Step 3: Now non-leaf nodes will be formed. In order to form these nodes, leaf nodes will be paired together from left to right, and the hash of these pairs will be calculated. Firstly hash of H1 and H2 will be computed to form H12. Similarly, H34 is computed. Values H12 and H34 are parent nodes of H1, H2, and H3, H4 respectively. These are non-leaf nodes.

H12 = Hash(H1 + H2)

H34 = Hash(H3 + H4)

Step 4: Finally H1234 is computed by pairing H12 and H34. H1234 is the only hash remaining. This means we have reached the root node and therefore H1234 is the Merkle root.

H1234 = Hash(H12 + H34)

6. Benefits of Merkle Tree

- **Efficient verification:** Merkle trees offer efficient verification of integrity and validity of data and significantly reduce the amount of memory required for verification. The proof of verification does not require a huge amount of data to be transmitted across the blockchain network. Enable trustless transfer of cryptocurrency in the peer-to-peer, distributed system by the quick verification of transactions.
- **No delay**: There is no delay in the transfer of data across the network. Merkle trees are extensively used in computations that maintain the functioning of cryptocurrencies.
- **Less disk space:** Merkle trees occupy less disk space when compared to other data structures.
- **Unaltered transfer of data:** Merkle root helps in making sure that the blocks sent across the network are whole and unaltered.
- **Tampering Detection:** Merkle tree gives an amazing advantage to miners to check whether any transactions have been tampered with.

7. Use of Merkle Tree in Blockchain

- In a centralized network, data can be accessed from one single copy. This means that nodes do not have to take the responsibility of storing their own copies of data and data can be retrieved quickly.
- However, the situation is not so simple in a distributed system.
- Let us consider a scenario where blockchain does not have Merkle trees. In this case, every node in the network will have to keep a record of every single transaction that has occurred because there is no central copy of the information.
- This means that a huge amount of information will have to be stored on every node and every node will have its own copy of the ledger. If a node wants to validate a past transaction, requests will have to be sent to all nodes, requesting their copy of the ledger. Then the user will have to compare its own copy with the copies obtained from several nodes.

- Any mismatch could compromise the security of the blockchain. Further on, such verification requests will require huge amounts of data to be sent over the network, and the computer performing this verification will need a lot of processing power for comparing different versions of ledgers.
- Without the Merkle tree, the data itself has to be transferred all over the network for verification.
- Merkle trees allow comparison and verification of transactions with viable computational power and bandwidth. Only a small amount of information needs to be sent, hence compensating for the huge volumes of ledger data that had to be exchanged previously.

8. Use Cases of Merkle Tree

Here are the most impactful use cases of Merkle Tree:

1. Blockchain: Light Clients & SPV

In networks like Bitcoin and Ethereum, full nodes store hundreds of gigabytes of data. However, mobile wallets (Light Clients) cannot do this.

- **The Mechanism:** Instead of downloading all transactions, a mobile wallet only downloads the Merkle Root (a tiny 32-byte hash) found in the block header.
- **The Use Case:** If someone sends you money, the wallet uses a Merkle Proof. It requests only the few specific hashes needed to "climb the tree" to the root. If the calculation matches the root in the header, the transaction is verified as valid without ever seeing the rest of the block.

2. Version Control: Git

Git is essentially a massive Merkle Tree (specifically a Merkle DAG).

- **The Mechanism:** Every file in a Git repository is hashed. Folders (directories) are also hashed based on the hashes of the files they contain.
- **The Use Case:** When you run git status or git pull, Git doesn't compare every line of every file. It compares the Merkle Root of your current folder with the one on the server. If they match, the folders are identical. If they don't, it quickly moves down the tree to identify exactly which file changed. This makes Git incredibly fast even for giant projects like the Linux kernel.

3. Distributed Databases: Anti-Entropy (Cassandra & DynamoDB)

Databases like Apache Cassandra or Amazon DynamoDB replicate data across many servers globally to ensure it's never lost.

- **The Mechanism:** Occasionally, these replicas get out of sync due to network glitches. To fix this, nodes exchange Merkle trees of their data.
- **The Use Case:** By comparing roots, nodes can instantly tell if they are in sync. If a mismatch is found, they compare the child hashes to narrow down the specific "range" of data that is different. This allows the system to repair itself by transferring only the missing/corrupted data chunks rather than re-sending the entire multi-terabyte database.

4. P2P File Sharing: BitTorrent & IPFS

When downloading a 50GB file from 100 different strangers on BitTorrent, you need to ensure no one is sending you "junk" or malicious data.

- **The Mechanism:** The .torrent file contains a Merkle Root of the entire file.
- **The Use Case**: As each small "piece" of the file arrives, it is hashed. Using the Merkle Tree, the client can verify that the piece belongs to the original file immediately—before the whole download finishes. If one piece is corrupt, the client only discards that 1MB chunk instead of the whole 50GB.

5. Certificate Transparency (SSL/TLS)

To prevent rogue authorities from issuing fake SSL certificates (like a fake "https://www.google.com/search?q=google.com" certificate), all issued certificates are logged in public "Certificate Transparency" logs.

The Use Case: These logs use Merkle Trees so that any browser (Chrome, Safari) can efficiently verify that a certificate is actually recorded in the official log without having to download the millions of other certificates in that log.

Programs and Output:

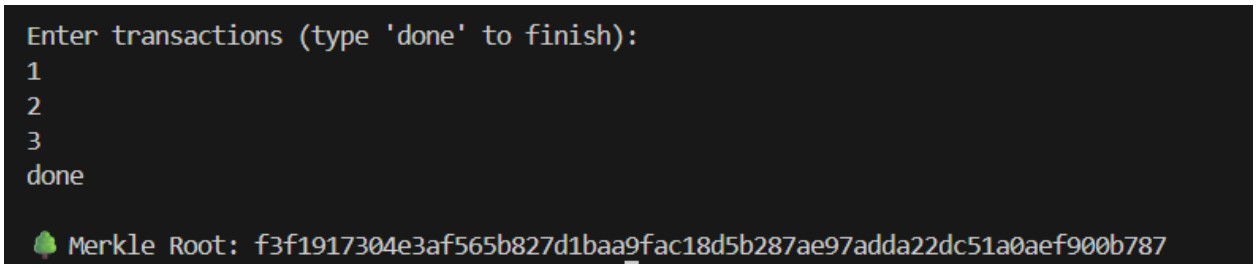
```
import hashlib

# 1. SHA-256 Hash Generation
def sha256_hash(data):
    return hashlib.sha256(data.encode()).hexdigest()

# 2. Hash Generation with Nonce (Mining simulation)
def hash_with_nonce(data, nonce):
  text = data + str(nonce)
  return sha256_hash(text)

# 3. Proof of Work (find nonce for leading zeros)
def proof_of_work(data, difficulty):
  print("\n🔍 Mining started...")
  prefix = "0" * difficulty
  nonce = 0
  while True:
   new_hash = hash_with_nonce(data, nonce)
   if new_hash.startswith(prefix):
     return nonce, new_hash
   nonce += 1

# 4. Merkle Tree & Merkle Root

def merkle_root(transactions):

   hashes = [sha256_hash(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 = hashes[i] + hashes[i+1]
        new_level.append(sha256_hash(combined))
      hashes = new_level
   return hashes[0]

# Main

if __name__ == "__main__":

 msg = input("Enter a string: ")
 print("\nSHA-256 Hash:", sha256_hash(msg))
```

```
nonce = int(input("\nEnter a nonce: "))
print("Hash with nonce:", hash_with_nonce(msg, nonce))

difficulty = int(input("\nEnter difficulty (number of leading zeros):"))
nonce, result_hash = proof_of_work(msg, difficulty)
print("\n✅ Proof of Work Found!")
print("Nonce:", nonce)
print("Hash:", result_hash)

print("\nEnter transactions (type 'done' to finish):")
transactions = []
while True:
 tx = input()
 if tx.lower() == "done":
   break
 transactions.append(tx)

if len(transactions) > 0:
 root = merkle_root(transactions)
 print("\n🌳 Merkle Root:", root)
else:
 print("No transactions entered.")
```

```
PS C:\Users\siddi\Desktop\blockchain Lab> python lab1.py
Enter a string: mahvish

SHA-256 Hash: 4c5c095768c5ba030ba8f0d452988a83ec6db3789c1ccb0a39cf80d030898bb1
```

```
Enter a nonce: 2
Hash with nonce: 9c04ea0baf682a9f738e085f7077c6cfc63ac24651f654ebd48f48c76c24afd5
```

```
Enter difficulty (number of leading zeros):4

🔍 Mining started...

✅ Proof of Work Found!
Nonce: 13712
Hash: 0000a7c9eec0d1f0b6e5851c391c388754685c867f0c01e8016f0cb23359aeb0
```



Tasks Performed:

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.
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.
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.
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.

Conclusion:

Through these tasks, the foundational cryptographic pillars of blockchain technology were successfully implemented. Data integrity and network consensus were demonstrated through the development of **SHA-256 hashing** and **Proof-of-Work** simulations. Additionally, the efficient verification of large-scale transaction data was achieved by constructing a **Merkle Tree**, illustrating how individual algorithms are combined to maintain a secure, immutable ledger.