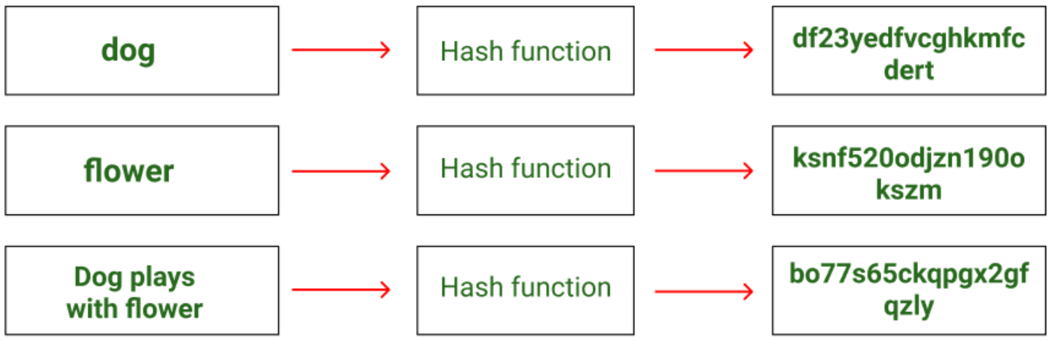
Aim: Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root tree hash.

Theory:

1. Cryptography Theory

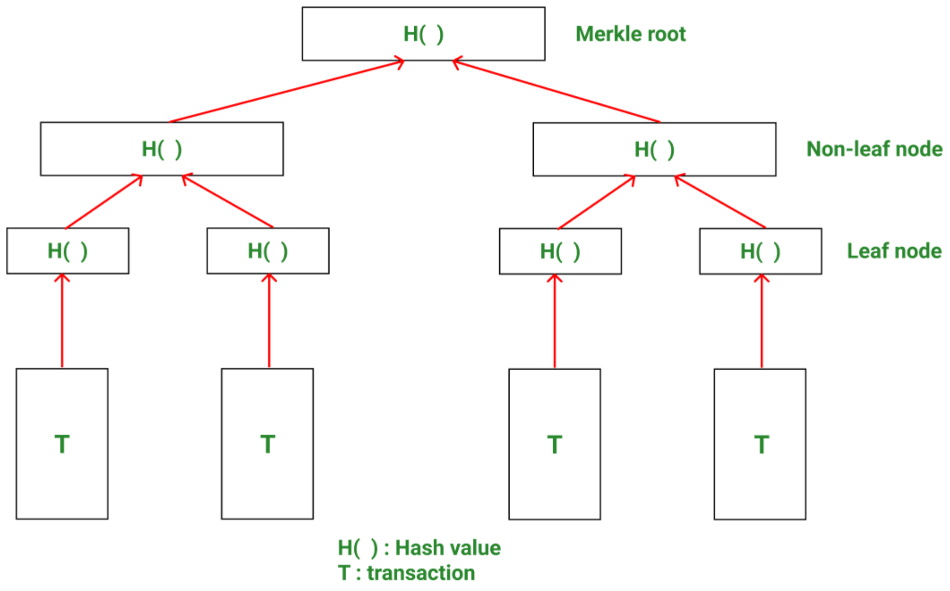
   Cryptography is the practice and study of techniques for secure communication in the presence of adversarial behavior. It involves converting readable information (plaintext) into an unreadable format (ciphertext) to prevent unauthorized access. The process of converting plaintext to ciphertext is called encryption, and the reverse process is called decryption.



Key Principles of Cryptography

1. Confidentiality: Ensures that information is accessible only to those authorized to have access.
2. Integrity: Ensures that information is not altered during transmission.
3. Authentication: Confirms the identities of the parties involved in communication.
4. Non-repudiation: Ensures that the sender cannot deny sending the message

2. Merkle tree structure



1. A blockchain can potentially have thousands of blocks with thousands of transactions in each block. Therefore, memory space and computing power are two main challenges.

2. It would be optimal to use as little data as possible for verifying transactions, which can reduce CPU processing and provide better security, and this is exactly what Merkle trees offer.

3. In a Merkle tree, transactions are grouped into pairs. The hash is computed for each pair and this is stored in the parent node. Now the parent nodes are grouped into pairs and their hash is stored one level up in the tree. This continues till the root of the tree. The different types of nodes in a Merkle tree are:

- Root node: The root of the Merkle tree is known as the Merkle root and this Merkle root is stored in the header of the block.
- Leaf node: The leaf nodes contain the hash values of transaction data. Each transaction in the block has its data hashed and then this hash value (also known as transaction ID) is stored in leaf nodes.
- Non-leaf node: The non-leaf nodes contain the hash value of their respective children. These are also called intermediate nodes because they contain the intermediate hash values and the hash process continues till the root of the tree.

4. Bitcoin uses the SHA-256 hash function to hash transaction data continuously till the Merkle root is obtained.

5. Further, a Merkle tree is binary in nature. This means that the number of leaf nodes needs to be even for the Merkle tree to be constructed properly. In case there is an odd number of leaf nodes, the tree duplicates the last hash and makes the number of leaf nodes even.

3. Merkle root

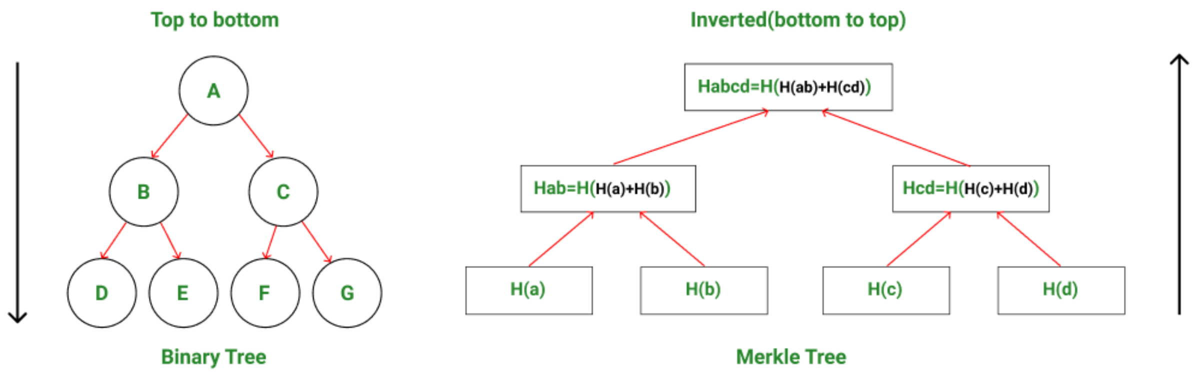
A Merkle root is the result of hashing the transactions in a block, pairing those hashes, and hashing them again until a single hash remains. Some blockchains use it to verify transactions without hashing and pairing hashes to compare Merkle roots generated by other nodes. This technique reduces the time needed to verify the transactions included in a block.

The Merkle root is used to verify transactions because a blockchain node only needs to check select blocks within the Merkle tree. This is called the Merkle proof.

4. 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.



- In order to check whether the transaction has tampered with the tree, there is only a need to remember the root of the tree.
- One can access the transactions by traversing through the hash pointers and if any content has been changed in the transaction, this will reflect on the hash stored in the parent node, which in turn would affect the hash in the upper-level node and so on until the root is reached.
- Hence the root of the Merkle tree has also changed. So Merkle root which is stored in the block header makes transactions tamper-proof and validates the integrity of data.
- With the help of the Merkle root, the Merkle tree helps in eliminating duplicate or false transactions in a block.
- It generates a digital fingerprint of all transactions in a block and the Merkle root in the header is further protected by the hash of the block header stored in the next block.

5. 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.
- Since the transactions are stored in a Merkle tree which stores the hash of each node in the upper parent node, any changes in the details of the transaction such as the amount to be debited or the address to whom the payment must be made, then the change will propagate to the hashes in upper levels and finally to the Merkle root.
- The miner can compare the Merkle root in the header with the Merkle root stored in the data part of a block and can easily detect this tampering

6. 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.
7. Use cases of merkle tree

1. Blockchain and Cryptocurrencies

Merkle trees are used to store and verify transactions within a block.

- Enable efficient transaction verification
- Support lightweight clients (SPV)
- Detect data tampering

Example: Bitcoin uses a Merkle root to verify transactions without storing the full block.

2. Data Integrity Verification

Merkle trees help ensure that data has not been altered.

- Quickly detect corrupted or modified data
- Efficient for large datasets
- Only changed data needs re-verification

Example: Verifying the integrity of large file backups.

3. Distributed Systems and Data Synchronization

Used to compare and synchronize data across multiple systems.

- Minimizes data transfer
- Efficient comparison of large datasets
- Ensures consistency across nodes

Example: Synchronizing data between peer-to-peer network nodes.

4. Version Control Systems (Git)

Merkle trees ensure integrity and traceability of code changes.

- Every commit is cryptographically linked
- Any change alters the root hash
- Prevents unnoticed history modification

Example: Git uses a Merkle DAG to maintain secure version history.

5. Cloud Storage and Secure File Systems

Merkle trees verify data stored in remote systems.

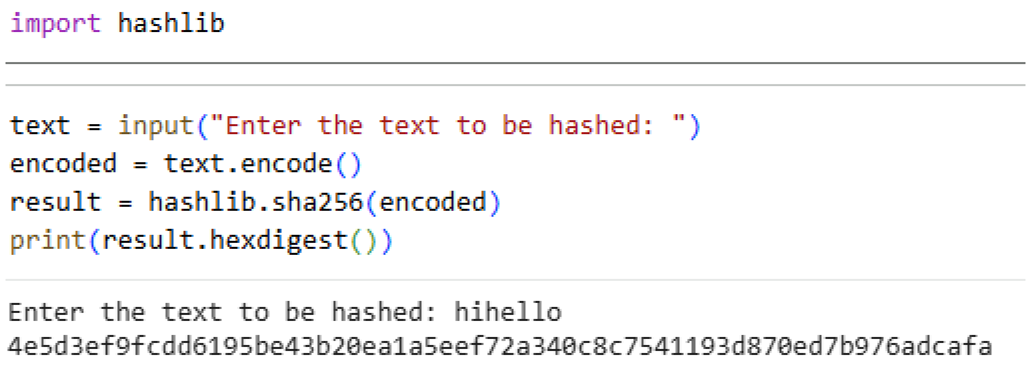
- Enables proof of data possession
- Detects unauthorized changes

- Supports secure storage architectures

Example: IPFS and ZFS use Merkle trees for data integrity.

Code:

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

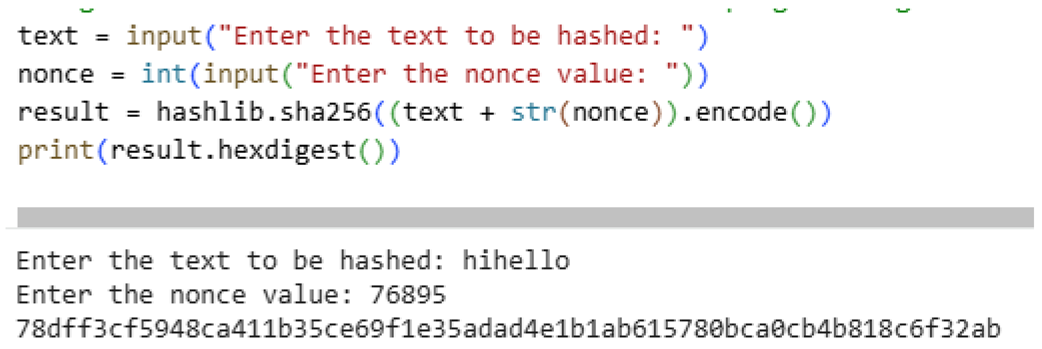


```python
import hashlib
```

```python
text = input("Enter the text to be hashed: ")
encoded = text.encode()
result = hashlib.sha256(encoded)
print(result.hexdigest())
```

```
Enter the text to be hashed: hihello
4e5d3ef9fcdd6195be43b20ea1a5eef72a340c8c7541193d870ed7b976adcafa
```

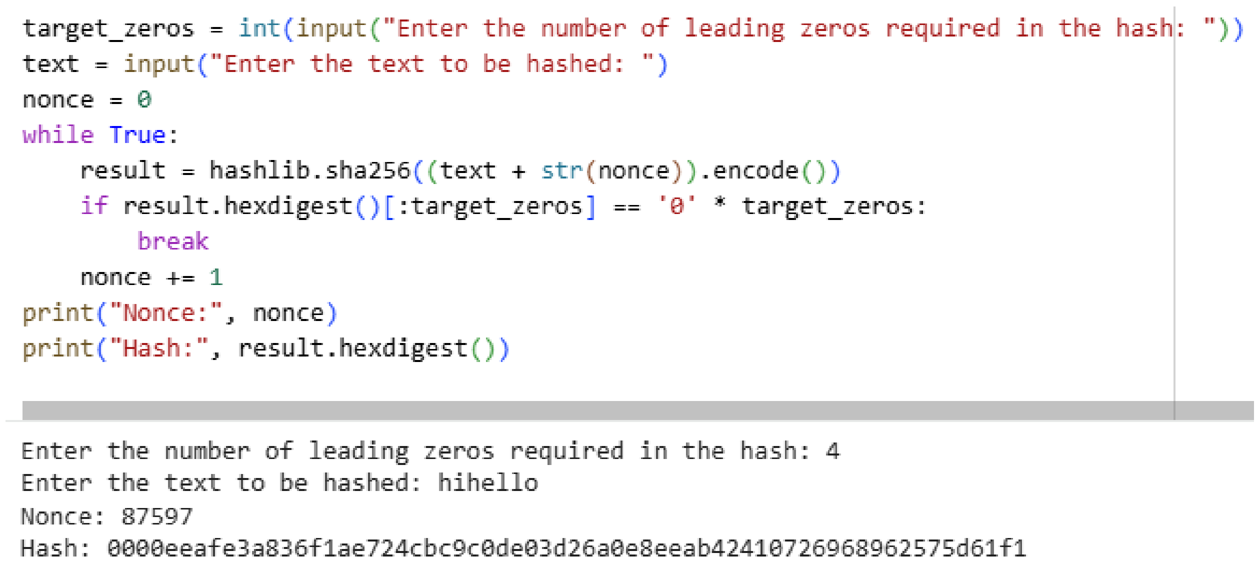
2. Target Hash Generation with Nonce: Created a program to generate a hash code by concatenating a user input string and a nonce value to simulate the mining process.



```python
text = input("Enter the text to be hashed: ")
nonce = int(input("Enter the nonce value: "))
result = hashlib.sha256((text + str(nonce)).encode())
print(result.hexdigest())
```

```
Enter the text to be hashed: hihello
Enter the nonce value: 76895
78dff3cf5948ca411b35ce69f1e35adad4e1b1ab615780bca0cb4b818c6f32ab
```

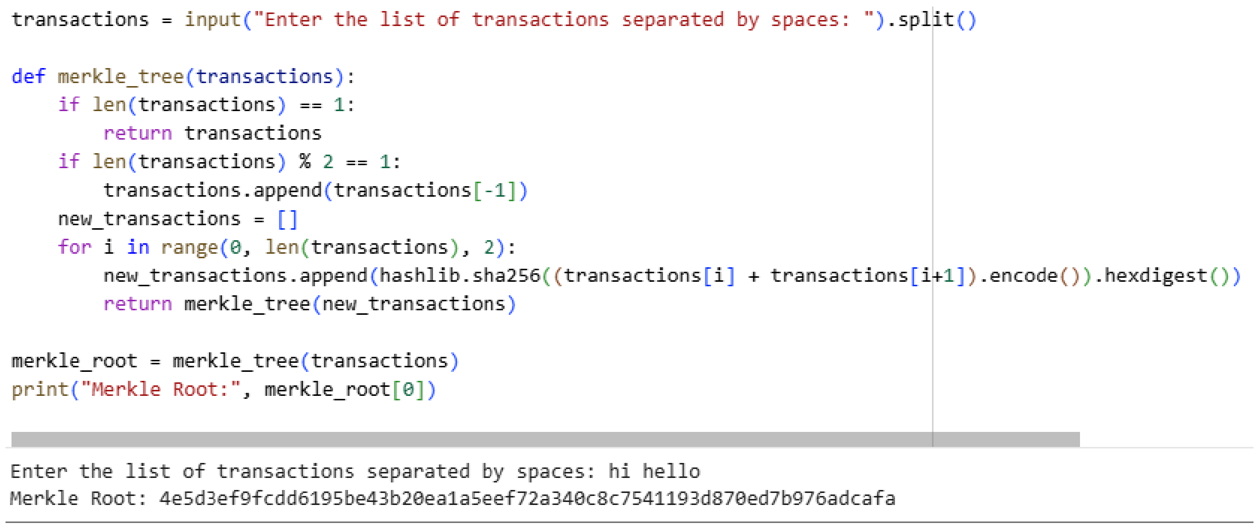
3. Proof-of-Work Puzzle Solving: Implemented a program to find the nonce that, when combined with a given input string, produces a hash starting with a specified number of leading zeros.



```
target_zeros = int(input("Enter the number of leading zeros required in the hash: "))
text = input("Enter the text to be hashed: ")
nonce = 0
while True:
    result = hashlib.sha256((text + str(nonce)).encode())
    if result.hexdigest()[:target_zeros] == '0' * target_zeros:
        break
    nonce += 1
print("Nonce:", nonce)
print("Hash:", result.hexdigest())
```

```
Enter the number of leading zeros required in the hash: 4
Enter the text to be hashed: hihello
Nonce: 87597
Hash: 0000eeafe3a836f1ae724cbc9c0de03d26a0e8eeab42410726968962575d61f1
```

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.



```
transactions = input("Enter the list of transactions separated by spaces: ").split()

def merkle_tree(transactions):
    if len(transactions) == 1:
        return transactions
    if len(transactions) % 2 == 1:
        transactions.append(transactions[-1])
    new_transactions = []
    for i in range(0, len(transactions), 2):
        new_transactions.append(hashlib.sha256((transactions[i] + transactions[i+1]).encode()).hexdigest())
        return merkle_tree(new_transactions)

merkle_root = merkle_tree(transactions)
print("Merkle Root:", merkle_root[0])
```

```
Enter the list of transactions separated by spaces: hi hello
Merkle Root: 4e5d3ef9fcdd6195be43b20ea1a5eef72a340c8c7541193d870ed7b976adcafa
```

Conclusion: This experiment helped in understanding the core cryptographic concepts used in blockchain technology. SHA-256 ensures data integrity, Proof of Work provides security through computational effort, and Merkle Trees enable efficient transaction verification. These concepts collectively form the foundation of secure and decentralized blockchain networks.