EXPERIMENT 01

AIM : Write a Python program to understand SHA and Cryptography in Blockchain and 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 input.
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.
- **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.
- **Avalanche Effect** – Even a very small change in the input results in a completely different hash output.
- **One-Way Nature** – It is practically impossible to retrieve the original input data from the generated hash value.

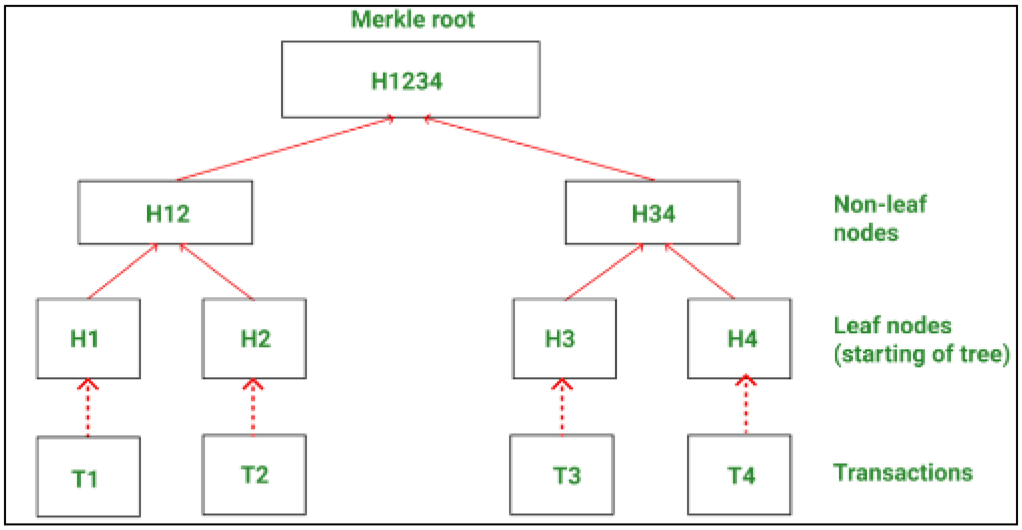
3. Merkle Tree (Hash Tree)

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. 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.



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 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 also changes.

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 modified, 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 called the **Merkle Root**, ensuring efficient and secure verification.

Steps Involved

Hashing Transactions

Each transaction is hashed using SHA-256 to form leaf nodes.

Example:
Transactions: T1, T2, T3, T4
Hashes: H(T1), H(T2), H(T3), H(T4)

Pairing and Hashing

Hashes are paired, concatenated, and hashed again to form parent nodes.
If the number of transactions is odd, the last hash is duplicated.

Example:

- Pair 1: H(T1) + H(T2) → H12
- Pair 2: H(T3) + H(T4) → H34

Building the Tree

Parent hashes are combined again to form higher-level hashes.

Example:
H12 + H34 → H1234

Creating the Merkle Root

The process repeats until one hash remains, which is the **Merkle Root**.

Verification

To verify a transaction, a **Merkle Proof** is used. Only a few hashes along the path are needed.

Example:
To verify T1 → Use H(T2) and H34 with H(T1).
If the calculated hash matches the Merkle Root, the transaction is valid.

Tamper Detection

If any transaction changes, its hash changes, affecting all parent hashes up to the Merkle Root.

Example:
T3 changes → H(T3) changes → H34 changes → H1234 changes → Merkle Root mismatch.

7. Benefits of Merkle Tree

- **Efficient Verification:**
  Transactions can be verified using only a few hashes instead of checking the entire block.
- **Strong Data Integrity:**
  Any change in a transaction changes the Merkle Root, making tampering easy to detect.
- **Reduced Storage Requirements:**
  Only the Merkle Root is stored in the block header, saving storage space.
- **High Scalability:**
  Large numbers of transactions can be handled without slowing down verification.
- **Improved Security:**
  Cryptographic hashing ensures transactions cannot be altered without detection.

8**. Use of Merkle Tree in Blockchain**

- Enables fast transaction verification using Merkle Proofs.
- Helps detect data tampering through Merkle Root comparison.
- Reduces storage needs by storing only the Merkle Root.
- Supports lightweight (SPV) nodes for efficient blockchain operation.

9. Use Cases of Merkle Tree

- **Blockchain and Cryptocurrencies:**
  Summarizes all transactions in a block using a single Merkle Root.
- **SPV Nodes:**
  Allows transaction verification without downloading full blocks.
- **Distributed File Systems:**
  Used in systems like Git to track and verify file changes.
- **Data Integrity Verification:**
  Ensures data integrity in cloud storage and P2P networks.
- **Peer-to-Peer Networks:**
  Used in BitTorrent to verify file chunks.
- **Secure Auditing and Logging:**
  Ensures logs remain unchanged over time.

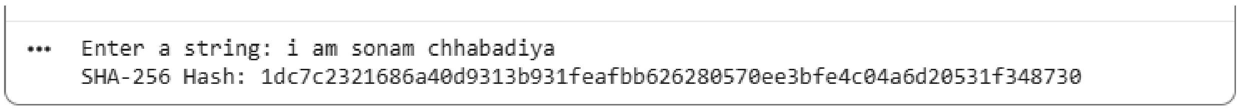
Colab Notebook:

https://colab.research.google.com/drive/1KgwKWa-gqMSq_cOp85It8qR92CmVVGtI?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.

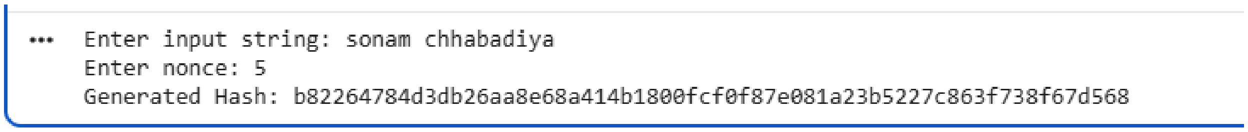
```
import hashlib
def sha256_hash(data):
return hashlib.sha256(data.encode()).hexdigest()
message = input("Enter a string: ")
print("SHA-256 Hash:", sha256_hash(message))
```



```
Enter a string: i am sonam chhabadiya
SHA-256 Hash: 1dc7c2321686a40d9313b931feafbb626280570ee3bfe4c04a6d20531f348730
```

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.

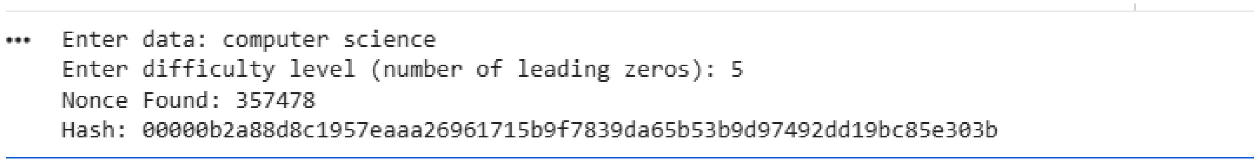
```
import hashlib
def hash_with_nonce(data, nonce):
text = data + str(nonce)
return hashlib.sha256(text.encode()).hexdigest()
data = input("Enter input string: ")
nonce = int(input("Enter nonce: "))
print("Generated Hash:", hash_with_nonce(data, nonce))
```



```
Enter input string: sonam chhabadiya
Enter nonce: 5
Generated Hash: b82264784d3db26aa8e68a414b1800fcf0f87e081a23b5227c863f738f67d568
```

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 proof_of_work(data, difficulty):
    prefix = '0' * difficulty
    nonce = 0
while True:
        text = data + str(nonce)
        hash_result = hashlib.sha256(text.encode()).hexdigest()
        if hash_result.startswith(prefix):
            return nonce, hash_result
        nonce += 1
data = input("Enter data: ")
difficulty = int(input("Enter difficulty level (number of leading zeros): "))
nonce, hash_result = proof_of_work(data, difficulty)
print("Nonce Found:", nonce)
print(" valid proof of work hash:", hash_result)
```



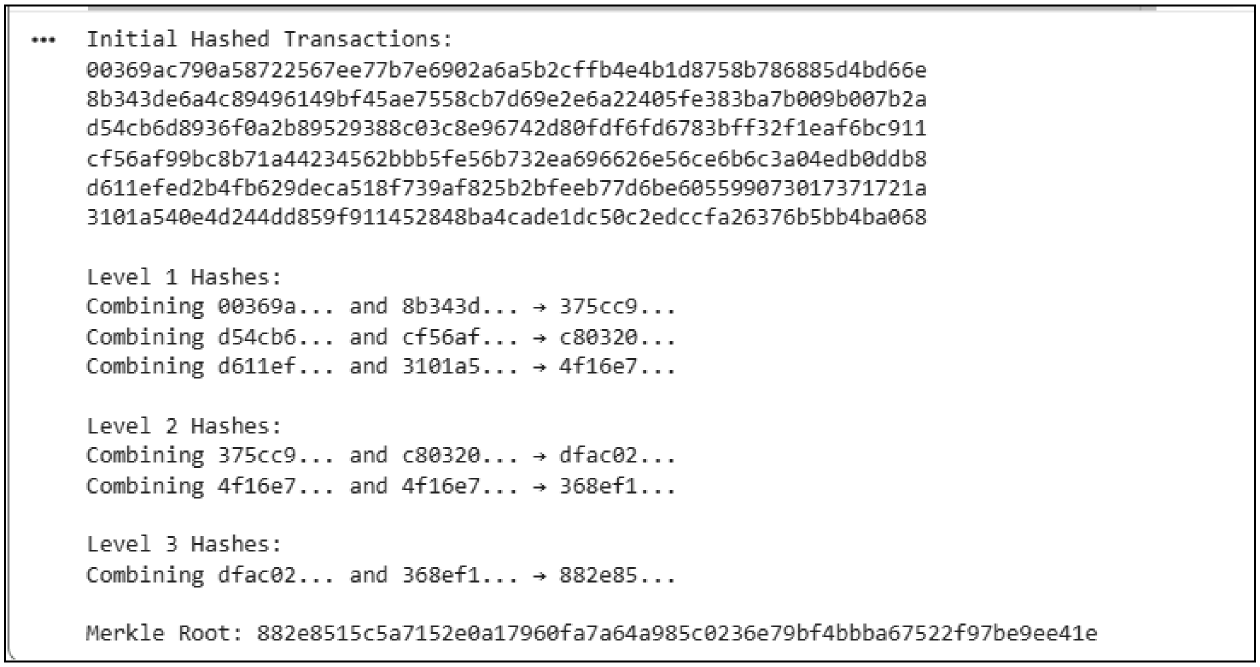
```
Enter data: computer science
Enter difficulty level (number of leading zeros): 5
Nonce Found: 357478
Hash: 00000b2a88d8c1957eaaa26961715b9f7839da65b53b9d97492dd19bc85e303b
```

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 build_merkle_tree(transactions):
    if len(transactions) == 0:
        return None
# Initial hashing of transactions
    hashed_transactions = [
        hashlib.sha256(t.encode('utf-8')).hexdigest()
        for t in transactions
    ]
 print("Initial Hashed Transactions:")
    for h in hashed_transactions:
        print(h)
  current_level = hashed_transactions
     level = 1
  while len(current_level) > 1:
        print(f"\nLevel {level} Hashes:")
# Duplicate last hash if odd count
        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 {current_level[i+1][:6]}... "
                  f"→ {hash_combined[:6]}...")

        current_level = new_level
        level += 1
return current_level[0]
transactions = [
    "User1 pays User2 120 BTC",
    "User3 pays User4 75 BTC",
    "User5 pays User6 200 BTC",
    "User7 pays User8 60 BTC",
    "User9 pays User10 90 BTC",
    "User11 pays User12 40 BTC"
]
merkle_root = build_merkle_tree(transactions)
print("\nMerkle Root:", merkle_root)
```



```
Initial Hashed Transactions:
00369ac790a58722567ee77b7e6902a6a5b2cffb4e4b1d8758b786885d4bd66e
8b343de6a4c89496149bf45ae7558cb7d69e2e6a22405fe383ba7b009b007b2a
d54cb6d8936f0a2b89529388c03c8e96742d80fdf6fd6783bff32f1eaf6bc911
cf56af99bc8b71a44234562bbb5fe56b732ea696626e56ce6b6c3a04edb0ddb8
d611efed2b4fb629deca518f739af825b2bfeeb77d6be605599073017371721a
3101a540e4d244dd859f911452848ba4cade1dc50c2edccfa26376b5bb4ba068

Level 1 Hashes:
Combining 00369a... and 8b343d... → 375cc9...
Combining d54cb6... and cf56af... → c80320...
Combining d611ef... and 3101a5... → 4f16e7...

Level 2 Hashes:
Combining 375cc9... and c80320... → dfac02...
Combining 4f16e7... and 4f16e7... → 368ef1...

Level 3 Hashes:
Combining dfac02... and 368ef1... → 882e85...

Merkle Root: 882e8515c5a7152e0a17960fa7a64a985c0236e79bf4bbba67522f97be9ee41e
```

Conclusion

This experiment successfully demonstrated the use of SHA-256 hashing and Merkle Tree construction in blockchain technology. It showed how cryptographic hash functions ensure data integrity, security, and immutability. The Merkle Tree efficiently summarizes all transactions into a single Merkle Root, enabling fast verification and easy detection of tampering. Thus, SHA-256 and Merkle Trees play a vital role in maintaining the reliability and security of blockchain systems.