Ria Chaudhari D20A14

Blockchain Lab Exp 1

AIM: Cryptography in Blockchain, Merkle root Tree Hash

Theory:

1.Cryptographic Hash functions in Blockchain

A **cryptographic hash function** is a mathematical algorithm that converts input data of any size into a fixed-length output (called a *hash* or *digest*).In blockchain, the most commonly used hash function is **SHA-256**.

Properties of Cryptographic Hash Functions

Deterministic – Same input always produces the same hash.
Fixed Output Size – No matter the input size, output length is fixed.
Pre-image Resistance – Cannot reverse the hash to find original input.
Collision Resistance – Hard to find two different inputs with the same hash.
Avalanche Effect – Small input change → completely different hash.

2.What is a Merkle Tree?

A Merkle Tree (Hash Tree) is a tree structure where:

- Each leaf node is a hash of a transaction.
- Each non-leaf node is a hash of its two child hashes.
- The top hash is called the Merkle Root.

It is used to efficiently verify data integrity in blockchain.

3.What is a Cryptographic Puzzle and explain the Golden Nonce

A cryptographic puzzle is a mathematical problem that:

- Is hard to solve
- Is easy to verify

In blockchain, this is used in Proof-of-Work (PoW).

Miners must find a number called a nonce such that:

```
Hash(Block Data + Nonce) → starts with required number of zeros
```

Example:

```
0000ab34f567...
```

The difficulty increases as required leading zeros increase.

What is the Golden Nonce?

The Golden Nonce is the special nonce value that solves the cryptographic puzzle.

It is the number that makes the block hash satisfy the required difficulty condition.

Example:

```
Block data + 45231 → hash starts with 00000
```

Here, `45231` is the Golden Nonce.

Miners try millions of nonces until they find this one.

4.How does a Merkle Tree work?

Let's say we have 4 transactions:`T1, T2, T3, T4`

Step 1: Hash all transactions

```
H1 = hash(T1)
H2 = hash(T2)
H3 = hash(T3)
H4 = hash(T4)
```

Step 2: Combine pairwise and hash again

```
H12 = hash(H1 + H2)
H34 = hash(H3 + H4)
```

Step 3: Final combination

```
Merkle Root = hash(H12 + H34)
```

If the number of transactions is odd → duplicate the last transaction.

This final hash (Merkle Root) is stored inside the block header.

5.Benefits of Merkle Tree

1. Efficient Verification:You don't need all transactions to verify one transaction.

2. Saves Memory:Lightweight nodes (SPV nodes) only store block headers.

3. Fast Detection of Tampering:If one transaction changes → Merkle Root changes.

4. Scalability:Efficient even with thousands of transactions.

6.Use cases of Merkle Tree

1. Blockchain (Bitcoin, Ethereum):Used to store transaction hashes.

2. Distributed Systems:Used in IPFS and distributed storage verification.

3. Git Version Control:Git uses Merkle Trees to track file changes.

4. Data Integrity Verification:Used to verify large datasets efficiently.

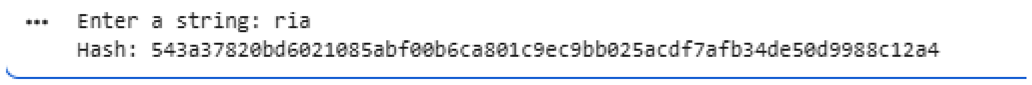
5. Peer-to-Peer Networks:Ensures file chunks are not modified.

Codes:

1.Python program that uses the hashlib library to create the hash of a given string:

```
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)
```

Output:

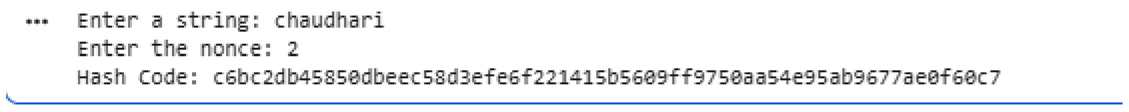


```
Enter a string: ria
Hash: 543a37820bd6021085abf00b6ca801c9ec9bb025acdf7afb34de50d9988c12a4
```

2.Program to generate required 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)
```

Output:

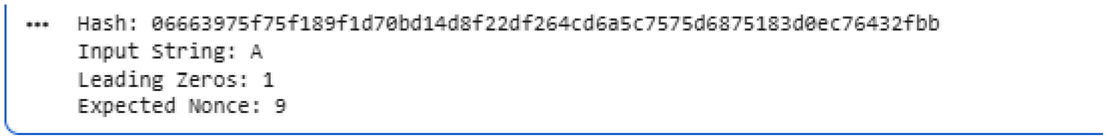


```
Enter a string: chaudhari
Enter the nonce: 2
Hash Code: c6bc2db45850dbeec58d3efe6f221415b5609ff9750aa54e95ab9677ae0f60c7
```

3.Python code for Solving Puzzle for leading zeros with expected nonce and given string

```
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 = "A"
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)
```

Output:

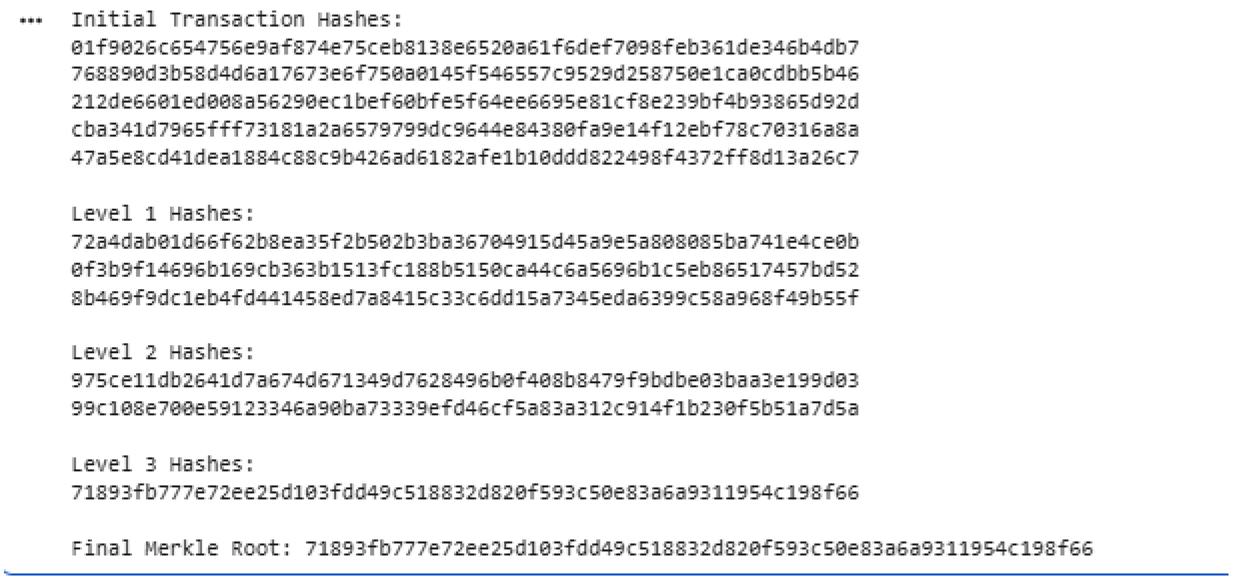


```
Hash: 06663975f75f189f1d70bd14d8f22df264cd6a5c7575d6875183d0ec76432fbb
Input String: A
Leading Zeros: 1
Expected Nonce: 9
```

4.Generating Merkle Tree for given set of Transactions

```
import hashlib

def build_merkle_tree(transactions):
    if len(transactions) == 0:
        return None

    # Step 1: Hash all transactions first
    print("Initial Transaction Hashes:")
    transactions = [
        hashlib.sha256(tx.encode('utf-8')).hexdigest()
        for tx in transactions
    ]
    for h in transactions:
        print(h)
    print()
  # Step 2: Build the Merkle Tree
    level = 1
    while len(transactions) > 1:
        print(f"Level {level} Hashes:")

        if len(transactions) % 2 != 0:
            transactions.append(transactions[-1])

        new_transactions = []

        for i in range(0, len(transactions), 2):
            combined = transactions[i] + transactions[i+1]
            hash_combined =
hashlib.sha256(combined.encode('utf-8')).hexdigest()
            print(hash_combined)
            new_transactions.append(hash_combined)

        print()
        transactions = new_transactions
        level += 1

    return transactions[0]
# Example usage with valid transactions
transactions = [
    "Ria -> Sneha : $200",
    "Bob -> Dave : $500",
    "Dave -> Ria : $100",
    "Eve -> Alice : $300",
    "Ria -> Bob : $50"
]
merkle_root = build_merkle_tree(transactions)
print("Final Merkle Root:", merkle_root)
```

Output:



```
Initial Transaction Hashes:
01f9026c654756e9af874e75ceb8138e6520a61f6def7098feb361de346b4db7
768890d3b58d4d6a17673e6f750a0145f546557c9529d258750e1ca0cdbb5b46
212de6601ed008a56290ec1bef60bfe5f64ee6695e81cf8e239bf4b93865d92d
cba341d7965fff73181a2a6579799dc9644e84380fa9e14f12ebf78c70316a8a
47a5e8cd41dea1884c88c9b426ad6182afe1b10ddd822498f4372ff8d13a26c7

Level 1 Hashes:
72a4dab01d66f62b8ea35f2b502b3ba36704915d45a9e5a808085ba741e4ce0b
0f3b9f14696b169cb363b1513fc188b5150ca44c6a5696b1c5eb86517457bd52
8b469f9dc1eb4fd441458ed7a8415c33c6dd15a7345eda6399c58a968f49b55f

Level 2 Hashes:
975ce11db2641d7a674d671349d7628496b0f408b8479f9bdbe03baa3e199d03
99c108e700e59123346a90ba73339efd46cf5a83a312c914f1b230f5b51a7d5a

Level 3 Hashes:
71893fb777e72ee25d103fdd49c518832d820f593c50e83a6a9311954c198f66

Final Merkle Root: 71893fb777e72ee25d103fdd49c518832d820f593c50e83a6a9311954c198f66
```

Conclusion:

Cryptographic hash functions secure blockchain data by ensuring integrity and immutability. Merkle Trees efficiently organize and verify large numbers of transactions using a single Merkle Root.Cryptographic puzzles in Proof-of-Work maintain network security by requiring computational effort.Together, these mechanisms make blockchain systems secure, transparent, and tamper-resistant.