Atharva Mulam
Div: D15A INFT
Roll No: 37

Blockchain Experiment 1

Aim: Cryptography in Blockchain, Merkle root Tree Hash

Task to be performed :
Make a copy of this Google Colab Notebook
Try to solve the errors in each of the 4 Programs
In the 4th Program - Constructing a Merkle Tree Root Hash, modify the code as follows:
Update the transactions list with valid entries.

Sample Transactions to be considered
-- T1 : Alice → Bob : $200;
-- T2 : Bob → Dave : $500;
-- T3 : Dave → Eve : $100
-- T4 : Eve → Alice : $300;
-- T5 : Roo → Bob : $50

Hash the transactions before combining them in the for-loop
-- Print all the intermediate hash during the construction of the Merkle Tree Root Hash

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

XXXXXXX #error statement; 3 Error

```
[ ]  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: Atharva
Hash: 51056b6a6a7b468483de6de32b530b482ab397dc83ec34c2ebdcf24bf6b4321d
```

Program 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)
```

```
Enter a string: Atharva
Enter the nonce: 1
Hash Code: 183b60a68d42a78344f533b262121a47733195b9b38de084636aabe6ef9b4073
```

Program 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_code):
            print("Hash:",hash_code )
            return nonce

        nonce += 1

# Get user input
input_string = "Atharva"
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: edf22d1ec0ff111186f057339320e33f05894e5cc7052d5e3acc5c53214cba91
Input String: Atharva
Leading Zeros: 1
Expected Nonce: 0
```

Program 4) Generating Merkle Tree for given set of Transactions

```python
import hashlib

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

    if len(transactions) == 1:
        return transactions[0]

    # Recursive construction of the Merkle Tree
    while len(transactions) > 1:
        if len(transactions) % 2 != 0:
            transactions.append(transactions[-1])

        new_transactions = []
        for i in range(0, len(transactions), 2):
            # Hash individual transactions
            hash_left = hashlib.sha256(transactions[i].encode('utf-8')).hexdigest()
            hash_right = hashlib.sha256(transactions[i+1].encode('utf-8')).hexdigest()
            # Combine hashes of left and right transactions
            combined = hash_left + hash_right
            # Calculate hash of combined hashes
            hash_combined = hashlib.sha256(combined.encode('utf-8')).hexdigest()
            print("Intermediate Hash:", hash_combined)
            new_transactions.append(hash_combined)

        transactions = new_transactions

    return transactions[0]

# Transactions
transactions = ['Alice -> Bob : $200', 'Bob -> Dave : $500', 'Dave -> Eve : $100', 'Eve -> Alice : $300', 'Roo -> Bob : $50']

merkle_root = build_merkle_tree(transactions)
print("Merkle Root:", merkle_root)
```

```
Intermediate Hash: 471608d3d6f2a642ff3a84dfaeeaebb1c271a9aed5b37a82d61c4784558e80e9
Intermediate Hash: a61c504d72e6ce69b2ec8791ff7469cc004704f727f8dbbf420fadbc4e4c0de7
Intermediate Hash: 314ca746b3dafe63a1e0b939ec3042df6270de7703aa0eb7e699a3f2c2e7c18e
Intermediate Hash: df1c179b63a628649527891307e58f637f5a54852315a667e66496c25f6f25e5
Intermediate Hash: e379f69aa639e6c8fde8e6f87031fd4c6d59cb4f906fb9bb579a23e918f43b17
Intermediate Hash: f13d86b724264acb22a2d35d8def30f577ed2b58b59f5643504da696783c6fdd
Merkle Root: f13d86b724264acb22a2d35d8def30f577ed2b58b59f5643504da696783c6fdd
```

Conclusion: In this experiment we understood the construction of a Merkle Tree Root Hash involves iteratively hashing individual transactions to form a tree structure and syntactical errors were fixed.