Netaji Subhas University of Technology

Sector - 3, Dwarka, Delhi-110075

Practical file

INITE52 : Blockchain

Submitted by :

Prashant Sharma	2021UIN3317
Vinayak	2021UIN3319
Sunaina	2021UIN3310
Gaurav kumar	2021UIN3312
Dhruv kumar singh	2021UIN3324

Submitted to : satish kumar singh

Practical 1

Aim : Design a Merkle tree for structured and unstructured data using SHA-256/SHA-512.

Merkle Tree Basics

- Merkle Tree Structure: A Merkle Tree is a binary tree where each non-leaf node is a hash of its child nodes.
- Leaves: The leaf nodes are hashes of individual data blocks (transactions in blockchain).
- Parent Nodes: Non-leaf nodes are hashes of concatenated child nodes.

PSEUDOCODE:- Merkle Tree for unstructured Data

```python
import hashlib

def sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()

# Given unstructured data
data = ["Data1", "Data2", "Data3", "Data4"]

# Step 1: Generate leaf hashes
leaf_hashes = [sha256(d) for d in data]

# Step 2: Build tree by hashing in pairs
while len(leaf_hashes) > 1:
    # Pair adjacent nodes and hash together
    temp_hashes = []
    for i in range(0, len(leaf_hashes), 2):
        # If odd number of nodes, duplicate the last one
        left = leaf_hashes[i]
        right = leaf_hashes[i+1] if i+1 < len(leaf_hashes) else leaf_hashes[i]
        temp_hashes.append(sha256(left + right))
    leaf_hashes = temp_hashes

# Final root hash
root_hash = leaf_hashes[0]
print("Merkle Root Hash:", root_hash)
```

PSEUDOCODE:- Merkle Tree for Structured Data

Steps:-

1. Hash each field and transaction individually.
2. Combine hashes in pairs, creating parent nodes until only the root hash remains.

main.cpp Share Run

```
import hashlib

def sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()

# Example structured data
structured_data = {
    "name": "Alice",
    "email": "alice@example.com",
    "balance": "1000",
    "transactions": [{"id": "1", "amount": "250"}, {"id": "2", "amount": "300"}]
}
# Step 1: Generate hash for each field
hashes = []
for key, value in structured_data.items():
    if isinstance(value, list):  # handle transactions array separately
        for transaction in value:
            transaction_hash = sha256(str(transaction))
            hashes.append(transaction_hash)
    else:
        field_hash = sha256(f"{key}:{value}")
        hashes.append(field_hash)

# Step 2: Build tree by hashing in pairs
while len(hashes) > 1:
    temp_hashes = []
    for i in range(0, len(hashes), 2):
        left = hashes[i]
        right = hashes[i+1] if i+1 < len(hashes) else hashes[i]
        temp_hashes.append(sha256(left + right))
    hashes = temp_hashes

# Final root hash
root_hash = hashes[0]
print("Merkle Root Hash:", root_hash)
```

Practical 2

Aim : Design a Block Structure for Healthcare, Agriculture and Financial Transaction

1)Healthcare Block Structure:

- **Block Header**
  - **Previous Block Hash**: Links to the previous block in the chain.
  - **Timestamp**: Records the creation time of this block.
  - **Merkle Root**: Root hash summarizing all patient-related transactions in the block.
  - **Nonce**: Number used for the proof-of-work consensus.
- **Block Body**
  - **Patient ID**: Encrypted identifier for patient privacy.
  - **Doctor ID**: Identifier for the attending doctor.
  - **Treatment Details**: Encrypted information about diagnoses, procedures, or treatments.
  - **Visit Date**: Date of the healthcare visit.
  - **Lab Reports Hash**: Hash of any related lab or medical test results.

2)Agriculture Block Structure:

- **Block Header**
  - **Previous Block Hash: Link to the previous block in the chain.**
  - **Timestamp: Records the data collection time.**
  - **Merkle Root: Root hash summarizing agricultural data in this block.**
  - **Nonce: Number for validating consensus.**
- **Block Body**
  - **Farm ID: Unique identifier for the farm or farming entity.**
  - **Crop Type: Type of crop grown (e.g., wheat, rice).**
  - **Planting Date: Date the crop was planted.**
  - **Harvest Date: Date of crop harvesting.**
  - **Pesticide Usage: Record of any pesticides used.**

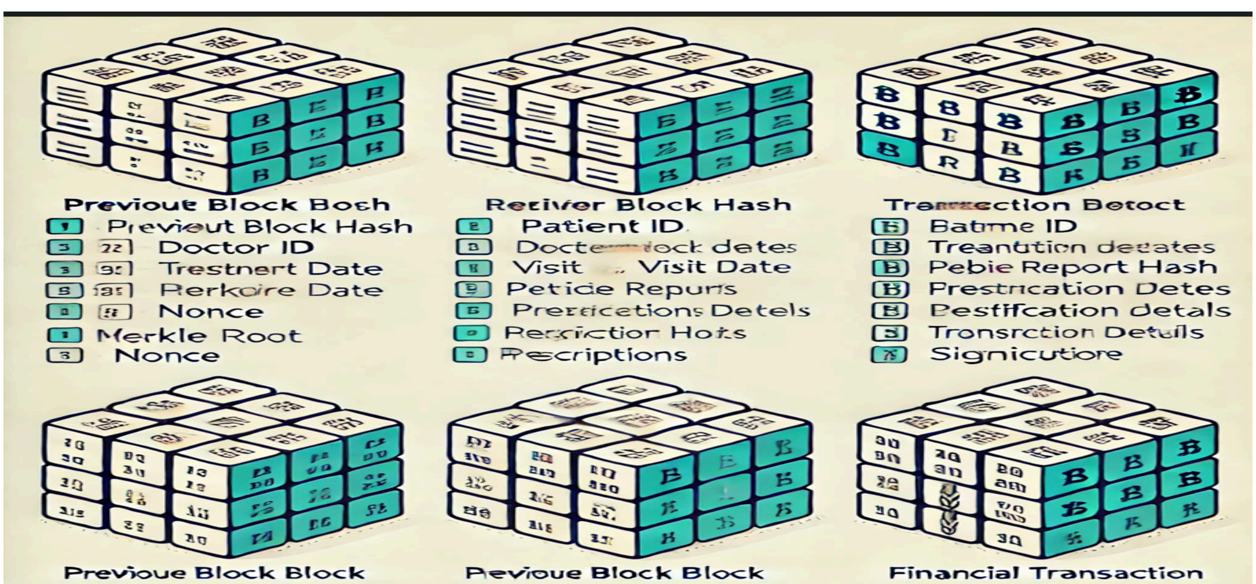
Fertilizer Details: Information on fertilizers applied.

3). Financial Transaction Block Structure

For financial transactions, the block stores sensitive data about transfers, payments, or credits, ensuring that all transactions are secure, transparent, and verifiable.

Financial Transaction Block Structure:

- **Block Header**
    - **Previous Block Hash: Links to the previous block in the chain.**
    - **Timestamp: Time of the transaction recording.**
    - **Merkle Root: Root hash summarizing all financial transactions in this block.**
    - **Nonce: Value used in the consensus process.**
- **Block Body**
    - **Transaction ID: Unique identifier for each transaction.**
    - **Sender Account: Encrypted identifier for the sender's account.**
    - **Receiver Account: Encrypted identifier for the receiver's account.**
    - **Amount: Amount transferred in the transaction.**
    - **Transaction Type: Specifies the transaction type (e.g., credit, debit, transfer).**
    - **Transaction Fee: Any fees associated with the transaction.**
    - **Signature: Digital signature for verifying authenticity.**



Practical 3

AIM :- Blockchain Implementation using Single node(Creation, Validation and Authentication by individually).

1. Block Structure

Each block in the blockchain will have:

- **Index: Position of the block in the chain.**
- **Timestamp: Time at which the block was created.**
- **Data: The transaction or data the block holds.**
- **Previous Hash: Hash of the previous block in the chain (providing immutability).**
- **Hash: The hash of the current block, used for validation and authentication.**

2. Blockchain Class

This class manages the chain, validates the chain, and ensures that blocks are added only when validated.

3. Block Class

This will represent each individual block, including how it calculates the hash.

4. Validation

A function that checks if the current chain is valid by verifying:

- **Each block's hash.**
- **The `previous_hash` value.**
- **That each block's hash matches its data.**

5. Authentication

This involves using digital signatures (RSA or ECC) to ensure that data within a block has been authenticated and hasn't been tampered with. For simplicity, we will use SHA256 here for hashing, but in a real-world application, you'd implement a public/private key system for more secure authentication.

main.cpp Share Run

```
import json
import time
from datetime import datetime

class Block:
    def __init__(self, index, previous_hash, timestamp, data, hash_value):
        self.index = index
        self.previous_hash = previous_hash
        self.timestamp = timestamp
        self.data = data
        self.hash_value = hash_value

    def calculate_hash(self):
        # Returns the hash of the block using SHA256
        block_string = f"{self.index}{self.previous_hash}{self.timestamp}{json.dumps(self.data)}"
        return hashlib.sha256(block_string.encode('utf-8')).hexdigest()

class Blockchain:
    def __init__(self):
        self.chain = []
        self.create_genesis_block()

    def create_genesis_block(self):
        # Create the first block (genesis block)
        genesis_block = Block(0, "0", str(datetime.now()), {"data": "Genesis Block"}, "0")
        genesis_block.hash_value = genesis_block.calculate_hash()
        self.chain.append(genesis_block)

    def add_block(self, data):
        last_block = self.chain[-1]
        new_index = last_block.index + 1
        new_timestamp = str(datetime.now())
        new_hash = ""
        new_block = Block(new_index, last_block.hash_value, new_timestamp, data, new_hash)

```

main.cpp Share Run

```
        # Calculate the hash of the new block
        new_block.hash_value = new_block.calculate_hash()

        self.chain.append(new_block)

    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Check if current block's hash is correct
            if current_block.hash_value != current_block.calculate_hash():
                print(f"Invalid hash at block {current_block.index}")
                return False

            # Check if the current block's previous hash is the same as the previous block's hash
            if current_block.previous_hash != previous_block.hash_value:
                print(f"Invalid previous hash at block {current_block.index}")
                return False

        return True

    def authenticate_block(self, block_index, signature):
        block = self.chain[block_index]
        block_data = json.dumps(block.__dict__, sort_keys=True)
        block_hash = hashlib.sha256(block_data.encode('utf-8')).hexdigest()

        # Simulate authentication: In real life, this would compare the hash with a signature
        # generated from a private key.
        if block_hash == signature:
            return True
        else:
            print(f"Authentication failed for block {block_index}")
            return False
```

```

# Testing the Blockchain

blockchain = Blockchain()

# Add a few blocks
blockchain.add_block({"data": "First Block Data"})
blockchain.add_block({"data": "Second Block Data"})

# Validate the blockchain
print("Blockchain valid:", blockchain.is_chain_valid())

# Simulate authentication (use the hash as a placeholder for signature)
block_data = json.dumps(blockchain.chain[1].__dict__, sort_keys=True)
block_signature = hashlib.sha256(block_data.encode('utf-8')).hexdigest()

# Authenticate block 1
print("Block 1 authentication:", blockchain.authenticate_block(1, block_signature))

```

Practical 4

Aim : To create an ERC20 Token

Code:

blockcahin_exp_3.py > ERC20Token > __init__

```python
class ERC20Token:
    def __init__(self, name, symbol, decimals, initial_supply):
        self.name = name
        self.symbol = symbol
        self.decimals = decimals
        self.total_supply = initial_supply * (10 ** decimals)
        self.balances = {}
        self.allowances = {}

        creator_address = "owner"
        self.balances[creator_address] = self.total_supply

    def balance_of(self, account):
        return self.balances.get(account, 0)

    def transfer(self, sender, recipient, amount):
        if self.balance_of(sender) < amount:
            return "Insufficient balance"

        self.balances[sender] = self.balance_of(sender) - amount
        self.balances[recipient] = self.balance_of(recipient) + amount
        return f"{amount} tokens transferred from {sender} to {recipient}"

    def approve(self, owner, spender, amount):
        if owner not in self.allowances:
            self.allowances[owner] = {}

        self.allowances[owner][spender] = amount
        return f"{spender} approved to spend {amount} tokens on behalf of {owner}"

    def allowance(self, owner, spender):
        return self.allowances.get(owner, {}).get(spender, 0)

    def transfer_from(self, spender, owner, recipient, amount):
        allowed_amount = self.allowance(owner, spender)

```

```python
        if allowed_amount < amount:
            return "Allowance exceeded"
        if self.balance_of(owner) < amount:
            return "Owner's balance is insufficient"

        self.balances[owner] = self.balance_of(owner) - amount
        self.allowances[owner][spender] -= amount
        self.balances[recipient] = self.balance_of(recipient) + amount
        return f"{amount} tokens transferred from {owner} to {recipient} by {spender}"


token = ERC20Token("MyToken", "MTK", 18, 1000)

print("Owner balance:", token.balance_of("owner"))

print(token.transfer("owner", "alice", 100))

print(token.approve("owner", "bob", 50))

print("Allowance of bob by owner:", token.allowance("owner", "bob"))

print(token.transfer_from("bob", "owner", "charlie", 30))

print("Owner balance:", token.balance_of("owner"))
print("Alice balance:", token.balance_of("alice"))
print("Charlie balance:", token.balance_of("charlie"))
```

Output:

```
PS C:\Workspace_V\New folder> python -u "c:\Workspace_V\New folder\blockcahin_exp_3.py"
Owner balance: 1000000000000000000000
100 tokens transferred from owner to alice
bob approved to spend 50 tokens on behalf of owner
Allowance of bob by owner: 50
30 tokens transferred from owner to charlie by bob
Owner balance: 999999999999999999870
Alice balance: 100
Charlie balance: 30
PS C:\Workspace_V\New folder>
```

Practical 5

Aim : Implement blockchain in Merkle Trees using at least 25 blocks.

Code:

blockchain_exp_4.py > MerkleTree > build_merkle_tree

```
import hashlib
import random

class MerkleTree:
    def __init__(self, transactions):
        self.transactions = transactions
        self.root = self.build_merkle_tree(transactions)

    def build_merkle_tree(self, transactions):
        if len(transactions) == 1:
            return transactions[0]

        next_level = []
        for i in range(0, len(transactions), 2):
            left = transactions[i]
            right = transactions[i + 1] if i + 1 < len(transactions) else transactions[i]
            next_level.append(self.hash_pair(left, right))

        return self.build_merkle_tree(next_level)

    @staticmethod
    def hash_pair(left, right):
        return hashlib.sha256((left + right).encode()).hexdigest()
```

```
class Block:
    def __init__(self, index, transactions, previous_hash):
        self.index = index
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.merkle_tree = MerkleTree(transactions)
        self.merkle_root = self.merkle_tree.root
        self.block_hash = self.calculate_block_hash()

    def calculate_block_hash(self):
        data = str(self.index) + self.previous_hash + self.merkle_root
        return hashlib.sha256(data.encode()).hexdigest()

```

```python
class Blockchain:
    def __init__(self):
        self.chain = []
        self.create_genesis_block()

    def create_genesis_block(self):
        genesis_block = Block(0, ["Genesis Transaction"], "0" * 64)
        self.chain.append(genesis_block)

    def add_block(self, transactions):
        previous_hash = self.chain[-1].block_hash
        new_block = Block(len(self.chain), transactions, previous_hash)
        self.chain.append(new_block)

    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            if current_block.block_hash != current_block.calculate_block_hash():
                return False

            if current_block.previous_hash != previous_block.block_hash:
                return False

        return True

```

```python
blockchain = Blockchain()

for i in range(1, 26):
    transactions = [f"Tx {random.randint(1, 100)}" for _ in range(4)]  # Each block has 4 transactions
    blockchain.add_block(transactions)

for block in blockchain.chain:
    print(f"Block {block.index} | Merkle Root: {block.merkle_root} | Block Hash: {block.block_hash}")

print("\nIs blockchain valid?", blockchain.is_chain_valid())

```

Output:

```
PS C:\Workspace_V\New folder> python -u "c:\Workspace_V\New folder\blockchain_exp_4.py"
Block 0 | Merkle Root: Genesis Transaction | Block Hash: b8d311dafb1c901248ee862d4881a68bc4af2f1454370fc25af4dc290c866f64
Block 1 | Merkle Root: 4be60b2d18815ba4ff46558d2af680fa1b5977f1938c10822c598685b6c44555 | Block Hash: 7737a70f432ba9d8ec25334387a9df67e45bb6634ce417fef5ad91b6eb7b86c9
Block 2 | Merkle Root: 1339fcea68ed33b795a2bc06d83c8ba40ce9a2866febece7868a7381ab32394d | Block Hash: 6b4a4d03eedba6303798d9a27f49f147eb271a1e203cbb8db0803d36f02b4a06
Block 3 | Merkle Root: adb89cab141ee23d1ab06894d52bba0b79f36f636f0a8b7287b460290334a474 | Block Hash: 648b0450127b60efd1661340531669826f5b40a23569d0c628dcadf1b2174892
Block 4 | Merkle Root: 0dbff941bf9ebbd9346badf7dd8fba09ccb49dd2c414eb418d7b13c0c13a94d4 | Block Hash: 93fe99a7fb0543800c9776bd432d81b2125886d6cfa3150bd7194cc3c701678b
Block 5 | Merkle Root: 8109e5aa97a571a5f2656be7e6b7e36c40cdb29feffadd19684a38a53fe3deb5 | Block Hash: e7ead8f5df741e796a9ddd7c359df24bd833054a7fb2726223f09395e2034ef9
Block 6 | Merkle Root: d6304dea545b865b0e8ad71cb48ae43f73d33c1b5afb4dceabdc57985276620d | Block Hash: f2ea830e4b1ada5fdb5c2deb163eaf9a0281493bd9bef62161aa8369d62e8d3c
Block 7 | Merkle Root: 2bec6bc161bfab766b497a080d6c107a36545443447a075481caef06b92402a6 | Block Hash: 3a6f9d2df32a55f1957da92a339ae651c4b63cd37c563b03406a70b283d346cc
Block 8 | Merkle Root: ee31266814192e2221c277beb3f772a1e55331559f86bb6f61274c709da22d8d | Block Hash: 27c093d3460eee2121c2c308beb546a8f079718deee39768c28ce2d04651614d
Block 9 | Merkle Root: 1f99440d1b6a0e5b5215e3e6e08595870086b0c2588cd1f7cdc3ab124d1fa8c1 | Block Hash: 7496acc2af25c033936e1e955ef54ea4448716d63081b0858acaa4e77e650cef
Block 10 | Merkle Root: d0a37f484c9b88a1b982c64b3d66f176c2e1e06a5fbe889e64d7cc0bda3eb6a4 | Block Hash: d34bb4afdd2083bee42458e3d3fc3d83e39ca22cd4f09d19a3d9afd43c5717a9
Block 11 | Merkle Root: e37fa2ff0312f74191e12e28a1b7efd27d5ebe17aa0c6ebd145cc008cad97f74 | Block Hash: 52d78c2f9e3c6a722139882e6fea72e6421019dc759238690721a8af1e4a6386
Block 12 | Merkle Root: 7609a389689b6186e3a0bda7071fe982540cc514c40138da2021e77fb9a263bb | Block Hash: f3ea0cb90748fda0c10bdaa4ffcb787669f4ddccf6f5c1c853dadf3ee3e26412
Block 13 | Merkle Root: 130b59a78caa1e77311c9dac36eb961426d9fc10a385c6e444d8da83ab00643f | Block Hash: 089d0939d8ec8dc7cd35d8fe82ebd9f80b1b3e3f2abe2c47078a3376b6cf731d
Block 14 | Merkle Root: 34f4ea95441574fbeb79d0ebc017be7fed1cc219853276041a0f8ed36139b6ef | Block Hash: 44a0d1e5ca41f40c1bca3684776a9966c65d4d77cc0777a0a4cc5901dc681883
Block 15 | Merkle Root: de6e35664cb5c7aea485c4f726887d7f8690ea04b302d8654424f30403d3ba2b | Block Hash: 8e0478cef373518a11908f57fb7792bba56e92bd7538ada43b4667ffcdcf559a
Block 16 | Merkle Root: e98ff14106651bd5371ede5a2f0c390ea3509cbf5c31675fb4c92602c8a710aa | Block Hash: c03bd927184a1e7b6db4a199520a250261767f845fbff4aed60835f7661cb586
Block 17 | Merkle Root: bdbab8df617e48a2cd5b7cc60751742b1b6601cbfaa85f2cc3dccd786dbd1e01 | Block Hash: cb1bafb1c4b1a7d7c74b6c3d39f5868b5ba82c5d30655dd0303c4573ea92da12
Block 18 | Merkle Root: 3aae820c90511229c86532fb0cc7d06c66c6ac4876e7e9d5f97a5108d9aaa70a | Block Hash: 8ce8e3f41dc726f507dd537b61a585b26628c2803b49ff600762979dde8d1852
Block 19 | Merkle Root: 6f547da7d600a142ddb7160f9eac1261ced8cba257557b5caaeb95dd3536847b | Block Hash: aeacc02d56b19b6361fb03c78471431537812c8d32698e4a128fbec99c178413
Block 20 | Merkle Root: 82291f2e88a52887e4496ce38a1d8a94e12297de2b732fced598af1feffb261d | Block Hash: a48c20e0370635d2be4701bc2d58777697f506ca3cab2c02c32f6e7dff686045
Block 21 | Merkle Root: 78aca7b9b92d4031bac2cab6c7a30fea547193532a3ed891e41dfc6cfcc82673 | Block Hash: c0a4b07dcaf5ca47f54b975d7a5ef27b176504a658b62812b70a38f9adcb17d7
Block 22 | Merkle Root: ad80d70bfe732e8760ca3076fe284037e8b39ee7862df65f86e7e5a09808e27b | Block Hash: 3d5f8fc44a48eaad25c140f16f8dbeafd0937022c528665ff67be63319a4683c
Block 23 | Merkle Root: a83da000448f05db475c62cac46107c86dbc43cce5a15a5faa4fee02dbd7dd27 | Block Hash: e11f01c9e7ec9c16e73679ec8d5ed968686ad4f12a580159a12c5c42e96f0b45
Block 24 | Merkle Root: 3ff717f73f83de888df5c382f0d207569af75c5516cf8981790c86fa7c2cb37e | Block Hash: 1506bd9d9a2a65dc16e2a471b14fe80e9ea77bf4cfda80a040612d8391d5fb7f
Block 25 | Merkle Root: fe292cf2333fa6edfc171e849ed55ea3ce5953f14f79f470cf13580453b01088 | Block Hash: d27e18db9789d7af99061587955d4da33509a5ac8e27369c557f923fb23048f4

Is blockchain valid? True
PS C:\Workspace_V\New folder>
```

Practical 6

Aim : Implement Mining using Proof-of-Work and Proof-of-Authority in blockchain

Code:

```
import hashlib
import time
import random

class Block:
    def __init__(self, index, transactions, previous_hash, timestamp=None):
        self.index = index
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.timestamp = timestamp or time.time()
        self.nonce = 0  # Used for Proof of Work
        self.block_hash = None  # Set later during mining

    def calculate_hash(self):
        data = str(self.index) + self.previous_hash + str(self.timestamp) + str(self.transactions) + str(self.nonce)
        return hashlib.sha256(data.encode()).hexdigest()

class Blockchain:
    def __init__(self, difficulty=4, authorities=None):
        self.chain = []
        self.difficulty = difficulty
        self.authorities = authorities or ["Authority1", "Authority2"]  # For PoA
        self.create_genesis_block()

    def create_genesis_block(self):
        genesis_block = Block(0, ["Genesis Block"], "0" * 64)
        genesis_block.block_hash = genesis_block.calculate_hash()
        self.chain.append(genesis_block)
```

```
    def proof_of_work(self, block):
        target = "0" * self.difficulty
        while True:
            hash_result = block.calculate_hash()
            if hash_result[:self.difficulty] == target:
                block.block_hash = hash_result
                return block
            else:
                block.nonce += 1

    def proof_of_authority(self, block):
        authority = random.choice(self.authorities)
        block.block_hash = block.calculate_hash() + authority
        return block

    def add_block(self, transactions, consensus="PoW"):
        previous_hash = self.chain[-1].block_hash
        new_block = Block(len(self.chain), transactions, previous_hash)

        if consensus == "PoW":
            print(f"Mining block {new_block.index} with Proof of Work...")
            mined_block = self.proof_of_work(new_block)
        elif consensus == "PoA":
            print(f"Validating block {new_block.index} with Proof of Authority...")
            mined_block = self.proof_of_authority(new_block)

        self.chain.append(mined_block)
```

```python
    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            if current_block.block_hash != current_block.calculate_hash()[:len(current_block.block_hash)]:
                return False
            if current_block.previous_hash != previous_block.block_hash:
                return False
        return True

blockchain = Blockchain(difficulty=4)

for i in range(3):
    blockchain.add_block([f"Transaction {i + 1}"], consensus="PoW")

for i in range(3, 6):
    blockchain.add_block([f"Transaction {i + 1}"], consensus="PoA")

for block in blockchain.chain:
    print(f"Block {block.index} | Hash: {block.block_hash} | Previous Hash: {block.previous_hash}")

print("\nIs blockchain valid?", blockchain.is_chain_valid())
```

Output:

```
PS C:\Workspace_V\New folder> python -u "c:\Workspace_V\New folder\blockchain_exp_6.py"
Mining block 1 with Proof of Work...
Mining block 2 with Proof of Work...
Mining block 3 with Proof of Work...
Validating block 4 with Proof of Authority...
Validating block 5 with Proof of Authority...
Validating block 6 with Proof of Authority...
Block 0 | Hash: a6bedb7b0387316c0fab926bf19978a84054e402e8283dc2341ba5cbf9c28b2a | Previous Hash: 0000
000000000000000000000000000000000000000000000000000000000000
Block 1 | Hash: 00008576d2d7ced4498a1e491d3a5fbc3c29165d709564a6601abb38b6f1ecbd | Previous Hash: a6be
db7b0387316c0fab926bf19978a84054e402e8283dc2341ba5cbf9c28b2a
Block 2 | Hash: 0000687784bbb8f14caa9332a9538a8bc540438d23fc2e633dc367b5017c7cca | Previous Hash: 0000
8576d2d7ced4498a1e491d3a5fbc3c29165d709564a6601abb38b6f1ecbd
Block 3 | Hash: 0000a38017db103fee5c3f8ba43bc7603e8aa10037bed2b198f4f335e5693a70 | Previous Hash: 0000
687784bbb8f14caa9332a9538a8bc540438d23fc2e633dc367b5017c7cca
Block 4 | Hash: 62e2ea6dd8992482249c4ba6c682323a70908558aad251db54479125f5c0e64dAuthority2 | Previous
Hash: 0000a38017db103fee5c3f8ba43bc7603e8aa10037bed2b198f4f335e5693a70
Block 5 | Hash: 8cb9caeff7a36fa4cb4c2fc0d5d2e252c3889f8eebc03e5f43bdede0d2157936Authority2 | Previous
Hash: 62e2ea6dd8992482249c4ba6c682323a70908558aad251db54479125f5c0e64dAuthority2
Block 6 | Hash: f9acae87d3364be27a97cea18e013c79b3c24ae4daa3bd50756d357b727eec09Authority2 | Previous
Hash: 8cb9caeff7a36fa4cb4c2fc0d5d2e252c3889f8eebc03e5f43bdede0d2157936Authority2

Is blockchain valid? False
PS C:\Workspace_V\New folder>
```

Practical 7

Aim : Implement peer-to-peer network of n nodes for deploying in Blockchain.

Code:

```
import socket
import threading
import json

class Node:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.peers = []  # List of connected peers
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind((self.host, self.port))
        self.server.listen(5)
        print(f"Node started on {self.host}:{self.port}")

        # Start listening for incoming connections
        threading.Thread(target=self.listen_for_connections).start()

    def connect_to_peer(self, peer_host, peer_port):
        """Connect to another peer"""
        try:
            peer_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            peer_socket.connect((peer_host, peer_port))
            self.peers.append(peer_socket)
            print(f"Connected to peer {peer_host}:{peer_port}")
            threading.Thread(target=self.listen_to_peer, args=(peer_socket,)).start()
        except Exception as e:
            print(f"Failed to connect to peer {peer_host}:{peer_port} - {e}")

    def listen_for_connections(self):
        """Listen for incoming peer connections"""
        while True:
            client_socket, addr = self.server.accept()
            print(f"Received connection from {addr}")
            self.peers.append(client_socket)
            threading.Thread(target=self.listen_to_peer, args=(client_socket,)).start()

    def listen_to_peer(self, peer_socket):
        """Listen for messages from a peer"""
        while True:
            try:
                message = peer_socket.recv(1024).decode('utf-8')
                if message:
                    print("Received message:", message)
                    # Process received data (e.g., verify transactions)
            except:
                # Remove peer if connection is lost
                self.peers.remove(peer_socket)
                peer_socket.close()
                break
```

```
    def broadcast(self, message):
        """Broadcast message to all connected peers"""
        for peer in self.peers:
            try:
                peer.send(message.encode('utf-8'))
            except:
                # Remove peer if sending fails
                self.peers.remove(peer)
                peer.close()

    def create_transaction(self, data):
        """Create and broadcast a transaction"""
        transaction = json.dumps(data)
        self.broadcast(transaction)
        print(f"Broadcasted transaction: {transaction}")

# Example usage
if __name__ == "__main__":
    # Initialize nodes on different ports (run these in separate terminals for each node)
    node = Node('127.0.0.1', 5000)
    # To connect to other nodes, use `node.connect_to_peer('host', port)`
    # For example, node.connect_to_peer('127.0.0.1', 5001)
    # node.create_transaction({'sender': 'Alice', 'receiver': 'Bob', 'amount': 10})
```

Output:

Node 1 Terminal:

```
Node started on 127.0.0.1:5000
Connected to peer 127.0.0.1:5001
Received connection from ('127.0.0.1', 55002)
Received message: {"sender": "Alice", "receiver": "Bob", "amount": 10}
```

Practical 8

Aim : Study the Bitcoin Architecture and its opcode.

Overview:
Bitcoin Architecture :- The Bitcoin network is a decentralized P2P network where nodes (computers) communicate to validate and relay transactions, manage the blockchain (the ledger), and participate in mining.

Key Components

1. Blockchain:
   - The blockchain is a public, distributed ledger of all Bitcoin transactions. Each block contains a set of transactions, a timestamp, a reference to the previous block (hash), and a unique hash generated for the block.
   - Blocks are added to the blockchain in a linear sequence, and every block references the previous one, forming a chain.
2. Nodes:
   - Full nodes validate and relay transactions and blocks, ensuring that all rules (such as spending limits and block size) are followed. They also store a copy of the entire blockchain.
   - Lightweight or SPV (Simplified Payment Verification) nodes rely on full nodes to verify transactions.
3. Mining and Proof of Work:
   - Miners validate transactions and package them into blocks. They solve computationally intensive puzzles (Proof of Work) to secure the network and add new blocks.
   - When a miner finds a valid block, it broadcasts it to the network, and other nodes verify the block's validity before adding it to their copies of the blockchain.
4. Wallets:
   - Wallets store private and public keys, which users need to sign transactions and prove ownership of Bitcoin.

- Wallets interact with the Bitcoin network to send and receive transactions and manage balances.

5. Transactions:
    - Transactions are the movement of Bitcoin between addresses. Each transaction has inputs (sources of Bitcoin) and outputs (destinations).
    - Bitcoin uses a UTXO (Unspent Transaction Output) model, meaning each transaction consumes UTXOs from previous transactions as inputs and creates new UTXOs as outputs.

Common Opcodes in Bitcoin Script

1. **Data Manipulation Opcodes**:
    - `OP_DUP`: Duplicates the top item on the stack.
    - `OP_DROP`: Removes the top item from the stack.
    - `OP_SWAP`: Swaps the top two items on the stack.
2. **Arithmetic Opcodes**:
    - `OP_ADD`: Adds the top two items on the stack.
    - `OP_SUB`: Subtracts the top two items.
3. **Hashing Opcodes**:
    - `OP_HASH160`: Hashes the top stack item with SHA-256 followed by RIPEMD-160.
    - `OP_SHA256`: Hashes the top item with SHA-256.
4. **Logical and Conditional Opcodes**:
    - `OP_EQUAL`: Returns true if the top two items are equal.
    - `OP_VERIFY`: Fails the transaction if the top stack item is not `TRUE`.
    - `OP_IF` / `OP_ELSE` / `OP_ENDIF`: Conditional statements used to add logic.
5. **Cryptographic Opcodes**:
    - `OP_CHECKSIG`: Verifies a digital signature using a public key.
    - `OP_CHECKMULTISIG`: Verifies multiple signatures against multiple public keys, allowing for multi-signature transactions.

Practical 9

Aim : Write Bitcoin Script for the following:

a. Pay to pubkey hash

Locking Script (ScriptPubKey)
OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

`OP_DUP`: Duplicates the top item on the stack (the public key).
`OP_HASH160`: Applies the `RIPEMD-160` hash to the result of `SHA-256` hashing of the public key. This is what creates the Bitcoin address (PubKeyHash).
`<PubKeyHash>`: The actual public key hash (i.e., the recipient's Bitcoin address).
`OP_EQUALVERIFY`: Verifies that the top two items on the stack are equal (i.e., the public key hash matches the provided address).
`OP_CHECKSIG`: Verifies the digital signature using the public key.

Unlocking Script (ScriptSig)
<Signature> <PublicKey>

`<Signature>`: The digital signature created by the private key corresponding to the public key.
`<PublicKey>`: The public key that corresponds to the public key hash (Bitcoin address) provided in the locking script.

b. Use of different arithmetic, stack and flow control opcodes i.e. Addition, subtraction, max, min, if, else, swap, depth, drop, equal verify etc.

1. Arithmetic Opcodes

- **OP_ADD**: Adds two values.
- **OP_SUB**: Subtracts the second value from the first.
- **OP_NEGATE**: Negates a value.
- **OP_EQUAL**: Checks if two values are equal.
- **OP_EQUALVERIFY**: Checks if two values are equal and removes them from the stack if they are.

- **OP_MAX**: Takes the maximum of two values.
- **OP_MIN**: Takes the minimum of two values.

2. Stack Manipulation Opcodes

- **OP_DUP**: Duplicates the top value on the stack.
- **OP_SWAP**: Swaps the top two items on the stack.
- **OP_DEPTH**: Pushes the current stack depth onto the stack.
- **OP_DROP**: Removes the top item from the stack.
- **OP_PICK**: Copies an item from the stack by index.
- **OP_ROT**: Rotates the top three items on the stack.

3. Flow Control Opcodes

- **OP_IF**: If the value on top of the stack is true, the next part of the script will execute.
- **OP_ELSE**: If the preceding `OP_IF` evaluated false, the `OP_ELSE` part is executed.
- **OP_ENDIF**: Ends the `OP_IF` or `OP_ELSE` block.

Bitcoin Script Example:

```
<value1> <value2> OP_ADD                     # Add two numbers
OP_DUP OP_2 OP_MOD OP_EQUALVERIFY            # Check if result is divisible by 2 (even number)
OP_IF
   OP_SUB 5                                  # If even, subtract 5
OP_ELSE
   OP_ADD 5                                  # If odd, add 5
OP_ENDIF
```

Practical-10

Aim : Study and highlight the features of Solidity Framework

Solidity is a high-level, statically typed programming language designed specifically for writing smart contracts on the Ethereum blockchain. It is influenced by JavaScript, Python, and C++, making it accessible to developers familiar with these languages.

Key Features of Solidity:

- Smart Contract Oriented:

    Statically Typed Language: Variables must have their type specified at compile time, allowing for error detection before runtime.
- Ethereum Virtual Machine (EVM) Compatibility:

    Solidity code is compiled down to bytecode that the EVM can execute, ensuring seamless deployment on the Ethereum blockchain.
- Inheritance and Libraries:

    Supports multiple inheritance and the use of libraries, enhancing code modularity and reusability.
- Access Control Modifiers:

    Includes visibility modifiers (public, private, internal, external) to manage function and variable accessibility.
- Custom Data Structures:

    Allows for the creation of complex data structures such as mappings, structs, and dynamic arrays.
- Event Logging:

    Supports event emissions that log to the blockchain, enabling a mechanism for asynchronous communication and data monitoring.
- Safe Arithmetic:

    Includes libraries to prevent integer overflows and underflows, ensuring mathematical operations remain secure

Basic Solidity example

```
pragma solidity >=0.8.2 <0.9.0;

contract HelloWorld {
    string public message;

    // Constructor to set the initial message
    constructor(string memory _message) {    infinite gas 315800 gas
        message = _message;
    }

    // Function to update the message
    function updateMessage(string memory _newMessage) public {    infinite gas
        message = _newMessage;
    }
}
```

Practical-11

Aim : To Implement the following Programs in Solidity

A. First App
B. Primitive Data Types
C. Variables
D. Reading and Writing to a State Variable

A. First App (Basic Smart Contract):

```
pragma solidity >=0.8.2 <0.9.0;

contract FirstApp {
    string public message = "Hello, Ethereum!";

    function setMessage(string memory _newMessage) public {     infini
        message = _newMessage; // Set a new message
    }

    function getMessage() public view returns (string memory) {     i
        return message;
    }
}
```

B. Primitive Data Types

```
pragma solidity >=0.8.2 <0.9.0;

contract PrimitiveDataTypes {
    bool public isAvailable = true;
    uint8 public smallNumber = 42;
    uint256 public bigNumber = 123456789;
    int256 public signedNumber = -123;
    address public ownerAddress = msg.sender;
    bytes32 public hashValue = keccak256(abi.encodePacked("Solidity"));
}
```

C. Variables

```
pragma solidity >=0.8.2 <0.9.0;

contract VariableExample {
    // State variable
    uint256 public stateVar = 100;

    function useLocalVariable() public pure returns (uint256) {    359 gas
        uint256 localVar = 50; // Local variable        return localVar;
    }

    function getGlobalVariable() public view returns (address) {    379 gas
        return msg.sender; // Global variable
    }
}
```

D. Reading and Writing to a State Variable:

```
pragma solidity >=0.8.2 <0.9.0;

contract StateVariableRW {
    uint256 public storedNumber;

    // Function to write to the state variable
    function setNumber(uint256 _num) public {    22514 gas
        storedNumber = _num;
    }

    // Function to read the state variable
    function getNumber() public view returns (uint256) {    2454 gas
        return storedNumber;
    }
}
```

Practical-12

Aim : To Implement the following Programs Solidity Framework

A. If / Else
B. For and While Loop
C. Mapping

A. If / Else

```
pragma solidity >=0.8.2 <0.9.0;

contract IfElseExample {
    function checkNumber(int256 _num) public pure returns (string memory) {
        if (_num > 0) {
            return "Positive";
        } else if (_num == 0) {
            return "Zero";
        } else {
            return "Negative";
        }
    }
}
```

B. For and While Loop

```
contract LoopExample {
    function sumNumbers(uint256 _n) public pure returns (uint256) {    infinite gas
        uint256 sum = 0;
        for (uint256 i = 0; i <= _n; i++) {
            sum += i;
        }
        return sum;
    }

    function whileLoop(uint256 _n) public pure returns (uint256) {    infinite gas
        uint256 sum = 0;
        uint256 i = 0;
        while (i <= _n) {
            sum += i;
            i++;
        }
        return sum;
    }
}
```

C. Mapping

```
pragma solidity >=0.8.2 <0.9.0;

contract MappingExample {
    mapping(address => uint256) public balances;

    function setBalance(address _addr, uint256 _amount) public {
        balances[_addr] = _amount;
    }

    function getBalance(address _addr) public view returns (uint256) {
        return balances[_addr];
    }
}
```