**PRACTICAL 1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Implement Doubly Linked List in Append only Mode. | | |

**Program**

import java.util.Scanner;

public class DLL {

static Node head;

static class Block {

int rollNo;

String name;

String branch;

Block(int roll, String nameIn, String branchIn) {

rollNo = roll;

name = nameIn;

branch = branchIn;

}

}

static class Node {

Node prev;

Block data;

Node next;

Node(Block std) {

prev = null;

data = std;

next = null;

}

}

public static DLL addBlock(DLL list, Block data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

newNode.prev = temp;

}

return list;

}

public static void printList(DLL list) {

System.out.println("\nThe List is: ");

Node current = head;

while (current != null) {

System.out.println(current.data.rollNo + " - " + current.data.name + " - " + current.data.branch);

current = current.next;

}

}

public static void main(String[] args) {

DLL list = new DLL();

Scanner sc = new Scanner(System.in);

Block b1 = new Block(1, "Harsh", "CSE");

list = DLL.addBlock(list, b1);

for (int i = 1; i <= 3; i++) {

System.out.printf("Enter data for student %d:\n", i + 1);

int roll = sc.nextInt();

sc.nextLine();

String name = sc.nextLine();

String branch = sc.nextLine();

Block b = new Block(roll, name, branch);

list = DLL.addBlock(list, b);

}

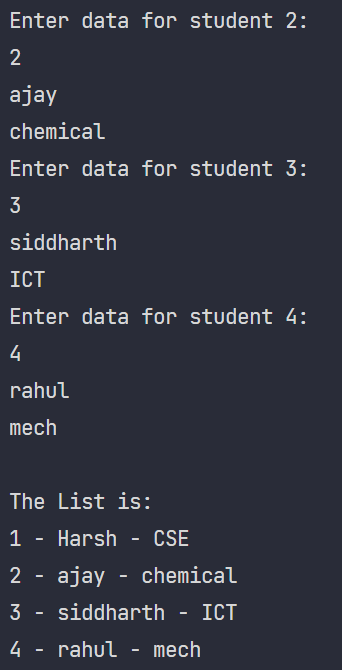
printList(list);

sc.close();

}

}

**Output:**



**PRACTICAL 2**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Implement SHA-1 and apply it on Doubly Linked List data. | | |

**Program**

import hashlib

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.prev = None

self.next = None

class DoublyLinkedList:

def \_\_init\_\_(self):

self.head = None

self.tail = None

def append(self, data):

new\_node = Node(data)

if self.head is None:

self.head = self.tail = new\_node

else:

new\_node.prev = self.tail

self.tail.next = new\_node

self.tail = new\_node

def get\_concatenated\_data(self):

current = self.head

data\_str = ""

while current:

data\_str += str(current.data)

current = current.next

return data\_str

def apply\_sha1(self):

concatenated\_data = self.get\_concatenated\_data()

sha1\_hash = hashlib.sha1(concatenated\_data.encode())

return sha1\_hash.hexdigest()

dll = DoublyLinkedList()

dll.append("Node1")

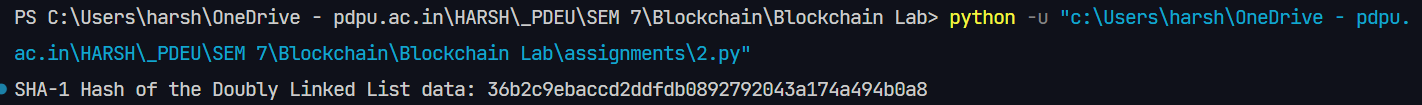
dll.append("Node2")

dll.append("Node3")

hash\_result = dll.apply\_sha1()

print("SHA-1 Hash of the Doubly Linked List data:", hash\_result)

**Output**



**PRACTICAL 3**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Exploring tools to understand the architecture of Blockchain. | | |

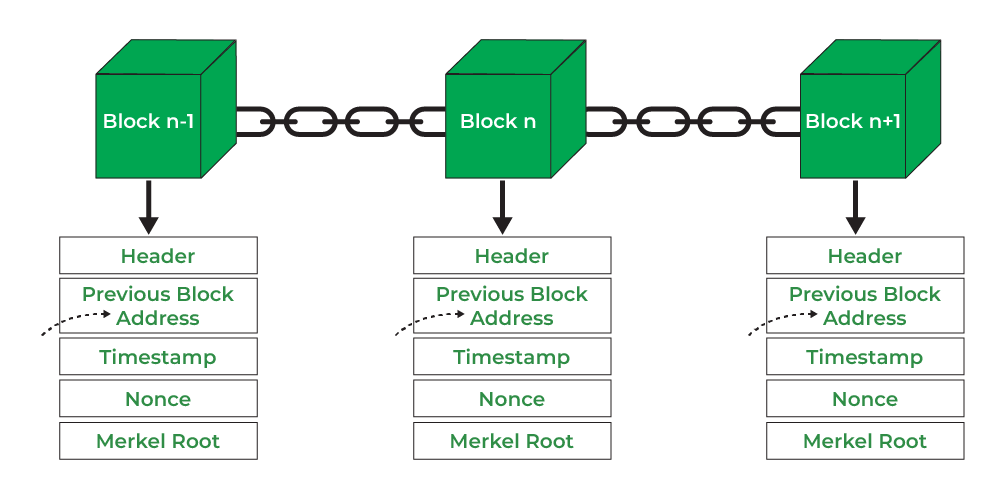
**Block**

A block in a blockchain is a digital record of transactions or data. Each block contains a list of transactions that have occurred within a specific period. The block also includes a reference to the previous block in the chain, creating a chronological order. This reference is typically a cryptographic hash of the previous block's contents.

**Blockchain**

Blockchain is a decentralized digital ledger that records and tracks transactions and assets in a business network. It's a shared, immutable database that stores a continuously growing list of ordered records, called blocks, which are linked using cryptography.

**Architecture of Blockchain**



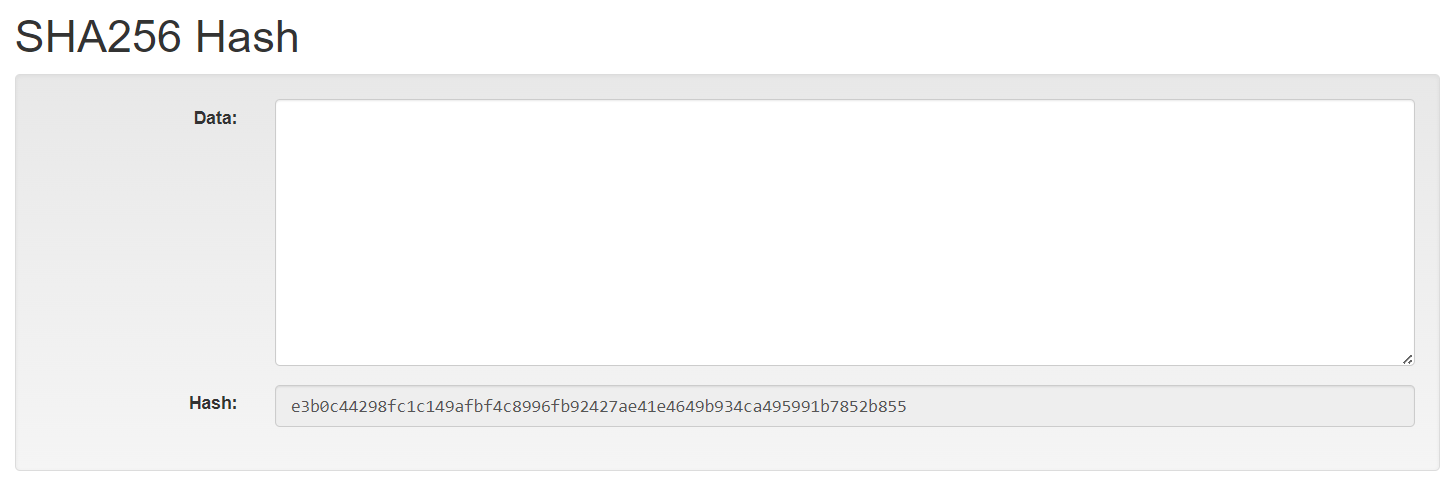
**Components of Block**

* **Block Header**: This contains metadata about the block, including:
* **Previous Block Hash**: A reference to the hash of the previous block in the chain.
* **Block Hash:** A unique identifier for the block generated by hashing the block header. This hash serves as the block's fingerprint and is used to link to the previous block, ensuring the chain's immutability.
* **Timestamp**: The time when the block was created.
* **Nonce**: A random number used in the mining process to ensure the hash meets certain conditions.

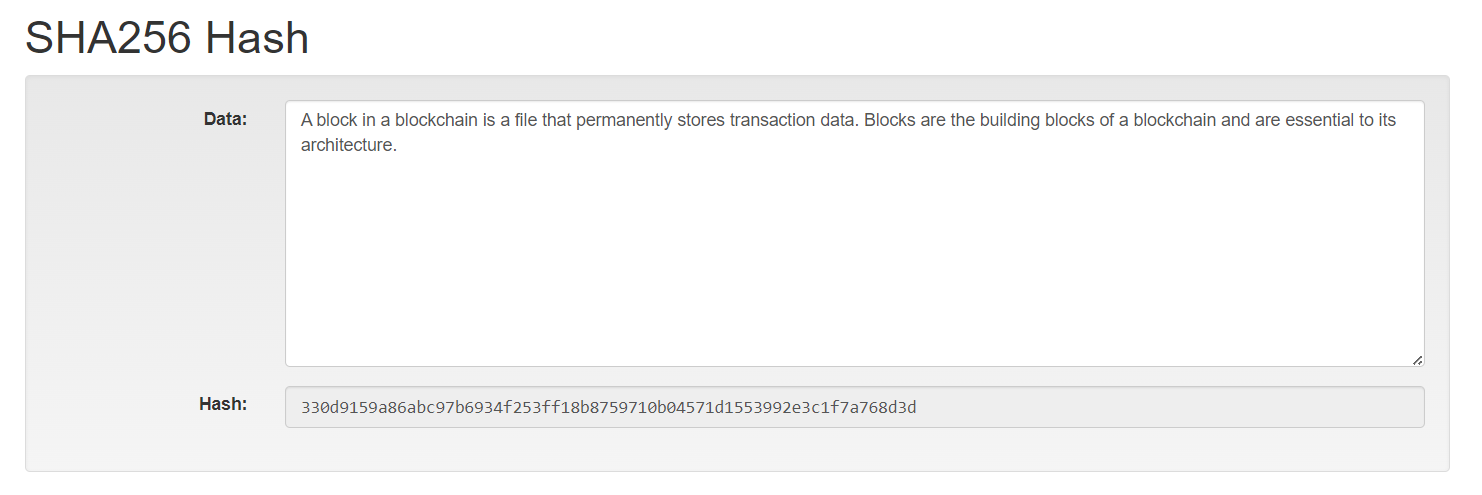
**Demonstration**

**SHA256**

* SHA256 Hash for Empty Data

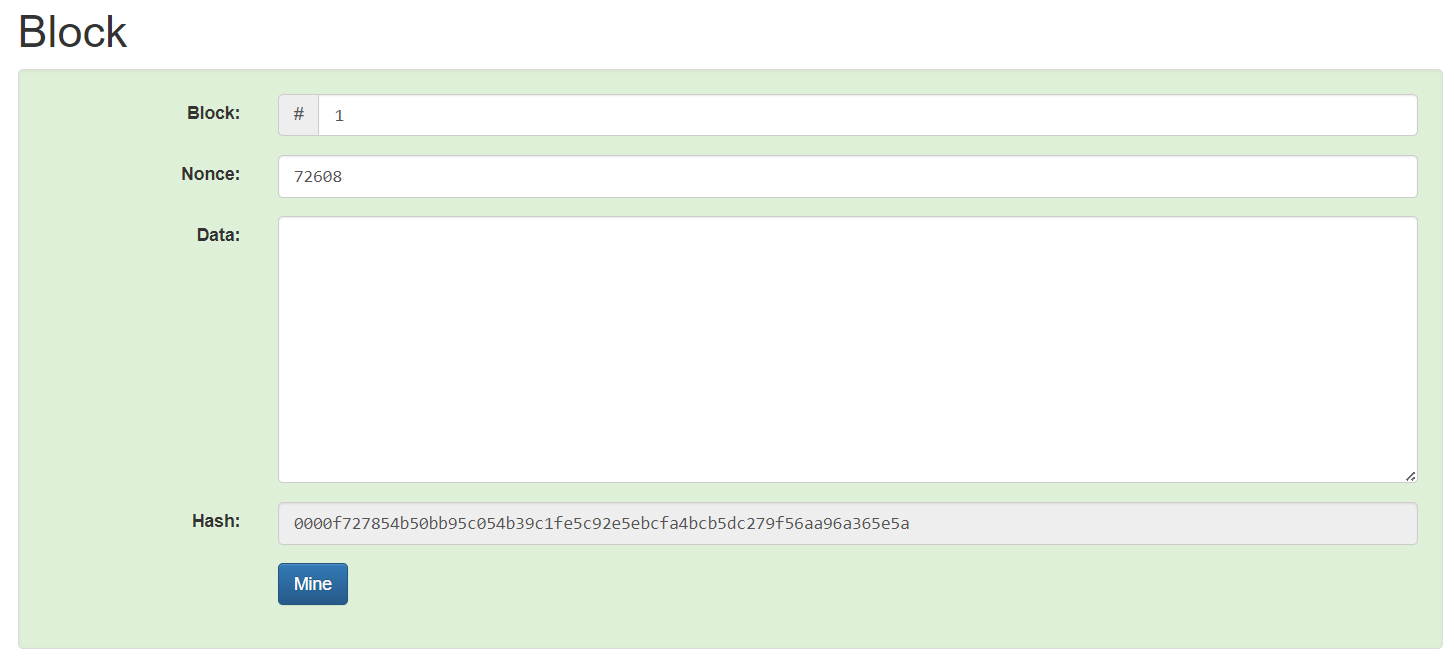


* SHA256 Hash for some data

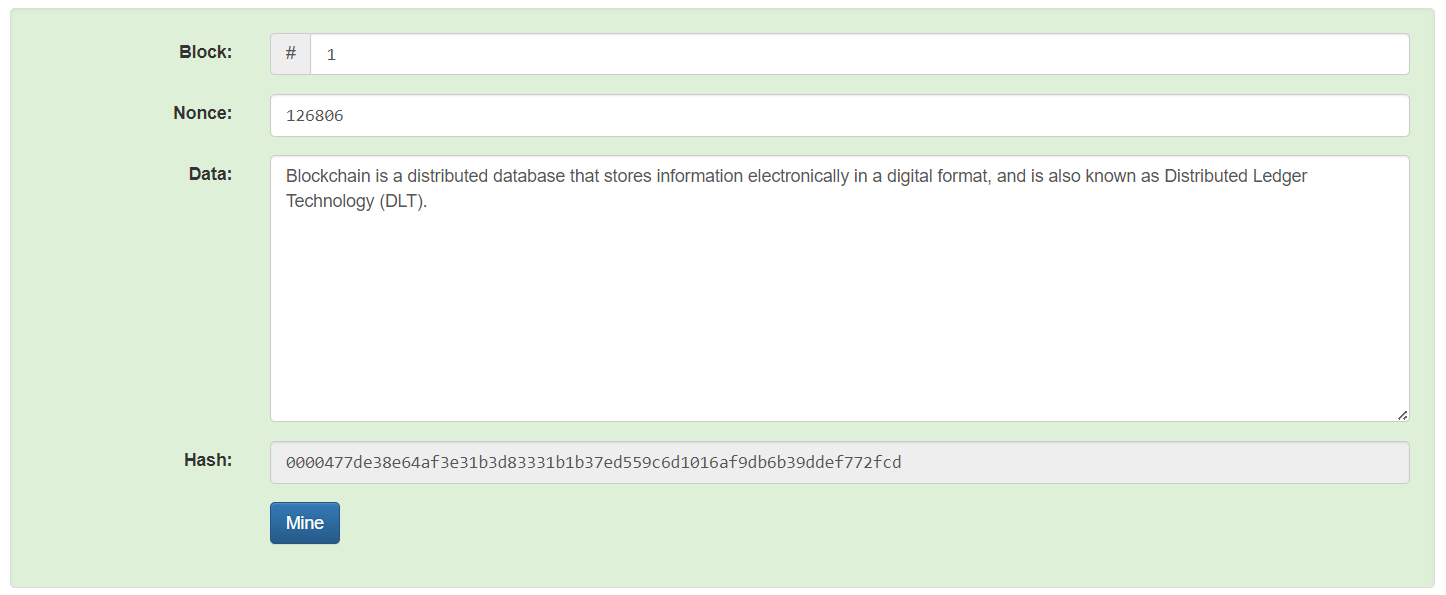


**Block**

* Empty Block

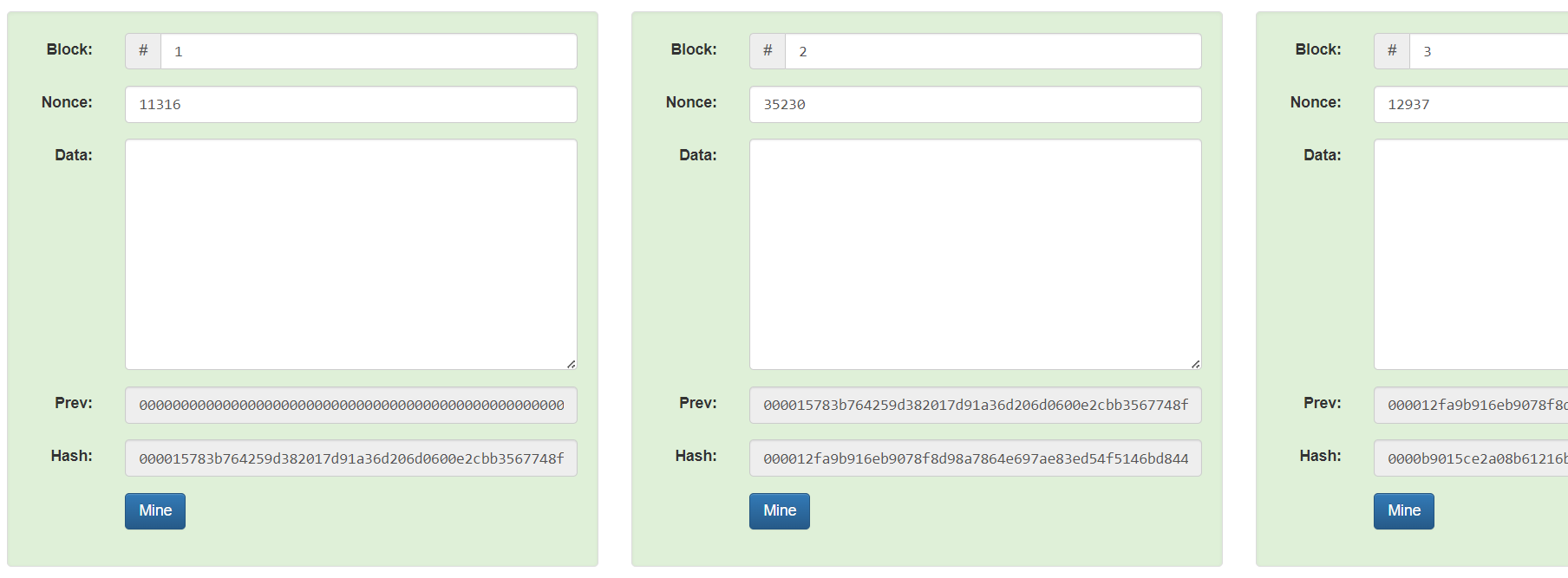


* Block - 1 after Mining

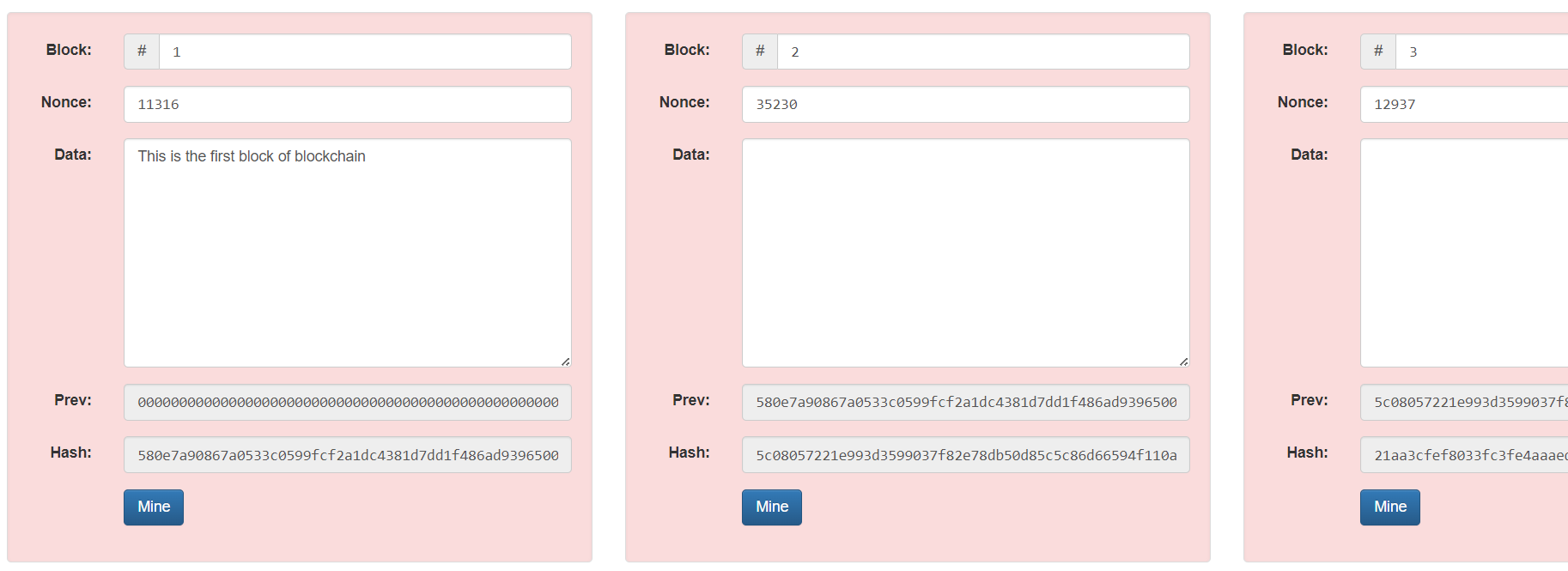


**Blockchain**

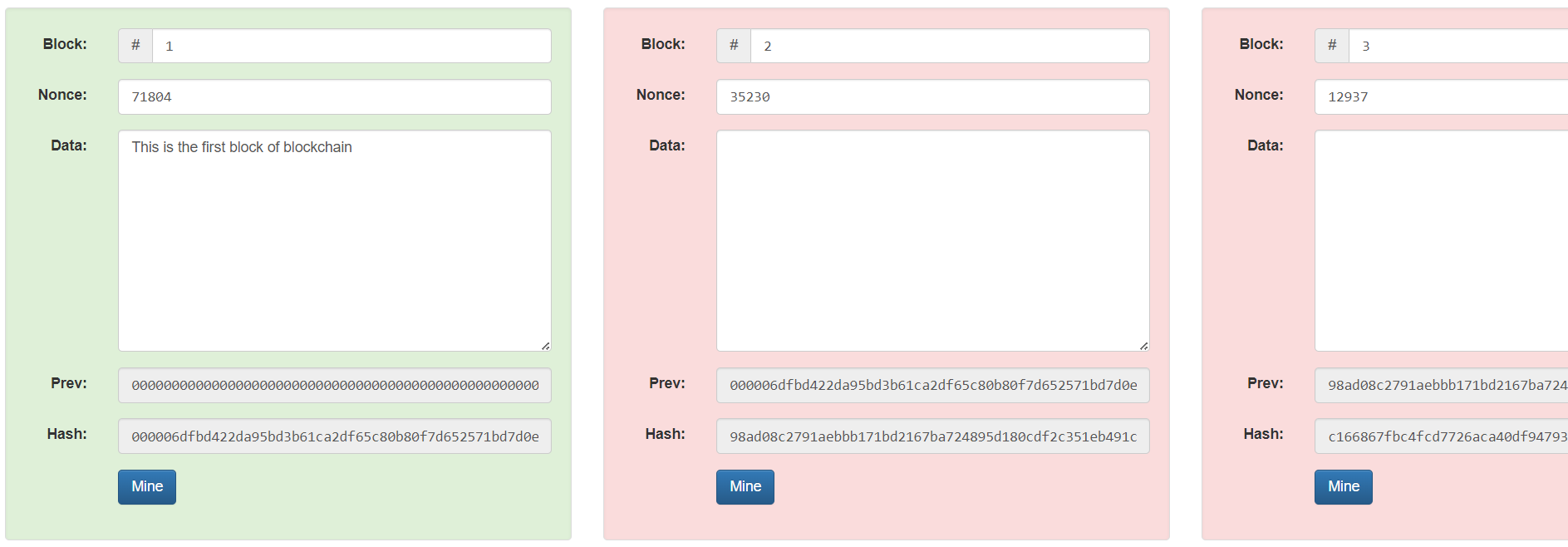
* Empty Blockchain

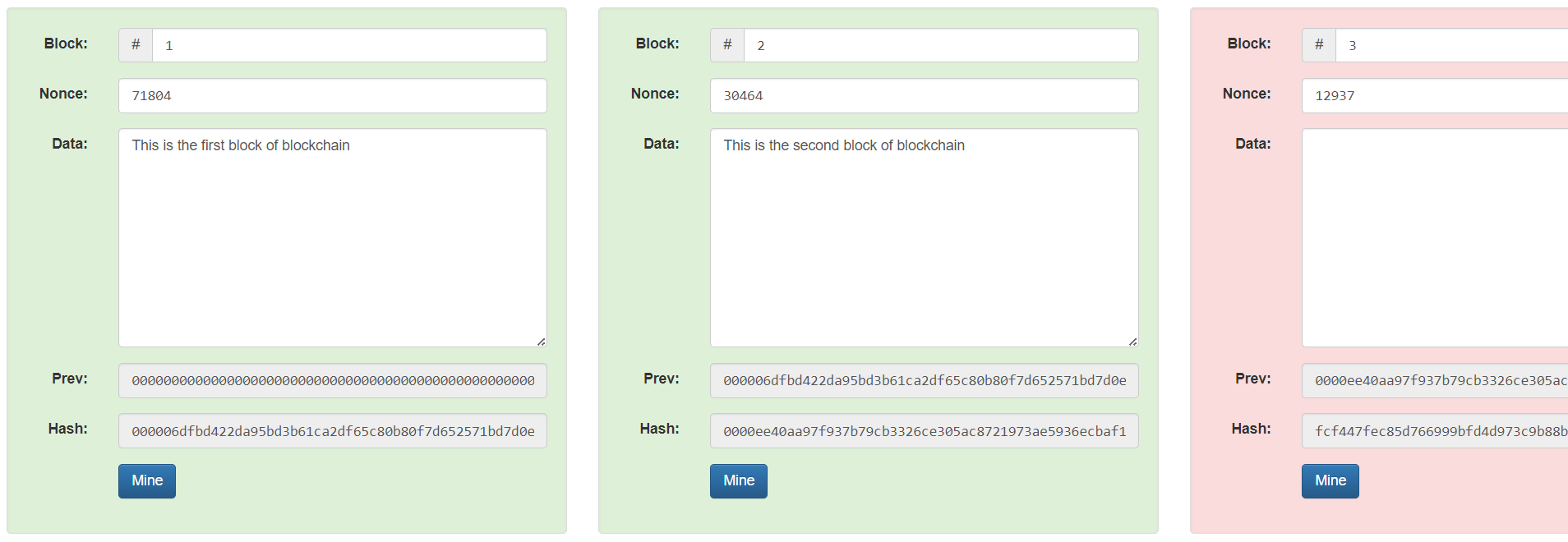


* Blockchain before Mining



* Blockchain after mining





**Significance of Leading zeros in a hash**

The leading zeros indicate the difficulty level set by the blockchain network. Miners must find a hash that meets this specific criterion. Miners repeatedly change the nonce (a random or semi-random number) and recompute the hash of the block until they find a hash that starts with the required number of leading zeros.

The network adjusts the difficulty level periodically (e.g., every 2016 blocks in Bitcoin) to ensure that blocks are mined at a consistent rate, typically every 10 minutes. This adjustment is achieved by increasing or decreasing the number of leading zeros required. The requirement for leading zeros makes it computationally expensive to find a valid hash, providing security to the network by making it difficult and resource-intensive to alter any previous blocks.

The process of finding a hash with the requisite number of leading zeros ensures that adding new blocks to the blockchain requires a significant amount of computational effort, thereby maintaining the integrity and security of the blockchain.

**PRACTICAL 4**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Implement PoW Consensus Mechanism on your own Blockchain. | | |

**Proof of Work (PoW)**

Proof of work (PoW) is a blockchain consensus mechanism that requires significant computing effort from a network of devices. The concept was adapted from digital tokens by Hal Finney in 2004 through the idea of "reusable proof of work" using the 160-bit secure hash algorithm 1 (SHA-1).

**Program**

import hashlib

import time

class Block:

def \_\_init\_\_(self, index, previous\_hash, data, nonce=0):

self.index = index

self.timestamp = time.time()

self.previous\_hash = previous\_hash

self.data = data

self.nonce = nonce

self.hash = self.calculate\_hash()

def calculate\_hash(self):

block\_string = f"{self.index}{self.timestamp}{self.previous\_hash}{self.data}{self.nonce}".encode()

return hashlib.sha256(block\_string).hexdigest()

def \_\_str\_\_(self):

return (

f"Block Index : {self.index}\n"

f"Timestamp : {time.ctime(self.timestamp)}\n"

f"Previous Hash : {self.previous\_hash}\n"

f"Hash : {self.hash}\n"

f"Data : {self.data}\n"

f"Nonce : {self.nonce}\n"

f"{'-'\*41}"

)

class Blockchain:

def \_\_init\_\_(self):

self.chain = [self.create\_genesis\_block()]

def create\_genesis\_block(self):

return Block(0, "0", "Genesis Block")

def get\_latest\_block(self):

return self.chain[-1]

def add\_block(self, data):

latest\_block = self.get\_latest\_block()

new\_block = Block(len(self.chain), latest\_block.hash, data)

new\_block = self.proof\_of\_work(new\_block)

self.chain.append(new\_block)

def proof\_of\_work(self, block, difficulty=4):

while block.hash[:difficulty] != "0" \* difficulty:

block.nonce += 1

block.hash = block.calculate\_hash()

return 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.hash != current\_block.calculate\_hash():

return False

if current\_block.previous\_hash != previous\_block.hash:

return False

return True

def \_\_str\_\_(self):

return "\n".join(str(block) for block in self.chain)

blockchain = Blockchain()

while True:

data = input("Enter transaction data for the new block (or 'q' to quit): ")

if data.lower() == "q":

break

blockchain.add\_block(data)

print("\nBlock added successfully!")

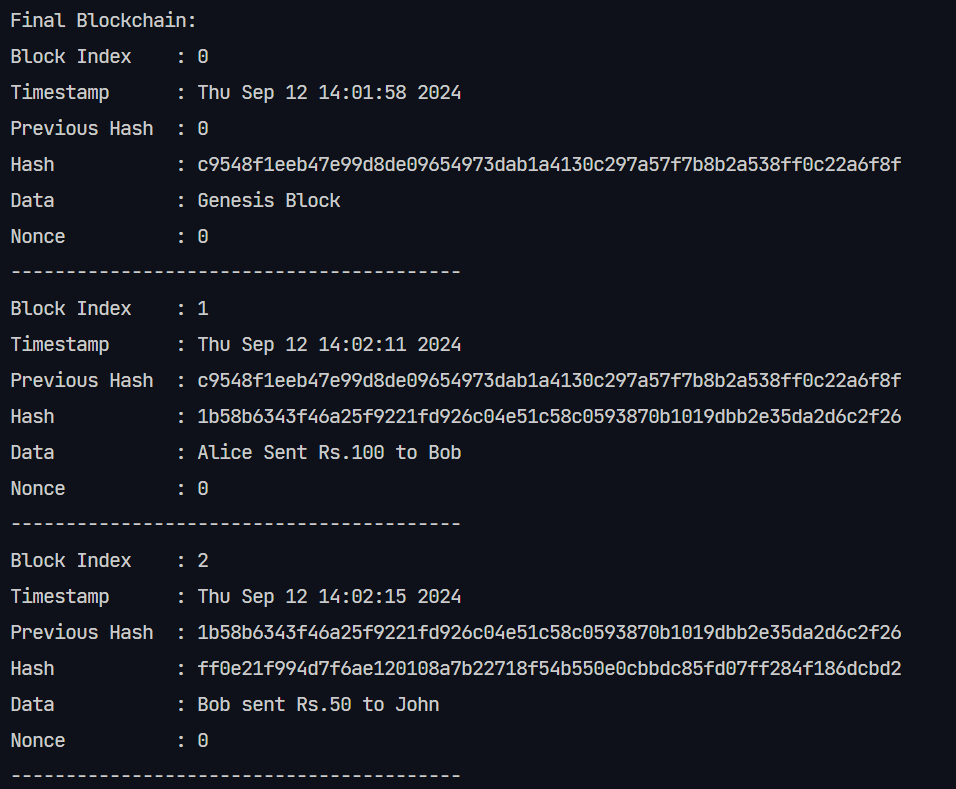
print("\nFinal Blockchain:")

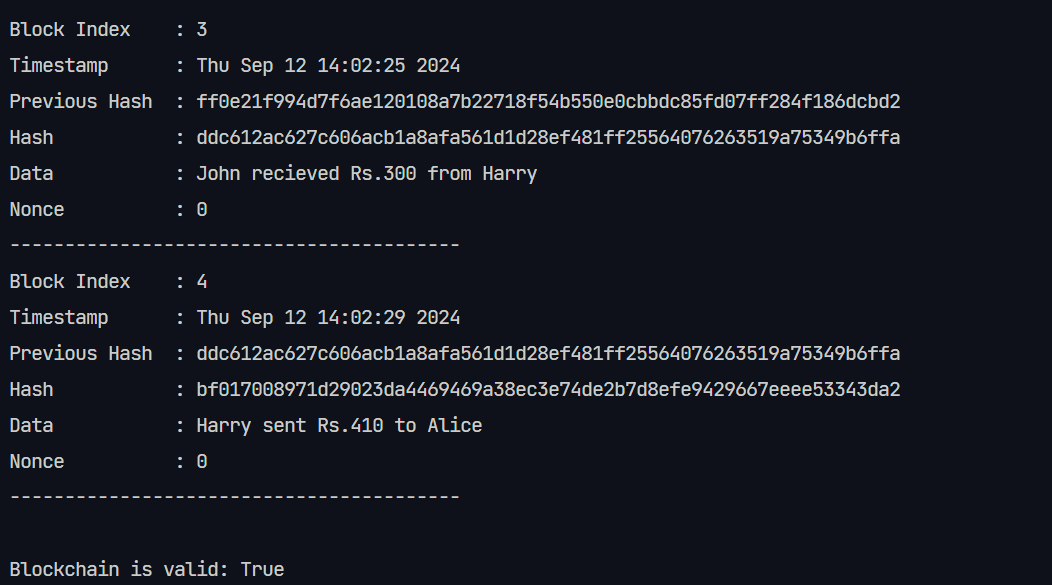
print(blockchain)

print("\nBlockchain is valid:", blockchain.is\_chain\_valid())

**Output**







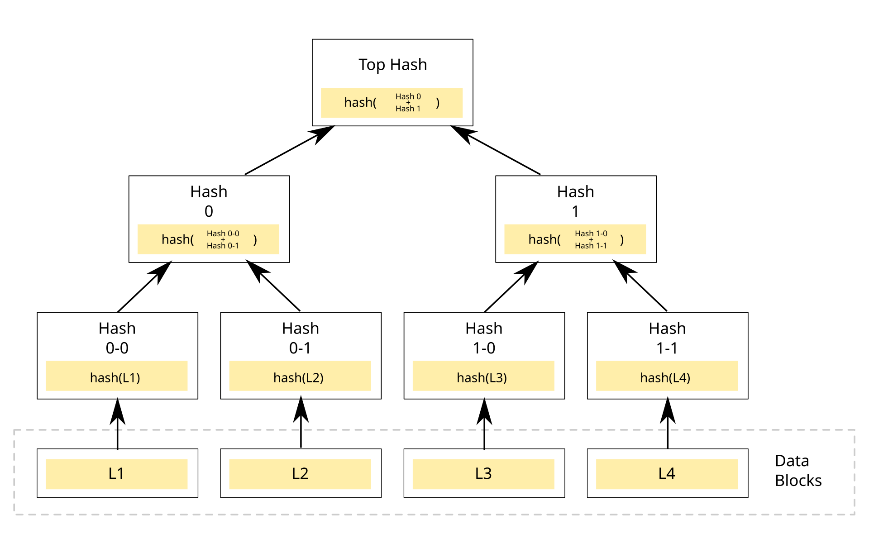
**PRACTICAL 5**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Implementation of Merkle Tree. | | |

**Merkle Tree**

A Merkle tree is a data encryption structure used in data management applications where data is sent through a hashing algorithm in different ways to create a hash that represents all of the data in a file.

|  |  |  |
| --- | --- | --- |
| **Complexity** | **Average** | **Worst** |
| **Space** | O(n) | O(n) |
| **Search** | O(log2(n)) | O(logk(n)) |
| **Insert** | O(log2(n)) | O(logk(n)) |
| **Delete** | O(log2(n)) | O(logk(n)) |



* Merkle tree also known as hash tree is a data structure used for data verification and synchronization.
* It is a tree data structure where each non-leaf node is a hash of its child nodes.
* All the leaf nodes are at the same depth and are as far left as possible.
* It maintains data integrity and uses hash functions for this purpose.

**Program**

import hashlib

class MerkleTreeNode:

    def \_\_init\_\_(self, value):

        self.left = None

        self.right = None

        self.value = value

        self.hashValue = hashlib.sha256(value.encode("utf-8")).hexdigest()

def buildTree(leaves, f):

    nodes = []

    for i in leaves:

        nodes.append(MerkleTreeNode(i))

    while len(nodes) != 1:

        temp = []

        for i in range(0, len(nodes), 2):

            node1 = nodes[i]

            if i + 1 < len(nodes):

                node2 = nodes[i + 1]

            else:

                temp.append(nodes[i])

                break

            f.write(

                "Left child : " + node1.value + " | Hash : " + node1.hashValue + " \n"

            )

            f.write(

                "Right child : " + node2.value + " | Hash : " + node2.hashValue + " \n"

            )

            concatenatedHash = node1.hashValue + node2.hashValue

            parent = MerkleTreeNode(concatenatedHash)

            parent.left = node1

            parent.right = node2

            f.write(

                "Parent(concatenation of "

                + node1.value

                + " and "

                + node2.value

                + ") : "

                + parent.value

                + " | Hash : "

                + parent.hashValue

                + " \n"

            )

            temp.append(parent)

        nodes = temp

    return nodes[0]

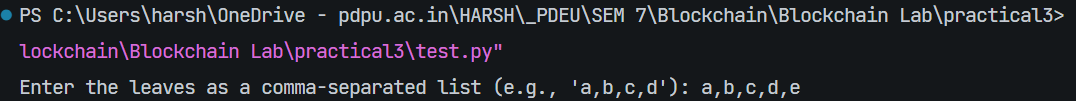
inputString = input("Enter the leaves as a comma-separated list (e.g., 'a,b,c,d'): ")

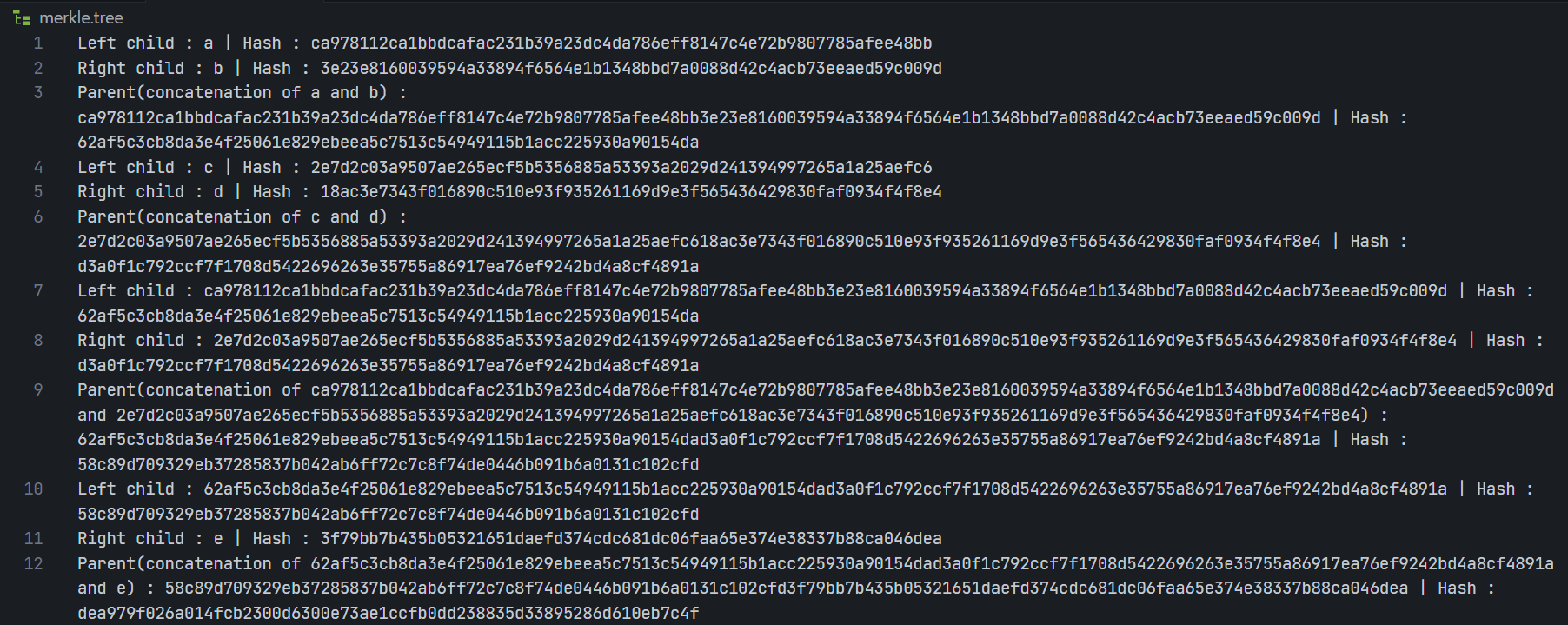
leaves = inputString.split(",")

with open("merkle.tree", "w") as f:

    root = buildTree(leaves, f)

**Output**





**PRACTICAL 6**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Implement PoS Consensus Mechanism on your own Blockchain. | | |

**Proof of Stake (PoS)**

Proof of stake (PoS) is a consensus protocol in blockchains. It is a way to decide which user or users validate new blocks of transactions and earn a reward for doing so correctly. A **stake** is value/money we bet on a certain outcome. The process is called staking.

**Program**

import hashlib

import time

import random

class Block:

def \_\_init\_\_(self, index, previous\_hash, data, validator, nonce=0):

self.index = index

self.timestamp = time.time()

self.previous\_hash = previous\_hash

self.data = data

self.validator = validator

self.nonce = nonce

self.hash = self.calculate\_hash()

def calculate\_hash(self):

block\_string = f"{self.index}{self.timestamp}{self.previous\_hash}{self.data}{self.validator}{self.nonce}".encode()

return hashlib.sha256(block\_string).hexdigest()

def \_\_str\_\_(self):

return (

f"Block Index : {self.index}\n"

f"Timestamp : {time.ctime(self.timestamp)}\n"

f"Previous Hash : {self.previous\_hash}\n"

f"Hash : {self.hash}\n"

f"Data : {self.data}\n"

f"Validator : {self.validator}\n"

f"Nonce : {self.nonce}\n"

f"{'-'\*41}"

)

class Validator:

def \_\_init\_\_(self, name, stake):

self.name = name

self.stake = stake

class Blockchain:

def \_\_init\_\_(self):

self.chain = [self.create\_genesis\_block()]

self.validators = [] # List of validators

def create\_genesis\_block(self):

return Block(0, "0", "Genesis Block", "None")

def get\_latest\_block(self):

return self.chain[-1]

def add\_validator(self, name, stake):

validator = Validator(name, stake)

self.validators.append(validator)

def choose\_validator(self):

total\_stake = sum(v.stake for v in self.validators)

chosen\_value = random.uniform(0, total\_stake)

cumulative\_stake = 0

for validator in self.validators:

cumulative\_stake += validator.stake

if cumulative\_stake >= chosen\_value:

return validator.name

def add\_block(self, data):

latest\_block = self.get\_latest\_block()

chosen\_validator = self.choose\_validator()

new\_block = Block(len(self.chain), latest\_block.hash, data, chosen\_validator)

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.hash != current\_block.calculate\_hash():

return False

if current\_block.previous\_hash != previous\_block.hash:

return False

return True

def \_\_str\_\_(self):

return "\n".join(str(block) for block in self.chain)

blockchain = Blockchain()

blockchain.add\_validator("Validator1", 10) # Validator with 10 units of stake

blockchain.add\_validator("Validator2", 20) # Validator with 20 units of stake

blockchain.add\_validator("Validator3", 5) # Validator with 5 units of stake

while True:

data = input("Enter transaction data for the new block (or 'q' to quit): ")

if data.lower() == "q":

break

blockchain.add\_block(data)

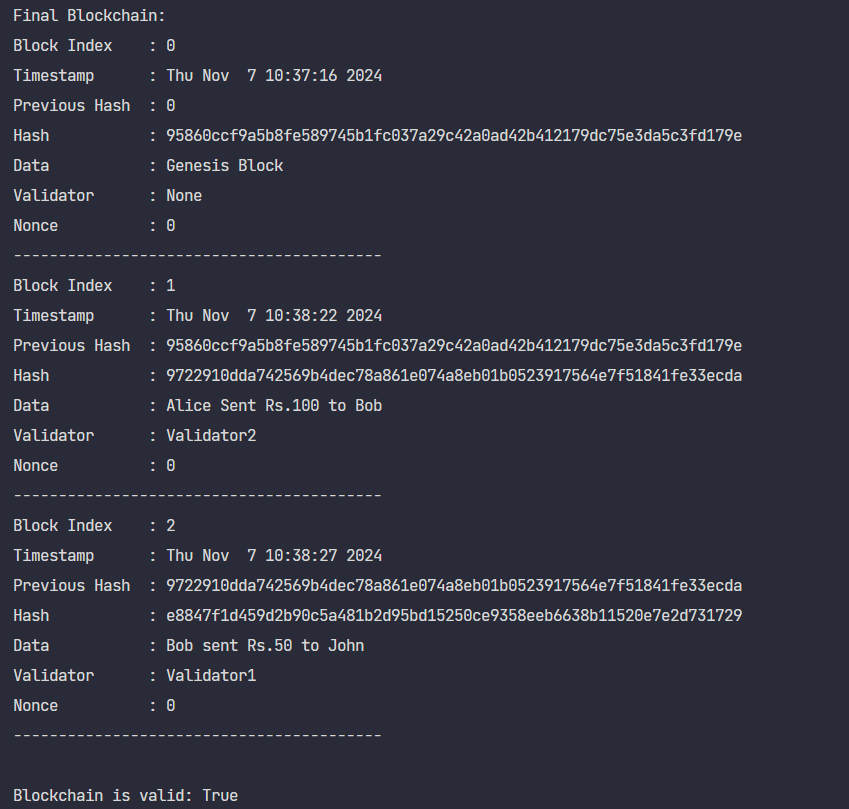
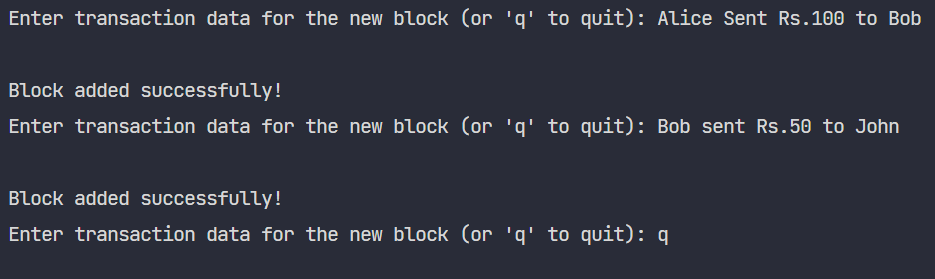
print("\nBlock added successfully!")

print("\nFinal Blockchain:")

print(blockchain)

print("\nBlockchain is valid:", blockchain.is\_chain\_valid())

**Output**



**PRACTICAL 7**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Learn about Smart Contracts and create a Hello World Program in Solidity. | | |

**Smart Contract**

A Smart Contract is a computer program that directly and automatically controls the transfer of digital assets between the parties under certain conditions. A smart contract works in the same way as a traditional contract while also automatically enforcing the contract. Just like a traditional contract is enforceable by law, smart contracts are enforceable by code.

**Solidity**

Solidity is a programming language designed specifically for writing smart contracts on the Ethereumblockchain. Smart contracts are self-executing programs that automatically enforce the terms of a contract when certain conditions are met. Solidity allows developers to write these contracts in a secure, transparent, and decentralized manner.

* Solidity is a high-level programming language designed for implementing smart contracts.
* It is a statically typed object-oriented (contract-oriented) language.
* Solidity is highly influenced by Python, C++, and JavaScript which run on the Ethereum Virtual Machine (EVM).
* Solidity can be used to create contracts like voting, blind auctions, crowdfunding, multi-signature wallets, etc.

**Program**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract HelloWorld {

string enter;

function set(string memory value) public {

enter = value;

}

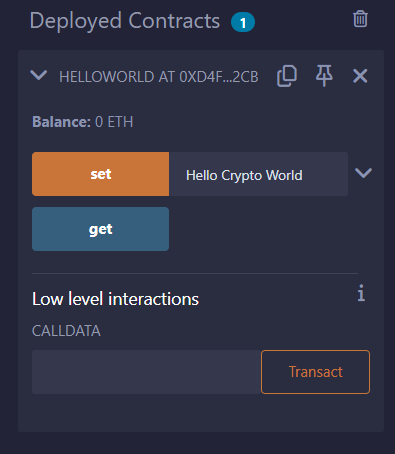
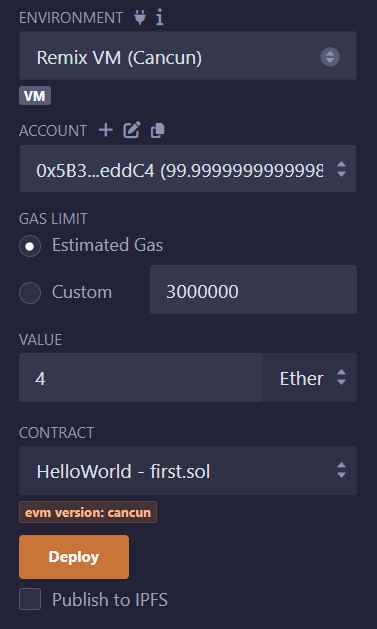
function get() public view returns(string memory) {

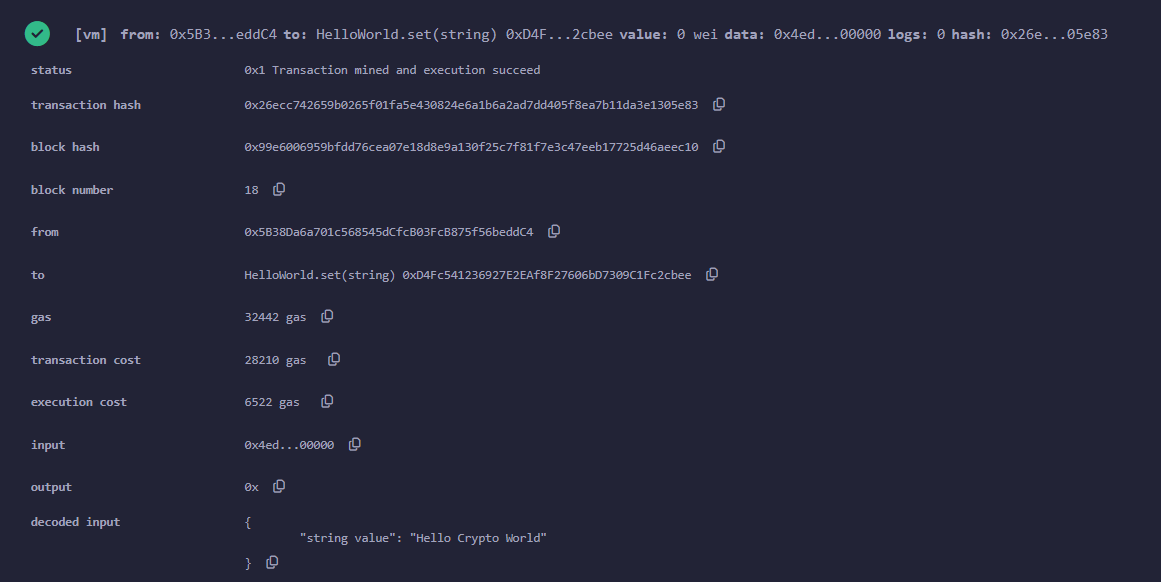
return enter;

}

}

**Output**





**PRACTICAL 8**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Implement Smart Contract in Solidity to send ether from one person to another. | | |

**Smart Contract**

A Smart Contract is a computer program that directly and automatically controls the transfer of digital assets between the parties under certain conditions. A smart contract works in the same way as a traditional contract while also automatically enforcing the contract. Just like a traditional contract is enforceable by law, smart contracts are enforceable by code.

**Program**

pragma solidity ^0.8.0;

contract SimpleTransfer {

    event Transfer(address indexed from, address indexed to, uint256 amount);

    function sendEther(address payable \_to) public payable {

        require(msg.value > 0, "You need to send some ether");

        require(\_to != address(0), "Invalid address");

        \_to.transfer(msg.value);

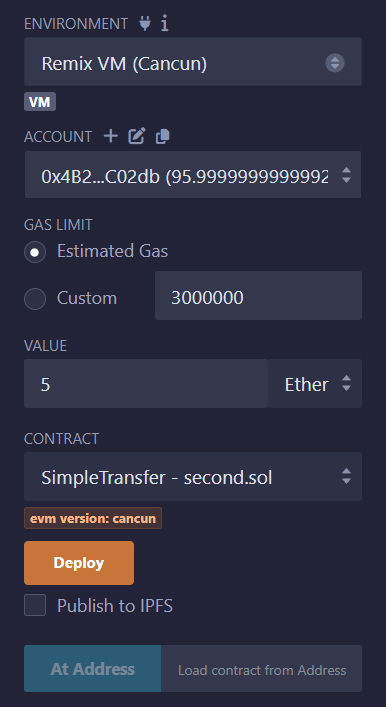
        emit Transfer(msg.sender, \_to, msg.value);

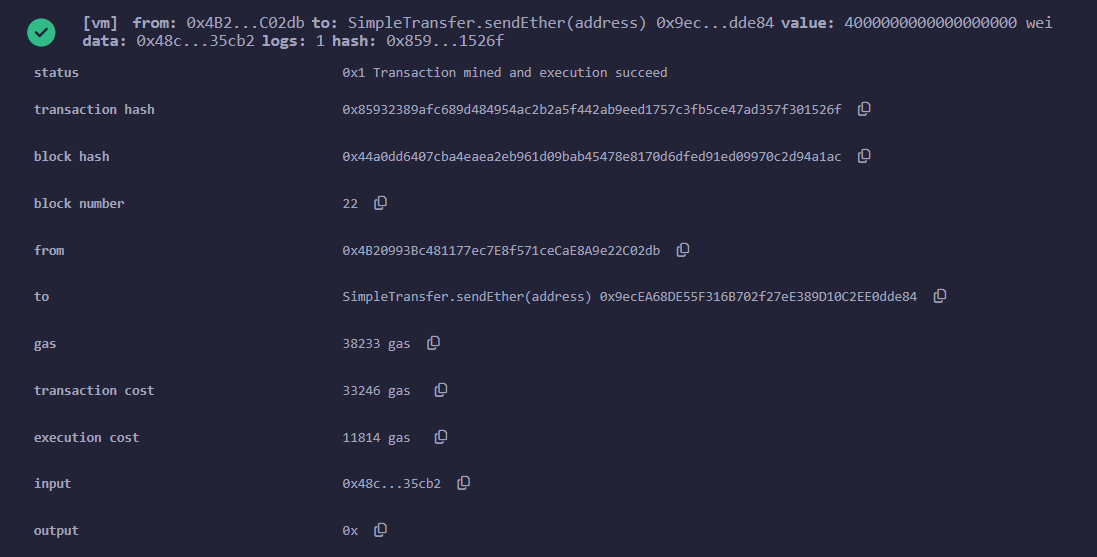
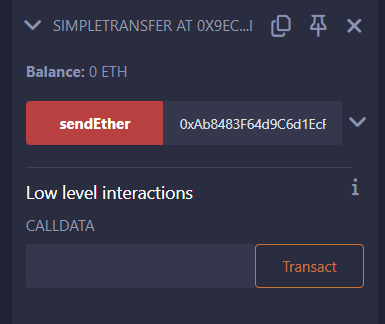
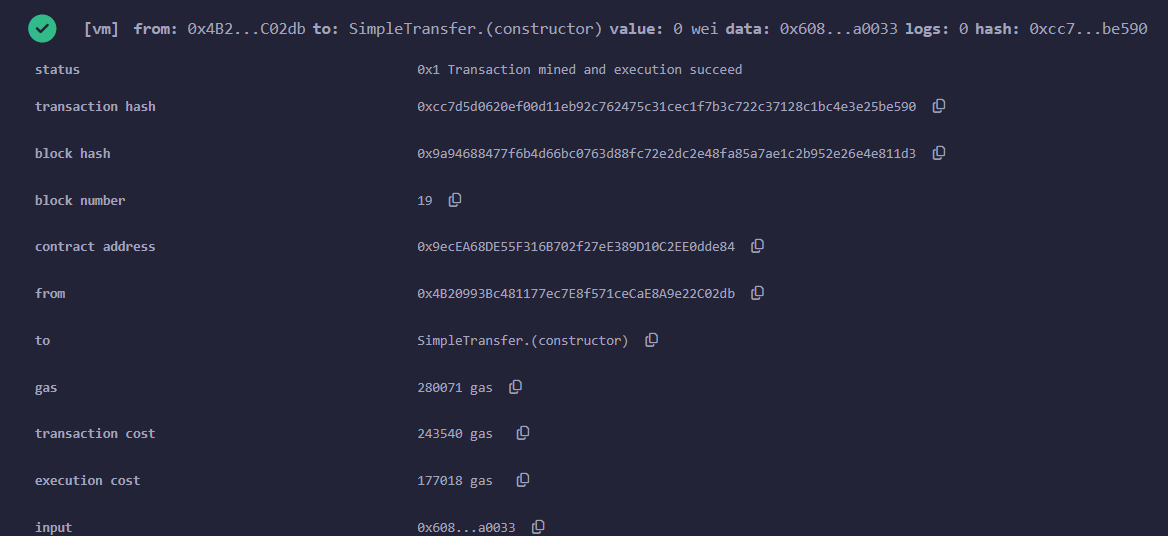
    }

    receive() external payable {}

}

**Output**





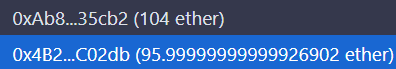


Figure: 4 ethers sent from **0x4B2…C02db** to **0xAb8…35cb2**

**PRACTICAL 9**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Roll No.:** | 21BCP359 |
| **Division:** | 6 | **Batch:** | G11 |
| **Aim:** | Implement your own blockchain in Solidity. | | |

**Program**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract BasicBlockchain {

    struct Block {

        uint index;

        uint timestamp;

        string data;

        string previousHash;

    }

    Block[] public chain;

    constructor() {

        // Create the Genesis block

        createGenesisBlock();

    }

    function createGenesisBlock() private {

        // Genesis block with hardcoded values

        chain.push(Block({

            index: 0,

            timestamp: block.timestamp,

            data: "Genesis Block - CrowdFunding Records",

            previousHash: "0"

        }));

    }

    function addBlock(string memory data) public {

        Block memory latestBlock = chain[chain.length - 1];

        uint newIndex = latestBlock.index + 1;

        uint newTimestamp = block.timestamp;

        string memory previousHash = latestBlock.previousHash;

        chain.push(Block({

            index: newIndex,

            timestamp: newTimestamp,

            data: data,

            previousHash: previousHash

        }));

    }

    function getBlock(uint index) public view returns (Block memory) {

        require(index < chain.length, "Block does not exist.");

        return chain[index];

    }

    function getChainLength() public view returns (uint) {

        return chain.length;

    }

}

**Output**

